



School of Computing

CS3219 Software Engineering Principles and Patterns

AY21/22 Semester 1

Final Report

Team 37

Team Members	Student No.
Wong Qin Liang	A0200341E
Loo Yeou Tzer	A0200248U
Lim Zi Qiang, Marcus	A0200325B
Zen Lee Yijie	A0202290W

1 Introduction	4
1.1 Background	4
1.2 Purpose	4
1.3 Description of Application	4
2 Individual contributions	5
3 Application Requirements	6
3.1 Functional Requirements (FRs)	6
3.2 Non-functional Requirements (NFRs)	8
4 Documentation	14
4.1 Architectural Design	14
4.1.1 Parent MVC	14
4.1.2 Nested MVC	16
4.1.2 Design decisions	17
4.2 Devops design	19
4.2.1 AWS Elastic Compute Cloud (EC2)	19
4.2.2 AWS Elastic Beanstalk (ELB)	19
4.2.3 Continuous Integration / Continuous Delivery (CI/CD)	20
4.2.4 AWS Relational Database Service (RDS)	21
4.3 Database design	21
4.3.1 Choice of Database System	21
4.3.2 Database schema	22
4.3.3 Setting up and accessing the database	24
4.4 Server design	25
4.4.1 Feature 1 - Video Synchronization (Server)	25
4.4.2 Feature 2 - Authentication	29
4.4.3 Design decisions	30
4.5 Frontend design	32
4.5.1 Code structure	33
4.5.2 Feature 1 - Video Synchronization (Client)	33
4.5.3 Feature 2 - Live text chat	34
4.5.4 Feature 3 - Authentication	35
4.5.5 Design decisions	36
4.6 Technical choices	38
4.6.1 Tech stack	38
4.6.2 Libraries and Packages	39
4.6.3 Coding standards	39
4.7 Development process	40
4.7.1 Project plan	40
4.7.2 Development schedule	40
4.8 Project management	43

5 Possible improvements and enhancements	43
6 Reflections and Learning points	43
6 Appendix	45
6.1 PeerWatch's uptime through Uptrends	45
6.2 UI Mockup	47

1 Introduction

1.1 Background

With the current COVID-19 pandemic, people are not allowed to go out in big groups or have a huge number of visitors to their house. The next best way to spend time with your friends or loved ones is through online entertainment and watching YouTube videos is one of the most popular ways to get entertainment. The standard way of streaming such content to one another is through the share-screen feature of popular voice-chat applications like Discord. However, these methods are susceptible to low quality feed, and often require users to pay a premium for better streaming resolution. It also induces a latency cost as one of the users has to be capable of streaming to other users while retrieving data from YouTube at the same time.

1.2 Purpose

The purpose is to create a web application that helps to relive the cinema experience of watching shows with your friends and relatives.

This can be achieved in multiple ways. For example, (i) by directly streaming the video content with others in a fast and seamless way or (ii) by using the well-established streaming platform to carry the heavy lifting of streaming the content to others. Despite many popular applications used the first way, it required at least one of the users to stream their content out from their device. This can incur a significant amount of bandwidth that not every user's device is capable of. For this project, we shall use method (ii), this allows us to delegate the streaming part onto streaming platforms like YouTube, restricting user's bandwidth cost to strictly fetching video feed from YouTube. As this application is focused on accessibility, developing it as a web application is a natural choice.

1.3 Description of Application

A web application that allows users to create shared rooms where they can stream and watch YouTube videos together. Inside these rooms, users will provide a YouTube video link and that video will be playing for every user in the room. The video player will then be synchronized across all users, i.e. users will be watching the video at the same timing, and any user actions such as pause, fast-forward, speed-up will take effect for all users. The users can also communicate with one another through a chat box in the room.

A general use case of PeerWatch is as follows:

John is under Stay Home Notice but wants to spend time with his friend, Peter. He creates an account and logs in. After logging in, John creates a room with a unique room code. He informs Peter of this room code through Telegram. Peter then proceeds to create an account, log in, then join the same room as John using this unique room code. Afterwards, they search for interesting YouTube videos and insert the link in a text box provided in the room which will play the video with both of their players synced. After they feel tired, they can inform each other in

the chat box and leave the room by navigating back to the home page. This will bring them back to the landing page. From this page, they can log out.

