

CS3219 Software Engineering Principles and Patterns

AY23/24 Sem 1

Project Report Group 32

Team Members	Student No.	Email
Anthony Neo Ming Hong	A0234936X	E0726936@u.nus.edu
Caden Cheong Jun Kai	A0231110J	E0701969@u.nus.edu
Chia Jeremy	A0234860E	E0726860@u.nus.edu
Lo Ho Yin	A0241049L	E0775647@u.nus.edu
Lim Jia Yi Venus	A0241055R	E0775653@u.nus.edu

Table of Contents

1. Background and Purpose of Project.....	5
2. Sub-group Contributions	6
3. Product Requirements	8
3.1. Functional Requirements	8
3.2. Non-Functional Requirements	11
4. Developer Documentation.....	13
4.1. Technology Stack	13
4.2. System Architecture	14
4.2.1. Local Deployment Architecture (without Docker).....	14
4.2.2. Local Deployment Architecture (with Docker).....	15
4.2.3. Cloud Deployment Architecture	16
4.3. Development Process.....	17
4.3.1. Sprint Iterations.....	17
4.3.2. Collaborative Tools	17
4.3.3. Continuous Integration/Continuous Development (CI/CD)	18
5. General Design Pattern and Decisions	21
5.1. Monolith vs Microservices	21
5.2. N-tiered Architecture and Controller-Service-Repository Pattern Within Each Microservice (Vertically-Sliced).....	22
5.3. API Gateway with Nginx.....	25
5.4. Use of ReactJS and Node.js with Express.js.....	25
5.5. Use of MongoDB	26
6. Implementation of Microservices	27
6.1. User Microservice	27
6.1.1. User Authentication and Authorization	27
6.1.2. User Schema	28
6.1.3. Design Considerations.....	29
6.2. Question Microservice	30
6.2.1. Question Schema	30
6.2.2. Question Generation System	31
6.3. Auth Microservice	33

6.3.1.	Open Authorization 2.0 Framework.....	33
6.3.2.	High-Level Overview.....	34
6.3.3.	JSON Web Token (JWT) Implementation	36
6.3.4.	Design Considerations.....	37
6.4.	Match Microservice	38
6.4.1.	Dynamic Queue Generation with a Common Queue.....	38
6.4.2.	Request-Reply Communication Pattern	39
6.4.3.	Interactions between Matching and Collaboration Service	40
6.5.	Collaboration Microservice	42
6.5.1.	Publisher-Subscriber Pattern for Real-time Collaboration	42
6.5.2.	Chosen Programming Languages for User	43
6.5.3.	Approach Used to Enable Rejoining of Collaboration Room	43
6.5.4.	Storage Infrastructure Considerations.....	46
6.6.	History Microservice.....	48
6.6.1.	Design Considerations.....	49
6.6.2.	History Model.....	53
6.6.3.	Deletion of Users and Questions.....	53
6.7.	Execution Microservice	57
6.7.1.	Code Execution Workflow	57
6.7.2.	Design Considerations.....	58
7.	Deployment Process.....	61
7.1.	Deployment Requirements.....	62
7.2.	Deployment Considerations	63
7.2.1.	AWS Application Load Balancer	63
7.3.	Stress Test on Deployment Server	64
8.	Suggestions for Improvements and Enhancements	65
8.1.	Voice Calls and Video Conferencing.....	65
8.2.	Expansion of Language Support.....	65
8.3.	Execution Microservice Roadmap.....	65
8.4.	Code Testing	66
9.	Reflections and Learning Points.....	67
9.1.	Importance of Design Patterns.....	67

9.2.	Importance of Software Requirements	67
9.3.	Room for Improvement	68

1. Background and Purpose of Project

Being students ourselves, we intimately grasp the challenges that job applications and technical interviews pose. It can be highly frustrating when you struggle with time constraints and conveying your thought process during coding interviews.

With such a vast array of interview questions, attempting to memorize solutions for all of them is an impractical solution. A more efficient approach is to engage in regular practice across various question categories, helping you understand the strategies necessary for interviews. However, going through this process alone can be isolating and tiresome.

Introducing PeerPrep, our project aimed at addressing these challenges. PeerPrep connects students, allowing them to participate in collaborative mock technical interviews. This fosters a cooperative environment, where both participants can actively work together in real-time to develop their solution to the provided question, enhancing their preparation.

PeerPrep empowers users to choose the difficulty level and programming language of their choice, enabling them to target their weaker areas more effectively. When paired with fellow students who share the same preferences, they can learn from each other's approaches to the same coding problems. Users can collaborate on real-time code development using a shared rich-text editor, similar to the tools used in actual technical interviews. Additionally, they can communicate through a real-time chat box to discuss their approaches or seek clarifications. This pairing not only alleviates the difficulty of solo interview preparation but also serves as an effective learning tool, making the most of the features offered by PeerPrep.

We would like to take this opportunity to thank our mentor, Ms. Vishruti Ranjan, for her unwavering support. It would have been difficult to navigate this journey without her guidance and advice. We would also like to extend our gratitude to Professors Akshay Narayan and Bimlesh Wadhwa for offering us the opportunity to engage in a project of captivating scope. Their provision of extensive knowledge in the field of Software Engineering has been invaluable and greatly assisted us throughout this project.

2. Sub-group Contributions

Member	Tasks Completed
Anthony Neo Ming Hong Chia Jeremy Lo Ho Yin	<p>N1: Communication service</p> <ul style="list-style-type: none"> Implement text-based chat service between users in a collaboration room Add event notification when there is a change in room state (i.e., Change in question, language, and users joining/leaving the room) <p>N3: Code Execution service</p> <ul style="list-style-type: none"> Implement execution service in a sandboxed environment Implement compiler / executor for Python, Java and Javascript Handle various cases of errors, such as timeout or code errors <p>N4: Enhance Question service</p> <ul style="list-style-type: none"> Enable filtering of questions by title, complexity, topics Enable retrieving of questions on the fly during a session initiation (collaboration room) <p>N11: API Gateway (Nginx)</p> <ul style="list-style-type: none"> Implemented an API Gateway that redirects API requests to relevant microservices
Caden Cheong Jun Kai Lim Jia Yi Venus	<p>N2: History service</p> <ul style="list-style-type: none"> Implement records of user's submission attempts, including their details Implement statistical figures such as the breakdown of submitted questions complexity, number of un-attempted questions and the heatmap of submission frequency <p>N5: Enhance Collaboration service</p> <ul style="list-style-type: none"> Provide a code editor with code formatting, syntax highlighting and code autocomplete for multiple languages Provide a boilerplate which users can reset their view to

	<ul style="list-style-type: none"> • Implement an added collaboration feature which shows the partner's live cursor position <p>N9: Deployment of PeerPrep on cloud services (AWS)</p> <ul style="list-style-type: none"> • Implement CI/CD pipeline which ensures continuous deployment on AWS • Implement Blue-Green deployment
--	--

3. Product Requirements

3.1. Functional Requirements

Table 1 highlights the functional requirements for our product, grouped according to their microservice:

- User Service
- Matching Service
- Question Service
- Collaboration Service
- History Service
- Execution Service

User includes both normal user and maintainers.

Table 1: Functional Requirements of our PeerPrep

S/N	Functional Requirement	Priority
User Service		
F1.1	The system should allow users to register a new account with PeerPrep	H
F1.2	The system should allow users to login to PeerPrep	H
F1.3	The system should allow users to edit their profile	
F1.3.1	<ul style="list-style-type: none">• The system should allow users to change their password	M
F1.3.2	<ul style="list-style-type: none">• The system should allow users to change their preferences, including the type of programming language and difficulty of questions	M
F1.3.3	<ul style="list-style-type: none">• The system should allow users to change their display name	M
F1.4	The system should allow users to logout from PeerPrep	H
F1.5	The system should allow users to deregister their account	M
F1.6	The system should allow maintainers to view all accounts	
F1.6.1	<ul style="list-style-type: none">• The system should allow maintainers to view all users' display names and emails	M
F1.6.2	<ul style="list-style-type: none">• The system should allow maintainers to view the created and last activity date of all accounts	L
F1.7	The system should allow maintainers to edit accounts	

F1.7.1	<ul style="list-style-type: none"> The system should allow maintainers to edit the account's display name 	M
F1.7.2	<ul style="list-style-type: none"> The system should allow maintainers to promote/demote other accounts to maintainer/normal user 	M
F1.8	The system should allow maintainers to deregister other accounts	M
Matching Service		
F2.1	The system should allow two users to matched based the same question difficulty and programming languages	H
F2.2	The system should keep the matching to within a fixed time limit	
F2.2.1	<ul style="list-style-type: none"> The system should show a timer indicating the time left 	M
F2.2.2	<ul style="list-style-type: none"> If there is a valid match, the system should automatically navigate both users to the page for collaboration 	H
F2.2.3	<ul style="list-style-type: none"> The system should inform the users that no match is available if a match cannot be found within the time limit 	H
F2.3	The system should provide a means for the user to exit the queue	M
F2.4	The system should ensure that users cannot be matched to themselves	H
Question Service		
F3.1	The system should allow users to view the entire list of questions	H
F3.2	The system should allow users to search the question repository	
F3.2.1	<ul style="list-style-type: none"> The system should allow users to search by question title 	H
F3.2.2	<ul style="list-style-type: none"> The system should allow users to search by question tags 	H
F3.2.3	<ul style="list-style-type: none"> The system should allow users to search by question difficulty 	H
F3.3	The system should allow users to sort the question repository	
F3.3.1	<ul style="list-style-type: none"> The system should allow users to sort the questions by difficulty 	H
F3.3.2	<ul style="list-style-type: none"> The system should allow users to sort the questions by title 	H
F3.4	The system should allow maintainers to add new questions to the repository	
F3.4.1	<ul style="list-style-type: none"> The system should allow maintainers to add question's title 	H
F3.4.2	<ul style="list-style-type: none"> The system should allow maintainers to add categories tagged to the question 	H
F3.4.3	<ul style="list-style-type: none"> The system should allow maintainers to add question's difficulty 	H
F3.4.4	<ul style="list-style-type: none"> The system should allow maintainers to add question's description, including text, hyperlinks, images 	H
F3.5	The system should allow maintainers to edit an existing question	
F3.5.1	<ul style="list-style-type: none"> The system should allow maintainers to edit question's title 	H

F3.5.2	<ul style="list-style-type: none"> The system should allow maintainers to edit categories tagged to the question 	H
F3.5.3	<ul style="list-style-type: none"> The system should allow maintainers to edit question's difficulty 	H
F3.6	<ul style="list-style-type: none"> The system should allow maintainers to edit question's description, including text, hyperlinks, images 	H
F3.7	The system should allow maintainers to remove an existing question	H
F3.8	The system should provide an extensive list of questions for users to practice	M
Collaboration Service		
F4.1	The system should provide a collaborative space for the matched users to work together in real-time	H
F4.2	The system should use a rich text editor that provides code formatting, syntax highlighting for multiple languages	M
F4.3	The system should allow the matched users to view where the other party is pointing to in real time	L
F4.4	The system should allow the matched users to communicate via a Chat feature	M
F4.5	The system should allow the users to leave the collaboration session	H
F4.6	The system should allow the users to rejoin the collaboration session if the room is still active and if they have not officially left the room	M
F4.7	The system should allow the users to save the code they have typed	M
History Service		
F5.1	The system should allow the users to view their submitted attempts	
F5.1.1	<ul style="list-style-type: none"> The system should allow the users to view the code of all their previous submitted attempts 	M
F5.1.2	<ul style="list-style-type: none"> The system should allow the users to view the submissions' detail, including the language used and the submission's datetime 	M
F5.1.3	<ul style="list-style-type: none"> The system should allow users to view the execution output of their submitted attempts, including the execution duration 	M
F5.1.4	<ul style="list-style-type: none"> The system should allow the user to easily reference the question 	M
F5.2	The system should allow the users to view the statistics of their past attempts	

F5.2.1	<ul style="list-style-type: none"> The system should allow the users to view the number of Easy, Medium, or Hard questions they have attempted 	M
F5.2.2	<ul style="list-style-type: none"> The system should allow the users to view the number of Easy, Medium, or Hard questions they have yet to attempt 	M
F5.2.3	<ul style="list-style-type: none"> The system should allow the users to view the frequency of their submissions 	M
Execution Service		
F6.1	The system should allow the users to execute their code	H
F6.2	The system should display the executed code's output if the code is valid	H
F6.3	The system should allow the users to view how long their code took to run	H
F6.3	The system should inform the users if the code is invalid	H
F6.4	The system should inform the users and timeout the code, if the executed code takes too long to run	M

3.2. Non-Functional Requirements

Table 2: Non-Functional Requirements of our PeerPrep

S/N	Non-Functional Requirements	Priority
Performance		
NF1.1	The system should be able to handle at least 100 concurrent users	H
NF1.2	The website should be loaded in 3 seconds on any given web page	H
NF1.3	Matching should be limited to 30 seconds	M
NF1.4	Collaboration between the users should be real time, with < 0.5s lag	H
NF1.5	The execution system should execute the user's code within 5 seconds upon submission to ensure a responsive user experience	H
Security		
NF2.1	Users' passwords should be stored as encrypted hashes in the database, meeting RFC 7519 ¹ standards	H
NF2.2	Users should only be able to perform functions that correspond to their roles (i.e., users cannot perform maintainer's roles)	H
NF2.3	All users should be logged in and authenticated before using PeerPrep	H
NF2.4	When a user logs out, all information stored within the browser should be cleared	H

¹ <https://datatracker.ietf.org/doc/html/rfc7519>

Accessibility		
NF3.1	The website should be accessible from the web remotely	H
NF3.2	The website should be able to work with an appropriate bandwidth internet of at least 20 Mbps (4G)	H
Compatibility		
NF4.1	The website should be compatible with top 3 most used browsers (Chrome, Safari, Edge)	H
NF4.2	PeerPrep should be compatible with most modern computers	H
NF4.2.1	<ul style="list-style-type: none"> PeerPrep should not take up more than 4GB of RAM 	M
Reliability		
NF5.1	The website should achieve 5 9's (99.999% up rate)	M
NF5.2	The execution system should handle code execution errors gracefully, providing informative error messages in case of invalid code	M
NF5.3	The execution system should be resilient to unexpected errors during code execution	M
NF5.4	The execution system should not allow code execution to be stuck in a loop, self-exiting after 5 seconds	M
NF5.5	The system should log relevant information about errors and critical events	H
Usability		
NF6.1	The website should be usable on various laptop devices	M
NF6.2	The user interface should be intuitive, making it easy for users to navigate around	H
NF6.3	The website should achieve the user's desired outcome without necessitating users to input more than two levels	H
NF6.4	The website should bring login credential entered to signup page when user navigates to signup page from the login page	L
NF6.5	Questions difficulty should be color coded such that it is intuitive for the user to identify the difficulty easily	H

4. Developer Documentation

4.1. Technology Stack

Our development technology stack is as follows:

Table 3: Non-Functional Requirements of our PeerPrep

	Technology required	Utilized by (Microservices)
Frontend (Web UI)	React.js, Bootstrap, Material UI	N.A.
Backend (All Microservices)	Express.js, Node.js	All
Database	Mongoose (MongoDB Atlas)	User, Question, History
Cache	Redis	Collaboration
Message Broker	RabbitMQ	Match
Pub-Sub Messaging	Socket.IO	Collaboration
Authentication and Authorization	OAuth, JWT	Auth
Encryption	Bcrypt	User
API Gateway	Nginx	N.A.
Containerization Tool	Docker, Docker-Compose	All
Deployment	AWS EC2, AWS Lambda, AWS CloudWatch, AWS Elastic IP	All
CI/CD	GitHub Actions	N.A.
Project Management Tools	GitHub Issues	N.A.
Testing	Postman	All

