



CS3219
Software Engineering Principles and Patterns
Final Project Report

Group 10

No.	Full Name (as in EduRec)	Student Number
1	Suryansh Kushwaha	A0262331R
2	Kim Yongbeom	A0179500H
3	Bhanuka Bandara Ekanayake	A0257875J
4	C Mahidharah Rajendran	A0252508H
5	Suresh Rubesh	A0253146J

Table of Contents

1. Declaration.....	4
2. Individual Contributions.....	5
3. Project scope (Product backlog).....	7
3.1 Functional Requirements (+ Fulfilment).....	7
F1: User Service.....	7
F2: Matching Service.....	7
F3: Question Service.....	9
F4: Collaboration Service.....	10
3.2 Non-Functional Requirements (+ Fulfilment).....	12
N1: User Service.....	12
N2: Matching Service.....	13
N3: Question Service.....	13
N4: Collaboration Service.....	14
3.3 Selected Nice-to-haves.....	15
N1: Real Time Chatting.....	15
N3: Code Execution.....	15
N4: Enhanced Collaboration Service.....	16
N6: Incorporation of Generative AI.....	16
N7: Cloud Deployment.....	17
4. Architecture Design.....	18
4.1 Overall Architecture.....	18
4.2 Project Structure & Overall Development.....	20
4.2.1 Development Process & Repository management.....	20
4.2.2 Project Organization.....	21
4.2.3 Project Inter-Component Interactions.....	22
4.2.4 MVC Architecture.....	23
4.3 Choice of databases.....	24
4.3.1 Question Service Databases.....	24
4.3.2 Matching Service Databases.....	26
4.4 Collaboration Service Mechanism.....	28
4.4.1 Collaboration Service Code Editors.....	28
4.4.2 Collaboration Service Synchronisation Means.....	29
5. Nice-to-have Implementations.....	33
5.1 N1 – Communication.....	33
5.1.1 Why Socket IO?.....	33
5.1.2 N1 Communication Implementation.....	34
5.2 N3 – Code Execution.....	35
5.3 N4 – Enhanced Collaboration Service.....	36
5.3.1 Multi-Language Syntax Highlighting.....	36
5.3.2 Multi-Language Code Completion Support.....	36



5.3.3 Collaboration UI and UX Enhancements.....	37
5.4 N6 – Incorporation of Generative AI.....	38
5.5. N7 – Cloud Deployment.....	42
6. Internal Microservices Design.....	44
6.1 Question Service Internal Design.....	44
6.1.1 Components.....	45
6.1.3 Evaluation of the Architecture (Pros & Cons).....	47
6.1.4 Deviation from MVC.....	48
6.1.5 Sequence Diagram of Question Service.....	49
6.2 Design of Matching Service.....	50
6.2.1 Components.....	50
6.2.2 Synchronous Communication.....	51
6.2.3 Command Query Responsibility Segregation (CQRS).....	54
6.2.4 Sequence Diagram of Question Service.....	56
7. Project plan.....	57
7.1 Gantt Chart.....	57

1. Declaration

We, the undersigned, declare that:

1. The work submitted as part of this project is our own and has been done in collaboration with the members of our group and no external parties.
2. We have not used or copied any other person's work without proper acknowledgment.
3. Where we have consulted the work of others, we have cited the source in the text and included the appropriate references.
4. We understand that plagiarism is a serious academic offence and may result in penalties, including failing the project or course.
5. We have read the [NUS plagiarism policy and the Usage of Generative AI](#).

Group Member Signatures:

Full Name (as in Edu Rec)	Signature	Date
Suryansh Kushwaha		10 Nov 2024
Kim Yongbeom		10 Nov 2024
Bhanuka Bandara Ekanayake		10 Nov 2024
C Mahidharah Rajendran		10 Nov 2024
Suresh Rubesh		10 Nov 2024

2. Individual Contributions

No	Full Name (as in Edu Rec)	Contributions
1	Suryansh Kushwaha	Question Service Backend Question Service Debugging Matching Service Backend Initial Setup Collaboration Service Debugging Tailwind CSS Integration UI/UX Enhancement of entire application Code Cleanup N2H: Enhanced Communication
2	Kim Yongbeom	Initial Repo setup Question service backend Matching service backend Dockerized Question service, User service Implemented docker-compose Debug question service Debug matching service N2H: Cloud Deployment
3	Bhanuka Bandara Ekanayake	Question Service Frontend Question Service Debugging Matching Service Database setup Matching Service Debugging Matching Service Containerisation Collaboration Service Backend Collaboration Service Debugging Collaboration Service Containerisation Enhancements for Question Service Frontend Enhancements for Question Service Backend N2H: Code Execution
4	C Mahidharah Rajendran	Question service design and plan Question service backend Question service database setup Question service sample data integration Question service backend integration Question service debugging User service frontend User service backend integration User service debugging Containerisation of user and question service Matching service design Matching service backend Matching service real-time database (Redis) integration Matching service algorithm implementation Matching service backend integration Matching service server-side-events implementation Matching service debugging Collab service design and plan Collab service backend

		Collab service backend integration Collab service debugging Generative-AI integration
5	Suresh Rubesh	Creation of Milestone 1 prototype in SwiftUI Question service Frontend (All Views) Question service Frontend integration Question service Api (for integration w matching service) User Service Frontend (All Views) Matching Service Frontend (All Views including Stub View) Matching service debugging Matching service integration with question service Implementation of Docker network bridge + Cross Origin Resource Sharing (using docker network bridge) for cross-service interactions Collab Service Frontend (All Views) Collab Service Code Editor implementation Collab Service overall design & implementation N2H - Code Editor Enhancements for Collab Service Enhanced syntax highlighting for 8 languages Autosuggestion for 8 languages Integration with Code Execution for 8 languages C, C++, C#, Java, JavaScript, Ruby, Kotlin, Python Overall Frontend <ul style="list-style-type: none"> - Routing and Navigation - Overall UI/UX incl CSS styling and behaviour and standardizations across the application

3. Project scope (Product backlog)

3.1 Functional Requirements (+ Fulfilment)

Functional Requirements	Priority	Sprint	PR
F1: User Service			
F1.1 User Initialization - Handling user registration, onboarding, and authentication			
F1.1.1 Allow users to create (i.e. sign up for) a new account with essential details such as username, email and password.	H	Sprint 2	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/21/
F1.1.2 Allow for the creation of an initial admin user via direct database access	M	Sprint 2	-
F1.2 User Authentication and Security - Securing account access			
F1.2.1 Implement secure login system for user to access their accounts based on their registration details (email & correct password)	H	Sprint 2	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/21/
F1.2.2 Allow for persistent identification of authenticated user app-wide	M	Sprint 2	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/21/
F1.3 User Profile Management - Managing and updating user account			
F1.3.1 Allow users to view their profile page, with essential details such as username, registered email and account creation date	M	Sprint 2	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/21/
F1.3.2 Allow user to check their admin status	L	Sprint 2	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/21/
F1.4 Admin Privileges - Admin access & management of user accounts			
F1.3.1 Allow for admin users to view a list of all users and their details	L	Sprint 2	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/21/
F1.3.2 Allow admin users to assign admin privileges to other users	L	Sprint 8	Pending
F1.3.3 Allow admin users to permanently delete user accounts	L	Sprint 8	Pending
F2: Matching Service			
F2.1 User Input - Capturing user preferences to facilitate accurate matching			
F2.1.1 Allow users to specify topic for question they want to work on	H	Sprint 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35

F2.1.2 Allow users to specify difficulty level for question	H	Sprint 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
F2.1.3 Ensure all specifications/preferences listed above are captured simultaneously	H	Sprint 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
F2.1.4 Retrieve available question topics and their corresponding difficulties from the question database	M	Sprint 8	(PR after report)
F2.1.5 Allow users to cancel their matching request to create a new one	M	Sprint 5	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/51
F2.2 Matching Algorithm - Core functionality of pairing users based on their inputs			
F2.2.1 Implement priority matching based on topic over difficulty by time in queue (after 15s accept matches with different difficulties)	H	Sprint 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
F2.2.2 Set fixed waiting time (e.g. 30 seconds) for finding a match	M	Sprint 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
F2.2.3 Break ties on difficulties if users indicated different difficulties	M	Sprint 4	(PR after report)
F2.3 User Feedback - Keeping users informed throughout the matching process			
F2.3.1 Provide visual and time feedback (e.g. animation) during waiting period	M	Sprint 5	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
F2.3.2 Handle and communicate successful matches to users	H	Sprint 5	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
F2.3.3 Handle and communicate unsuccessful matches to users	H	Sprint 5	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
F2.4 Integration - Connecting the Matching Service with other components of the system			
F2.4.1 Integrate with User Service to access user information (userId)	H	Sprint 5	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35

F2.4.2 Integrate with Question Service to assign question upon match	H	Sprint 5	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/53
F2.5 Queueing Mechanism - Managing multiple concurrent matching requests efficiently			
F2.5.1 Implement a queue system for managing user match requests	H	Sprint 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
F2.5.2 Ensure efficient processing of the queue for timely matching	M	Sprint 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
F2.5.3 Ensure users do not have to handle synchronisation issues	M	Sprint 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
F3: Question Service			
F3.1 Question Retrieval - Enable users to access question repository			
F3.1.1 Allow users to view table of questions available	H	Sprint 1	Feat/frontend v1 by bhnuka · Pull Request #12 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10
F3.1.2 Display full question upon clicking question title in search	H	Sprint 1	[Feat] Final UI Changes by sp4ce-cowboy · Pull Request #16 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10
F3.2 Integration - Ensuring question service works with application and services			
F3.2.1 Integrate with the Matching Service to find compatible questions and provide the appropriate questions for matched users in a collaborative session	H	Sprint 6	Rtdb v2 by sp4ce-cowboy · Pull Request #35 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10
F3.2.2 Integrate with the Collaboration Service to provide the full question display (F3.1.2) for matched users in a collaborative session	H	Sprint 6	Collaboration Service Frontend + Basic Implementation by sp4ce-cowboy · Pull Request #41 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10

F3.3 Question Management - Updating question repository			
F3.3.1 Implement an interface for users to add new questions to the repository	H	Sprint 1	Feat/frontend v1 by bhnuka · Pull Request #12 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10
F3.3.1.1 Allow users to add questions with various categories	H	Sprint 1	
F3.3.1.2 Allow users to specify question title	H	Sprint 1	
F3.3.1.3 Allow users to specify question description	H	Sprint 1	
F3.3.1.4 Allow users to specify question difficulty	H	Sprint 1	
F3.3.1.5 Prevent users from adding duplicate questions	H	Sprint 1	Add topics + Add question duplication check when updating qn by bhnuka · Pull Request #61 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10
F3.3.2 Allow users to update existing questions	H	Sprint 1	Feat/frontend v1 by bhnuka · Pull Request #12 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10
F3.3.2.1 Prevent users from adding duplicate questions by editing existing questions	H	Sprint 1	Add topics + Add question duplication check when updating qn by bhnuka · Pull Request #61 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10
F3.3.4 Allow users to remove outdated or inaccurate questions	H	Sprint 1	Feat/frontend v1 by bhnuka · Pull Request #12 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10
F3.4 Accessibility - Improving question discovery and user experience			
F3.4.1 Provide a preview for question's key details (e.g., title, difficulty, topic) without fully opening it while searching	M	Sprint 1	[Feat] Final UI Changes by sp4ce-cowboy · Pull Request #16 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10
F4: Collaboration Service			
F4.1 Collaboration Sync			

F4.1.1 The collaboration service supports real-time concurrent code editing for matched users.	H	Sprint 7	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
F4.2 Authentication & Authorization			
F4.2.1 The collaboration service allows authenticated and matched users to connect to the collaborative space.	H	Sprint 5	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
F4.2.2 The collaboration service prevents non-matched or non-authenticated users from connecting to the collaborative space.	H	Sprint 5	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
F4.3 Connection Handling			
F4.3.1 The collaboration service allows users to terminate the session at any time.	M	Sprint 5	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
F4.3.2 The collaboration service notifies the users if the connection to the space is lost.	M	Sprint 5	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
F4.3.3 Upon disconnection, the collaboration service automatically attempts to reconnect a user back to the collaborative space.	L	Sprint 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35