2 Individual contributions

Team member	Technical contributions
Wong Qin Liang	<ul style="list-style-type: none">● Setting up tech stack● Architectural Design● UI Design● Frontend UI, features and integration with backend● Video synchronization (1 server & multi servers)● Live text chat● Resolution support● Optimization (Lazy load, preload etc)● Browser testing
Loo Yeou Tzer	<ul style="list-style-type: none">● Live text chat<ul style="list-style-type: none">○ Viewing of users in the room.● Account login, registration, and password reset.● Account recovery via email.● Setup CI for tests● Wrote API tests
Lim Zi Qiang, Marcus	<ul style="list-style-type: none">● Design database schema● Backend APIs for rooms● Integration of account login, registration and password reset with database
Zen Lee Yijie	<ul style="list-style-type: none">● Set up CI/CD pipeline● Set up production environment (EC2, ELB)

Team member	Non-technical contributions
Wong Qin Liang	<ul style="list-style-type: none">● Documentation<ul style="list-style-type: none">○ FR / NFRs○ Architectural design○ Server design○ Frontend design○ Technical choices○ Project management○ Possible improvements and enhancements○ Reflections and Learning points● Organise and manage the tasks to be completed and bugs to track in GitHub Project
Loo Yeou Tzer	<ul style="list-style-type: none">● Documentation<ul style="list-style-type: none">○ FR / NFRs

	<ul style="list-style-type: none"> ○ Server design ○ Frontend design ○ Development process
Lim Zi Qiang, Marcus	<ul style="list-style-type: none"> ● Documentation on Database
Zen Lee Yijie	<ul style="list-style-type: none"> ● Documentation on Dev Ops

3 Application Requirements

3.1 Functional Requirements (FRs)

Based on the description of the application, we have identified five categories of FRs in decreasing importance: Video synchronisation, Room management, Live chat, Pages, and Authentication and Authorization. The requirements under each category are arranged in descending priorities and labelled accordingly.

S/N	Functional Requirement
Video Synchronisation	
F1.1	The application should allow users to play a YouTube video by providing a valid URL
F1.2	The application should allow users to watch the same Youtube video in sync with the other users in the room
F1.3	The application should allow users that just joined the room to load up the Youtube video that is playing in the room
F1.4	The application should pause the Youtube video for all other users in the room when any user is buffering and only resume the video when all users finish their buffering
F1.5	The application should allow users to adjust the playback settings and have the settings propagated to other users in the room (e.g. Play, Pause, Seek, Playback speed).
F1.6	The application should allow users to watch the Youtube video in full screen without affecting other users in the room
Rooms Management	
F2.1	The application should allow users to create a room that other users can join
F2.2	The application should allow users to join a room through a generated link or entering a valid code
F2.3	The application should allow the host to automatically pass their role to another user in the room once they left

F2.4	The application should close the room once all users have left
F2.5	The application should allow users to copy the room's link or code to share with others
F2.6	The application should prevent users from entering a room that they are already in
F2.7	The application should prevent users from entering a room that has already reached its maximum capacity
F2.8	The application should allow users to see who else is in the room by displaying a list of usernames.
F2.9	The application should allow the host to change the room's maximum capacity
F2.10	The application should allow the host to block and unblock users from using text chat
F2.11	The application should allow the host to block and unblock users from entering a Youtube video URL
F2.12	The application should allow the host to kick any user out of the room except themselves
Live chat	
F3.1	The application should allow users to talk to each other in the same room via text chat
F3.2	The application should allow users to identify one another's messages by attaching their username with it
F3.3	The application should allow users to be notified by a message when a new user enters the room
Pages	
F4.1	The application should allow users to access a page where they can watch Youtube videos together and chat with one another
F4.2	The application should allow users to access a page where they can create a room, join a room, register an account, and log into an account.
F4.3	The application should allow users to access a page where they can reset their account's password
F4.4	The application should allow users to return the previous page
F4.5	The application should redirect users to a fallback page whenever they enters a room that is either full or a room that they are already in
F4.6	The application should redirect users to a fallback page whenever they attempts to enter a room that cannot be found

F4.7	The application should redirect users, that have yet to login, to a fallback page whenever they try to access a page that requires logging in
F4.8	The application should redirect users to a fallback page whenever they access an URL route that was not supported
Authentication and authorization	
F5.1	The application should allow users to create an account with a display name, an email and a password
F5.2	The application should allow users to login to their account by entering their email and password
F5.3	The application should allow users to change their password through a page only accessible via a link sent to their email
F5.4	The application should allow users to request for a password reset link by providing an email that an account is associated with

3.2 Non-functional Requirements (NFRs)

Differing from FRs, NFR categories are established based on Quality Attributes. The attributes that we focused on are: Performance, Availability, Interoperability, Usability, Security, Integrity, Portability and Scalability. We determined which attributes to be prioritized by using a Requirement Prioritization table whereby these attributes are compared against one another to derive a score and through this score we know which are the more important attributes.