4.2. System Architecture

4.2.1. Local Deployment Architecture (without Docker)

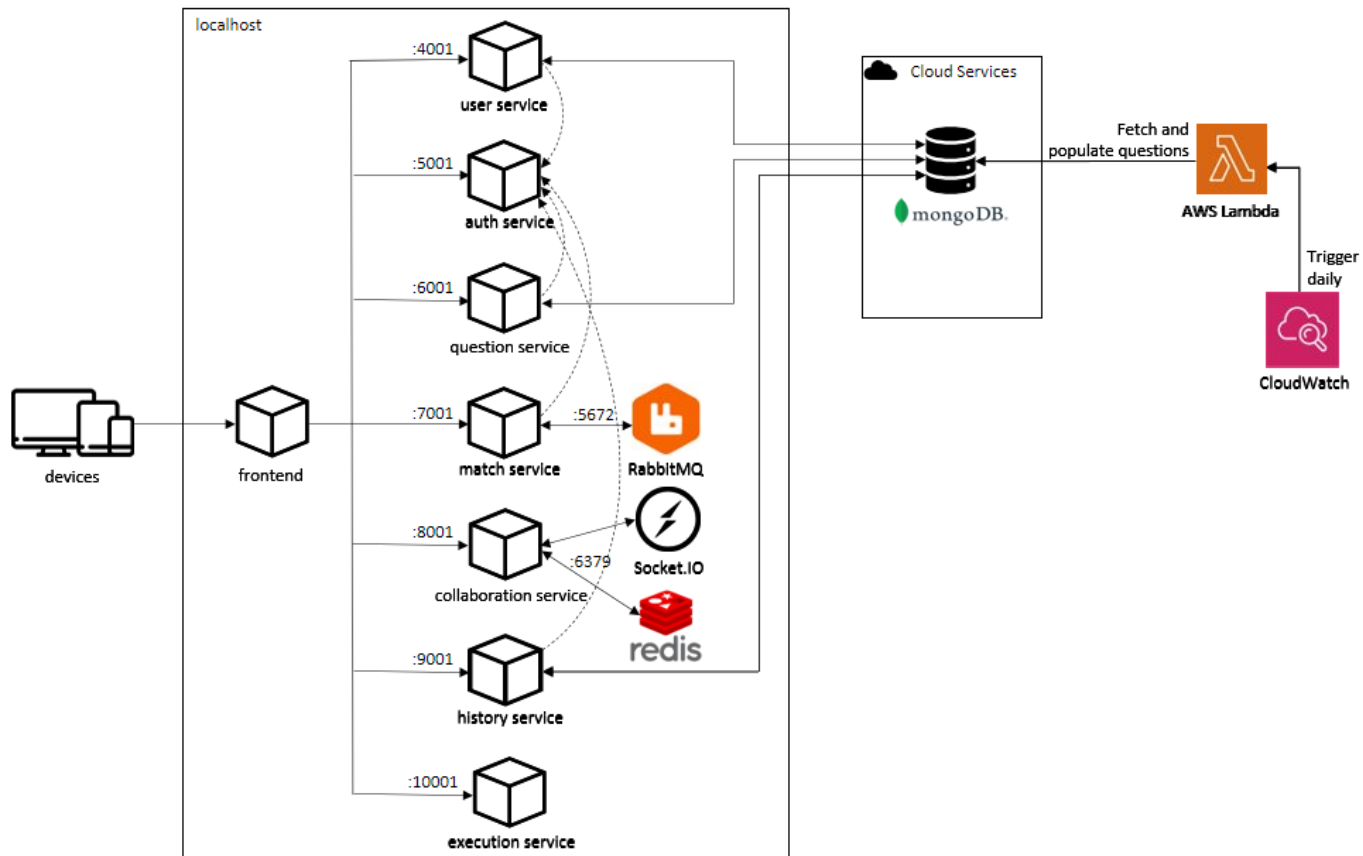


Figure 1: Local Deployment Architecture (without Docker)

4.2.2. Local Deployment Architecture (with Docker)

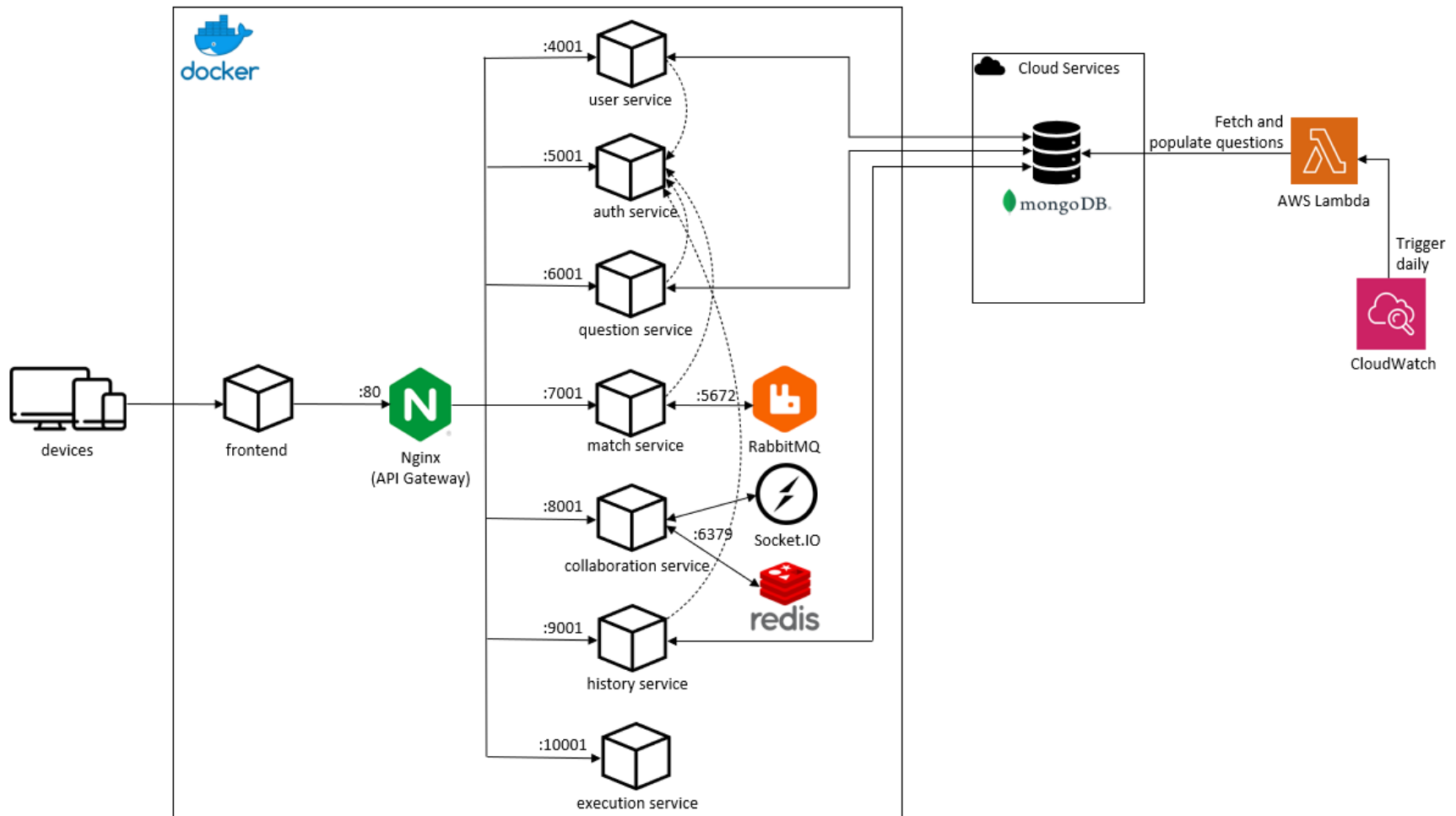


Figure 2: Local Deployment Architecture (with Docker)

4.2.3. Cloud Deployment Architecture

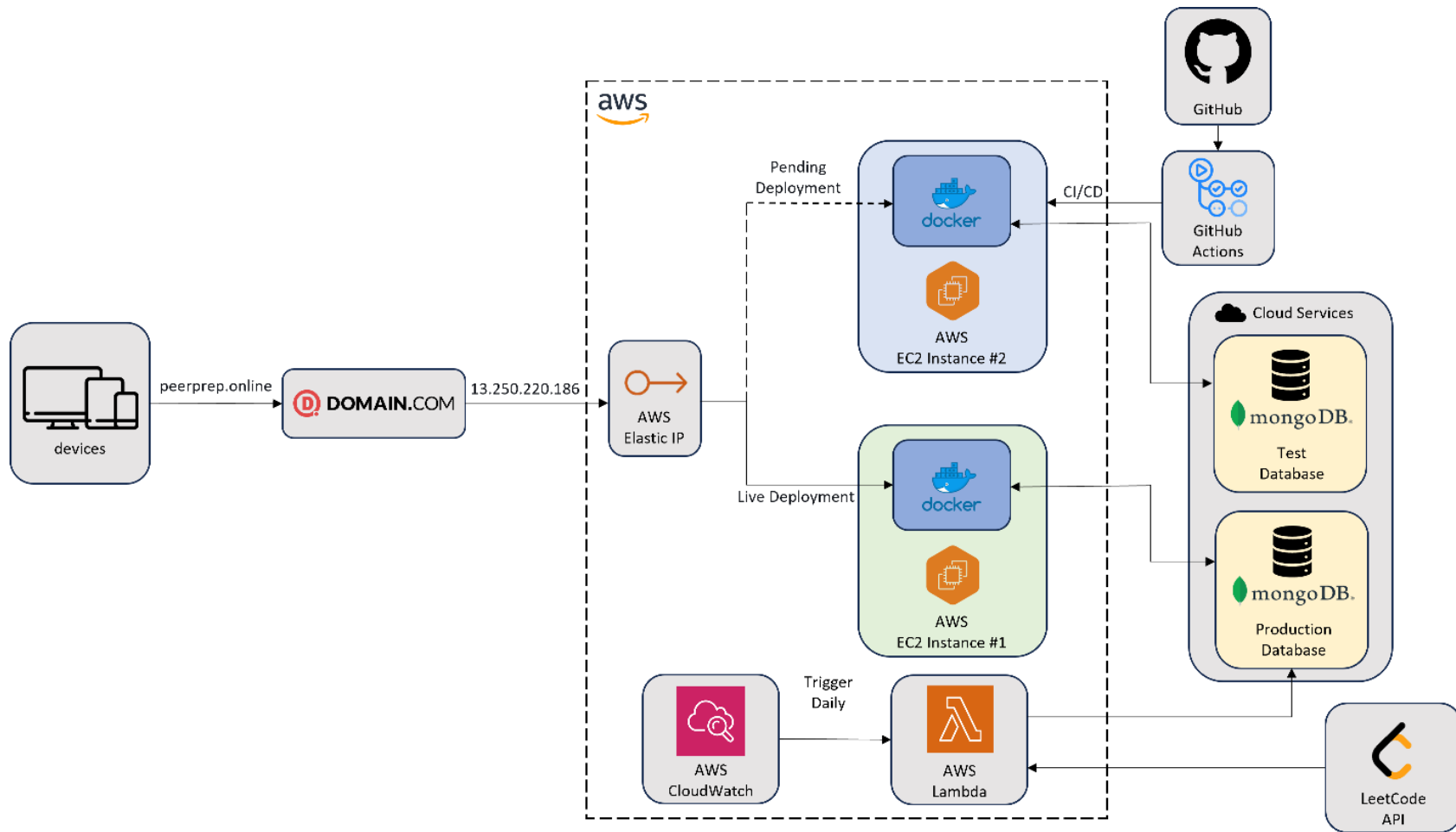


Figure 3: Cloud Deployment Architecture

4.3. Development Process

In our software development project, we seamlessly integrate Agile methodologies, Version Control Systems (VCS), and a weekly stand-up, fostering a dynamic and collaborative workflow. Adhering to Agile principles, we operate in two-week iterative sprints, promoting adaptability and collaboration. Git, our chosen VCS, streamlines code management, enabling parallel development and maintaining version history for traceability.

4.3.1. Sprint Iterations

Our commitment to efficiency is reinforced through a weekly Sunday night stand-up, enhancing team communication. This comprehensive approach empowers our team to iterate rapidly, uphold code integrity, and deliver high-quality software, all within the context of a two-week sprint and regular stand-up meetings, aligning closely with stakeholder expectations.

4.3.2. Collaborative Tools

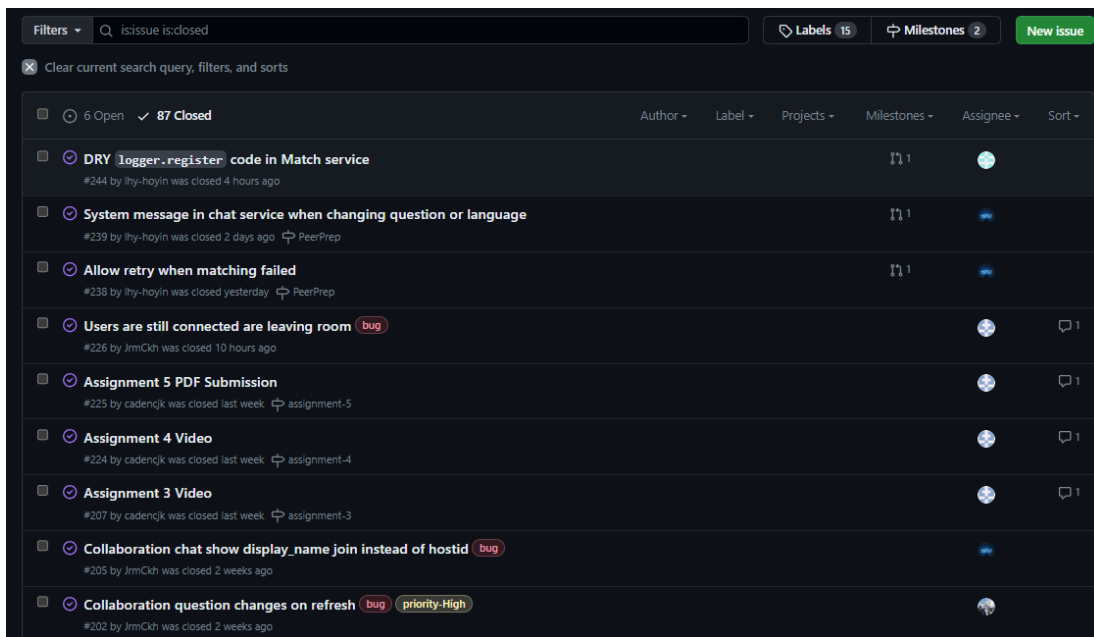


Figure 4: Our team's GitHub issue tracker

Our team employed various collaborative tools to assist in our development process. The GitHub issue tracker² played a pivotal role, facilitating the assignment of various features and bugs to specific team members (see Figure 4). This allowed us to assess the workload of individual team members and make necessary adjustments. Additionally, we implemented issues labels to streamline the search for a particular topic, such as different priority levels. Setting up multiple milestones enabled us to monitor the progress and meet the project's deliverable deadline effectively.

4.3.3. Continuous Integration/Continuous Development (CI/CD)

```
1  name: CI Pipeline
2
3  on: push
4  jobs:
5    build:
6      timeout-minutes: 10
7      runs-on: ubuntu-latest
8      steps:
9        - name: Checkout repository
10         uses: actions/checkout@v3
11
12        - name: Prepare Environment
13         run: mv template.env .env
14
15        - name: Start Containers
16         run: docker compose up -d
17
18        - name: Setup Node.js
19         uses: actions/setup-node@v3
20         with:
21           node-version: '18.x'
22           cache: 'npm'
23
24        - name: Stop Containers
25         if: always()
26         run: docker compose down
```

Figure 5: Our team's CI Pipeline

² <https://github.com/CS3219-AY2324S1/ay2324s1-course-assessment-g32/issues>

The team integrated a CI/CD pipeline into our workflow using GitHub Actions. Our CI pipeline³ (see Figure 5) is designed to automate the process of checking out code, preparing the environment, initiating Docker containers, configuring Node.js, and concludes with a cleanup step of stopping and removing containers. This automation enhances code integration, ensures consistency, and rapidly identifies and resolves potential issues.

```
1  name: Docker Deploy
2
3  on:
4    workflow_run:
5      workflows: ["CI Pipeline"]
6      branches: [dev]
7      types:
8        - completed
9
10 jobs:
11   deploy-packages:
12     if: ${github.event.workflow_run.conclusion == 'success' }
13     runs-on: ubuntu-latest
14     steps:
15       - uses: actions/checkout@v3
16       - name: Stop and remove images
17         uses: appleboy/ssh-action@master
18         with:
19           host: ${secrets.EC2_HOST}
20           username: ${secrets.EC2_USERNAME}
21           key: ${secrets.EC2_KEY}
22           script: |
23             cd ay2324s1-course-assessment-g32/
24             sudo docker-compose down
25             sudo docker system prune -a -f
26             cd ..
27             rm -rf -r ay2324s1-course-assessment-g32/
28       - name: Pull repository
29         uses: appleboy/ssh-action@master
30         with:
31           host: ${secrets.EC2_HOST}
32           username: ${secrets.EC2_USERNAME}
33           key: ${secrets.EC2_KEY}
34           script: |
35             git clone https://github.com/CS3219-AY2324S1/ay2324s1-course-assessment-g32.git -b dev || true
36       - name: Build and deploy
37         uses: appleboy/ssh-action@master
38         with:
39           host: ${secrets.EC2_HOST}
40           username: ${secrets.EC2_USERNAME}
41           key: ${secrets.EC2_KEY}
42           script: |
43             cd ay2324s1-course-assessment-g32/
44             cp template.env .env
45             sed -i 's|REACT_APP_HOST=http://localhost|REACT_APP_HOST=${secrets.EC2_IP}|g' .env
46             sudo docker-compose up -d || true
```

Figure 6: Our team's CD Pipeline

³ <https://github.com/CS3219-AY2324S1/ay2324s1-course-assessment-g32/blob/dev/.github/workflows/ci.yml>

Our CD pipeline⁴ (see Figure 6) is designed to deploy the most recent updates to an AWS environment following the successful execution of the CI pipeline. Together, these pipelines form an integrated workflow, establishing a continuous integration and deployment process. Leveraging Docker containers, this orchestrated workflow ensures a consistent and automated deployment on AWS, contributing to a seamless and dependable Blue-Green release cycle (explained further in Section 7: Deployment Process).

⁴ <https://github.com/CS3219-AY2324S1/ay2324s1-course-assessment-g32/blob/dev/.github/workflows/cd.yml>

5. General Design Pattern and Decisions

In this section, we'll offer an overview of our system's design pattern, emphasizing its impact on scalability, maintainability, and other relevant considerations. This aims to provide a clear insight into the strategic decisions guiding our software architecture.

5.1. Monolith vs Microservices

In the comparison between a monolithic and microservices architecture, a monolith is essentially a single application with a large, consolidated codebase. In contrast, a microservices architecture consists of multiple smaller, individual services. Monoliths offer advantages like easier deployment and development, as there's only one codebase to maintain. However, they suffer from disadvantages, primarily tight coupling between components, which makes them less adaptable to changes.

Microservices, on the other hand, help address these issues by breaking the application logic into smaller, loosely coupled microservices. This decreases the degree of coupling, increasing flexibility for changes, and benefiting development and deployment. Each microservice can implement its own tests, tech stack, and deployment, reducing the effort required for maintainability, bug fixes, and extensibility. Microservices also provide the added flexibility to scale efficiently.