3.2 Non-Functional Requirements (+ Fulfilment)

Non-Functional Requirements	Priority	Sprint	PR Link/Comments
N1: User Service			
N1.1 Performance - Speed and efficiency of the user account management operations			
N1.1.1 Handle creation of a new account within 1 second (i.e. turnaround time between request & success response.)	M	Sprint 2	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/21/
N1.1.2 Handle account login within 1 second. (i.e. turnaround time between request & success response.)	M	Sprint 2	
N1.1.3. Excluding time taken for user interactions with the app, the turnaround time between account registration and initial account login should be less than 2 seconds.	M	Sprint 2	
N1.1.4 Reflect profile updates (i.e. assignment of admin privileges) within 2 seconds.	M	Sprint 2	
N1.2 Scalability - System's ability to handle growth in users			
N1.2.1 Handling up to 500 concurrently logged in users without significant performance degradation.	L	Sprint 2	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/21/ Based on MongoDB free tier bandwidth.
N1.2.2 Handling up to 5000 total registered users without significant performance degradation in user registration operations.	L	Sprint 2	
N1.3 Reliability - Ensuring consistent and dependable user service			
N1.3.1 Maintain an uptime of 99.0% to ensure high availability of the application	M	Sprint 2	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/21/ Based on MongoDB free tier bandwidth.
N1.3.2 Ensure that user profile data remains accurate and uncorrupted during updates and deletions (by admin users), for up to 100 simultaneous updates for unique users	H	Sprint 3	
N1.3.3 Ensure that user profile data remains accurate and uncorrupted during updates and deletions (by admin	L	Sprint 2	
			https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/21/

users) when multiple admins modify details of the same user			Based on MongoDB's last-write wins policy where concurrent write operations will not corrupt data.
N2: Matching Service			
N2.1 Performance - Speed and efficiency of the matching service			
N2.1.1 Match users under 30 seconds	H	Sprint 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
N2.1.2 Handle at least 100 concurrent matching requests	H	Sprint 4	
N2.1.3 Maintain response time under 500 ms for 95% of requests	M	Sprint 4	
N2.1.4 Ensure no synchronisation issues with frontend and backend	H	Sprint 5	
N2.2 Reliability - Ensuring the matching service remains available and functional			
N2.3.1 Achieve 99.9% uptime for the matching service	H	Sprint 8	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
N2.3.2 Implement fault tolerance for queue system	H	Sprint 7	
N3: Question Service			
N3.1 Performance - Speed and efficiency of user experience			
N3.1.1 Retrieve and display a question on the full question display within 3 seconds for 95% of requests	H	Sprint 2	Feat/backend v1 by suryanshkushwaha · Pull Request #13 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10 Based on MongoDB free tier bandwidth
N3.1.2 Handle up to 100 concurrent users retrieving questions simultaneously without performance degradation	M	Sprint 2	
N3.2 Scalability - System's ability to handle large number of users and questions			
N3.2.1 Efficiently manage up to 1000 questions in the repository with minimal impact on retrieval time (<5 seconds)	H	Sprint 2	Same as above Based on MongoDB free tier bandwidth
N3.2.2 Support up to 1000 concurrent users browsing the question repository	M	Sprint 2	
N3.3 Reliability - Ensuring the question service remains accurate and available			
N3.3.1 Maintain 99.9% uptime for the question service to ensure high availability for users	H	Sprint 2	Feat/backend v1 by suryanshkushwaha · Pull Request #13 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10

N4: Collaboration Service

N4.1 Latency			
---------------------	--	--	--

N4.1.1 The collaboration service shall synchronise code changes between both users in <500 ms.	H	Sprint 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
N4.1.2 Text messages exchanged between the users should be synchronised in <500ms.	M	Sprint 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35

N4.2 Scalability			
-------------------------	--	--	--

N4.2.1 The collaboration service should support at least 1,000 concurrent users without significant performance degradation.	M	Sprint 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/35
--	---	----------	---

3.3 Selected Nice-to-haves

Nice-to-haves	Priority	PR
N1: Real Time Chatting		
N1.1 Real-Time Messaging - Enable real-time messaging for users within the same collaborative space		
N1.1.1 Implement messaging using Socket.io for low-latency communication	H	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/commit/e8af53d5c5d42d0f8e3fd822d40852425ae7d706
N1.1.2 Restrict message exchange to users within the same collaborative session	H	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/commit/e8af53d5c5d42d0f8e3fd822d40852425ae7d706
N1.2 User Experience Enhancements		
N1.2.1 Color-code messages based on sender and receiver for easy identification	H	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/commit/e8af53d5c5d42d0f8e3fd822d40852425ae7d706
N3: Code Execution		
N3.1 Code Execution - Running user-submitted code and returning results		
N3.1.1 Implement support for base64-encoded code submission and result retrieval	H	working judge0 api for code execution by bhnuka · Pull Request #45 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10
N3.1.2 Ensure code execution handles multi-line code input properly	H	
N3.1.3 Display code execution results in a separate output area	H	
N3.1.4 Implement logging for code execution requests and responses for debugging	H	
N3.2 Integration - Connecting code execution with other components		
N3.2.1 Integrate code execution service with the collaborative editor	H	working judge0 api for code execution by bhnuka · Pull Request #45 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10
N3.2.2 Integrate code execution with Question Service to retrieve and display test cases	M	Testcase addition to collabservice ui by bhnuka · Pull Request #62 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10
N3.3 User Experience - Enhancing the code execution process for a seamless user experience		

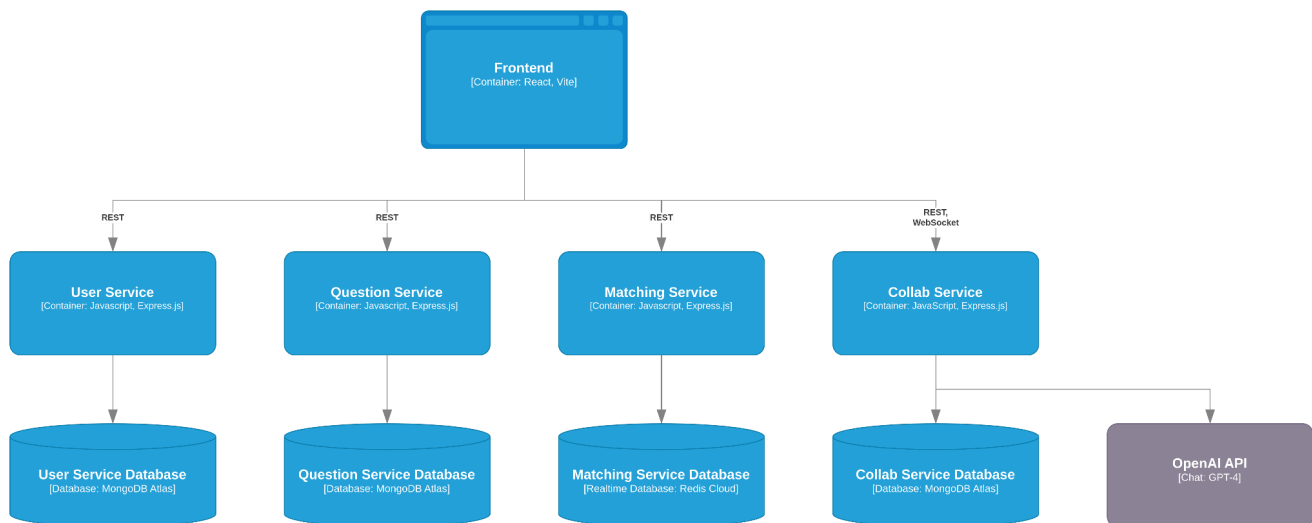
N3.3.1 Code execution is per user (i.e. if one user runs the code, the other user does not see the output unless they too choose to run the code on their end)	H	working judge0 api for code execution by bhnuka · Pull Request #45 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10
N3.3.2 Add language selection dropdown for users to choose programming language	H	
N3.3.3 Allow code to be run various languages, namely, C, C++, C#, Python, Java, JavaScript, Ruby and Kotlin	H	
N4: Enhanced Collaboration Service		
N4.1 Multi Language Syntax Highlighting support		
N4.1.1 Implement syntax highlighting for 8 languages. (C, C++, C#, Java, JavaScript, Python, Ruby, Kotlin)	H	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/48
N4.1.2 Implement automatic code formatting features, such as auto-indentation and	H	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/48
N4.2 Multi Language Code Completion support		
N4.2.1 Implement “intellisense” like in-line code completion suggestions for 8 languages. (C, C++, C#, Java, JavaScript, Python, Ruby, Kotlin)	M	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/48
N4.2.2 Implement real time switching for all 8 languages such that the user can switch to any language at any time, and see the updated syntax highlighting & code completion without having to reload the page or join a new session	M	
N4.3 Collab UI General Enhancements & Coding UX Enhancements –		
N4.3.1 Implement ability to undo code changes that are visible on both users’ ends	H	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/48
N4.3.2 Add monospaced font and line number indication to editor to emulate feel of a real code editor	M	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/48
N4.3.3 Implement a unified language selector such that the user can change both syntax language and execution language for all 8 languages with one selector	M	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/48
N6: Incorporation of Generative AI		

N6.1 Generative AI implementation		
N6.1.1 Ensure generative AI assesses code based on style	H	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/53
N6.1.2 Ensure generative AI assesses code based on time-complexity	H	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/53
N6.1.3 Ensure generative AI assesses code based on space-complexity	H	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/53
N6.1.4 Provide question context for generative AI to asses code correctness	M	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/60
N6.1.5 Ensure generative AI provides hints if attempt can be better	M	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/53
N6.2 Integrate with frontend		
N6.2.1 Ensure users can intuitively access the generative AI easily	H	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/53
N6.2.2 Discourage users from prompting generative AI such that they retrieve answers via the frontend	H	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/53
N6.2.3 Integrate the frontend with other components in the collaboration frontend seamlessly	L	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g10/pull/53
N6.3 Generative AI reliability		
N6.3.1 Improve reliability of generative AI	H	(PR to come)
N6.3.2 Optimise generative AI behaves as requires	M	(PR to come)
N7: Cloud Deployment		
N7.1 DNS, SSL, and Load Balancing		
N7.1.1 Integrate automated renewal for SSL certificates on GCP to streamline the HTTPS process and reduce manual intervention.	H	Cloud deployment by Yongbeom-Kim · Pull Request #59 · CS3219-AY2425S1/cs3219-ay2425s1-p roject-g10
N7.1.2 Configure GCP Cloud DNS with failover settings for improved reliability and automatic domain resolution adjustments.	H	
N7.1.3 Use GCP's Cloud CDN to distribute static files globally, reducing latency for users in different regions.	H	
N7.2 Database and Caching		
N7.2.1 Schedule automated backups for MongoDB and Redis databases to	H	Cloud deployment by Yongbeom-Kim · Pull Request #59 ·

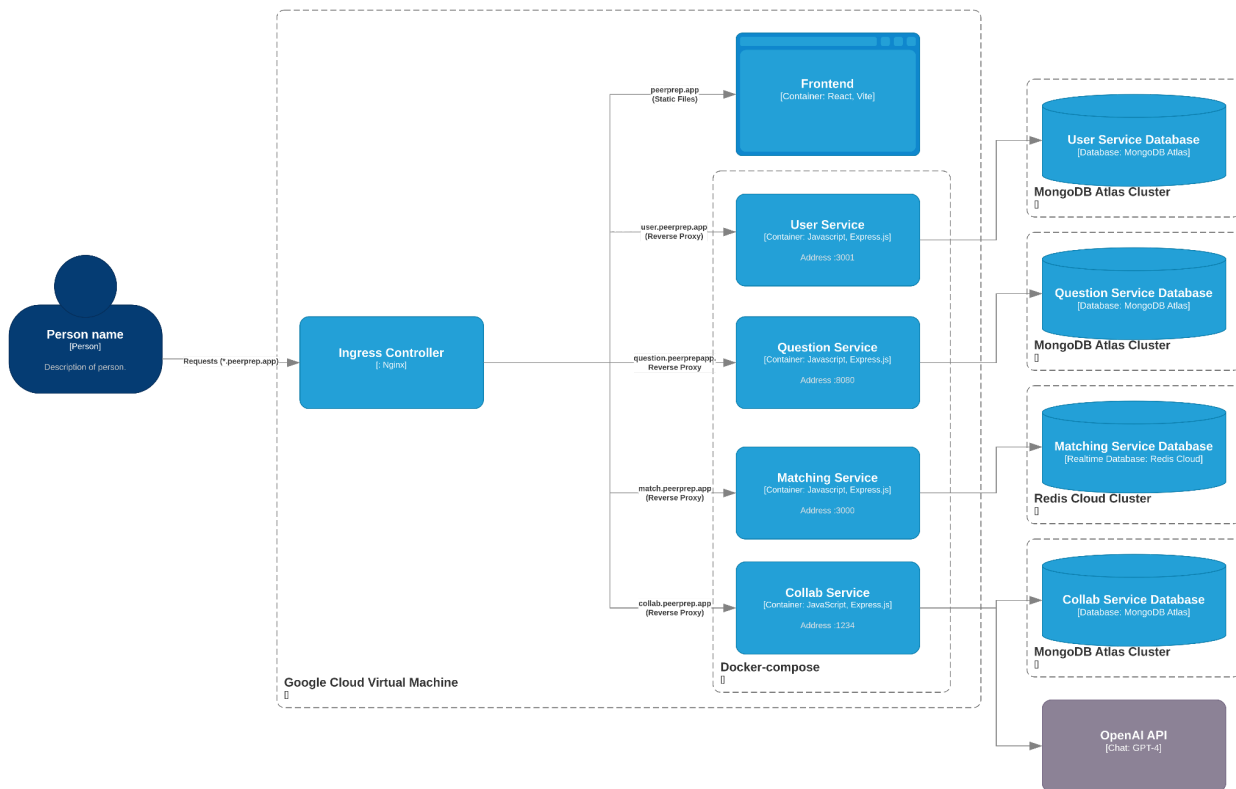
ensure data integrity and recovery options.		CS3219-AY2425S1/cs3219-ay2425s1-project-g10
N7.2.2 Implement Redis caching on GCP for frequently accessed data to improve response times and reduce load on the main database.	M	
N7.3 Nginx Configuration, Monitoring, and Logging		
N7.3.1 Implement monitoring and logging for Nginx to track and analyse traffic patterns, identify bottlenecks, and troubleshoot issues in real time.	H	Cloud deployment by Yongbeom-Kim · Pull Request #59 · CS3219-AY2425S1/cs3219-ay2425s1-project-g10
N7.3.2 Set up custom error pages in Nginx for a better user experience in case of downtime or connectivity issues.	M	

4. Architecture Design

4.1 Overall Architecture



We opted to follow a microservice architecture, with 4 microservices - User, Question, Matching and the Collaboration service. Each service has its own database, and the collaboration service also can make requests to GPT-4 via an OpenAI API key. Locally, the frontend is hosted on port :5173 with the Vite build tool, and each service is containerized and deployed with the Docker compose tool.