Attribute	Score	Performance	Availability	Interoperability	Usability	Security	Integrity	Portability	Scalability
Performance	6		<	<	^	<	<	<	<
Availability	2			^	^	^	<	^	<
Interoperability	5				<	^	<	<	<
Usability	5					<	<	^	<
Security	4						<	^	<
Integrity	4							^	<
Portability	4								^
Scalability	1								

As PeerWatch targets a niche audience with limited use cases, the importance of availability and scalability has been dropped in favor of performance, interoperability and usability. These attributes will not only help to provide users with a better watching experience, but also provide a more usable application as it is likely that many new users are non-tech savvy friends or relatives of an existing user. The security and integrity aspects of the application are also given lower priority as the primary purpose of having an account is to help identify each user and does not store any other personal details of the user, other than their email and password.

As such, our NFR categories will be prioritised in this order: Performance, Interoperability, Usability, Portability, Security, Integrity, Availability and Scalability. The following table list those categories and their corresponding NFRs in descending order of importance:

S/N	Non-Functional Requirement
Performance	
NF1.1	<p>The application should sync the video between users within a 2 second delay.</p> <p>Reason: According to a research, less than 2 seconds is the duration where many application users remember information from a response. The more detailed the information is, the greater the need for response to be less than 2 seconds. Since visual content like videos comprise a significant amount of info, the limit is essential to keep all the users that are watching the same video to be on the same frequency.</p> <p>Reference: <i>The Art of Application Performance Testing</i> by Ian Molyneaux</p>
NF1.2	<p>The application should ensure the live text chat messages that were sent are received in less than 2 seconds</p> <p>Reason: Similar reason as above, conversation between users is also something that requires all users to be on the same frequency for.</p>
NF1.3	<p>The application should load the webpage within 2 to 4 seconds.</p> <p>Reason: According to the same research above, a duration of 4 seconds can sometimes inhibit a user and frustrate them while a duration of less than 2 seconds is more for tasks requiring high concentration. For a task that requires low concentration such as accessing a page, 2 to 4 seconds is a reasonable duration.</p>
NF1.4	<p>The application should allow at most 15 users in a room and prevent any other user from joining once the room is full.</p> <p>Reason: Having a maximum capacity per room allows us to determine our approach towards handling user communication and optimization. This capacity is set to 15 as we are expecting the application to be mainly used among friend circles or family circles.</p>
NF1.5	<p>The application should check whether a user is active in a room and kick them out if they cannot respond to a prompt in time</p> <p>Reason: To prevent an idle user from unnecessarily taking up the server's resources when they are not utilizing it.</p>
NF1.6	<p>The application should ensure its performance scoring in Google Lighthouse to be equals or above 90</p> <p>Reason: To improve the exposure of PeerWatch, SEO is essential. For Google search result ranking, having a high score in Lighthouse helps the app to achieve a higher rank than other search results.</p>
Interoperability	
NF2.1	The application should accept all official formats of links to Youtube videos (eg.