In terms of system architecture, the team opted for a microservices pattern to modularize various components of the web application. The alternative, a monolithic pattern, would offer easier deployment and streamlined debugging and testing but would be challenging to scale and involve more complexity for iterative development in a team.

The modular nature of microservices makes it easier to split work, allows asynchronous development of features from different components, and simplifies scalability. It aligns with the open-close principle, enabling extensibility while being closed to modification. Choosing the microservices pattern ensures a clear separation of concerns as each microservice operates independently. In contrast, this separation of concerns is not as explicit in a monolithic pattern due to the nature of a monolithic application being a single unit.

5.2. N-tier Architecture and Controller-Service-Repository Pattern Within Each Microservice (Vertically-Sliced)

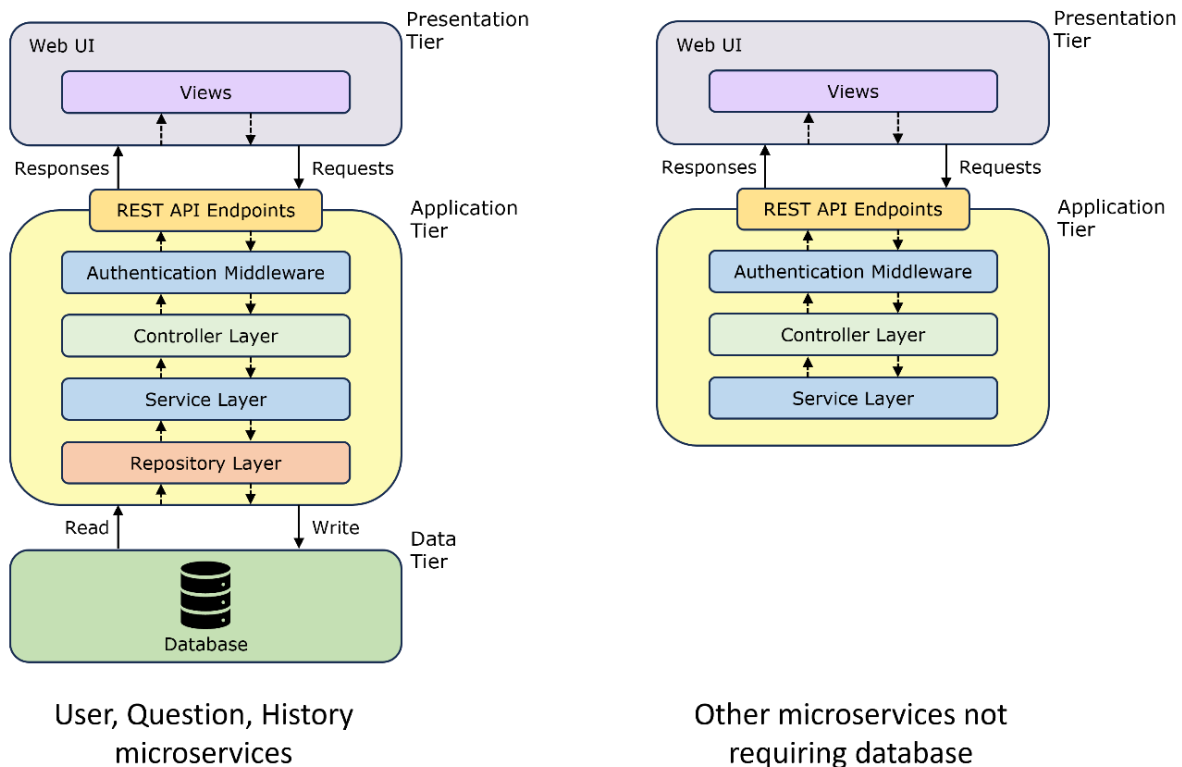


Figure 7: N-tier architecture of each microservice after vertically slicing

The architecture of each microservice is structured as a Single Page Application (SPA) following a N-tier model. The first tier, the presentation layer, undertakes tasks such as rendering web pages, managing local app state, and executing client-side logic. The second tier, our application logic layer, embraces the Controller-Service-Repository design pattern. In this layer, REST API endpoints facilitate client interaction, and requests are sequentially processed through the Controller, Service, and Repository layers. The third tier, the data layer, assumes responsibility for providing and managing persistence. This facilitates a clear separation of concerns in terms of user interface, application processing, and data management.

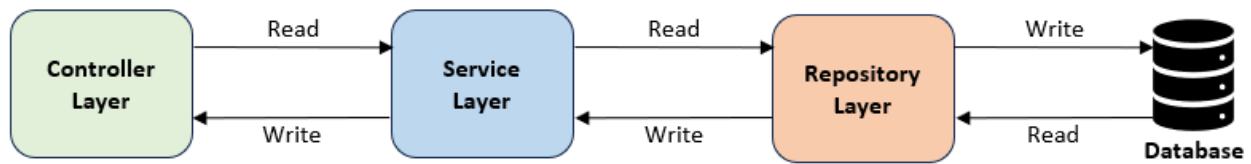


Figure 8: Controller-Service-Repository Pattern

We further enhance our design's modularity, scalability, and overall integrity by implementing the Controller-Service-Repository (CSR) pattern for each service in our architecture.

Within the CSR pattern, each component functions independently, with distinct responsibilities. The controller layer manages request processing and calls upon the relevant services. The service layer is responsible for performing the business logic, while the repositories layer oversees the storage and retrieval of data from a persistent store, such as a database.

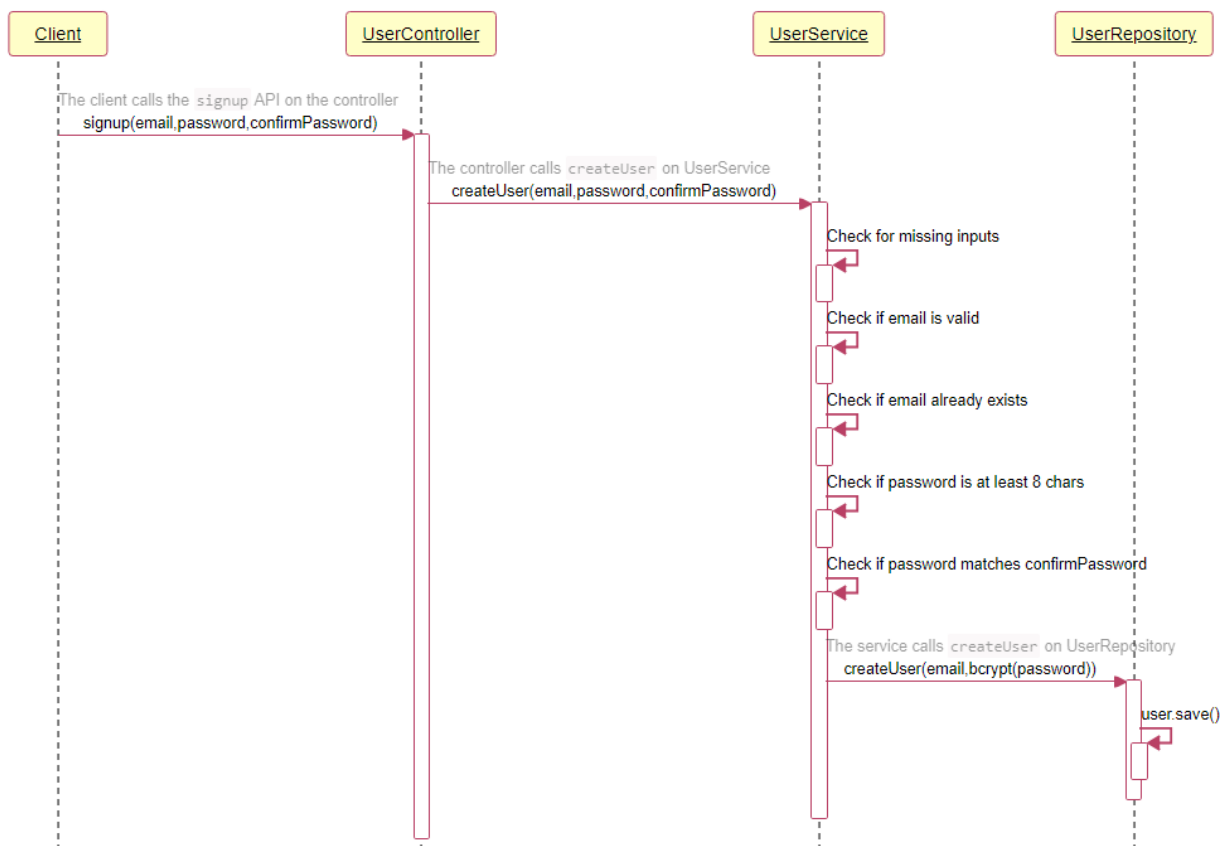


Figure 9: Sequence Diagram of User Microservice Controller-Service-Repository Pattern for User Signup

As depicted in Figure 9, when a user initiates a signup process, the UserController takes charge of processing the API request and delegates it to the UserService. The UserService manages all aspects of the business logic, including tasks such as validating the email. On the other hand, the UserRepository is responsible for handling the storage operations associated with the signup process.

This architecture promotes modularity and maintainability by allowing the addition of new controllers, services, or repositories without impacting existing components. This adheres to the Open Closed Principle, making our program more extensible, which emphasizes the ability to extend functionality without necessitating extensive modifications.

An example showcasing the benefits of CSR for our team emerged when MySQL was initially used to store user's login information in our Assignments' system. Subsequently, for PeerPrep, the benefits of CSR became evident when we migrated to MongoDB, requiring only a single adjustment in the repository layer to modify the database implementation. This eliminated the need for modifications in both the controller and service layers.

5.3. API Gateway with Nginx

The use of the Facade pattern is evident when an API Gateway is employed to present a unified interface, which efficiently routes incoming API requests to the relevant microservices.

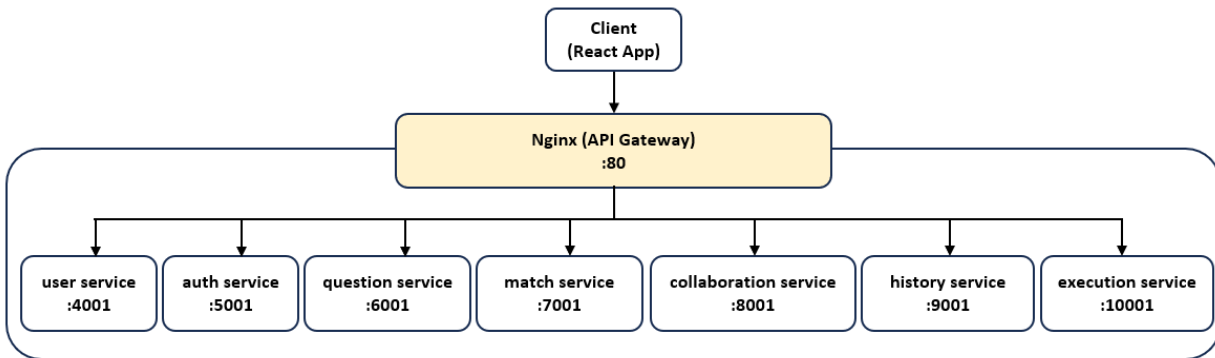


Figure 10: How Facade Pattern is used in API Gateway

When a web client initiates a request, it leverages the API Gateway acting as a facade to manage the incoming API request and direct it to the relevant service. This design pattern effectively conceals our services from the client's viewpoint. The client's sole requirement is to be familiar with a single address, that of Nginx, instead of needing to manage the addresses of multiple individual services. This abstraction helps improve the organization, security, and manageability of the overall system architecture.

5.4. Use of ReactJS and Node.js with Express.js

Our decision to employ ReactJS as the primary frontend framework for PeerPrep is based on a combination of factors. This choice is driven by its popularity in the developer community, coupled with our team's familiarity with it. Moreover, it has many open-source modules for us to leverage on to enhance the functionality and user experience of PeerPrep.

In addition to ReactJS, we are also opting for Node.js and Express.js as our backend technologies. Node.js is a runtime environment for us to run JavaScript on the server side, allowing us to use Express.js to build the backend API.

5.5. Use of MongoDB

NoSQL databases, such as MongoDB, offer support for large volumes of data and high performance. These attributes can offer significant advantages to our PeerPrep app, especially considering the nature of the data it stores.

One notable case where NoSQL, and specifically MongoDB, shines is in the storage of our "questions" collection, which encompasses questions obtained from the LeetCode API. This collection may grow significantly over time, if LeetCode introduces a large amount of new questions to its repository. We could face potential challenges with a traditional SQL database in terms of scalability.

However, with MongoDB, these concerns are alleviated. It is designed to scale seamlessly, making it a robust choice for our PeerPrep. If LeetCode continues to add more questions, we can confidently rely on MongoDB to scale and adapt, thus minimizing any impact on our application's performance and data management.

6. Implementation of Microservices

6.1. User Microservice

The User microservice oversees key user-related functions within PeerPrep, encompassing tasks like sign-up, login, and the management of account details and preferences. Authentication can only be done through this microservice. Upon the creation of a PeerPrep account, a unique identifier is automatically generated through an incremental process and then assigned to the account. Notably, for enhanced security, the user's password undergoes both hashing and salting procedures before being stored in the system. This ensures that sensitive user credentials are securely handled, aligning with best practices for safeguarding user information within the PeerPrep system.

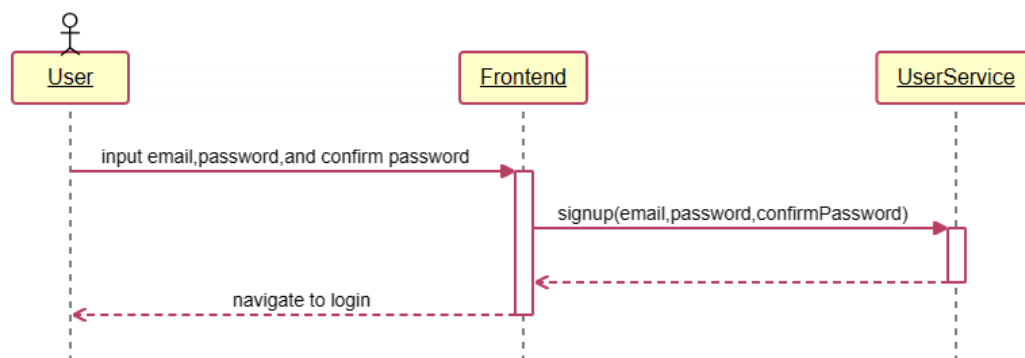


Figure 11: Sequence Diagram for User Signup

6.1.1. User Authentication and Authorization

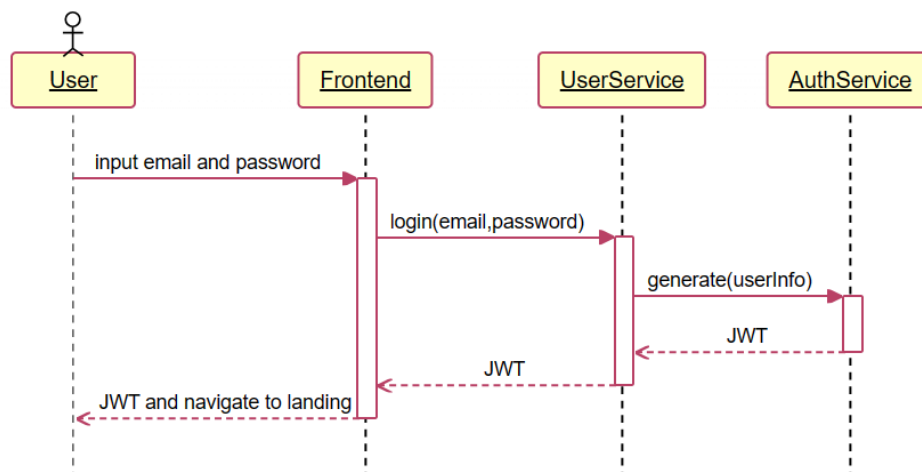


Figure 12: Sequence Diagram of how OAuth works when user attempts login

Whenever a user logs in, our authentication process is facilitated by this User microservice, and the authorization process is facilitated by our OAuth framework within the Auth microservice. The accompanying sequence diagram outlines the systematic process of generating access tokens for each user. Importantly, this mechanism ensures that only authenticated users possessing valid access tokens gain access to our microservices, reinforcing the security and controlled access within our system.

Our application supports two user roles: Maintainer and Normal User, each with distinct permissions and access levels within our application. The role of every user is stored in this microservice. In general, Maintainers have broader capabilities compared to Normal Users, who have more limited permissions and rights. Table 4 outlines the specific actions that Maintainers are allowed to access, whereas Normal Users do not have access to these resources. Access rights are determined based on the information contained in the access token, specifically the `isMaintainer` boolean.

Table 4: A list of actions and the microservices responsible for the actions

Action	Microservice
Add Question	Question
Delete Question	
Edit existing Question	
Delete (Deregister) other Users	User
List all the Users	
Promote/Demote other Users	

6.1.2. User Schema

Our User database houses user-related data, including the `"isMaintainer"` field, which serves as an indicator of whether a user holds a regular user or maintainer role. This distinction enables our application to allocate appropriate access rights to various resources. Additionally, the `"language"` and `"complexity"` fields store user preferences, allowing the application to remember the user's language and complex options when matching with other users. As a result, users do not have to manually set their language and complexity preferences, as these options are conveniently preselected by default.