As for its cloud deployment, both the frontend, as well as all backend services are deployed on a single virtual machine configured in Google Cloud, with the DNS records for `peerprep.app` (and subdomains) pointing to the public IP address of the virtual machine. There is a bare-metal Nginx service that acts as an ingress controller, routing requests to the frontend and all services based on the host name of the URL. For instance, a request to `peerprep.app` is routed to the frontend, whereas a request to `user.peerprep.app` is routed to the user service.

Further details and design decisions are noted under the “Cloud Deployment” subsection, under “Nice-to-have Implementations”. The other MongoDB and Redis databases were hosted in their own clusters, on MongoDB Atlas and Redis Cloud respectively.

4.2 Project Structure & Overall Development

While it is true that software architecture is a critical element of software engineering, we feel that the workflows surrounding the creation of the software product can be equally important at times. While they might not be directly related to our software architecture, the tools and workflows we incorporated into the project have played a crucial role in eventually allowing us to implement our desired architecture. As such, we have dedicated this section of our report to briefly elaborate the “logistics” of our project, to lay forth the context for the remaining parts of this report.

4.2.1 Development Process & Repository management

We opted for a **monolithic repository** to house all our microservices to simplify development and enhance collaboration across the team. By keeping all microservices in a single repository, we eliminated the overhead of managing multiple repositories, ensuring that dependencies, shared libraries, and configurations were consistent and accessible to everyone. This approach also streamlined setting up CI/CD pipelines and centralised project management, such as secrets management, write access and access rights, etc.

To maintain code quality and manage contributions effectively, we implemented **branch protection** and followed a structured **branching workflow**. Each major feature was developed in its own branch, and individual team members branched off from these feature branches for their contributions. This allowed parallel development without risking instability in the main branch and made it easier to integrate changes into the feature branch after reviews. This strategy ensured that contributions were cohesive, reduced merge conflicts, and enabled the smooth integration of contributions while upholding team consensus standards for quality and consistency.

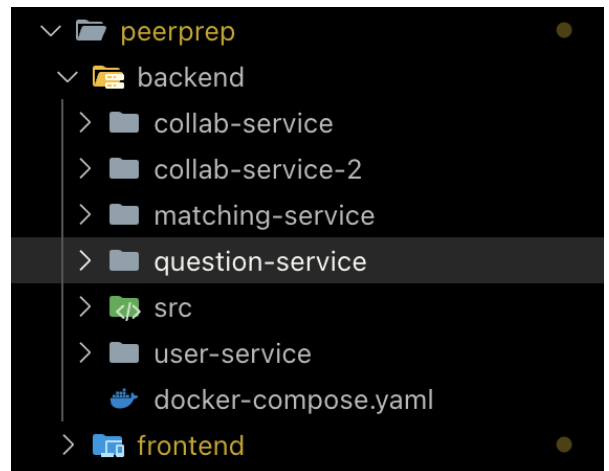
Additionally, not using a forking workflow allowed us to more easily pull the branches of team members who required assistance with particular features. Given this, as an additional layer of protection to our codebase, we cloned our **main** branch into a default **dev** branch that was used as the base for all our changes. All our changes would be merged into this **dev** branch, and when it was time for us to submit a milestone, we open a Pull Request from the **dev** branch to the **main** branch. This allowed us to continue working on the next milestone through the **dev** branch whilst having a functional main branch that can serve as a “checkpoint” for our progress, essentially a compromise between a pure branching workflow and a pure forking workflow.

4.2.2 Project Organization

From the outset, we recognised that the file structure and organisation of our project would have a significant impact on our workflows and overall cohesiveness as a team, especially with regard to conflict, given that we decided against using git submodules.

Backend Directories

As can be seen in the image on the right, we have split each microservice into a discrete directory. Each directory represents a standalone microservice, containing all relevant information such as the `.gitignore` file, the `dockerfile` and service-specific secrets.



As you might be able to tell, at one point we had to refactor our Collaboration microservice. It was at this point that we felt the benefits of using a microservices architecture, as all it took was for us to “swap out” the old collab service with the new one. Eventually, we managed to fix the original Collaboration microservice, and we were able to “swap” it back into our project.

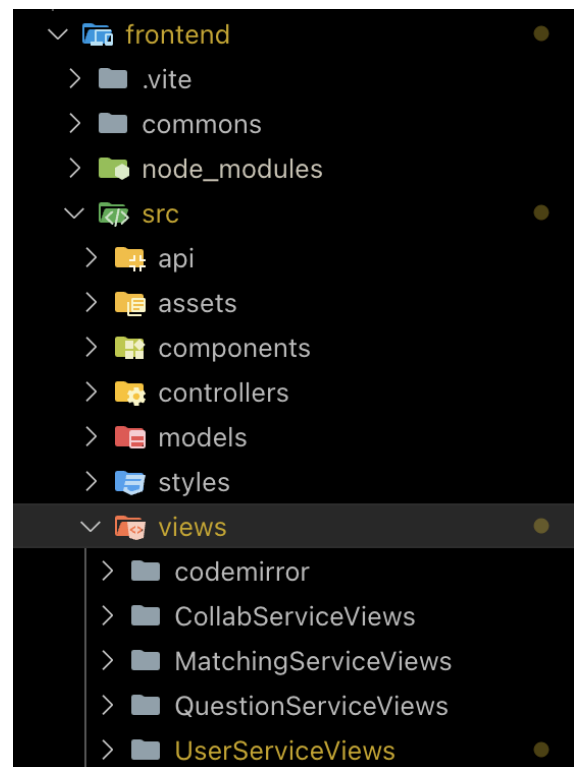
Frontend

Unlike the backend, we chose not to individually containerise the frontend as it made more sense to unify the views and simplify the development process.

By organising the **views** into a single centralised folder called frontend, we streamlined the development and ensured consistent UI components across the application, by way of standardised CSS styling.

Meanwhile, the **models** and **controllers** were placed in their respective service folders in the backend, keeping the data management and business logic distinct from the presentation layer.

Each microservice is assigned its own unique **Views** directory within which the views pertaining to that service are stored. The `api` directory contains the



files that allow for programmatic communication between the Frontend and the relevant backend service.

4.2.3 Project Inter-Component Interactions

Inter-Service Communication

Naturally, we would need microservices to be able to communicate with each other. For example, the matching service has to find a random question from the question service upon a successful match.

In order to facilitate inter-service communication, we define a network like **my-network** (shown in the docker-compose.yaml file on the right), it creates an isolated network environment for the services within that Docker Compose configuration. Services within this network can communicate using service names (e.g., user-service, question-service) as hostnames. The **bridge** driver is the default for single-host networks, and it allows containers to communicate with each other through a virtual network on the same Docker host.

Additionally, we modified our core server files where applicable to allow for Cross-Origin Resource Sharing (CORS) such that certain services are able to communicate with certain others in a safe and predictable manner.

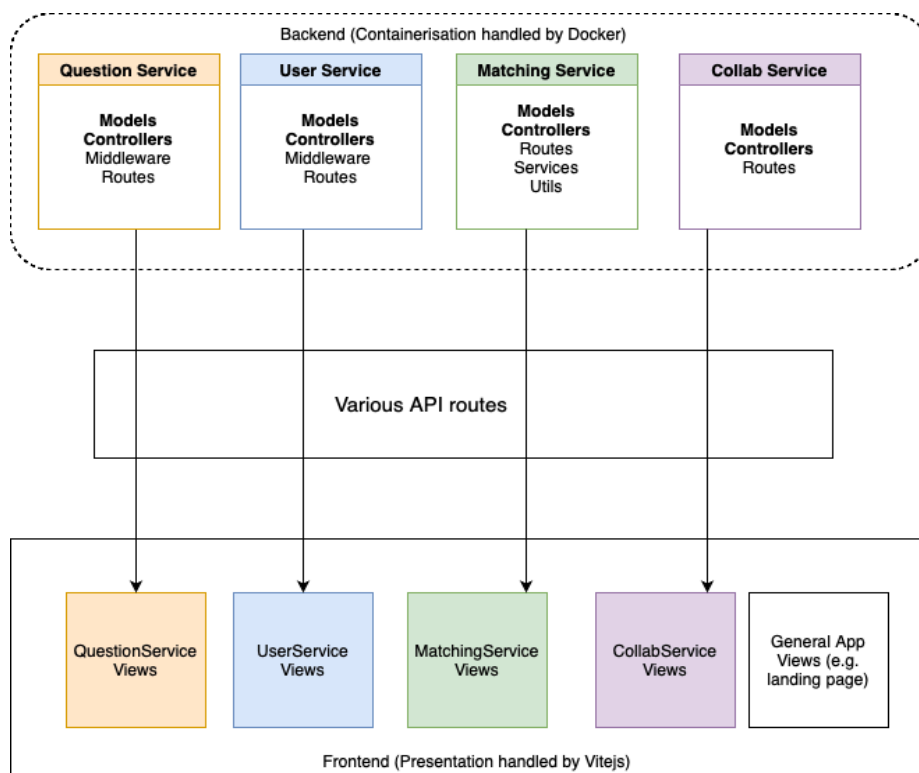
```

1 services:
2   user-service:
3     build:
4       context: ./user-service
5       dockerfile: Dockerfile
6     ports:
7       - "3001:3001"
8     env_file:
9       - ./user-service/.env
10
11   question-service:
12     build:
13       context: ./question-service
14       dockerfile: Dockerfile
15     ports:
16       - "8080:8080"
17     env_file:
18       - ./question-service/.env
19
20   matching-service:
21     build:
22       context: ./matching-service
23       dockerfile: Dockerfile
24     ports:
25       - "3000:3000"
26     env_file:
27       - ./matching-service/.env
28
29   collab-service:
30     build:
31       context: ./collab-service
32       dockerfile: Dockerfile
33     ports:
34       - "1234:1234"
35       - "3003:3003"
36     env_file:
37       - ./collab-service/.env
38
39 networks:
40   my-network:
41     driver: bridge
42

```

Frontend-Backend Interaction

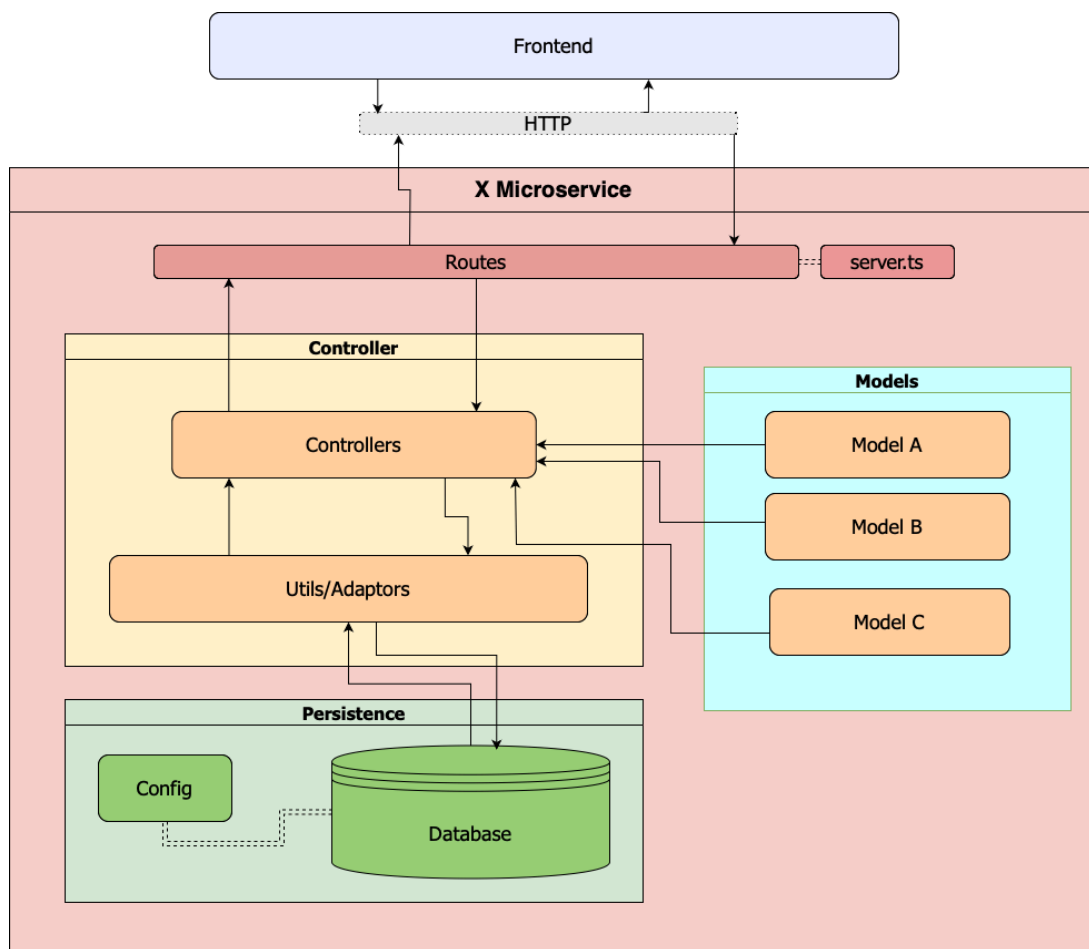
The diagram below shows an overview of how the various backend services (each containing minimally Models and Controllers) interact with their respective Views through various API routes.