	<p>direct, shared, with timestamp)</p> <p>Reason: There are many ways that users can get the link to a Youtube video. It can be directly from the URL bar in their browser, a shortened link via YouTube's share function, or a link to a recently watched video that includes a timestamp. The users will naturally expect these links to all work properly with the application.</p>
NF2.2	<p>The application should be kept up to date with the Youtube API</p> <p>Reason: It is understandable and expected that YouTube's API will change over time. Those changes may incur adjustments on how the application approaches the embedding, synchronisation and playback of a video. To ensure the application's features remain functional, the application needs to keep up to date.</p>
Usability	
NF3.1	<p>Users should be able to navigate to any page in the site within 3 clicks</p> <p>Reason: To prevent users from going through too many unnecessary pages and affect their experience.</p>
NF3.2	<p>The application should ensure the contrast ratio of at least 4.5:1 for text and 3:1 for UI components.</p> <p>Reason: According to Web Content Accessibility Guidelines (WCAG), contrast and color use are vital to a website's accessibility, especially to users with visual disabilities. To ensure PeerWatch remains visually perceivable, we will adhere to the contrast ratio established by WCAG 2.</p> <p>Reference: Contrast and Color Accessibility - Understanding WCAG 2 Contrast and Color Requirements</p>
NF3.3	<p>The application should ensure its text content to be of font size at least 1em.</p> <p>Reason: 1em is equivalent to the font size defined by the browser's user preference. This will not only allow the font size to be comfortable for the user, but also em is a scalable unit that can allow us to easily scale down for mobile devices if needed.</p> <p>Reference: CSS Font-Size: em vs. px vs. pt vs. percent</p>
NF3.4	<p>The application should indicate the success or failure of a user's action with meaningful and clear messages</p> <p>Reason: To prevent users from being confused whenever there are any errors happening and assure that their actions have been received successfully.</p>
NF3.5	<p>The application should ensure its accessibility scoring in Google Lighthouse to be equals or above 90</p> <p>Reason: Same reason as NF1.6</p>

Portability	
NF4.1	<p>The application should be accessible via the common browsers (eg. Chrome, Firefox, Edge, Opera, Safari)</p> <p>Reason: To support a wider user base, making the application accessible to all their browsers is essential</p>
NF4.2	<p>The application should be accessible on different operating systems (eg. Windows, Mac, Linux)</p> <p>Reason: To support a wider user base, making the application accessible to all their Oses is essential. Safari is also one of the top used browsers which is only exclusive on mac, so in a way we need to support different operating systems as well.</p>
NF4.3	<p>The application should remain usable for different desktop resolutions: 1920x1080 px, 1366x768 px, 1536x864 px, and 1280x720 px.</p> <p>Reason: Users can use screens of varying sizes to access the application. To ensure the application remains usable for the majority, the UI must be responsive towards the most commonly used resolutions for a browser window</p> <p>Reference: Browser Display Statistics</p>
Security	
NF5.1	<p>The application should ensure that any transaction involving user-sensitive information uses signed JWT tokens</p> <p>Reason: JWT allows us to sign tokens using a secret, allowing us to verify whether the sender can be trusted. This will prevent any potential attacker from gaining access to user-sensitive information by listening to the transactions made by the user or the application.</p>
NF5.2	<p>The application should ensure that a user's password is salted and hashed before it is send for transaction</p> <p>Reason: To protect against attackers who gain read-only access to the database that stores the verification required for passwords</p>
NF5.3	<p>The application should ensure that a password must be at least 8 characters long, containing a mix of letters and digits</p> <p>Reason: 8 characters is what people are accustomed to for passwords. If we ask for more, users may protest by choosing witty passwords which ruin its purpose. A mix of letters and digits are sufficient here as an enforcing of uppercase/lowercase mix or punctuation will be noticeably harder to type on mobile devices.</p>
NF5.5	<p>The application should ensure that the code for each room must follow the Universally Unique IDentifier (UUID) standard</p>

	<p>Reason: This standard not only allows us to uniquely identify each room with near certainty, but also makes it nearly impossible for any nefarious user from bruteforcing room code to invade a user's room.</p> <p>Reference: rfc4122</p>
NF5.6	<p>The application should ensure that a generated password reset link consists of a hashed random identifier.</p> <p>Reason: To prevent nefarious users from brute forcing a valid reset link and thus, compromising a user's account.</p>
NF5.7	<p>The application should ensure that the password reset link is only sent to the user's registered email address.</p> <p>Reason: Since only the account owner has access to their own email address, it is reasonable to assume that by sending a reset link through that medium, their account will not be compromised with unauthorized password resets.</p>
NF5.8	<p>The application should ensure the password remains hidden when the user is typing it on the register and login page</p> <p>Reason: To prevent a simple on-looker from identifying a user's password.</p>
Integrity	
NF6.1	<p>The application should ensure that the database it uses are different between a development and a production environment</p> <p>Reason: To prevent developers from not only accidentally deleting or contaminating the data in a production server, but also prevent them from directly compromising the user's data. As such, security is maintained as well.</p>
NF6.2	<p>The application should backup the database on a daily basis.</p> <p>Reason: This is so that there will be a backup of the user accounts and prevent any unforeseen situations where data might get corrupted, resulting in users not being able to login to their accounts.</p>
NF6.3	<p>The application should ensure that the database backup process happens at an off-peak period</p> <p>Reason: Lower possibility of any new users registering and not having their data backed up while the backup process is ongoing. Also, there might be a possibility of some form of drop in the application's performance due the database backup process. So it would be good to have this database backup done during an off-peak period to reduce the possibility of this affecting the users.</p>
NF6.4	<p>The application should enforce the validity of fields when registering an account, logging in and resetting password</p>