UserModel	
_id	Mongoose.Schema.Types.ObjectId
displayName	String
email	String
password (hashed)	String
isMaintainer	String
language	String
complexity	String
createdAt	Date
updatedAt	Date

Figure 13: MongoDB (Mongoose) Models Schema Diagrams for users Database

6.1.3. Design Considerations

6.1.3.1. Use of NoSQL Database Instead of SQL Database

Despite having a defined schema for our user database, we made the choice to opt for MongoDB, a NoSQL database, rather than a SQL database. This decision was driven by the desire to streamline maintenance and maintain consistency across various collections, particularly the "users" and "histories" collections. In the event of a user deletion from the "users" collection, we aim for a corresponding deletion of their associated "histories" documents. Achieving this level of consistency would be arduous if the different tables were housed in different types of databases.

Furthermore, if we were to store the "users" data in a SQL database, it would result in a single table, limiting our ability to leverage the robust features inherent in SQL databases, such as the ability to perform join queries, maintain ACID properties, and ensure data integrity. Utilizing MongoDB for the "users" collection also offers additional benefits, including enhanced scalability if the number of users increases significantly in the future.

6.2. Question Microservice

The Question microservice plays a pivotal role in storing and delivering sample technical interview questions. It offers functionality, including the generation of a random interview question based on the user-selected difficulty level. Additionally, this microservice empowers system administrators by providing tools to manage the data within the questions database, ensuring the seamless maintenance of the list of recommended questions.

6.2.1. Question Schema

QuestionModel	
_id	Mongoose.Schema.Types.ObjectId
title	String
complexity	String
description	String
tags	[String]

LeetCodeQuestionModel	
_id	Mongoose.Schema.Types.ObjectId
leetcodeId	Number
title	String
complexity	String
description	String
tags	[String]

Figure 14: MongoDB (Mongoose) Models Schema Diagrams for Manual and LeetCode Questions

Our database accommodates two types of questions: those sourced from LeetCode and those manually created. The key distinguishing factor is the inclusion of the LeetCode ID field. Regardless of the source, both types of questions can be edited or deleted manually by maintainers, providing flexibility and control over the question set.

6.2.2. Question Generation System

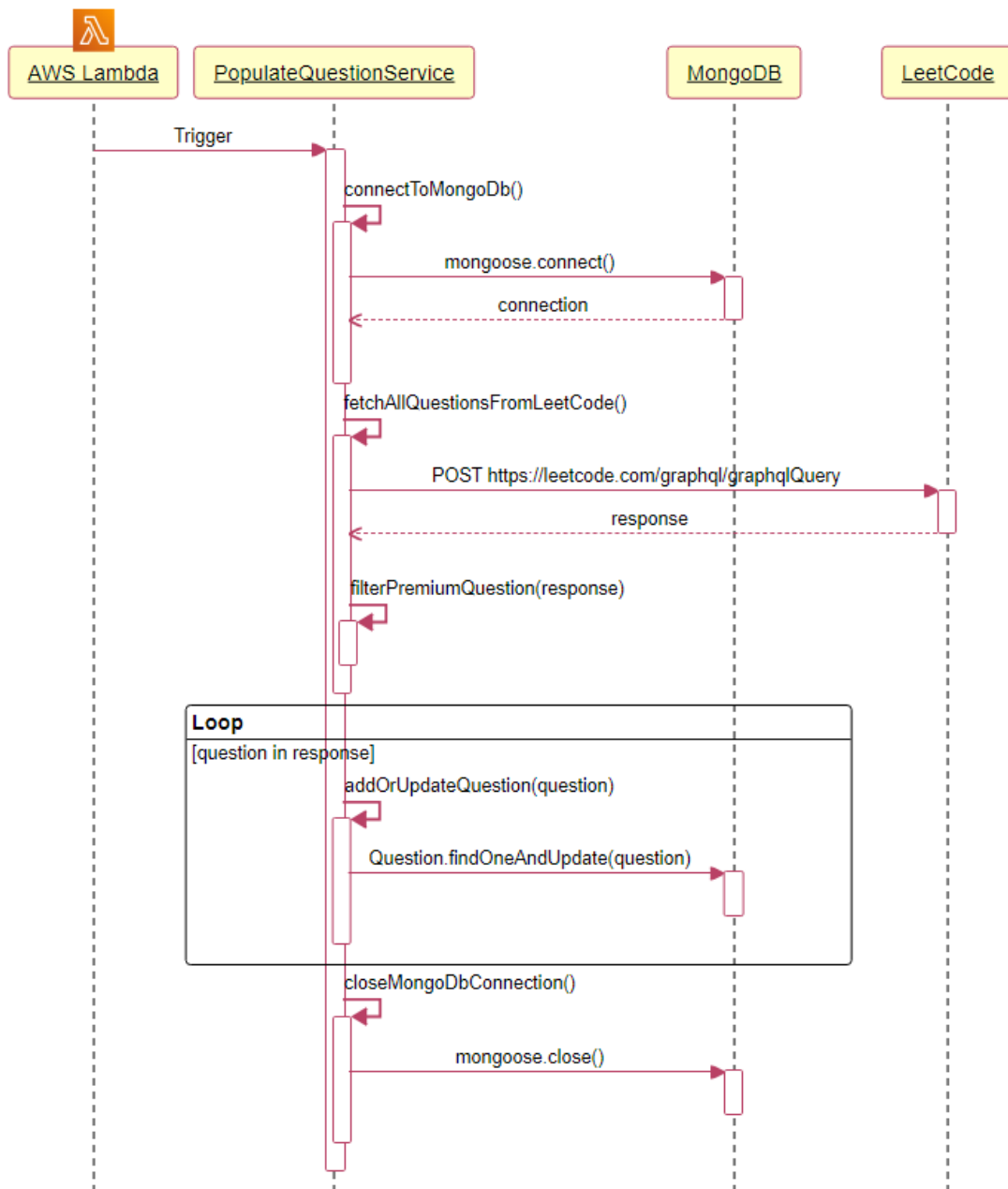


Figure 15: Sequence Diagram of AWS Lambda Serverless Function Populating Questions

Our question generation system offers dual capability, allowing maintainers to manually create questions through the front end and leveraging an AWS Lambda serverless function for automated question generation. The serverless function queries data from LeetCode's GraphQL

API⁵, a publicly accessible resource, to populate our database. This serverless function⁶ is scheduled to trigger daily at 12am, checking the LeetCode API for question data. Before insertion into our database, the function verifies if the question already exists based on the LeetCode ID obtained from the API.

⁵ <https://leetcode.com/discuss/general-discussion/1297705/is-there-public-api-endpoints-available-for-leetcode>

⁶ <https://github.com/CS3219-AY2324S1/ay2324s1-assignment-6-g32>

6.3. Auth Microservice

The Auth microservice is designed as a centralized authorization server, streamlining the user authorization process. It serves as a single point of authorization for all our microservices and implemented with respect to the OAuth2.0 framework to standardize with the current industrial best practices. This microservice concerns itself only with authorization. Authorization is done before communication is established between different microservices that require privileged access only.

To illustrate, authorization is required before establishing communications between frontend and Question microservice. This ensures that only users of PeerPrep can request questions from the database in Question microservice.

6.3.1. Open Authorization 2.0 Framework

Open Authorization 2.0⁷ (OAuth 2.0) is a widely adopted industry-standard authorization framework designed to facilitate secure and controlled access to web resources. It addresses the challenge of delegated access in web and mobile applications by allowing a user to grant limited access to their resources without revealing confidential credentials like passwords. Key components include the Resource Owner (PeerPrep's users), Client Application (PeerPrep), Authorization Server (Auth Microservice), and other microservices of PeerPrep. An abstract diagram is shown in Figure 16 below.

⁷ <https://oauth.net/2/>

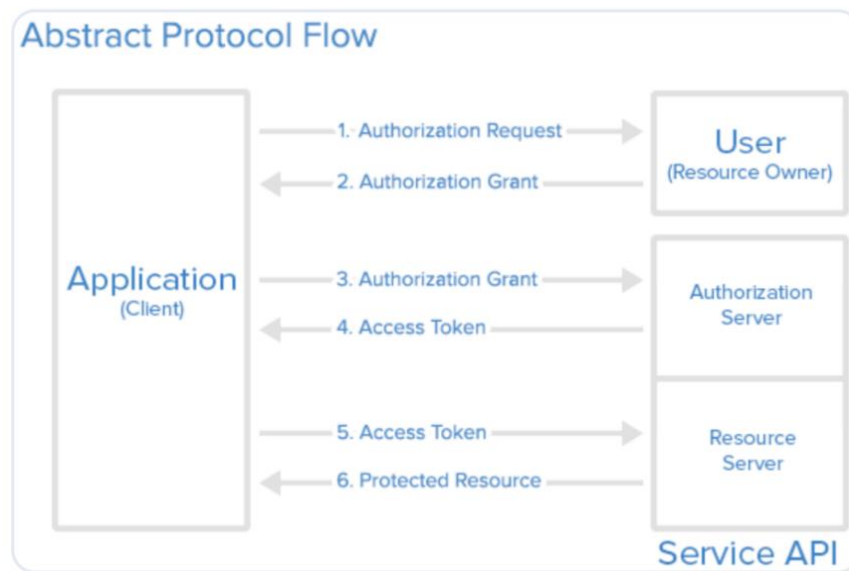


Figure 16: OAuth 2.0 Protocol Flow⁸

Access Tokens play a pivotal role in OAuth 2.0, serving as credentials that represent the user's authorization for the client to access specific resources. These tokens are typically short-lived and enable the client to interact with protected resources on behalf of the user.

OAuth 2.0's flexibility and extensibility make it a preferred choice for secure authorization in a variety of scenarios. It is widely implemented by major web services and APIs, ensuring a robust framework for controlling and managing access to user data. Implementing and configuring OAuth 2.0 securely is crucial to safeguard against unauthorized access and protect user information.

6.3.2. High-Level Overview

The implementation of Auth microservice uses JSON Web Token (JWT) which are stored locally in browser cookies on users' machines. A new JWT is generated on instance of logging-in. Referring to Figure 12, to get this access token, the user can go through the User microservice to sign up their account, which will redirect them to this Auth microservice to generate and return a valid JWT.

When accessing API endpoints that require user authentication or require users to possess maintainer privileges, the HTTP request directed to these specific API endpoints undergoes an

⁸ <https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2>

initial check at the Auth API endpoints. This check involves verifying whether the JWT provided in the HTTP Authorization request header was generated by our Auth microservice.

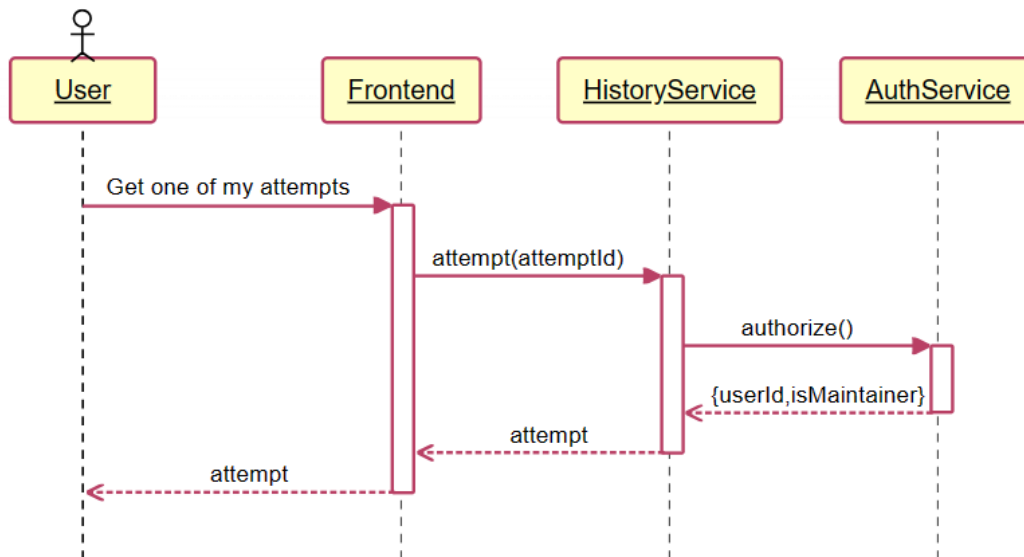


Figure 17: Sequence Diagram of how OAuth works to give authorization to actions that require users to be authenticated

For those endpoints that mandate users to be maintainers, the Auth microservice performs additional verification steps. It decodes the JWT to extract the value of the isMaintainer field, ensuring that the user indeed possesses a maintainer status.

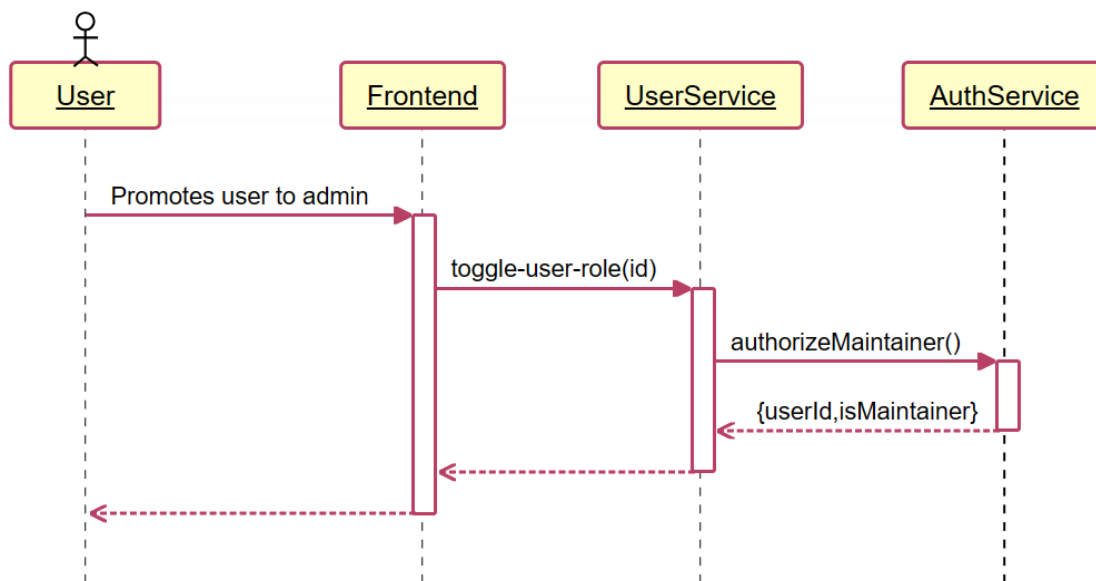


Figure 18: Sequence Diagram of how OAuth works to give authorization to maintainer actions

The Auth microservice functions as a middleware in other microservices. Additionally, the client application can call upon the Auth microservice to grant authorization for specific resources within the application.

6.3.3. JSON Web Token (JWT) Implementation

The JWTs produced by the Auth microservice follow a standardized format that comprises two main components: the header and the payload. The header is presented as a JSON object with specific attributes, denoted as follows:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Figure 19: JWT Token Header

In this representation, "alg" specifies the algorithm employed for the token's signature. As the token signature is just to ensure the authenticity of the message, we have chosen HS256 as our algorithm. As a symmetric signing method, it is secure and fast. "typ" signifies the type of token, which is "JWT" for JSON Web Token.

Concurrently, the payload is an additional JSON object that encapsulates user information. This payload incorporates the user's unique identifier, designated as "userId," a boolean indicator denoting whether the user holds maintainer privileges ("isMaintainer"), and a timestamp indicating when the token was issued ("iat"). The overall structure of the payload is delineated as:

```
{
  "userId": "<id of user>",
  "isMaintainer": <true/false boolean>,
  "iat": "<time at which token was issued>"
}
```

Figure 20: JWT Token Payload

Essentially, the header and payload form the content of the JWT, encapsulating essential information for secure authentication and data exchange to prove the identity of the user for authorization purposes.

6.3.4. Design Considerations

6.3.4.1. Centralized Auth

Our decision to establish a standalone Auth microservice stems from the need for an explicit separation of roles. By doing so, we ensure that all authorization processes consistently pass through the Auth microservice. Without this separation, it would likely be intertwined with the User microservice, given its user-related nature. However, this integration would result in multiple microservices that require user authentication to depend on the User microservice, potentially granting access to our users database, which poses a security risk due to the storage of user credentials.

This design choice also grants us the flexibility to scale independently in the future. If the Auth microservice were integrated within the User microservice, scaling would require expanding the entire User microservice, which may not be the most efficient or ideal approach.

6.3.4.2. Own OAuth Microservice vs Third-Party OAuth Services

Despite the convenience offered by third-party OAuth services, such as Auth0, the team chose to construct our own OAuth microservice due to several considerations. Building our dedicated OAuth microservice provided us with meticulous control over authentication and authorization processes, addressing security and privacy concerns. Furthermore, this decision provided flexibility in integration, minimized external dependencies, and allowed for the accommodation of specific business logic unique to PeerPrep. Constructing our OAuth microservice empowered us to create a tailored authentication system precisely aligned with the application's needs, underscoring our commitment to security, flexibility, and control.

6.4. Match Microservice

The Match microservice plays the pivotal role of pairing two users based on their chosen difficulty and language preferences. To facilitate this matching process, the microservice employs RabbitMQ as its queuing technology. Each match request is transmitted as a packet to the designated match queue. When two users are aligned in their chosen criteria, the queue facilitates their connection, resulting in a successful match.