4.2.4 MVC Architecture

We chose to model our folder structure within each microservice to emulate the Model-View-Controller (MVC) architecture for all our microservices to maintain a clear separation of concerns and ensure scalability. However (as will be explained later in the Question Service microservice design explanation below in this report), it is not always possible nor feasible to implement a pure MVC architecture, especially one that is a subset of an overarching microservices architecture.

The diagram below is a simplified depiction of the interaction between various components within a generic microservice that (mostly) conforms to the MVC architecture, including how the models, controllers, and the views (Frontend) interact.



For some microservices, additional “Adaptors” are created to abstract away the interaction between the controller and the database. Additionally, some microservices can omit the use of an external database and simply use local storage for persistence (for e.g. sessions in Collab Service). Routes are added to facilitate interaction where necessary (for e.g. Backend MC to Frontend V).

4.3 Choice of databases

Database-Server-Per-Service Pattern

To achieve loose coupling across services, we adopted a database-server-per-service pattern. This approach allows each service to operate independently with the database allocated to it, promoting modularity and thus, reducing inter-service dependencies, allowing for both future scalability and better maintainability.

However, there is one exception to this, which is the collaborative microservice. We felt that matching and collaborative microservices were highly coupled. Given our rather simple implementation of collab service and the lack of need of a persistent data storage for the collab service, we made the decision to keep track of collaborative sessions on the same Redis database used in the matching service as both these services were highly coupled (where a user can only exclusively either be in a matching queue or a session). Hence, to ensure data integrity across these 2 microservices, we opted to not follow the convention for this service, especially since the data in these 2 services need not be persisted, and in-built persistence mechanisms were sufficient for our functional and nonfunctional requirements.

Ensuring Data Independence

To maintain data independence and separation of concerns, we configured separate .env files for each service. This setup allows each service to define specific configurations, access controls, and database settings, ensuring that data handling is secure, independent, and tailored to each service's requirements.

4.3.1 Question Service Databases

MongoDB vs Alternatives		
Features	MongoDB	Other Databases (e.g., MySQL, PostgreSQL, Cassandra)
Data Storage	Document-oriented NoSQL database with dynamic schemas and flexible data structures.	MySQL and PostgreSQL use relational models, which require predefined schemas, limiting flexibility. Cassandra uses wide-column storage, which is good for scalability but lacks flexibility in schema design.
Scalability	Horizontal scaling through sharding allows MongoDB to handle very large datasets and high throughput.	MySQL and PostgreSQL require vertical scaling or complex sharding configurations, while Cassandra scales well but lacks the flexibility MongoDB offers in terms of schema.

Query Flexibility	Rich querying support for a variety of complex queries, such as range queries, text search, and aggregation.	MySQL and PostgreSQL offer powerful querying but with stricter schema constraints. Cassandra focuses more on high-volume writes but lacks the query flexibility MongoDB offers.
Indexing	Highly efficient indexing: supports single-field, compound, multikey, geospatial, text	MySQL and PostgreSQL have traditional indexing; optimised for relational constraints. While Cassandra has limited indexing, mainly focuses on primary key for fast writes
Transactions	ACID transactions for ensuring data consistency across multiple documents or collections.	PostgreSQL and MySQL also support ACID transactions but require a relational schema. Cassandra offers limited support for transactions.
Ease of Integration with Node.js	Official Node.js driver and Mongoose ORM streamline integration with Node.js applications.	MySQL, PostgreSQL, and Cassandra can be integrated with Node.js but require additional libraries and setup. MongoDB's integration is more seamless.
High Availability and Reliability	Built-in replication and automatic failover ensure redundancy and reliability.	PostgreSQL and MySQL require additional configuration for replication, while Cassandra offers high availability but with complexity in setup.
Security	Robust built-in security features (encryption, authentication, and auditing) protect sensitive data.	MySQL and PostgreSQL provide security but may need additional configuration for certain compliance standards. Cassandra offers security features but is more complex to manage.

Owing to the benefits stated above, we opted to use MongoDB as our database for Question Service.

- **Flexibility and Schema Design:** MongoDB's flexible schema meets our NFRs, which don't require SQL, and accommodates quick schema modifications, which is ideal given our limited development time.

- **Node.js Integration:** MongoDB's seamless integration with Node.js, including official drivers and the Mongoose ORM, streamlines our development workflow, reducing setup complexity.
- **Scalability and Cloud Deployment:** MongoDB supports horizontal scaling from the outset, aligning with our plans to scale effectively. Furthermore, MongoDB Atlas offers a managed cloud solution that ensures high availability, fault tolerance, and better uptime than we might achieve with our limited cloud resources for hosting.

4.3.2 Matching Service Databases

Redis vs RabbitMQ		
Features	Redis	RabbitMQ
Data Storage	Operates as an in-memory database (with optional disk persistence for specific use cases), making it fast for both reads and writes. This offers the lowest latency for databases.	Primarily uses disk-based storage, with options for memory caching. Slower than Redis but persistent, thus offering fault tolerant and managed clustering solutions.
Performance	Ideal for real-time (sub-millisecond latency) applications needing rapid response times due to its in-memory nature, at up to 200 million operations per second.	Handles more complex queueing well but may add latency due to persistence.
Reliability	Provides solutions like multi-instance replication and cluster support, but less robust for persistent storage.	Provides strong message delivery guarantees, with retries, acknowledgments, or DLQs (Dead Letter Queues). RabbitMQ promises reliability with such functionality.
Flexibility	Supports data structures (hashes, sorted sets) alongside Pub/Sub and basic queues for easy customizability to use case.	Provides advanced messaging features like exchanges, routing, and multi-protocol support inbuilt, which may pose a steep learning curve.
Queueing Capabilities	Sorted queues with priority and atomic transactions using sorted sets (ZSET), but no advanced queue management features.	Allows implementation of complex queueing algorithms, including priority and delayed queues, built-in.
Event-Driven Notifications	Not inherently event-driven, making synchronisation simpler but with fewer notification options	Naturally supports event-driven workflows, allowing more flexible synchronisation and notification options

Scalability	Scales well with straightforward queueing, given lower memory and processing overhead	Scales well for complex queueing tasks but may require additional infrastructure as queues grow
Native Cloud Solutions	Provides scalable, resilient (with 99.999% uptime SLA) cloud solutions that capitalises on low latency operations. Provides an easy configuration.	Reliable cloud options (AWS MQ, Tanzu RabbitMQ) but more complex to configure.
Ease of set-up	Straightforward setup with minimal configuration for real-time and basic queueing needs. Cloud set-up is straightforward and flexible.	Requires careful setup, especially for reliability and scalability features. Cloud set up is relatively manual and requires some configuration.

After considering the merits of each database service, we decided to go forward with Redis, owing to the reasons shown below.

Choice of Database for Matching Service: Redis	
High-Speed In-Memory Operations	Redis operates as an in-memory database, allowing it to perform both read and write operations with low latency. Moreover, Redis offers an API for atomic operations with Multi and Exec commands, that allow transactions in the database that ensure data integrity and avoid synchronisation issues. This is important in a matching service where users are actively waiting for a match, as even slight delays can impact the user experience
Short-Lived Data Handling	The matching service requires only temporary data storage, holding user information in queues for a maximum of 30 seconds. Redis's in-memory nature suits this need as it's designed for fast data storage rather than long-term persistence. Even in an event of full data loss (unlikely with the cloud service provided, as it boasts a 99.999% uptime as mentioned), users will just have to send in a matching request again after they are immediately notified of their lost match request.
Responsiveness	Since responsiveness is crucial, Redis's high-speed data handling allows the system to manage queues and respond quickly, reducing the waiting time and improving the overall user experience. With this responsiveness, we were able to achieve our functional requirement to allow users to delete a matching request and then create a new one, which was a feature very well implemented due to its responsiveness with the real-time database.
Easy set-up	All things considered, while RabbitMQ offers abstracted functionality that was useful in our implementation, the ease of set up was a major deciding factor for us, especially with respect to the cloud setup. However, manual

	functions were created to implement queuing mechanisms, but were actually found to be robust and easy to use after good folder structuring, designing and implementing of our algorithm.
Scalability	RabbitMQ does provide more intuitive and straightforward scaling (horizontal scaling) as taught to us in lecture, Redis also allows scaling with PUB/SUB functionality. Overall, we found Redis cloud set up to be easy to use and scalable if necessary, making it an easy, reliable and robust choice compared to RabbitMQ, considering the short development time in our hands. Redis was also able to handle our Non-Functional requirements with respect to concurrent matches and user requests, thus was the chosen database for our microservice.

4.4 Collaboration Service Mechanism

In designing a collaborative, real-time code editor, we evaluated multiple tools and frameworks to support efficient code editing, synchronisation, and communication. The choice of components was guided by key criteria, including performance, scalability, privacy, and ease of integration.

Our evaluation involved comparing various code editors and synchronisation mechanisms.

4.4.1 Collaboration Service Code Editors

Code Editors Comparison		
Editor	Advantages	Disadvantages
CodeMirror	Lightweight and fast, suitable for most web applications Multi-language support with customisable syntax highlighting Advanced features like autocompletion, hints, and theme support Easy integration with React using react-codemirror2, plus active community and documentation	Limited advanced features compared to Monaco (e.g., lacks IntelliSense natively) Customisation can require more manual setup than other editors
Monaco	Comprehensive feature set which includes IntelliSense, error detection, and advanced syntax highlighting High compatibility with modern	Heavy and resource-intensive, leading to slower load times Compatibility issues on older browsers

	languages and advanced language services Has a react-friendly wrapper for easier integration and state management in React applications	
ProseMirror	Highly customisable for structured document editing CRDT-based collaborative editing support Strong plugin support	Lacks native syntax highlighting or linting Requires significant setup to adapt for code editing

In implementing the Collaboration Service, we encountered a unique challenge that we hadn't faced in previous microservices. The main issue was that the code editor in this service was tightly coupled with the synchronisation mechanism, which added complexity to the decision-making process. Unlike standard CRUD or stateless microservices, where components interact more independently, here, the editor and synchronisation mechanism were interdependent, making it necessary to compare both elements together rather than separately.

4.4.2 Collaboration Service Synchronisation Means

WebSockets vs P2P		
Features	WebSockets	P2P
Scalability	Highly scalable with centralised server infrastructure	Scalability can be challenging as each client must handle direct connections with multiple peers
Privacy	Does not expose user IP addresses to other clients, enhancing privacy and minimising security risks	Exposes client IP addresses to other peers, which can raise privacy and security concerns
Latency	May introduce slight latency due to the use of a server	Direct communication between peers can reduce latency in small networks but suffers more when scaled up
Traffic Management	Centralised control over traffic flow and message rates	Decentralised, where each client manages its own traffic, which can lead to unpredictable performance in high-traffic scenarios

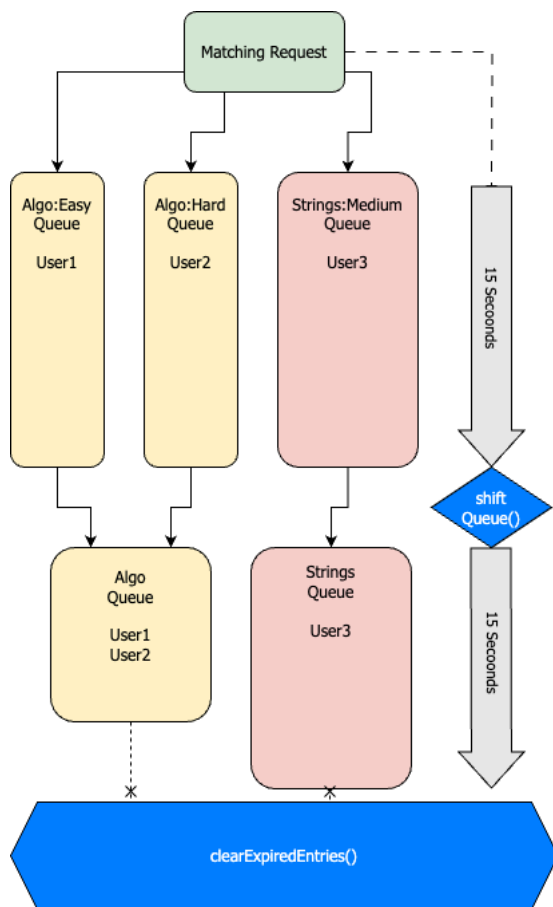
After considering the merits of each code editor and synchronisation options, we decided to go forward with the CodeMirror (specifically react-codemirror2), owing to the reasons shown below.