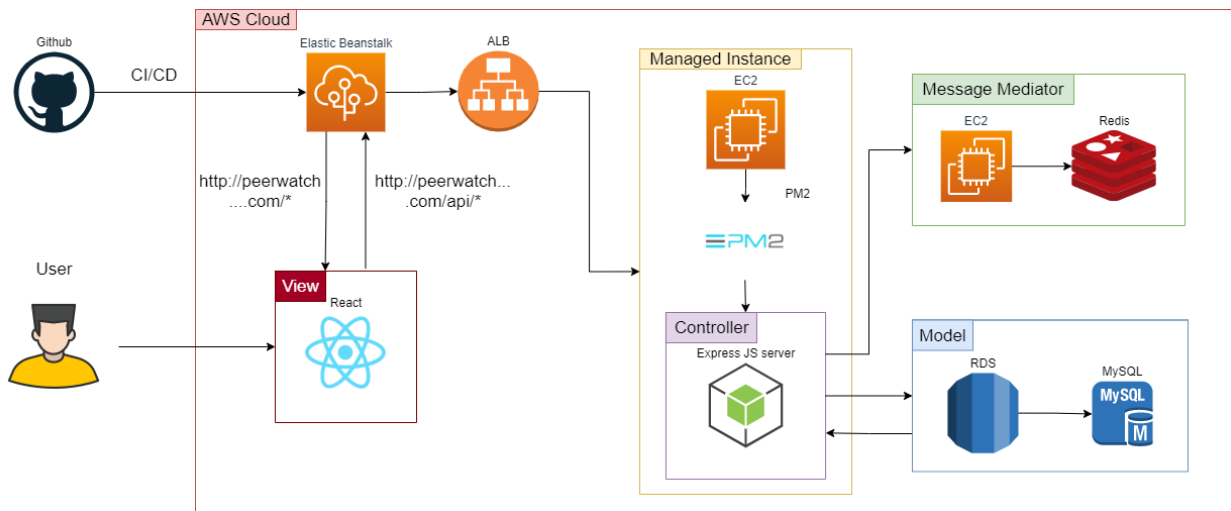
	Reason: This is to ensure that users have filled out all the necessary information before they can register or update their information.
NF6.5	<p>The application should enforce the validity of fields when a host is changing the setting of a room</p> <p>Reason: This is to prevent users from setting an invalid capacity for the room that may unintentionally lock users out of a room</p>
Availability	
NF7.1	<p>The application should be able to restart within 15 minutes if it goes down</p> <p>Reason: An application can go down for many reasons, including uncontrollable ones like a service provider going down. To ensure the user can continue using the application, it must be able to recover by restarting</p>
NF7.2	<p>The application should attain at least 95% availability for all users with an internet connection</p> <p>Reason: Our application should be up for almost all the time for users to enjoy the service whenever they wish to unless there is maintenance going on.</p>
Scalability	
NF8.1	<p>The application should be horizontally scalable by network out traffic using a load balancer to support at least 1 Mb/s for each replica</p> <p>Reason: As this application is rather niche, we do not expect a large number of users, at most 5,000 named users. Based on the general practice, concurrent users are estimated with the 100:10:1 rule, which states that for every 1,000 named users, 100 are quite active and about 10 are concurrent at any moment in time. So, we will have around 50 concurrent users at any moment but we will add an additional 25 concurrent users as a buffer.</p> <p>In terms of network out traffic, the heaviest message will be a timing broadcast from the server to all users in the room. With a generous estimate, it will consume around 100B (16B for room ID, rest for event name and timing). Assuming that all 75 users are watching videos, it consumes 75,000B/s of network out traffic. Since there will also be smaller messages to support functionalities like chat and room management, 100,000B/s should be more than sufficient for the estimated user base. Therefore, each replica should support at least 0.8 Mb/s of network out traffic.</p> <p>Reference: https://www.ibm.com/docs/en/cognos-analytics/10.2.2?topic=SSEP7J_10.2.2/com.ibm.swg.ba.cognos.crn_arch.10.2.2.doc/c_arch_estimatingconcurrentusers.html How many server instances will I need to support my user base? - LiveSwitch </p>