6.4.1. Dynamic Queue Generation with a Common Queue

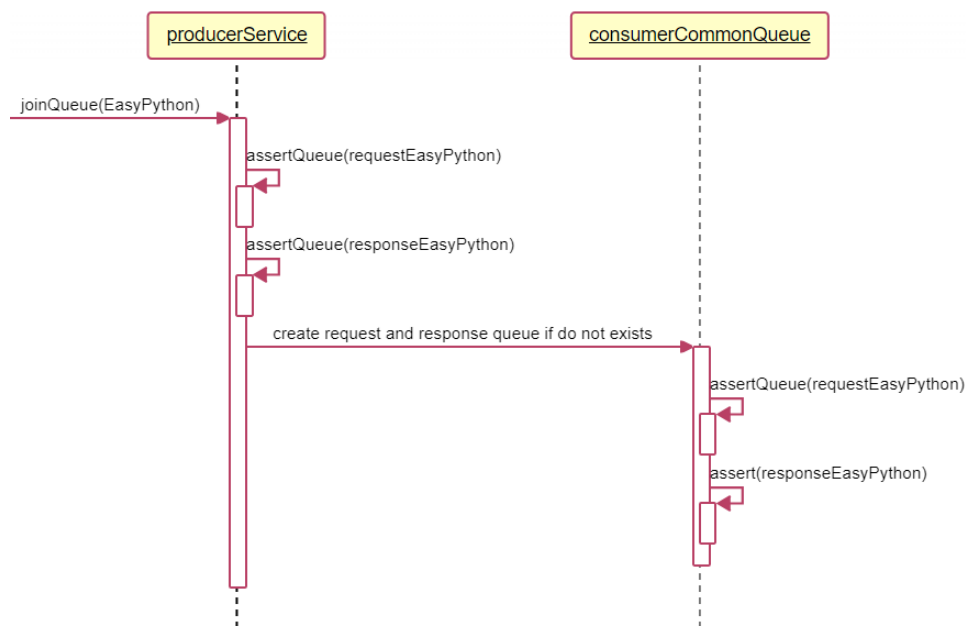


Figure 21: Sequence Diagram of Our Common Queue Establishing EasyPython Queues

In our Matching implementation, we have implemented a common queue for the dynamic creation of queues as required during runtime. Figure 21 provides a visual representation of a simplified sequence diagram, illustrating how the common queue facilitates the establishment of EasyPython queues. When a client initiates a "join queue" request, it dynamically creates a request and response queue on the producer side and sends it to the consumer's (server) common queue. Both the consumer and producer ensure the existence of these queues, creating them if not already present.

The dynamic creation of queues through a common queue enhances the efficiency of our system by eliminating the necessity for manual queue specifications on both the client and server sides.

This approach simplifies maintenance, requiring only the management of queue types on the frontend, effortlessly translating into queue creation. For instance, envision three sets of matching criteria, each with three items, resulting in a clear 27 permutations of queues. Importantly, as the number of criteria increases, the system's adaptability scales accordingly, necessitating more queues to accommodate diverse matching conditions without the burden of hardcoded configurations.

Moreover, once the queues are inactive for a predetermined period, they are automatically deleted to optimize resource utilization and maintain system efficiency. This adaptive process ensures that the system remains responsive to changing conditions, promoting efficient resource usage and overall system reliability.

6.4.2. Request-Reply Communication Pattern

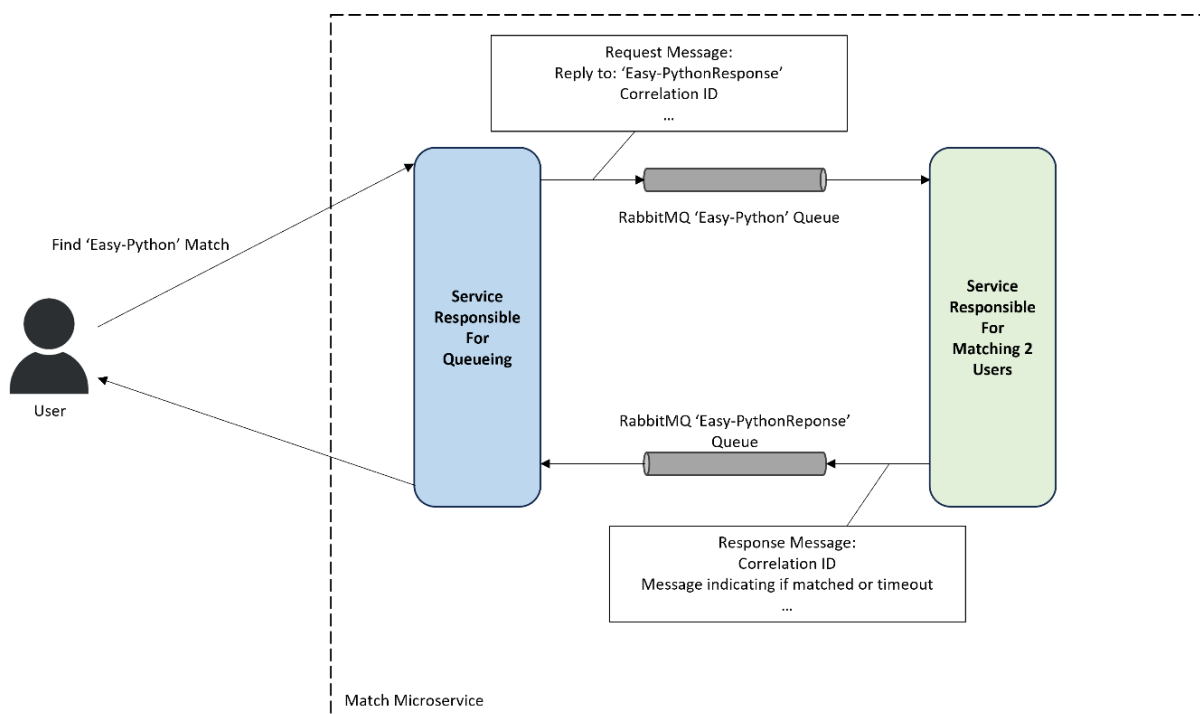


Figure 22: Match Microservice using RabbitMQ⁹

Our RabbitMQ implementation employs a synchronous Request-Reply communication pattern to facilitate the process of client matching. This pattern, akin to a Remote Procedure Call (RPC),

⁹ <https://www.rabbitmq.com/tutorials/tutorial-six-python.html>

involves clients initiating remote procedures or operations on the server and patiently waits for the results. RabbitMQ acts as the message broker, efficiently orchestrating the distribution of requests and responses between clients and the server. The integration of RabbitMQ into the request-reply paradigm aligns with RPC principles, ensuring smooth and synchronous communication across distributed components.

When a client initiates an RPC request, it sends the request to the common queue, dynamically establishing both the request and callback queues if they do not already exist. The RPC request is dispatched with properties like ``reply_to`` pointing to the callback queue and ``correlation_id`` holding a unique identifier for each request. This request is directed to the request queue. Meanwhile, the RPC worker, acting as a server, patiently awaits incoming requests on the designated queue. Upon receiving a request, it diligently executes the assigned task and responds by sending a message containing the result back to the client, utilizing the queue specified in the ``reply_to`` field.

The client actively monitors its callback queue for incoming messages, checking the ``correlation_id`` property upon detection. When a match is found with the unique identifier of the original request, the client seamlessly returns the corresponding response to the application. This well-coordinated exchange ensures an efficient and synchronous RPC interaction between the client and the server.

6.4.3. Interactions between Matching and Collaboration Service

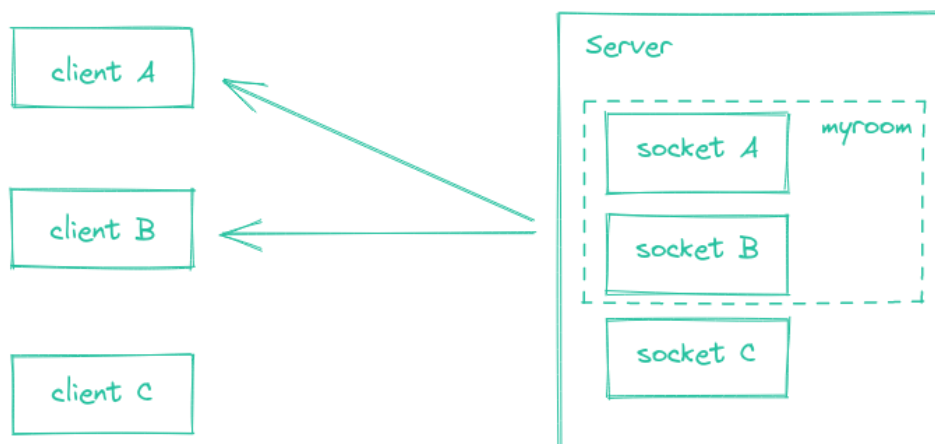


Figure 23: How rooms work in Socket.IO¹⁰

¹⁰ <https://socket.io/docs/v3/rooms/>

Upon the discovery of a match between two users, a distinct room ID is generated. Subsequently, both clients join this room using the `join` method in Socket.IO. The implicit creation of rooms is a feature of Socket.IO, ensuring that the specified room is automatically generated if it does not exist. Within these rooms, clients utilize sockets to establish connections, fostering real-time communication and collaboration in the context of the identified room. This mechanism facilitates a structured and isolated environment for coordinated interaction between the matched clients.

6.5. Collaboration Microservice

The Collaboration microservice offers a dedicated platform for real-time collaboration among matched users, facilitating features like Chat, collaborative code editing, and dynamic question and language changes. It utilizes Socket.IO as its primary tool to ensure seamless real-time engagement. Meanwhile, Redis operates as the Cache service, optimizing data retrieval and storage to enhance overall performance.

6.5.1. Publisher-Subscriber Pattern for Real-time Collaboration

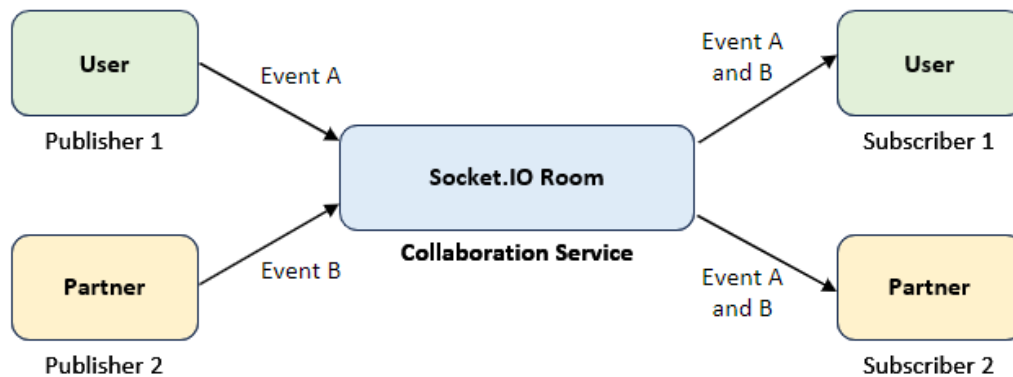


Figure 24: Pub-Sub in Collaboration Service

Real-time collaboration among clients is facilitated through the Publisher-Subscriber pattern in Socket.IO's Room, which adopts an event-driven structure. Within our collaboration service, clients subscribe to Socket.IO room to receive updates from publishers. While Socket.IO naturally broadcasts updates to all members in a room, encompassing the sender, our implementation incorporates explicit checks to selectively filter out updates made by the client itself when necessary. This pattern is selected for its low-latency communication capabilities, contributing to a loosely coupled architecture. Publishers and subscribers can operate independently without requiring knowledge of each other, fostering a scalable and flexible collaboration environment.

While considering message patterns, the team also considered the Observer pattern, but we realized that publishers do not necessarily need awareness of the subject or subscriber. Hence, the Publisher-Subscriber pattern emerges as the more fitting choice.

6.5.2. Chosen Programming Languages for User

Most used programming languages among developers worldwide as of 2023

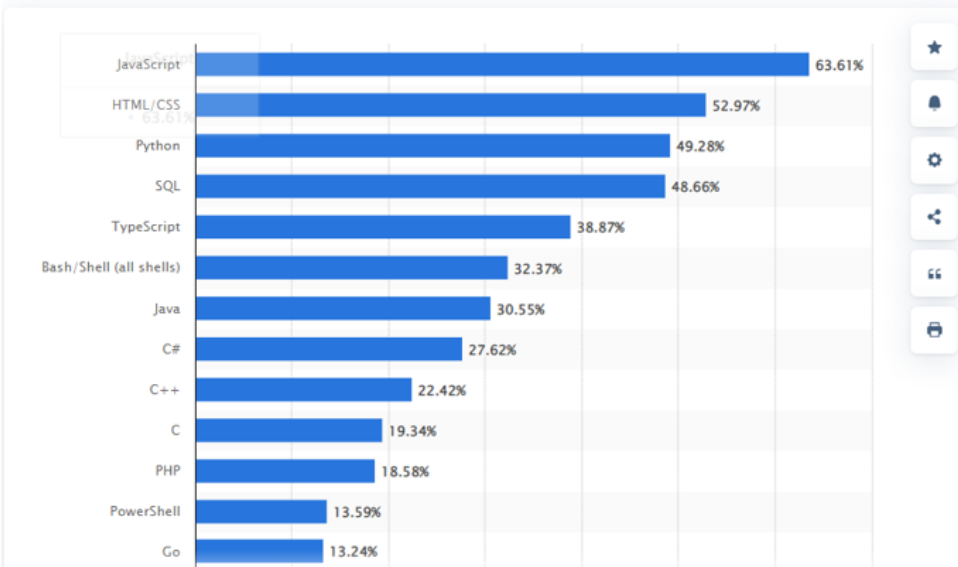


Figure 25: Most Used Programming Languages as of 2023 from Statista¹¹

The team has chosen to integrate three programming languages — Python, Java, and JavaScript — into our PeerPrep program. This choice is in line with Figure 25, highlighting these languages as among the most widely used by developers worldwide as of 2023. Notably, they are the top 3 languages in the list that are used to address algorithmic-related challenges. This deliberate selection underscores our commitment to utilizing languages that are not only popular but also well-suited for addressing the complexities inherent in algorithmic problem-solving. Future iterations of PeerPrep might support more programming languages.

6.5.3. Approach Used to Enable Rejoining of Collaboration Room

One of the functionalities we desire to achieve is to enable users to rejoin a collaborative room even after navigating away¹² from the collaboration page. The challenge arises when another user is actively typing in the room, necessitating the re-joiner to synchronize with the latest room state. This state encompasses the typed code, selected question, execution output, chat messages, and the chosen programming language for the code editor.

¹¹ <https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/>

¹² Eg. clicking back or going to another webpage; i.e does not explicitly leave room

These were 2 approaches that the team could have taken:

6.5.3.1. Peer-Sharing Synchronization

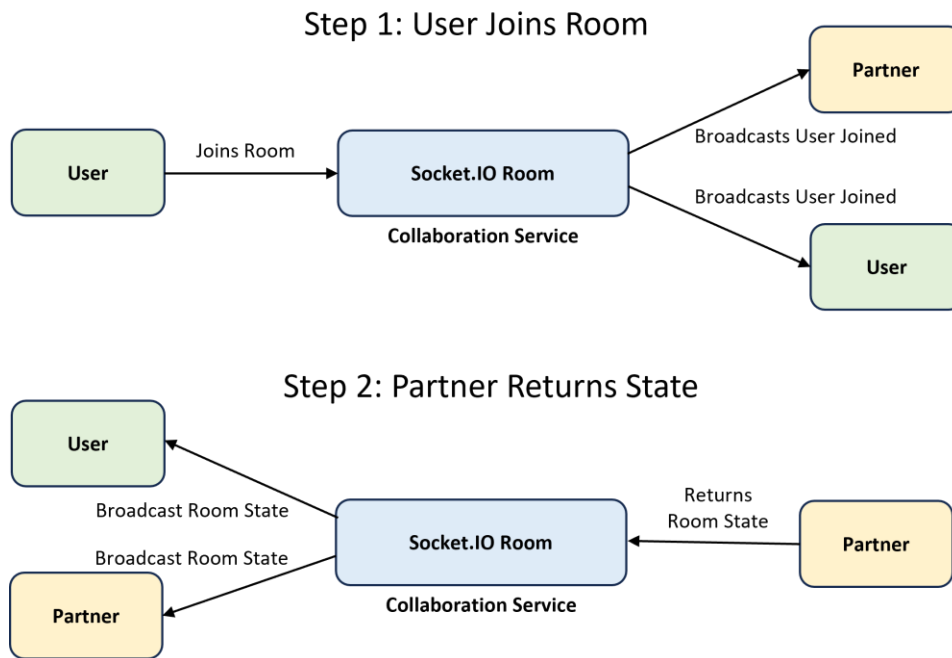


Figure 26: Peer-Sharing Synchronization

In approach 1, Peer-Sharing Synchronization, when User #1 rejoins the room, User #1 will send a special socket message requesting User #2 to broadcast the last room state.

Table 5: Pros and Cons of Peer-Sharing Synchronization

Pros	Cons
<ul style="list-style-type: none"> • Easy to implement. • Require no extra storage. 	<ul style="list-style-type: none"> • Requires more work from the client to broadcast the state. • Relies on client's storage (i.e. client may have cleared browser cache). • Difficult to ascertain who has the last updated state. • Unable to retrieve information if both clients leave the room.