Choice and Mechanism Used for Collaboration	
Code Editing: CodeMirror (React integration using react-codemirror2)	
React-Friendly Integration	react-codemirror2 provides a wrapper for CodeMirror, making integration with React apps straightforward and efficient
Multi-Language Support	CodeMirror supports syntax highlighting for multiple languages, which is essential for collaborative coding across different languages
Enhanced User Experience	CodeMirror allows for advanced features like autocompletion, hints, and theme customisation, creating a more intuitive and helpful coding environment
Less overhead	Less overhead than Monaco, making it more suitable to meet NFRs of responsiveness
Real-Time Synchronisation: Yjs, CodemirrorBinding, WebSockets (y-websocket)	
Efficient Real-Time Data Syncing	Yjs uses Conflict-Free Replicated Data Types (CRDTs) to handle real-time data synchronisation without conflict issues, making it ideal for collaborative editing
Direct Linkage to Editor	CodemirrorBinding directly binds Yjs data structures to CodeMirror, ensuring that all edits appear instantly for each participant, providing a seamless collaborative experience
Community and Stability	<p>WebSockets can manage traffic flow, control message rates, and handle a high volume of users without overwhelming individual clients</p> <p>WebSockets offer better infrastructure for adding features like authentication, traffic management, and message filtering, which could have been more challenging in a P2P setup</p>

Unlike P2P connections, WebSockets do not expose the user's IP address to other clients, enhancing privacy and minimising security risks by keeping network details hidden
--

4.5 Matching Algorithm

Our matching algorithm is designed to be compatible with redis, as it is our database and implementation decision. Redis offers sorted sets that we will use to queue users. This allows users that have joined the queue earlier to be prioritised (as we sort by the relevant timestamp upon creating the match request). It also offers hashsets, which we will use to keep track of users in all queues (for easy and fast querying of user status) and session data.



ALGORITHM

As shown on the diagram, our algorithm prioritises matching by topic & difficulty by adding users to a topic:difficulty queue for the first 15 seconds. We also add users to a user hashset to keep track of their presence in a queue.

We implemented a `shiftQueue` function in `redisUtils`, such that it checks if entries in each of these queues have elapsed 15 seconds, and shifts them to a more generic topic-only queue.

Finally, we implemented a `clearExpiredEntries` function that clears the topic only queues after entries there have existed for 15 seconds or more. We also remove entries from the user hashset, indicating that they are no longer in the queue.

In the diagram, we see that users 1 and 2 spend 15 minutes in their respective specific queues, then join the general algorithm queue, where they will get matched. User 3 however is not matched after 30s.

Note that functions `shiftQueue`, `clearExpiredEntries`, `dequeueUser`, `deleteSession`, `enqueueUser` and others are atomic, ensuring data integrity in the database, so that abstracted data structures (like the hashsets to keep track of users' queueing status) are perfectly synchronised with the actual data structures in the database.

Our implementation also means that users who send in a matching request will not find a match just by topic in the first 15 seconds. While this may be argued to be a flawed approach,

we found this to be aligned with our functional requirements, where we wanted to design a way such that users are matched by both topic and difficulty ideally, and then as a backup, matched by just topic. This means that in the same scenario, if user2's request was sent in 15 seconds later than user1, they would not be matched. We find this reasonable as we cannot compromise on finding an ideal match for user2 just so that user1 is matched within the 30 seconds.

5. Nice-to-have Implementations

5.1 N1 – Communication

To implement the real-time communication enhancement feature for seamless interactions among users within the same session or room, I selected **Socket.IO** as the core technology for enabling real-time, bidirectional communication. This decision was guided by several factors, including Socket.IO's robust performance, reliability, and compatibility with a wide range of browsers, which ensured an efficient user experience compared to other available technologies.

5.1.1 Why Socket IO?

Aspect	Socket.IO	Other Competitors (e.g., WebSockets, Pusher, Firebase)
Real-Time Communication	Provides real-time, bidirectional communication with low latency and persistent connections.	WebSockets offer similar features but lack automatic fallback mechanisms. Pusher and Firebase require additional integrations for fallback.
Fallback Mechanism	Automatic fallback from WebSocket to HTTP long-polling if WebSocket is unavailable.	WebSockets lack this feature, and in the case of Pusher or Firebase, fallback can be complex and manual.
Room Management	Built-in room management to group users for private messaging and broadcasting.	While WebSockets support rooms, handling them requires additional code, while Pusher/Firebase offer limited room functionality.
Cross-Browser Compatibility	Handles browser-specific issues seamlessly for consistent user experience.	WebSockets might face issues with older browsers. Pusher and Firebase generally ensure cross-browser compatibility but can add extra overhead.

Event-Driven Architecture	Simple event-driven system with connect, disconnect, and custom events like messages.	WebSockets provide basic event handling but often require more manual setup for scalability. Pusher/Firebase have more predefined events but less flexibility for custom events.
Scalability	Scalable with built-in support for clustering and horizontal scaling.	While WebSockets can be scaled, Socket.IO's built-in clustering and scaling methods are more robust compared to Firebase and Pusher.
Ease of Use	Easy-to-implement with a simple API and minimal configuration for both client and server.	Firebase and Pusher require additional setup for more complex features, whereas WebSockets can be more complex to implement.

5.1.2 N1 Communication Implementation

Implementation Overview	
Component	Description
Server-Side Setup	server.js manages connections, assigns users to rooms, and routes messages. Events like connection, message, and disconnect handle user sessions securely.
Client-Side Setup	Chat.tsx establishes client-server connections, assigns users to specific rooms, and enables message reception. The client displays messages dynamically in the interface.
UI Customization	Sender and receiver messages are colour-coded (handled in Chat.tsx), allowing users to distinguish their messages visually, improving readability and engagement within the chat.

User Management and Message Handling	
Aspect	Description
User Connections	The server assigns each user to a room using their session ID via Socket.IO's join method, ensuring room-specific communication and privacy.
Message Broadcasting	Uses Socket.IO's method to emit messages only to users in the same room, maintaining relevance and privacy in communication.
Disconnections	When a user disconnects, Socket.IO removes them from the room, efficiently managing server resources and preventing memory leaks.

5.2 N3 – Code Execution

Initially, the code execution service was only created with JavaScript code execution in mind. The initial execution involved sending code from the frontend, from `CollabServiceIntegratedView`, to a server on the backend where it was executed and sent back to the frontend. However, upon implementation of the code syntax service, which allows you to change languages, we started considering alternative code execution methods.

Judge0 vs Local Backend Execution		
Feature	Judge0 (via RapidAPI)	Local Backend Execution
Language Support	Supports a wide variety of programming languages, making it flexible for multi-language collaborative environments	Limited by the languages installed on the backend server, often requiring complex setup and maintenance for multi-language support
Scalability	Offloads execution to an external service, reducing load on the backend server and supporting high-concurrency execution	Executes on the backend, which may limit scalability and increase load on the server, especially with many concurrent users
Execution Time	May have variable response times depending on external service load, although it generally supports asynchronous execution well	Generally faster for small scale, as it avoids external service latency, but less efficient with multiple simultaneous executions

Judge0 was chosen over local backend execution to enable multi-language support, reduce backend load, and enhance security, despite the need for polling and potential external service costs. As such, we made the following changes:

Aspect	Implementation Details
Execution Workflow	<p>Code to be executed is encoded in base64 and sent as a POST request to Judge0 for execution</p> <p>The service polls for results every second, checking until execution is complete</p> <p>After receiving the result, it decodes the output from base64 if necessary</p>
Test Case Table Enhancement	<p>A test case table was added to the collaboration service page, allowing users to compare expected results with actual output</p> <p>Test cases are fetched from MongoDB.</p>

5.3 N4 – Enhanced Collaboration Service

The Enhanced Collaboration Service provides a multi-language, real-time code editing environment. By incorporating multi-language syntax highlighting, intelligent code suggestions, and various UI improvements, we have created an intuitive experience for collaborative coding.

Our early decision to use CodeMirror (i.e. react-codemirror) turned out to be really useful as it came with certain builtin functionality that allowed us to easily add the enhancements.

5.3.1 Multi-Language Syntax Highlighting

Implementation and Mechanism

CodeMirror enables language-specific syntax highlighting through language “modes,” each of which parses the syntax rules for different languages. By enabling support for eight languages—C, C++, C#, Java, JavaScript, Python, Ruby, and Kotlin—we ensure that developers working in various languages can see syntax in a familiar format, reducing the cognitive load when reading or editing code.

How Syntax Highlighting Works in CodeMirror

Under the hood, CodeMirror uses **tokenizers** within each language mode to parse the input code. Tokenizers break the code down into individual “tokens,” each representing a keyword, variable, operator, or other syntax element. Each token is then associated with a specific CSS class based on its role in the language, and CodeMirror applies the corresponding styling. The modularity of CodeMirror’s design allows us to dynamically change the language mode as users switch languages, without reloading the editor or losing the current session context.

5.3.2 Multi-Language Code Completion Support

Intelligent Code Suggestions with Inline Completion

To enhance the user experience, we implemented **IntelliSense-like code completion** using CodeMirror’s hinting API. Inline code completion provides contextual suggestions as users type, helping them write code more efficiently and with fewer syntax errors. This feature works by detecting specific characters and keywords within the language mode, then querying a predefined list of valid tokens to generate a dropdown of possible suggestions.

How Code Suggestions Work in CodeMirror

The CodeMirror hinting system listens for specific keystrokes or patterns, which for our use case we defined to be all characters (except new space). While it could have been constrained to a specific key binding such as Ctrl+Space to invoke, we decided that automatic suggestions would be the least interruptive to the coding experience.

When triggered, CodeMirror queries a set of completion functions for the active language, which returns a list of possible completions based on the cursor's context. These suggestions are rendered in a dropdown beneath the cursor, allowing users to select and autocomplete. For this, we leveraged CodeMirror's **hint options** in combination with language-specific dictionaries, creating an "IntelliSense" effect that adapts based on the active language mode.

5.3.3 Collaboration UI and UX Enhancements

Collaborative Undo Functionality

In a multi-user environment, undo functionality must be synchronised across all participants. Traditional single-user undo stacks could not address this requirement effectively, so we implemented a **collaborative undo** feature using the **Yjs library**. Yjs, based on **CRDTs (Conflict-Free Replicated Data Types)**, allows for synchronised document changes across distributed clients. By binding Yjs's CRDT structure directly to CodeMirror via **CodemirrorBinding**, each undo action is automatically propagated to all participants, maintaining a consistent shared state.

Editor UX Improvements

To create an authentic coding experience, we added **monospaced fonts** and **line numbers**, emulating traditional code editors like Visual Studio Code. This enhancement improves readability and user familiarity, as monospaced fonts align characters vertically, and line numbers help users quickly navigate and reference code.

Real-Time Language Switching/Unified Language Selector

To support multi-language environments, we added a **real-time language switcher**. This feature updates both the syntax highlighting and code completion based on the selected language. Internally, it dynamically reloads CodeMirror's mode and hinting configurations without requiring a page reload or session reset, allowing users to fluidly switch languages during a collaborative session. This real-time adaptation is essential for cross-language collaboration and ensures that users do not have to exit their current workflow to switch languages.

To allow the user to easily access this functionality, we designed a **unified language selector** that allows users to choose both the syntax and execution language simultaneously. This selector provides a streamlined interface, ensuring that users do not have to toggle between multiple dropdowns. Under the hood, it triggers both the syntax mode update in CodeMirror and the backend configuration for code execution, maintaining consistency between what users see and what executes on the server with judge0.

5.4 N6 – Incorporation of Generative AI

In approaching this nice to have, I wanted something tailor-made to our use case, which is a collaborative code editing platform that is to help students prepare for technical software interviews. Upon brainstorming, I found that generative AI, in particular, Large Language Models (LLMs) is typically used in the following ways.

1. To read code and the given question, and reply as instructed by the user or by a predefined instructional prompt, usually to assess the correctness and effectiveness of the code with its proposed solution.
2. To autocomplete code in a predictive manner to avoid repetitive and straightforward tasks such as explicit syntax typing, writing comments or writing repetitive code (like console log/print statements). Often it tends to try to aid the user by providing ambitious and long suggestions.

Here is a detailed comparison that I did to finalise my chosen implementation.

Feature	Pre-prompted LLM agent	In-Editor Code Autocompletion (e.g. CoPilot)	AI Chatbots
Purpose	Guides users holistically by evaluating code on style, correctness, and complexity, promoting structured feedback and learning.	Focuses on assisting users with quick code suggestions, syntax fixes, and boilerplate generation.	Provides conversational assistance, answering coding questions and debugging issues interactively.
Flexibility	Limited to predefined prompts and evaluations, where users cannot modify instructions, ensuring consistent guidance.	Highly flexible, allowing users to freely write and modify code with intelligent suggestions.	Flexible as users can ask open-ended questions and explore various problem-solving approaches.
Customisability	Minimal customizability to ensure the AI aligns with the educational goals of holistic	Highly customizable; users control the editor environment and can adapt suggestions based	Highly customizable; users can ask detailed, specific questions or tweak their interaction with

	learning and structured supervision.	on preferences.	the chatbot.
Focus on Learning	Promotes user learning by assessing their code comprehensively and nudging them toward self-improvement rather than direct solutions.	Limited learning focus; assists with code completion but does not evaluate or provide comprehensive feedback.	Promotes short-term learning by answering direct questions but does not ensure users understand holistically.
Ease of Use	Simplified workflow with Codemirror binding, single-click execution, and structured prompts for user guidance.	Seamless in-editor suggestions that blend naturally with the coding process, reducing interruptions.	Easy to interact with, though requires users to frame specific queries for effective assistance.
Educational Impact	Strong focus on guiding users toward independent problem-solving by providing feedback instead of direct solutions.	Minimal educational impact. In fact, counter-intuitive as it automates repetitive tasks but does not guide users on improving their coding style.	Moderate educational impact; answers promote understanding but may encourage dependency on the chatbot.