4 Documentation

4.1 Architectural Design

PeerWatch is driven by a monolithic architecture similar to the Single Page Application (SPA) variant of the Model View Control (MVC) pattern. This architecture consists of two MVC patterns where one pattern is nested within the other pattern's view and that view is on the Client side. The following two sections will present the design for the overall architecture with the parent MVC pattern and the design for the nested MVC pattern:

4.1.1 Parent MVC



When the developers commit changes to the repository, continuous integration (CI) and contiguous deployment (CD) will be performed to build static assets for the frontend and test the application using Mocha and Chai. If there is no issue, the application will be deployed to a AWS Elastic Beanstalk (ES) environment.

This environment creates EC2 instances to host our application, provides a load balancing service to help scale up by creating more instances, and distributes the incoming traffic among the instances. This service allows us to horizontally scale up our key computing unit, the server (controller) of the application, based on customizable metrics. In our case, we scale the environment based on the outgoing traffic of the server as that is the main performance bottleneck for our video synchronization service. Therefore, we can fulfill the scalability requirement mentioned in [NF8.1](#).

In each instance, it runs a PM2 process that runs and monitors an Express server that acts as a controller and serves the static assets which form the view to the users. The process provides us to handle any unforeseen termination of the server by restarting the server whenever it detects that the server is down. The restarting is relatively short as it is on an instance by instance basis and in the meantime, the ES's load balancer can route the traffic to another server. This allows us to keep up an uptime and availability that adheres to [NF7.1](#) and [NF7.2](#). To

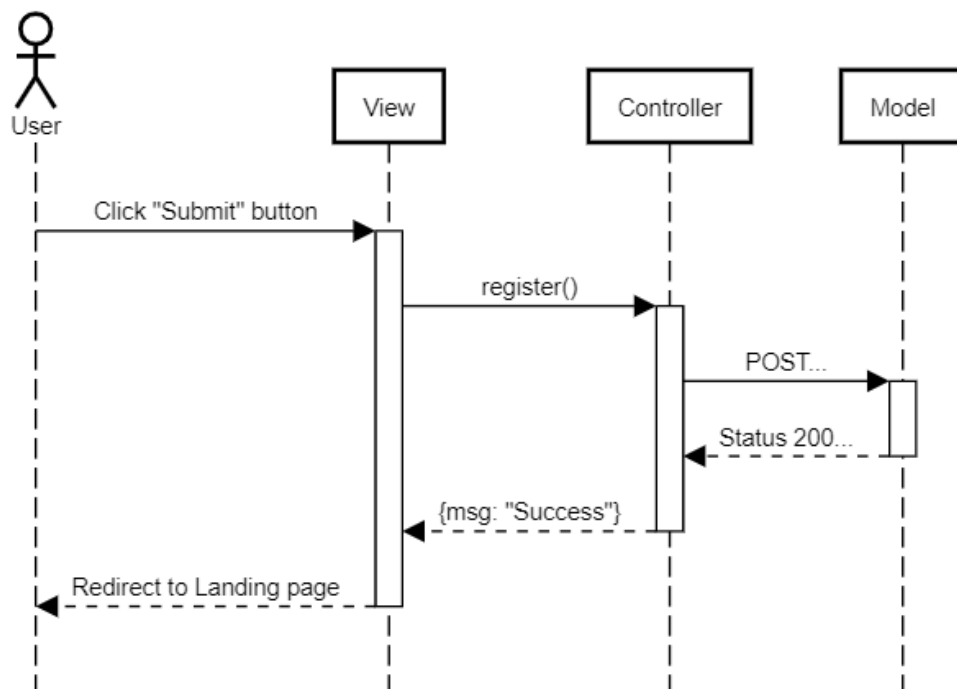
monitor these metrics, we made use of a third party service called Uptrend which measures the metrics by pinging and accessing our server and database. You may look at some of our metric data in [6.1 PeerWatch's uptime through Uptrends](#).

The server instances are connected to the MySQL instances managed by the AWS Relational Database Service (RDS) environment and it serves as a model that stores and provides persistent data such as user credentials, room occupancy, and room settings. These data can then be created, retrieved, updated, or deleted via API controllers implemented by the servers. The databases are also backed up 5am daily in case of a need to recover after unforeseen data corruption and this helps to satisfy [NF6.2](#) and [NF6.3](#).