6.5.3.2. Redis Cache Storage

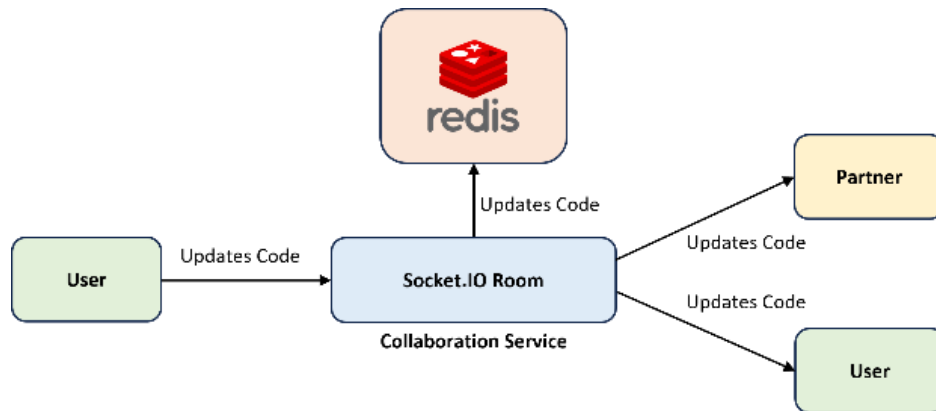


Figure 27: Redis Cache Storage, update room states

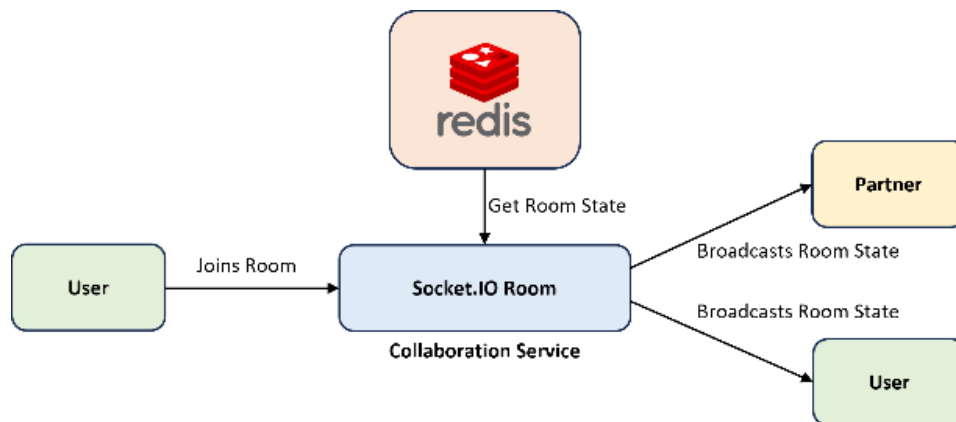


Figure 28: Redis Cache Storage, get room states

In approach 2, Redis Cache Storage, when a user updates the state of the room, the Collaboration Service will push the updated attribute to Redis Cache. Upon User #1 rejoining the room, the user will retrieve the latest cached state from Redis, removing the necessity for the other user to broadcast the last room state.

Table 6: Pros and Cons of Redis Cache Storage

Pros	Cons
<ul style="list-style-type: none"> Requires less work from the client to broadcast the state. Higher reliability with specialized storage system. 	<ul style="list-style-type: none"> Difficult to implement. Require extra storage backend.

<ul style="list-style-type: none"> • Always provide the last updated state. • Able to handle the case when both users leave the room temporarily. 	<ul style="list-style-type: none"> • Require a way to clear the room storage if there is no activity for a prolonged period.
---	---

Ultimately, our team has decided to implement the storage approach, primarily because Peer-Sharing synchronization faces a critical limitation in handling the case when both users temporarily leave. This limitation could potentially impact website's usability, as users might encounter situations where their code is erased entirely upon returning to the room.

6.5.4. Storage Infrastructure Considerations

For our storage infrastructure, the following considerations were considered:

Table 7: Pros and Cons of Considered Storage Infrastructures

Technology	Pros / Cons
MongoDB Atlas	Pros: <ul style="list-style-type: none"> • Easy to add on to the existing User, Question, and History database. • Require no extra storage locally. • Has built-in functions to clear room data after expiry. Cons: <ul style="list-style-type: none"> • Slow to read and write, dependable on Internet bandwidth.
MySQL	Pros: <ul style="list-style-type: none"> • Fast read and write speed as it is stored locally. Cons: <ul style="list-style-type: none"> • Need to install additional dependency. • Require additional storage locally. • No direct way to clear room data after expiry, require triggers.
Redis	Pros: <ul style="list-style-type: none"> • Fastest read and write speed as it is using in-memory cache. • Has built-in functions to clear room data after expiry. Cons: <ul style="list-style-type: none"> • Need to install additional dependency. • Require more memory. • Not persistent storage.

Weighing the pros and cons for each technology, we have decided to go with Redis due to its fast in-memory data store. This is crucial for collaborative use cases, given that coders typically type

at a speed ranging from 40 to 70 words per minute¹³. Therefore, we require storage technology capable of managing this substantial volume of changes, especially if there are multiple rooms open and making changes. Furthermore, we are unconcerned about the drawbacks of transient storage, as the room state will be automatically cleared after 2 hours of inactivity.

¹³ <https://www.hackerrank.com/the-average-typing-speed-for-programmers/#:~:text=How%20fast%20do%20programmers%20type,they%20should%20be%20exceeding%20this.>

6.6. History Microservice

The History microservice allows the user to save their attempt in the Collaboration room, including its executed output and run time duration. This service offers a comprehensive platform in facilitating the code review process, allowing users to reflect on their past mistakes and identify areas to improve their code efficiency. Our team firmly believes that code review is the most effective learning process, aligning with the core objective of this project.

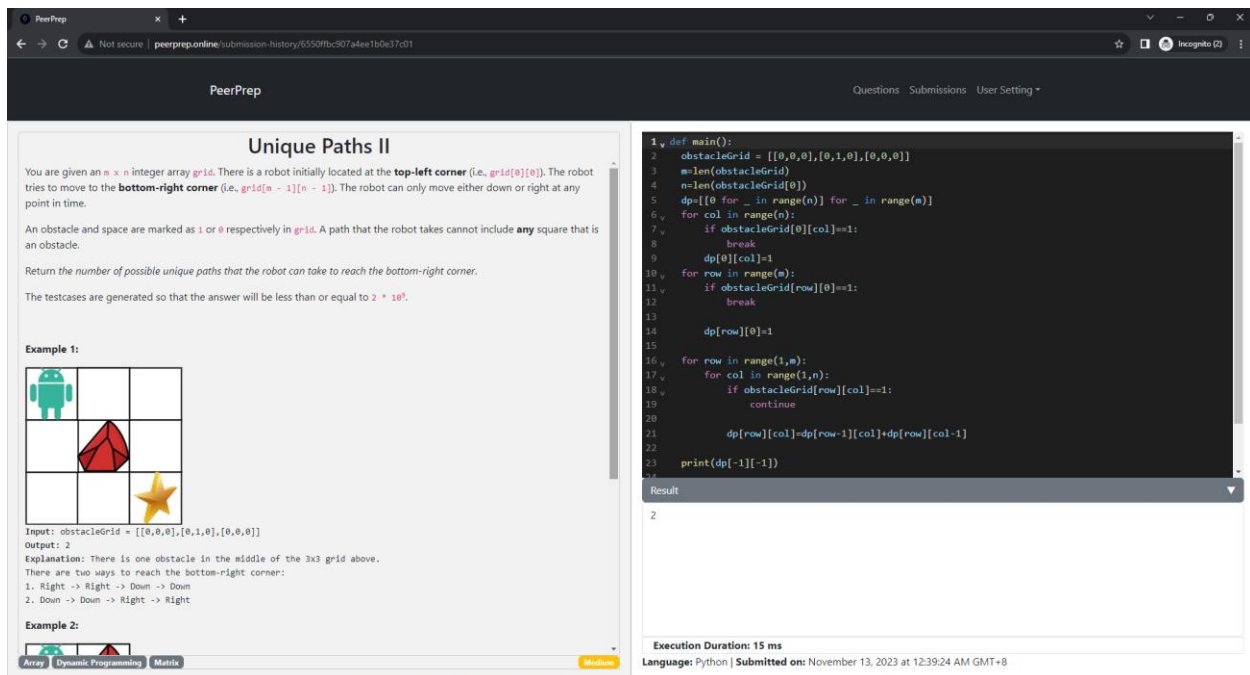


Figure 29: Screenshot of Attempted History

In addition to storing the past coding attempts, the History microservice provides a feature aimed at enhancing user engagement and facilitating the monitoring of progress – Statistical graphs. These graphical representations offer a visual overview of what the user has achieved so far throughout their coding progress, aiding them in understanding their proficiency level.

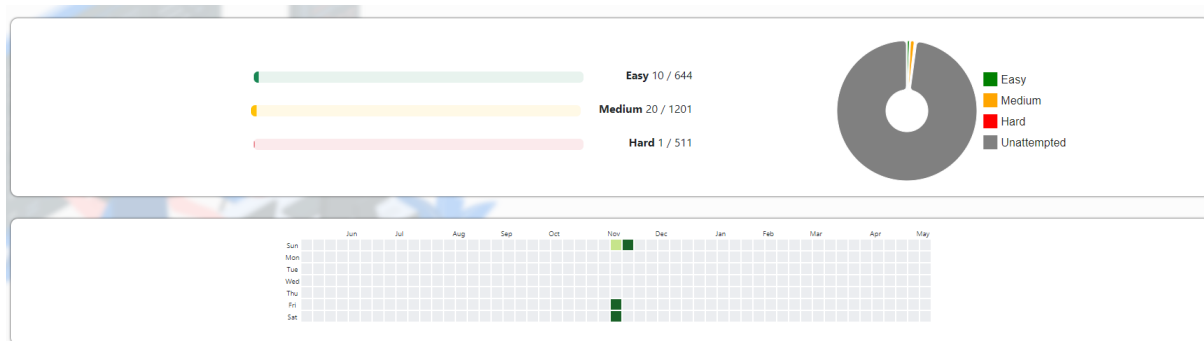


Figure 30: Statistical Diagrams for Progress Tracking

One of the key components of these graphs is a pie and bar chart that presents a breakdown of the user's attempted questions, based on difficulty levels. These charts provide users an immediate understanding of their coding proficiency, and users can quickly assess if they are leaning towards the easier or more challenging questions. Additionally, the charts complement progress tracking by displaying the number of questions they have yet to attempt, differentiated by difficulty levels. This information not only celebrates users' achievements but also serves as a motivational tool to encourage users to explore a wider variety of difficulty levels.

The heatmap feature is another statistical tool that serves as a powerful addition to our platform, offering users a dynamic visual representation of their submission patterns. The heatmap displays a timeline of up to 6 months ago, allowing users to pinpoint the specific periods they were more active. By utilizing a color gradient, the heatmap also signifies the intensity of code submissions for a particular day. This visual distinction aids users in quickly interpreting the data and recognizing different trends.

6.6.1. Design Considerations

For History service to generate the statistics for the attempted questions, several design options were considered for implementation:

6.6.1.1. Storing the Complete Snapshot of the Question into Database

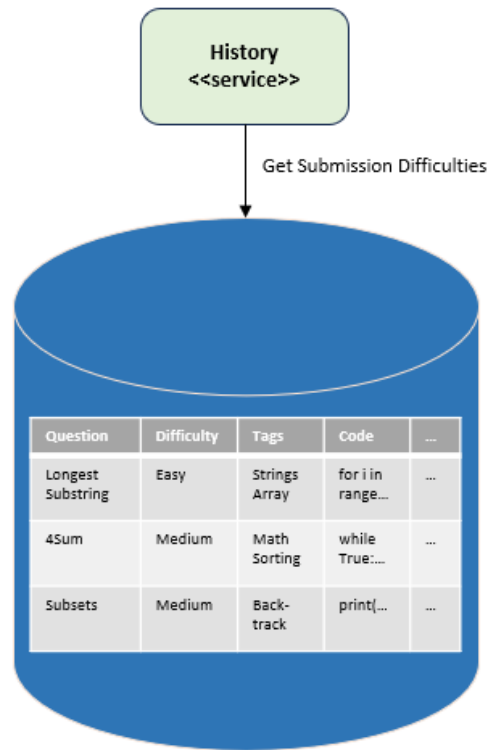


Figure 31: Snapshot Design

The first design choice involves the History service storing a snapshot of the question details at the time of submission, along with the submission attempt into a database. When statistics need to be generated, it directly calculates the count of Easy, Medium, and Hard submitted questions from its database.

Table 8: Pros and Cons of Snapshot Design

Pros	Cons
<ul style="list-style-type: none">Loosely coupled, does not require the retrieval of question difficulties from another database.	<ul style="list-style-type: none">Redundancy in storage due to duplicated question information.Static snapshot, when question details is updated, such as the difficulty level, this database must be updated.

6.6.1.2. Leveraging on the Question Database for Questions Retrieval

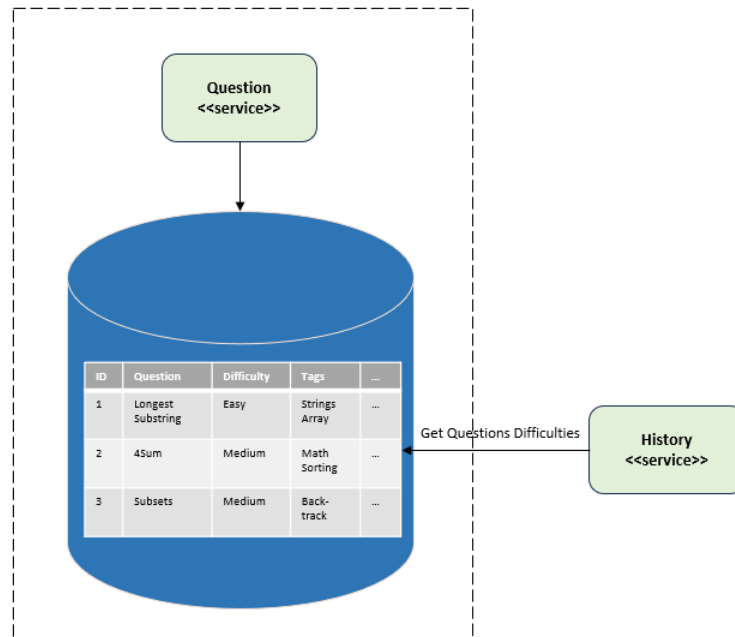


Figure 32: Leveraging on Question Database Design

The second design choice involves the History service directly fetching the difficulty levels of questions from the Question database. It then calculates the counts of each difficulty based on the submitted questions it has.

Table 9: Pros and Cons of Leveraging on Question Database Design

Pros	Cons
<ul style="list-style-type: none">• Dynamic retrieval of questions difficulties.• Easy implementation.	<ul style="list-style-type: none">• Implementation coupling.• If Question database changes, the data retrieval implementation of History service must change as well.

6.6.1.3. Using API Calls to the Question service for Questions Retrieval

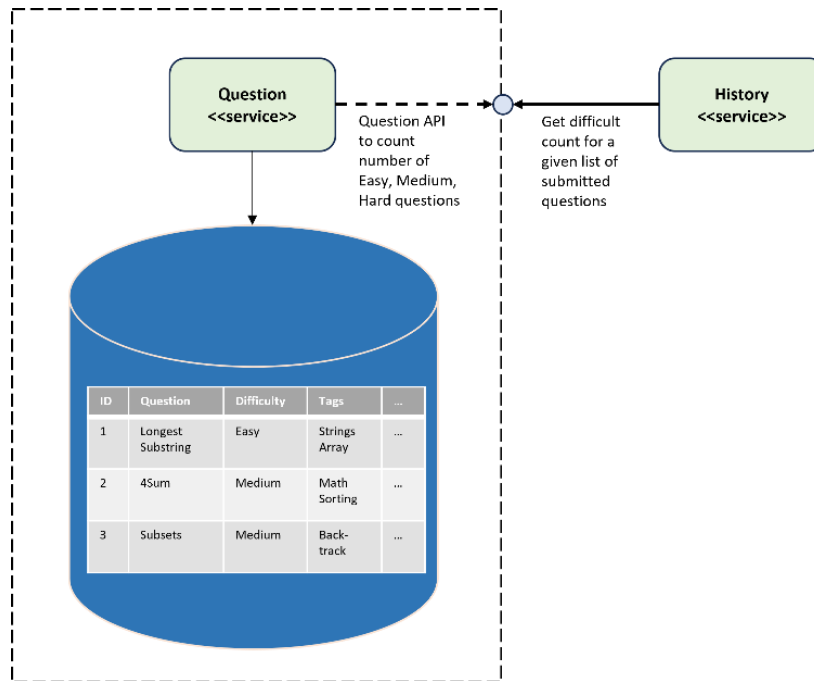


Figure 33: Using API Calls for Questions Retrieval

In our third design option, the History service initiates an API request to the Question service, supplying a list of question IDs that the user has submitted. The Question service then calculates the difficulty count based on the provided IDs and sends the response back to the History service.

Table 10: Pros and Cons of Using API Calls

Pros	Cons
<ul style="list-style-type: none">• Dynamic retrieval of question difficulties.• Reduced implementation coupling as API call should remain the same, even if Question database is changed.	<ul style="list-style-type: none">• Troublesome, must set up an additional API endpoint for Question service.

When conceptualizing this feature, our goal was to dynamically retrieve the attempted question. Specifically, if a maintainer updates the question's difficulty, the submission statistics should reflect the most recent difficulty level. Therefore, we have opted out of the first option. The second design was also dismissed due to our desire to avoid strong implementation coupling, particularly considering potential changes to the database instance in the future. The third design is the most beneficial to our architecture and is what we went for.