Upon the comparison above, it is clear that in editor autocompletion is extremely counter intuitive in helping a user learn to code in almost all aspects. Not only does it provide syntax, but often divulge solutions to the user, or distract the thought process of the user with a suggestion, as users will try to decipher the suggestion intuitively.

Moreover, an AI chatbot was not a very unique implementation. While it provides great flexibility and can be tailor made to respond to the user as they please, this again makes it counter-intuitive, especially given how predictive and accurate LLMs are today. With an easy to reach tool (as users can replicate this with another LLM chat-app), it will further tempt users to seek for easy help, instead of collaborating with their matched peer and bouncing ideas off each other.

A pre-prompted agent is thus what I decided on. Given the shortcomings of the previous proposed implementations, I actively combat them in my implementation, by prioritising ease of use, holistic learning and structured, reliable feedback.

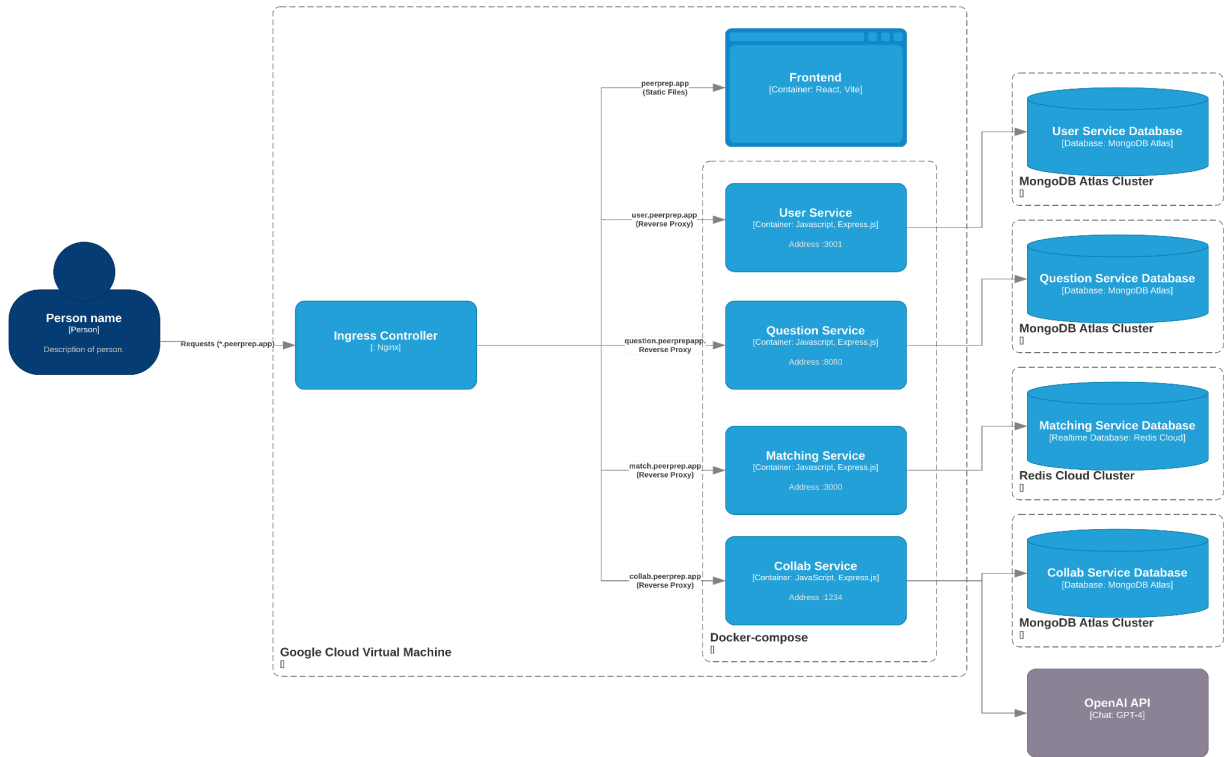
Component	Description
Codemirror Binding for Input	Enables direct input of code from the editor, eliminating the need for manual copy-pasting for a seamless, hassle-free process for users. It also ensures users attempt the question before they can seek for assistance, by being forced to input code such that they can access it, incentivising the coding process.
Question and Language Input	Allows users to input contextual information such as the coding question and programming language. This ensures the LLM has the necessary context to provide accurate assistance.
Single Button Execution	Simplifies functionality with a single-click process, delegating all logic to the backend. Maintains usability while abstracting complex processes from the user.
Instructional Prompt	Centralises logic to evaluate code holistically. Protects question solution details while providing constructive feedback on <ul style="list-style-type: none">- Time complexity- Space complexity- Code style- Correctness and optimisation hints This is designed to act like a code assessor for an interview, where comments are rather general and less detailed, guiding the user to focus and think carefully.
Restricting User-Defined Prompts	Prevents users from inputting custom prompts, focusing interactions on learning rather than prompting at each roadblock. Guides users towards independent problem-solving. The agent thus seeks to supervise, and not micromanage, similar to how an interviewer would approach assessing code.
Reliability Enhancements	Incorporates few-shot prompting, clear instructional design, and low response temperature to increase LLM reliability. Minimises issues caused by conversational variability with a single API call that does refer to history / previous prompts for a smaller, more focussed context

	window.
--	---------

This simple yet thoughtful addition of generative AI is, I believe, a well suited feature for our project. As mentioned, it improves on the responses of LLMs with proper, thoughtful prompting and adding of contextual knowledge, and incentivises users to code so that they can effectively use this feature, making the coding and learning process an easier and more enjoyable one.

5.5. N7 – Cloud Deployment

As mentioned at the start of this report, given below is the deployment infrastructure for each service in our application. All frontend/backend code is deployed on a Google Cloud Platform (GCP) virtual machine (e2-medium).



Service	Deployment Infrastructure
DNS	GCP - Cloud DNS/Domains
SSL Certificates	Certbot, on GCP Virtual Machine. Served by Nginx reverse proxy
Frontend	Static files on GCP Virtual Machine (e2-medium) behind Nginx reverse proxy
User Service	Docker compose on GCP Virtual Machine (e2-medium) behind Nginx reverse proxy
Question Service	
Collaboration Service	
Matching Service	
MongoDB Database	MongoDB Atlas Clusters
Redis Database	Redis Cloud Clusters
GPT-4 API	N/A

For each database (MongoDB, Redis), we chose to use their native cloud solutions (Atlas, Redis Cloud) as it was the most straightforward, managed solution. For a product of our intended scale (based on our NFR pertaining to scalability), it does not make sense to say, have a self-managed redis or MongoDB cluster.

For the choice of cloud platform, Google Cloud was chosen due to the disbursement of cloud credits in the course. We used these cloud credits to register the domain `peerprep.app`, which we then used to register SSL certificates with Certbot.

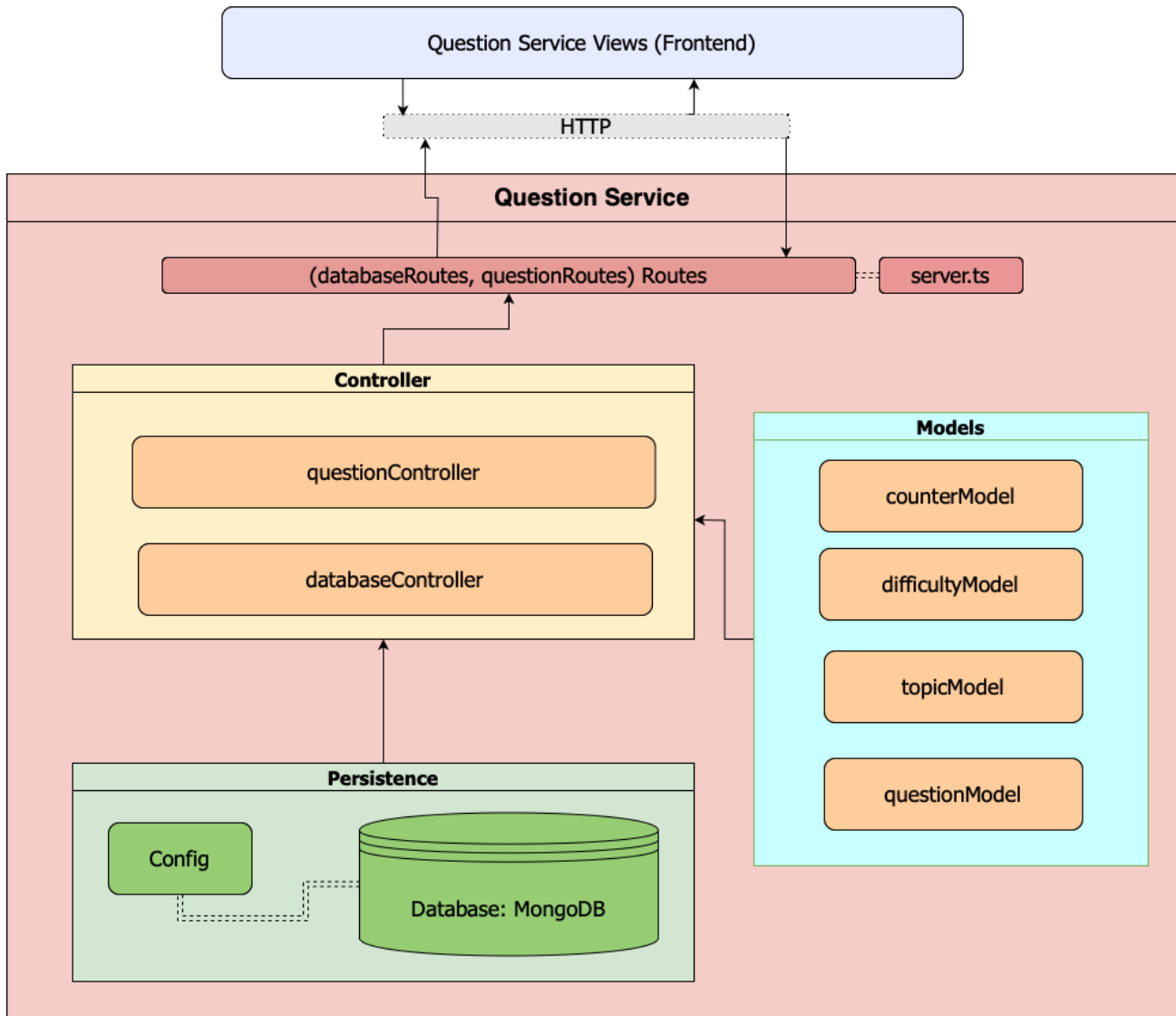
This is a preferable option to set up HTTPS via GCP with their managed certificates, since unlike AWS, we need to register load balancers and wait for GCP to automatically provision certificates, which may take up to hours, not to mention the error-prone nature of setting up the load balancers and HTTP proxies. For our deployment, the simpler option of using Certbot with Nginx verification makes much more sense.

Another option we considered for setting up the backend servers was to use the serverless platform Google Cloud Run - but similar to their managed certificates, GCP's serverless websockets only provide "best-effort", where websocket connections may be routed to other serverless instances, which requires more infrastructure to synchronise states with external storage. Since we could just host this on a stateful server, we deemed that the trade-off of state management was not worth it.

6. Internal Microservices Design

6.1 Question Service Internal Design

The Question Service follows a Model-View-Controller (MVC) structure with additional components for routing and frontend integration. The following sections describe each core component in detail. The diagram below is based on the general MVC diagram shown in the earlier part of the report.



6.1.1 Components

MODELS

These models define the data structure for entities within the Question Service, functioning as blueprints for data validation and database schema structuring. Below is a summary of the primary models used here.

Model	Description
counterModel.ts	Manages a counter to track question IDs, ensuring each question has a unique identifier.
difficultyModel.ts	Defines question difficulty levels (e.g., easy, medium, hard), standardising difficulty attributes.
questionModel.ts	Core model representing question attributes (title, description, categories, difficulty, etc.).
topicModel.ts	Represents topic categories for for each question

Here, we decided to model both the difficulty and topic even though they could have been implemented as enums and referenced within the question model itself. However, we realised that there are more use cases for these models as opposed to just being used in the context of a question.

For example, having a list of topics stored within the DB allows us to retrieve this list on the frontend where the user might interact with the topic, as opposed to having them either:

1. Hardcoded in the frontend OR
2. Retrieve all questions, and implement a method for finding unique topics

The first would be introducing unnecessary coupling to the system (between the frontend and backend), and the second might incur performance degradation for a large number of question stored, which would go against our NFRs.