6.6.2. History Model

HistoryModel	
_id	Mongoose.Schema.Types.ObjectId
userId	Mongoose.Schema.Types.ObjectId
questionId	Mongoose.Schema.Types.ObjectId
code	String
language	String
output	String
duration	Number
timestamp	Date

Figure 34: Mongoose Model of History

With the chosen design consideration, we created a Mongoose model to store the contents for every submission attempt. This includes the user who submitted the attempt, the attempted question, the final code which the user had submitted, the language chosen for the code editor, the output generated by the execution service, the duration which the execution service took to run, and the time stamp of the submission.

6.6.3. Deletion of Users and Questions

We had to consider the case when a user deregisters, or a particular question is deleted by a maintainer. In such a scenario, it would be efficient to free up storage space by deleting the corresponding history attempts as they are no longer relevant.

6.6.3.1. Event Driven Design using Event Brokers

A potential approach to address this situation is to introduce event brokers between Question microservice and History microservice, as well as between User microservice and History microservice.

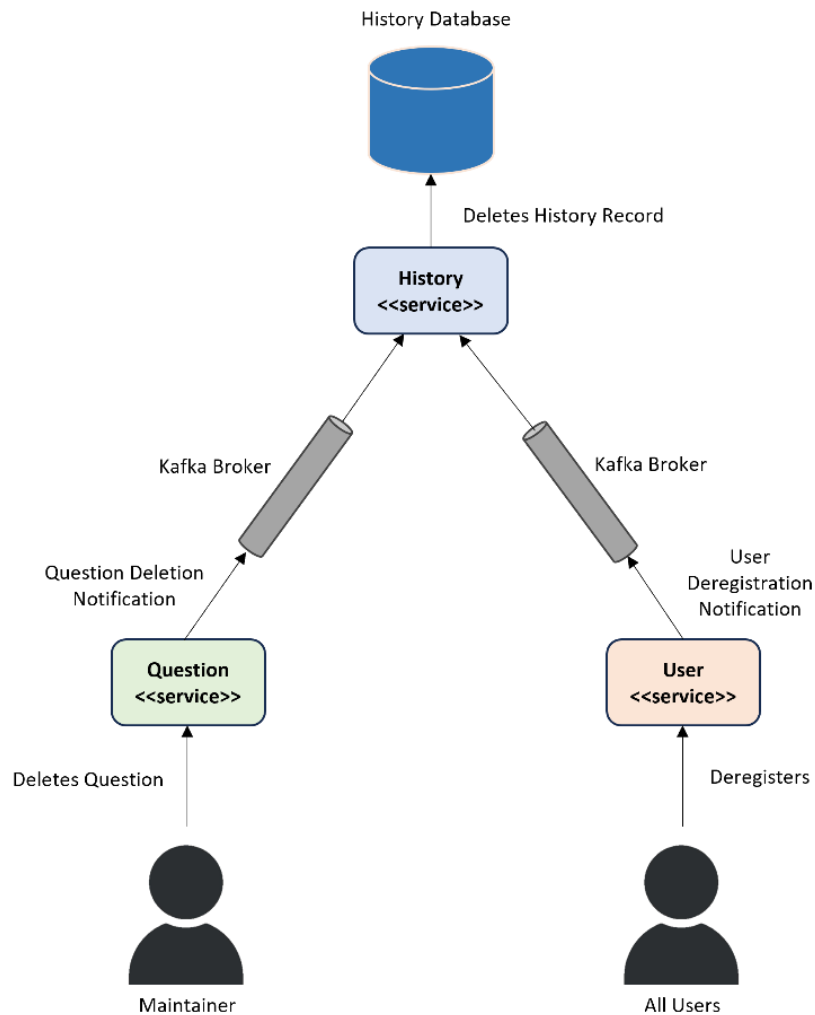


Figure 35: Event Brokers Solution

As depicted in Figure 35, the Question and User microservices can push a notification to a broker like Kafka. The History microservice subscribes to the channel, and upon receiving the notification, proceeds to delete the record from its database.

Table 11: Pros and Cons of Event Broker Solution

Pros	Cons
<ul style="list-style-type: none"> Decouples History service from Question and User services. 	<ul style="list-style-type: none"> Difficult to maintain consistency across remotely distributed systems. For example, if a system shuts down the History microservice and proceeds to delete a question, the History database will not be updated.

	<p>Another system that is connected to the same History database will get an out-of-date query.</p> <ul style="list-style-type: none"> • Need to locally install additional event brokers.
--	---

6.6.3.2. Serverless Synchronization of Databases

Given that we are using MongoDB Atlas for all our 3 databases, another approach is to utilize Atlas Triggers to delete the relevant entries in our History database.

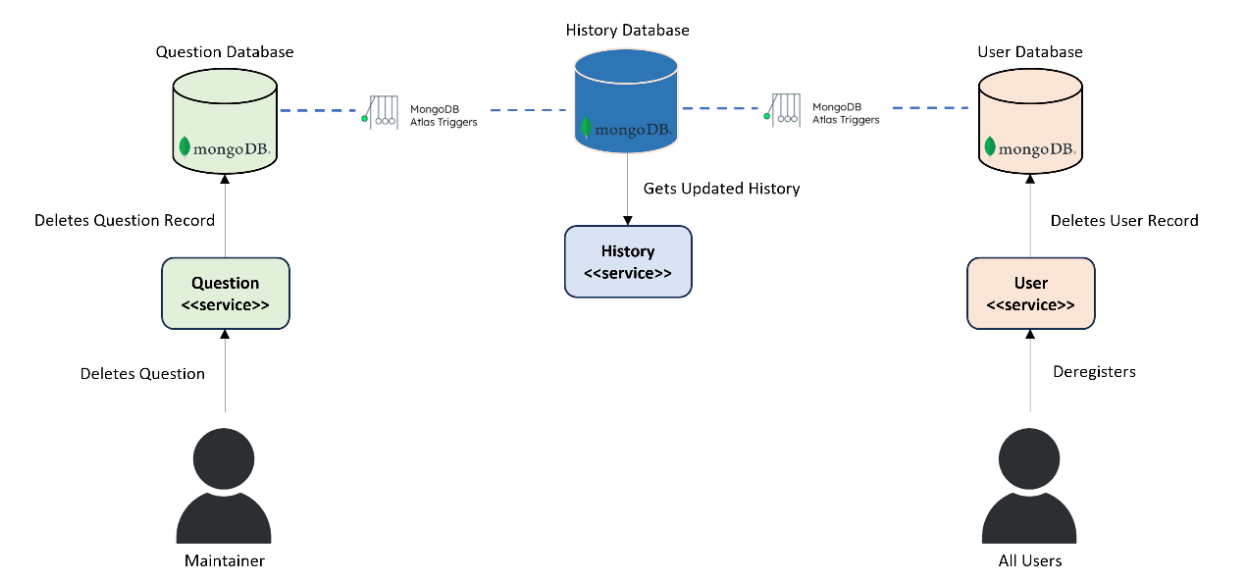


Figure 36: Synchronization of Databases with Atlas Triggers

Table 12: Pros and Cons of Atlas Triggers

Pros	Cons
<ul style="list-style-type: none"> • Take full advantage of Atlas Triggers. • Can maintain consistency of databases across distributed systems. • Serverless and free of charge. • Offers monitoring of trigger events. 	<ul style="list-style-type: none"> • Coupling across different databases.

Due to the high financial cost of running a centralized development server throughout the full 13 weeks of development, we faced budget constraints. Therefore, we could only develop the

application using our local machines most of the time. The remaining option available to obtain the updated History database was through the utilization of free Atlas Triggers.

6.7. Execution Microservice

The Execution microservice plays a crucial role in receiving code, executing it, and providing the user with the corresponding output. Currently, PeerPrep accommodates code execution in three languages: Python, Java, and JavaScript, which is in accordance with the options offered by the Collaboration microservice. The nature of PeerPrep is to provide code collaboration to aid students in their preparation. Thus, the most prominent software quality attributes in the design and implementation of this microservice includes security, maintainability, scalability. This microservice should be able to scale and accommodate a variety of languages to suit all students' coding needs, while acknowledging the risks of executing unknown foreign scripts.

6.7.1. Code Execution Workflow

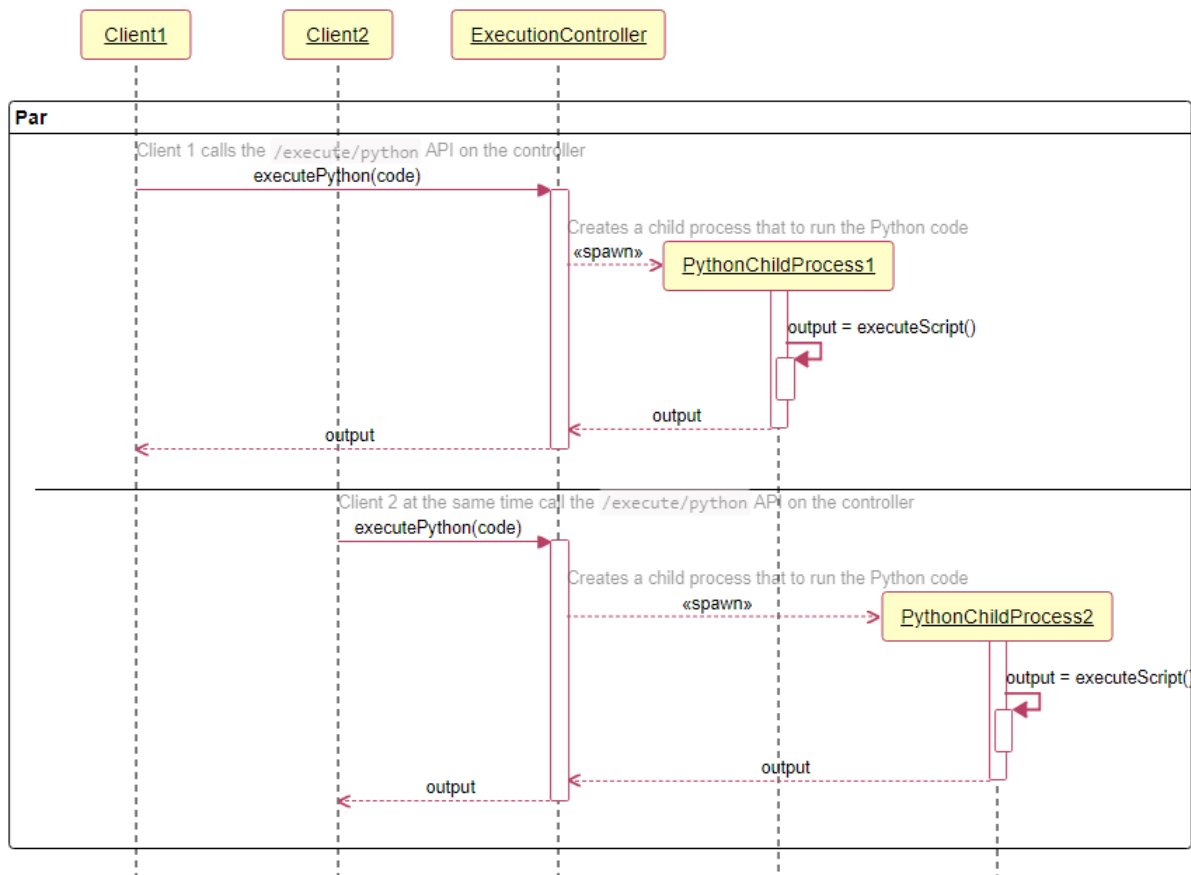


Figure 37: Simplified Sequence Diagram of a Python Code Execution Workflow

The figure above depicts a standard code execution workflow, utilizing Python for illustration. The concurrent execution of multiple users is facilitated by utilizing Child Processes, as mentioned above.

6.7.2. Design Considerations

6.7.2.1. Security

Implementing Execution as its own microservice has helped to decouple this from the Collaboration microservice. Since the Execution microservice only concerns itself with executing scripts, it does not have knowledge of who the user of the application is. Briefly, attacks occur when malicious code is executed to exploit vulnerabilities in a system, leading to unauthorized access, data breaches, service disruption, or control over the targeted system or application. Hence, the abstraction of code execution into its own service is justified as this allows for dedicated handling of these complexities.

Firstly, our approach employs the *spawn* function with explicitly specified interpreters, such as *python3*, *javac*, and *node*. This deliberate choice serves as a fundamental mitigation strategy against code injection risks by ensuring that only the intended programming language is executed. Any code not recognized by the specified interpreter returns an appropriate error response to notify the user. Spawn also executes the foreign scripts in a separate child process which limits interaction between Execution microservice process and the executing foreign script.

Secondly, the decision to rely mainly on specified interpreters to mitigate code injection is justified based on the scope of the project. Students are PeerPrep's target audience, and users are authenticated and authorized appropriately upon log-in to PeerPrep. Thus, a sophisticated code injected attack is unlikely based on the scope of this application. Furthermore, the microservice architecture ensures isolation of the code execution process from other microservices. Since only relevant APIs will be called from specified services, this guarantees the most restricted access required for a microservice. To further highlight this, Execution microservice's API is only referenced by the Collaboration microservice. Critical databases in other microservices cannot be referenced by the Execution microservice malicious scripts be executed. The current level of security provided by the Execution microservice is appropriate based on PeerPrep's scope and requirements.

In summary, isolating execution as its own microservice has the benefits of decoupling the complexities of executing different types of code into a standalone dedicated service. Executing foreign scripts in 'spawn' mitigates risks by executing foreign scripts in child processes in explicitly specified interpreters. This decision is justified through the scope and requirements of PeerPrep.

6.7.2.2. Code Errors

The Execution microservice is expected to handle code errors gracefully and notify users of the received errors appropriately, as stated in NF5.3 (Section 3.2: Non-Functional Requirements). Ideally, the interpreters specified in the 'spawn' functions that are used to execute the scripts will return all outcomes, including errors. However, there are edge cases that the interpreters cannot handle on its own. Execution microservice handles two edge cases, infinite loops, and empty codes.

Firstly, codes with infinite loops cannot be handled by 'spawn' alone. Executing infinite loops results in the child processes created by 'spawn' to run indefinitely. To mitigate this, Execution microservice employs a delayed function 'scriptTimeout' that will run after a constant MAX_EXECUTION_TIME is reached. This is defined as 5 seconds, as specified in Non-Functional Requirements NF1.4 (Section 3.2: Non-Functional Requirements). This function kills the child process(es) and returns a response to users notifying them that their code execution taking too long, as specified in Functional Requirements (Section 3.1).

The rationale behind incorporating a timeout in PeerPrep is driven by its design philosophy, which revolves around algorithmic problem-solving. As PeerPrep is dedicated to evaluating users' problem-solving skills, algorithmic proficiency, and coding abilities, the nature of challenges on the platform, commonly devoid of delays or time-based functions.

Secondly, empty code blocks do not return an output. Although this is not an explicit code error, returning a relevant response can improve the user experience. As specified in Functional Requirements (Section 3.1), users should be able to view the output of their code execution. This should also apply to empty code blocks. Execution microservice will explicitly check if the script it receives is empty and returns a relevant response notifying the user that the code executed is empty.

6.7.2.3. Dedicated Packages

During the implementation of Execution microservice, there were considerations in using dedicated node packages to handle code execution. The decision to implement custom functions instead of using existing node packages to handle execution are due to the following reasons: lack of extensive documentation, lack of variety of language packages, and long-term dependencies.

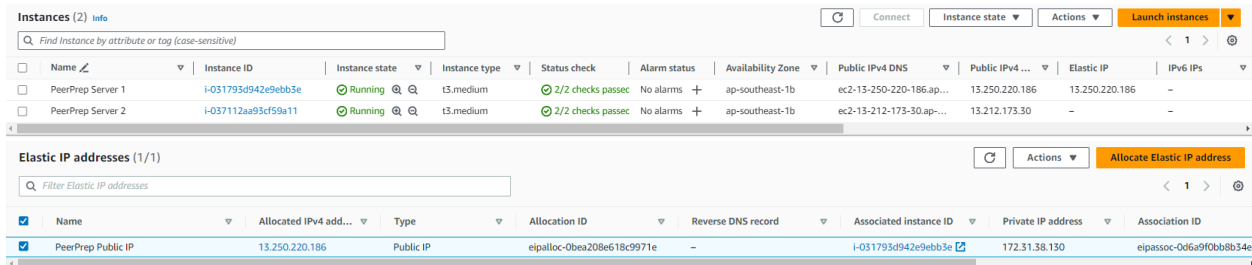
The first crucial consideration was the inadequacy of extensive documentation for existing Node packages tailored for code execution. The lack of comprehensive documentation posed challenges in terms of understanding the functionalities and nuances of these packages, potentially hindering effective integration, maintenance, and scalability. Given the pivotal role documentation plays in facilitating seamless development workflows, the team carefully weighed the trade-offs associated with limited documentation when evaluating existing packages.

Another key factor in creating custom execution functions was the observed scarcity in the variety of language packages available within the Node ecosystem. Scalability of Execution microservice should not be dependent on the availability of node packages. Direct implementation of execution functions ensures that PeerPrep can scale should the need for new languages arise. This consideration reinforced the decision to develop custom functions tailored to PeerPrep's specific needs for its target users and project requirements.

Long-term dependencies emerged as a pivotal concern in the decision-making process surrounding the Execution microservice with respect to PeerPrep's maintainability. Acknowledging the potential risks and limitations associated with dependencies on packages with ambiguous or unpredictable trajectories, custom execution functions are favored to prioritize long-term maintainability.

7. Deployment Process

To achieve reliability and reduce downtime, our team has decided to employ blue-green development in staging our changes. In our Amazon Web Service (AWS) console, we have 2 AWS EC2 instances running and an AWS Elastic IP leased (see Figure 38).



The screenshot shows the AWS Management Console. The top section displays two EC2 instances, both in a 'Running' state. The bottom section shows one Elastic IP address, 'PeerPrep Public IP', which is associated with 'PeerPrep Server 1'.