CONTROLLERS

Controllers handle business logic, connecting models with the application by processing requests and manipulating data accordingly. Within the question service,

Controller	Purpose and Description
databaseController.ts	Manages interactions with the database pertaining to topicModels and difficultyModels. Deals with database operations of question “metadata” like topic and difficulty.

questionController.ts	Central controller for question handling. Provides functions such as validateQuestion, fetchQuestions, createQuestion, updateQuestion, and deleteQuestion to facilitate CRUD operations, ensuring data integrity and validation. Also contains the functionality to retrieve a question as specified and requested by the matching service via CORS
-----------------------	---

ROUTES

Routes define the API endpoints exposed by the Question Service. They map incoming HTTP requests to the corresponding controller functions.

Route	Description
databaseRoutes.ts	Manages question metadata specific routes, such as those dealing with topicModel or difficultyModel, as mentioned above.
questionRoutes.ts	Maps requests related to questions, enabling clients to perform CRUD operations via questionController.

VIEWS (React Components)

The frontend of the Question Service is developed in React, with specific components handling user interactions, data rendering, and input collection.

React Component	Description
QuestionManagement	Integrates QuestionForm and QuestionList components to provide an interface for creating, editing, and listing questions.
QuestionForm	Collects input from users for question creation or updates, triggering handleSubmit for processing the form data.
QuestionList	Displays a list of questions and provides options to edit, delete, or select questions.
QuestionController.tsx	The frontend equivalent, communicates with the questionController on the backend.

React also complicates the MVC pattern because it tends to combine the View and Controller. In React, the UI (View) and business logic (Controller) are often mixed within a single component, such as a React Function Component (React.FC). This structure, while powerful, makes it challenging to maintain a strict separation between the view and controller logic. We have managed to circumvent this by separating out views wherever possible, like the above, and thus abstracting functionality common to the Views within a frontend gateway “Controller” that communicates with the actual question controller on the backend.

6.1.2 Components Interaction

Frontend-Backend Communication

Frontend components interact with backend services through HTTP requests. The **QuestionController** serves as the intermediary, invoking functions in `questionApi.ts` which uses Axios for API calls. Data validation, error handling, and state management. Key functionalities here are:

- **Data Validation:** The `validateQuestion` function in QuestionController (Frontend) ensures only valid data is submitted.
- **Error Handling:** Centralised in `questionApi.ts`, preventing cascading issues across components.
- **State Management:** The QuestionManagement component manages state, including active questions and those under edit, displaying real-time data, integrating QuestionForm and QuestionList into a singular, unified view.

6.1.3 Evaluation of the Architecture (Pros & Cons)

Advantages	Description
Separation of Concerns	The MVC structure ensures clear separation between data, business logic, and presentation, improving maintainability and readability.
Modularity	Each component (models, controllers, routes) has a defined responsibility, facilitating testing, updates, and scaling.
Scalability	The microservice architecture allows for scaling services independently according to demand, enhancing overall performance.
Centralised Error Handling	Errors are managed in a single location (<code>questionApi.ts</code>), improving reliability and simplifying debugging processes.

Challenges	Description
Increased Complexity	Microservices and MVC add structural layers, making initial setup and debugging more challenging, especially for inter-service communication.
Data Consistency Issues	Since each microservice manages its data, ensuring consistency across services can be challenging, particularly with interconnected data.
Increased Latency	The multiple layers (frontend, controllers, models, routes) involved in each request may introduce latency, impacting performance.

6.1.4 Deviation from MVC

In our implementation of the question service (and some other microservices where routing is required), we've adapted the vanilla MVC (Model-View-Controller) architecture to incorporate elements of VIPER (View-Interactor-Presenter-Entity-Router). This hybrid approach allows us to address the unique requirements of our microservices architecture and React's component-based structure.

Deviation from Vanilla MVC

In a traditional MVC setup, the Model represents data, the View displays the UI, and the Controller handles the logic between them. MVC works well for standalone, full-stack applications, where each component of the architecture can work in close coordination. However, in a microservices architecture, each service is specialised and operates independently, often relying on **inter-service communication** to complete tasks.

And as mentioned earlier, React also complicates the MVC pattern because it tends to combine the View and Controller. In React, the UI (View) and business logic (Controller) are often mixed within a single component, such as a React Function Component (React.FC). This structure makes it challenging to maintain a strict separation between the view and controller logic. For example, a react view, (React.FC) tends to use the `useEffect` closure to invoke certain functionality upon loading of the page, when strictly speaking, this invocation should be carried out by the controller, and the controller should afterwards invoke the view with the results of the execution.

Incorporating VIPER Elements

To address these challenges, we've incorporated elements of VIPER, a more modular and decoupled architecture pattern, commonly used for mobile (iOS) application development. VIPER separates concerns into five distinct roles: View, Interactor, Presenter, Entity, and Router. Each VIPER component has a specific responsibility, making it easier to manage complex workflows, which is particularly applicable in our context.

While we have largely maintained MVC, we've incorporated the "Router" aspect of VIPER (mainly, and to a smaller extent, the "Presenter"), and we use Express as the "Router" when dealing with API calls, and React's `useNavigate` when we want to navigate between various Views of different microservices (i.e. Presenter)

In VIPER, the Router directs requests to appropriate parts of the application. In our setup, Express routes (like `questionRoutes.ts`) act as routers, directing HTTP requests to the appropriate controller functions.

- The Router serves as the entry point for client requests, providing a defined interface for each service within the larger architecture.

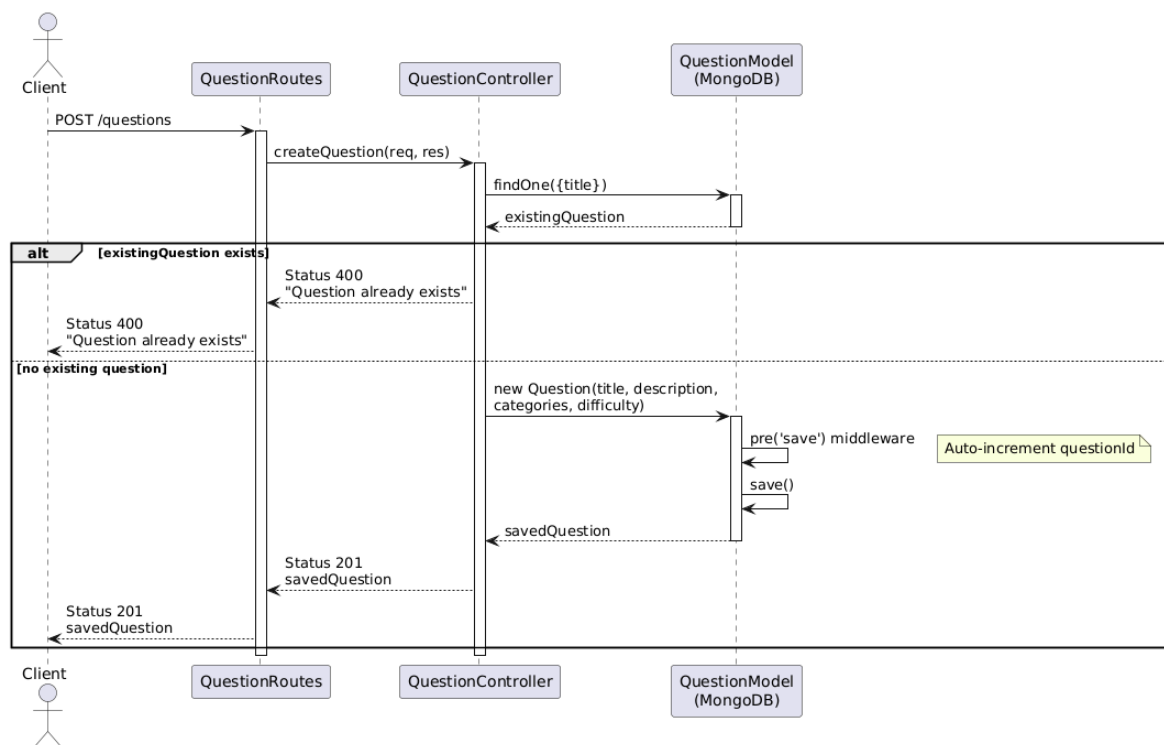
- This layer abstracts away the specifics of routing, so each service can independently define its API endpoints and communicate with other services.

Why We Deviate from Vanilla MVC

MVC is generally designed for full-stack applications where components are tightly coupled and co-located. In microservices architecture, each service is “isolated”, focusing on specific functionality and requiring explicit communication with other services, essentially the opposite where tight coupling is to be avoided. In MVC, communication between views is straightforwardly handled by the controller. However, that is not the case if the Models and Controllers exist separately from the View, together with heavy integration of persistence methods like Databases which require additional configuration, as is present in the context of our project.

Thus, the Frontend-Backend communication and sometimes even Backend-Backend becomes unavoidable. Our approach adapts MVC to microservices architecture by adding a “Router” to manage API endpoints, and a “Presenter” i.e. `useNavigate` in React to navigate between Views on the Frontend. This hybrid architecture leverages React’s component-driven approach (and extensive support) while incorporating aspects of the VIPER architecture where needed to compensate for the possible shortcomings of implementing MVC as a sub-architecture of an overarching microservices architecture.

6.1.5 Sequence Diagram of Question Service



Given below is a sequence diagram of an example flow of a client creating a new question.

6.2 Design of Matching Service

6.2.1 Components

The Matching Service follows (on an abstract level) a Model-View-Controller (MVC) structure with additional components for routing and frontend integration. The following sections describe each core component in detail.

1. Routes and server.ts

Server.ts acts as the entry point of the application, using apiRoutes.ts to specify the HTTP requests it listens to.

2. Controllers

Controllers are separated such that requestController.ts acts as the command controller related to matching requests, sessionController.ts acts as the command controller related to sessions. sseController.ts is a query controller, handling the consistent updating of queue and session statuses to the user using server-side events.

3. Services

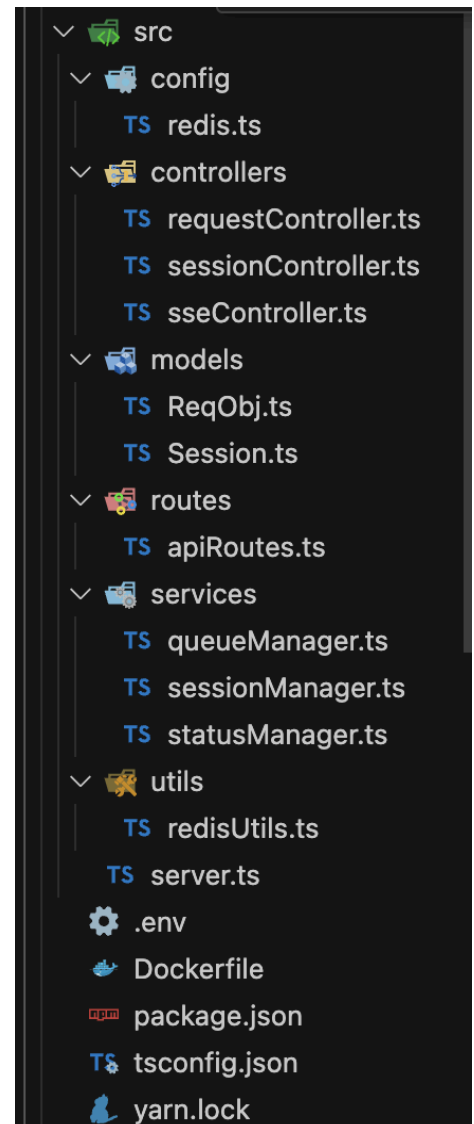
Services are similarly separated as controllers, acting as abstractions that interact with redisUtils.ts, holding the queueing logic and the matching algorithm, as well as other business logic.

4. Models

Models encapsulate the models used in this microservice, mainly a matching request, defined in ReqObj.ts and a session, defined in Session.ts. This provides an abstraction that can be used by various files and ensures type checking as we use typescript. When handling these objects. It is also exactly implemented on the frontend. This standardisation allows for easy integration with the frontend

5. Config

Holds the file redis.ts that handles connection with the database. It implements reconnection attempts when disconnected or when a connection error occurs.



6. Utils

This folder holds the redisUtils.ts file, responsible for all the direct interactions with the redis database. It holds an amount of business logic as well to accommodate atomic transactions in the database to ensure data integrity and synchronisation as stated in our functional and non-functional requirements.

As such, the core functionality lies in redisUtils.ts, while services form an abstraction of the methods in redisUtils.ts and handle the business logic of the application, where controllers use these methods defined in services and handle errors that are translated to api error to communicate effectively to the frontend. As such, we achieve a good separation of concerns, where a slight functional change or bug can be easily pinpointed and/or implemented/fixed.

6.2.2 Synchronous Communication

Feature	Async Communication	Sync Communication
Use Case	Best for real-time data processing , event-driven systems, and when decoupling components is essential.	Best for retrieving specific information or executing tasks that require immediate acknowledgment (e.g. CRUD operations).
Ease and Time Taken to Implement	Requires more time and effort due to event-driven architecture and managing message brokers (such as RabbitMQ).	Generally easier and faster due to the straightforward nature of request-response.
Debugging and Error Handling	Complex: Involves monitoring distributed systems, tracing message flows, and ensuring reliable delivery (e.g., retries, dead-letter queues).	Simpler: Errors are synchronous and handled in the same request-response cycle, making them easier to trace.

Support for Real-Time Data	Excellent: Ideal for scenarios needing low latency, real-time updates, and quick system reactions (e.g., live chats, notifications).	Limited: Introduces latency, as clients must wait for a response, making it unsuitable for real-time scenarios.
Is Compatible with CQRS Communication Pattern	Yes: Events naturally align with CQRS by decoupling write (command) and read (query) models.	Limited: Tight coupling in synchronous requests makes it harder to separate concerns in a CQRS setup.
Fault Tolerance	High: Systems can retry failed messages, use dead-letter queues, and resume processing without affecting the overall system.	Lower: If the responder service fails, the client request may timeout, requiring retries at the client level.
Dependency Management	Loose coupling: Components only listen for or emit events, reducing dependency on other services' internal details.	Tightly coupled: The client must know the exact API or endpoint of the responder, increasing dependencies.
System Coupling	Loose coupling: Asynchronous events decouple components, enabling independent updates and scaling.	Tight coupling: Clients and servers are tightly coupled, requiring changes on both ends for updates.
Performance Overhead	Low latency at scale: Efficient for real-time updates but adds overhead for managing brokers or message systems.	Higher latency at scale: Performance degrades with increased simultaneous requests due to blocking synchronous communication.
Response Guarantees	Eventual consistency: No immediate response; guarantees are managed via retries and acknowledgments.	Immediate response: Clients expect direct responses, but the system fails if the responder is unavailable.

Despite the plentiful merits asynchronous communication offers us, we chose to stick to synchronous communication due to the ease of implementation, given the tight schedule for development. Here is a detailed breakdown on our decision.

Reason	Explanation
Fulfilling Functional and Nonfunctional Requirements	Our use case requires immediate responses for matchmaking or status queries, which are achievable with a synchronous pattern. Redis ensures fast lookups, meeting functional needs, and our less ambitious non-functional needs.
Easy and Fast Implementation	Synchronous patterns are easier to implement since they follow a straightforward request-response flow, eliminating the need for complex message brokers or event-driven architectures.
Simplifies Testing and Debugging	Errors can be traced directly within the request-response cycle, reducing the need for distributed tracing tools and making debugging faster and simpler.
Ease of Delegation	Synchronous communication aligns with the team's familiarity and expertise, enabling better delegation of tasks and faster development.
Latency and Synchronisation by Redis	Redis handles real-time operations with low latency and synchronisation features, reducing the typical drawbacks of synchronous communication.
Fault Tolerance Not Required	Our use case does not demand high fault tolerance (e.g., retries, dead-letter queues). Moreover, with an out-of-the-box solution with RedisCloud, it offers fault tolerance that meets our requirements.
Response Guarantees Not Critical	Our system does not need strict guarantees for retries, acknowledgments, or durable message storage. Temporary errors can be addressed by intuitive retries (implemented in the frontend) initiated by the user.

Supports CQRS Pattern	The synchronous approach still enables us to follow the CQRS pattern by separating the query and command responsibilities at the service level.
-----------------------	---

Therefore, we stuck to a synchronous pattern. Moreover, our approach to the synchronous communication slightly emulates an event driven, asynchronous message pattern, by following closely to the CQRS pattern.

6.2.3 Command Query Responsibility Segregation (CQRS)

To ensure our matching service adheres to the CQRS principles, we structured our implementation by separating responsibilities for handling commands (write operations) and queries (read operations). This clear separation improves maintainability, scalability, and performance, especially for real-time data processing. Below, we outline the key implementation details and explain how each component aligns with CQRS principles.

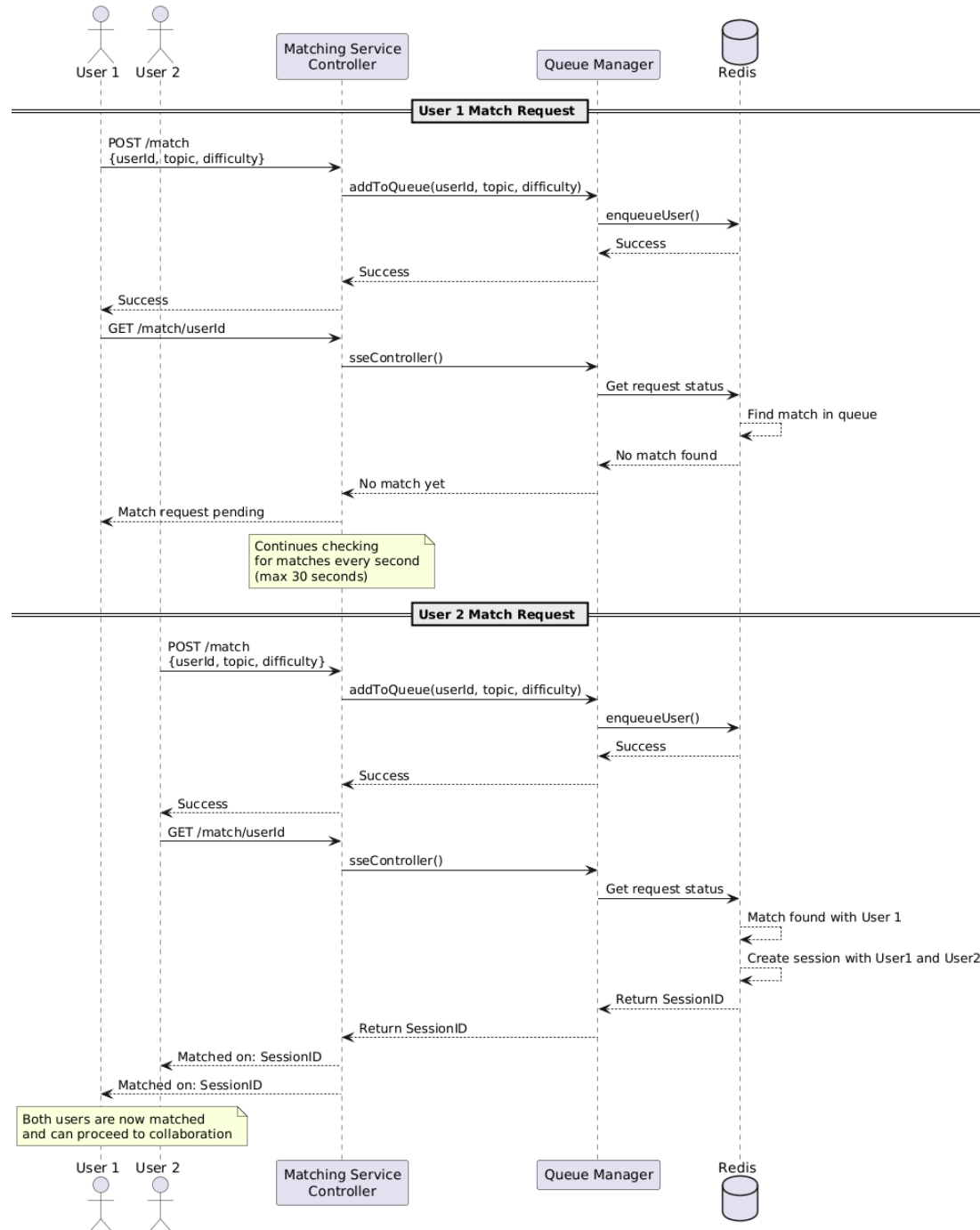
Implementation Aspect	Details	How It Achieves CQRS
API Route Separation	<ul style="list-style-type: none"> - Commands: POST <code>/matchingrequest</code> for creating requests, DELETE endpoints for deletions. - Queries: GET <code>/matchingrequest/:userId</code> (SSE for status updates). 	<ul style="list-style-type: none"> - Ensures write operations (create/delete) are isolated from read operations (real-time status retrieval). - Reduces coupling between query and command logic.
Recursive File Separation with respect to command and query patterns	<ul style="list-style-type: none"> - Command Controllers & Services: <code>requestController.ts</code>, <code>sessionManager.ts</code> handle data creation/deletion. - Query Services: <code>statusManager.ts</code>, <code>sseController.ts</code> handle data retrieval. 	<ul style="list-style-type: none"> - Logical separation ensures commands modify state without interfering with query workflows. - Query services handle only read-related logic.

Server-Sent Events (SSE)	<ul style="list-style-type: none">- <code>sseController.ts</code> streams real-time match updates with server-side events- Includes timestamps to emulate real time data retrieval	<ul style="list-style-type: none">- Decouples queries from the write side, as updates are consumed independently.
--------------------------	---	---

By following the CQRS pattern, we achieved a clean separation between commands and queries in the matching service. Commands are isolated in their respective controllers and services to modify the state, while queries leverage Server-Sent Events and prioritised logic to deliver real-time updates. This architecture not only simplifies maintainability and debugging but also scales efficiently by decoupling read-heavy operations from write-heavy workflows. Ultimately, this implementation enables the service to fully benefit from CQRS, such as enhanced performance, real-time responsiveness, and improved modularity. Overall, it allows the frontend to effectively subscribe to events in the database, while being able to write into it when allowed.

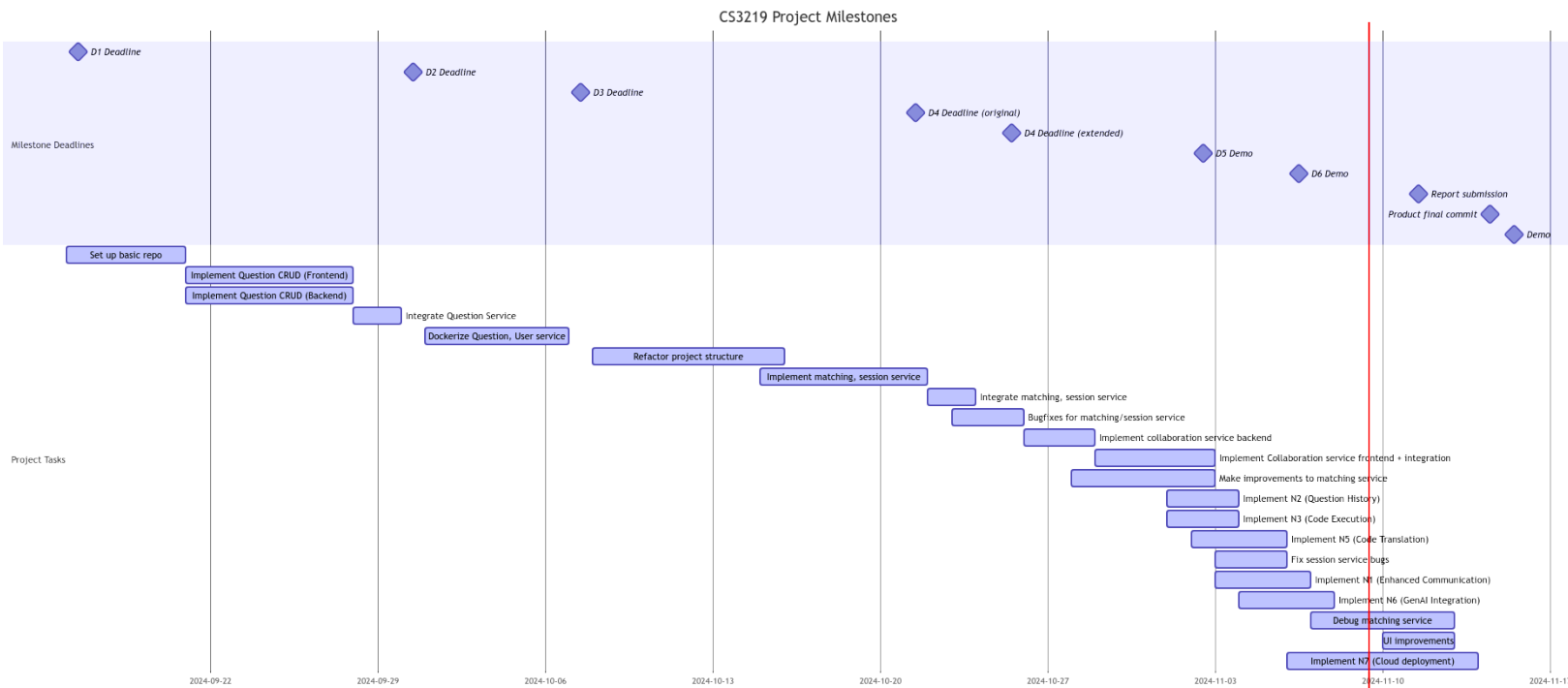
6.2.4 Sequence Diagram of Matching Service

This sequence diagram combines both our code structure and the CQRS principles we followed, where matching requests are sent separately and statuses of these matching requests are retrieved separately.



7. Project plan

7.1 Gantt Chart (Flipped so as to allow for a larger Gantt Chart)



Usage of AI

In our project, we have leveraged AI tools such as **ChatGPT** and **Claude AI** to enhance various aspects of the development process. These tools were utilised to improve efficiency and accuracy across multiple tasks, from data generation to code optimisation.

1. Sample Data Generation:

AI tools were used to automatically generate sample data, ensuring comprehensive coverage for testing various scenarios and edge cases.

2. Code Debugging and Error Detection:

ChatGPT and Claude AI helped identify potential bugs and errors, offering suggestions for efficient fixes, thus reducing manual debugging time.

3. Code Structuring and Optimisation:

AI tools analysed and suggested improvements for code structure and performance, leading to more efficient and maintainable code.

4. Code Vulnerability Assessment:

AI tools analysed the vulnerability of our code and provided suggestions to keep code more robust in terms of security.