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...	Elastic IP	IPv6 IPs
PeerPrep Server 1	i-031793d942e9ebb3e	Running	t3.medium	2/2 checks passed	No alarms	ap-southeast-1b	ec2-13-250-220-186.ap...	13.250.220.186	13.250.220.186	-
PeerPrep Server 2	i-037112aa93cf59a11	Running	t3.medium	2/2 checks passed	No alarms	ap-southeast-1b	ec2-13-212-173-30.ap...	13.212.173.30	-	-

Name	Allocated IPv4 add...	Type	Allocation ID	Reverse DNS record	Associated instance ID	Private IP address	Association ID
PeerPrep Public IP	13.250.220.186	Public IP	eipalloc-0bea208e618c9971e	-	i-031793d942e9ebb3e	172.31.38.130	eipassoc-0d6a9f0bb8b34e1

Figure 38: EC2 Instances and Elastic IP

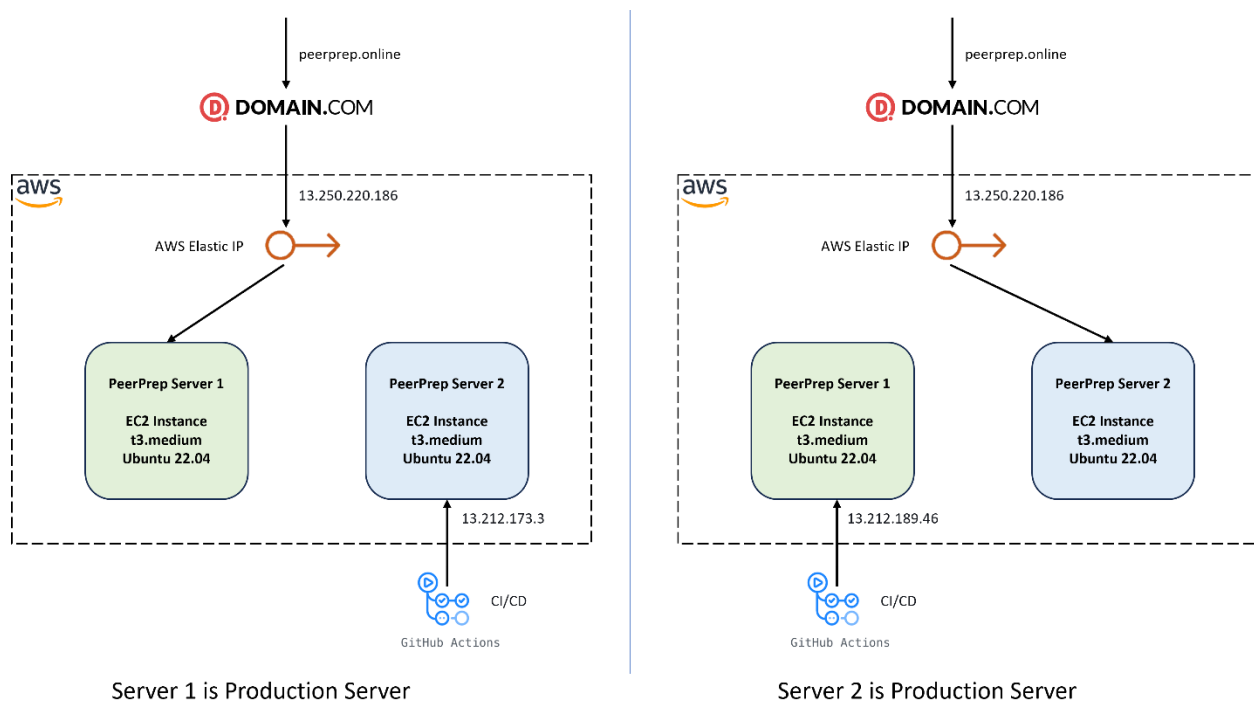


Figure 39: Overview of our Blue-Green Development

Both EC2 instances are identical in terms of their hardware and host software choices. We have set up our GitHub actions to deploy on our staging server automatically whenever there is a push to the master branch.

Once the team is satisfied with the results on the staging server, we will redirect our AWS Elastic IP service to the staging server. From this point onwards, the roles of both servers will switch: The staging server now becomes the actual production server, while the previous production server will be the staging server.

This cycle repeats whenever we wish to deploy a new code to the production server.

Table 13: Pros and Cons of Cloud Deployment with AWS EC2 Instances and AWS Elastic IP

Pros	Cons
<ul style="list-style-type: none">• Minimal down time while re-routing AWS Elastic IP. Elastic IP re-routing is instantaneous, compared to re-building of Docker on a single server.• Enable us to test extensively on the production-like environment before deployment, improves reliability.• Rollback capable, we can quickly revert to the previous server if something fails.	<ul style="list-style-type: none">• Requires the leasing of AWS Elastic IP, costing more money.

7.1. Deployment Requirements

We have conducted successful trials of deploying Docker containers on Windows, macOS, and Linux platforms. It's essential for the machine to maintain an active and robust internet connection to interact with MongoDB Atlas. Low internet latency is optimal to ensure a seamless user experience on the collaboration page.

7.2. Deployment Considerations

7.2.1. AWS Application Load Balancer

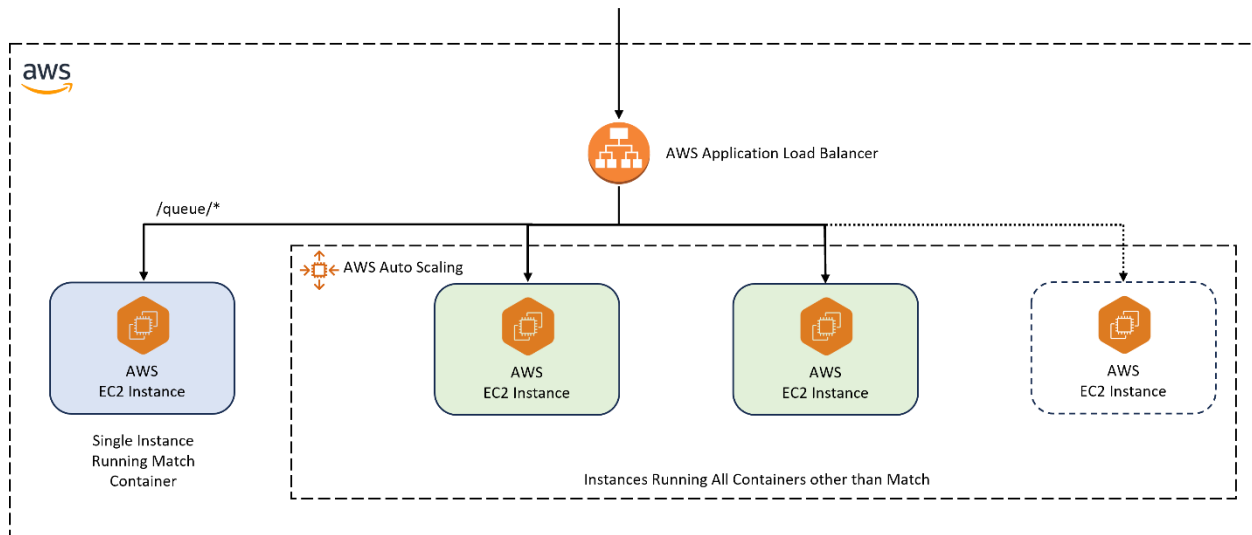


Figure 40: Load Balancing Across Multiple Instances

To balance the load from incoming traffic, we could deploy an outward-facing AWS Application Load Balancer that redirects traffic to different instances. This ensures that no single instance would be excessively burdened, improving the overall performance. Furthermore, we could allow horizontal scaling of instances using AWS Auto Scaling, deploying more instances if necessary. One point to note is that there should only be a single instance running the Match Container, because users should not be waiting in different waiting rooms despite having the same match criteria.

However, that option would not come cheap as the cost would multiply depending on the number of instances that is deployed. Furthermore, there is a need to pay for the additional AWS services which we are uncertain whether it would be claimable from the school. Therefore, we have decided to keep the cost low by deploying it in a single instance.

7.3. Stress Test on Deployment Server

Nevertheless, we are confident that our budget-friendly deployment would be more than capable of handling 100 users stated in NF1.1 (Section 3.2: Non-Functional Requirements). Our website has undergone successful stress testing utilizing Grafana k6.io¹⁴.

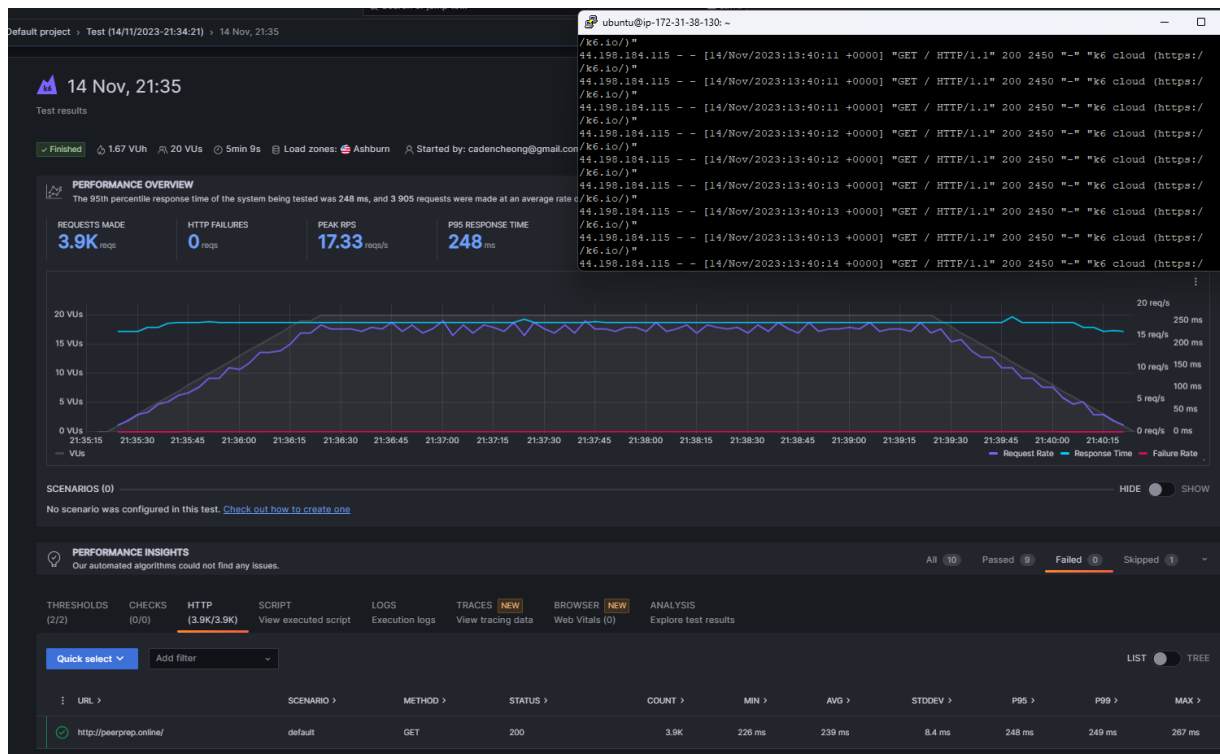


Figure 41: Stress Testing using k6.io

Shown in Figure 41, we had made over 3900 requests in a span of 5 minutes, surpassing our self-imposed requirement. None of the requests made received any errors, and the maximum response time was 267ms, meeting NF1.2 (in Section 3.2: Non-Functional Requirements). Therefore, the deployment of a load balancer with multiple instances is unnecessary in our specific context.

¹⁴ <https://k6.io/>

8. Suggestions for Improvements and Enhancements

8.1. Voice Calls and Video Conferencing

The intentional decision to exclude this feature in our current implementation aligns with our non-functional requirement (NFR) emphasizing bandwidth reduction, ensuring the website's optimal functionality in slow internet scenarios. However, the team acknowledges the potential benefits that voice calls and video conferencing could bring in, to provide a more authentic simulation and improve the overall user experience. A promising direction for future improvement involves integrating advanced technologies such as Socket.IO and WebRTC, in conjunction with established packages like PeerJs, to facilitate seamless peer-to-peer communication.

8.2. Expansion of Language Support

Currently, PeerPrep supports only Python, Java, and JavaScript, significantly limiting the number of users who can benefit from our platform. To address this constraint, our team aims to expand the range of supported programming languages. Enabling users to practice with their preferred and popular languages, such as TypeScript and C++, is a key enhancement we aspire to implement. This expansion aims to make PeerPrep more inclusive and adaptable to diverse language preferences, catering to a broader user base.

8.3. Execution Microservice Roadmap

For current requirements of this project, the level of abstraction of this microservice is appropriate. However, further abstractions will be needed should the project's target audience reach beyond students into a larger programming audience. Greater scrutiny of executing foreign scripts will also be needed at this scale. As such, there will be a need to improve implementation and design.

An advanced execution microservice that employs a myriad of interpreters highlights the need for further enhancements. Hence, virtual environments can be used to tackle the challenges associated with managing multiple asynchronous interpreters. This isolation mechanism improves security, optimizes resource usage, streamlines dependency management, and contributes to the scalability and reliability of the microservice architecture.

As shown in Figure 37 above, all the code execution occurs in 'ExecutionController'. As the Collaboration microservice scales to accommodate more languages, the code complexity of 'ExecutionController' will likely increase as well. This limits the scalability and maintainability of Execution microservice. An additional abstraction may help to ease this complexity. For example, a Factory design pattern may be utilized to instantiate the specific interpreter type dynamically on runtime as required.

8.4. Code Testing

Our current PeerPrep implementation lacks any form of code testing or code coverage, an aspect integral to a robust software development life cycle (SDLC). Code testing plays a pivotal role in bug identification, code quality assurance, and overall software reliability, and we recognize the need to introduce these practices as a vital future improvement. Additionally, incorporating code coverage tools will offer insights into the extent of code exercised by these tests, aiding in the identification of areas requiring additional scrutiny. This enhancement aligns with industry best practices, ensuring the ongoing evolution of PeerPrep as a reliable platform for users mastering technical interviews.

9. Reflections and Learning Points

9.1. Importance of Design Patterns

The implementation of PeerPrep has provided our team with valuable insights, particularly emphasizing the importance of design patterns in developing a scalable application. Building a system like PeerPrep requires careful attention to the chosen design pattern, emphasizing aspects such as a decoupled architecture and the assurance of a clear separation of concerns to achieve scalability.

Notably, the adoption of a microservices architecture has proven instrumental in overcoming common challenges by breaking down the application into smaller, independent services, which reduces coupling between services. This not only streamlines the development of a flexible and extensible system but also empowers individual services to evolve independently, fostering agility in our development processes.

Furthermore, our experience with the Controller-Service-Repository (CSR) design pattern firsthand highlighted its effectiveness in minimizing code modifications, by ensuring a separation of concerns and increase in cohesion, during the transition from our initial assignment setup to the development of PeerPrep, as mentioned earlier. This design pattern provided a structured framework that significantly facilitated a smoother migration, particularly during the transition from MySQL to MongoDB for databases. The inherent modularity in these architectures allowed for precise and targeted adjustments, effectively mitigating the impact of a substantial database transition on the overall codebase.

9.2. Importance of Software Requirements

The contents of lectures 2 and 3 on specifying software requirements, as well as lecture 6's content on DDD patterns, have helped to form the foundations and basis of this project.

By specifying software requirements, such as functional and non-functional requirements, the team gains a comprehensive understanding of the functionalities and features expected in the final product. This forms the basis for communication, collaboration, and decision-making throughout the project lifecycle.

Furthermore, the insights from Lecture 6, which delves into Domain-Driven Design (DDD) patterns, contribute to the robustness of the project. DDD patterns provide a systematic

approach to modeling and structuring the software based on the underlying business domain, which enabled the team to model PeerPrep as a microservice architecture effectively. This not only enhances the team's comprehension of the problem domain but also aids in aligning the software design with real-world scenarios, promoting a more effective and meaningful solution.

The knowledge acquired from these lectures has empowered the team to justify the proposed features with a profound understanding of user needs and system requirements. It has allowed for the establishment of a solid framework that ensures the delivered software meets both functional and non-functional expectations.

In summary, software requirements and DDD patterns have been instrumental in providing the team with a solid foundation for the PeerPrep project. This not only guides the development process but also ensures that PeerPrep aligns closely with the envisioned goals and requirements, ultimately contributing to the success of the project.

9.3. Room for Improvement

In hindsight, we felt that there were things where we could have done better during our development of PeerPrep.

Given that we were tasked with the assignments as scaffolding to start off for PeerPrep, we generally took a bottom-up approach for PeerPrep. This had assisted us in writing more reusable and modular code, as well as facilitating parallel development. However, it came with its own set of challenges as we did not have a very clear understanding of the required components in our project. Should we have the opportunity to revisit this project, we would strive to strike a balance between the bottom-up and top-down approaches, recognizing the strengths of both methodologies. Utilizing a top-down approach would have been beneficial to us as well since we knew the rough overall architecture. A hybrid approach would enable us to leverage our knowledge of the project's architecture while harnessing the benefits of incremental development and modular code.

Additionally, we think that we could put more time and effort into UI/UX of PeerPrep. We tend to dive straight into implementing features, prioritizing the technical aspects over the design and user-friendliness. Looking back, we realize that a more effective approach would involve thinking about the UI and UX right from the start. This means planning the front-end design alongside our initial ideas for how features should work. Tools like Figma can help us with this process. By doing this, we can create a more seamless and user-friendly experience for our PeerPrep users.