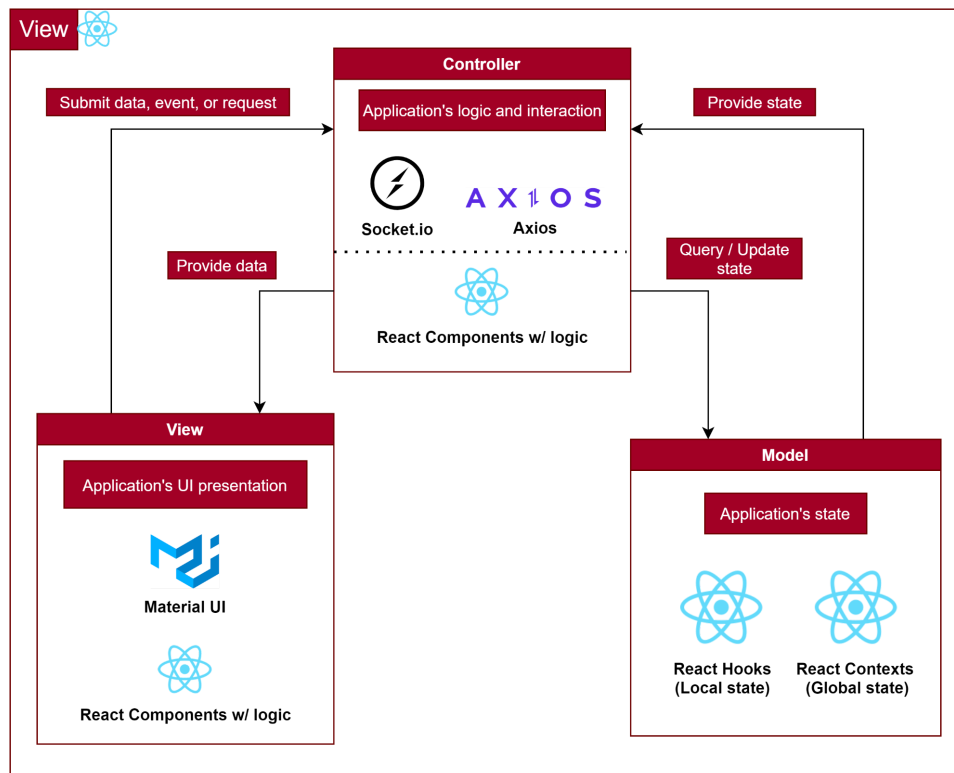
To provide video synchronization and live chat across servers on different instances, the server will not only require the usage of websockets, but also a mediator to manage the message transmission between the servers. This role is performed by a separate EC2 instance that hosts a Redis database which provides a pub-sub service necessary for message transmission across servers and the temporary storage required for synchronizing videos that were watched by users across servers.

As mentioned, the view is served by the Express server via static assets. These assets are then dynamically rendered as pages on the client side and can behave differently based on the data provided by the server. Through this view, users can submit actions or requests to the controller using the controller's APIs to create, retrieve, update, and delete data.

The following diagram summarizes an example of an interaction between the Model, View and Controller when the user is registering an account:



4.1.2 Nested MVC



Within the view of the parent MVC pattern, there is a nested MVC pattern that represents the client side of our application which we shall dub it as Client. The Client is driven by the React library and comprises three aspects, each representing a different aspect of the MVC pattern.

For the model aspect, the Client's storage can be thought of as a form of state persistence which is necessary to provide functionalities such as user authentication and video synchronisation. This persistence is achieved via the React library where the React Hooks allow the Client to persist local states like a button and the React Contexts allow the persistence of the global state such as whether the user has already logged in. This state information can be retrieved and updated by the controller aspect of the Client.

For the view aspect, the Client provides its user interface through assembling a mixture of customised React components and existing components from a UI library like Material-UI. Similar to the UI elements from a traditional view, these components will contain minimal logic in order to submit data, events or requests to the Controller aspect which provide the view with data to determine what to display next.

For the controller aspect, the Client makes use of React Components and third-party libraries like Axios and Socket.io to provide complex functionalities that involve communication and interaction between internal components and external parties such as the servers or other clients.

4.1.2 Design decisions

Throughout the design process of PeerWatch's architecture, there are mainly two design problems we faced and they are as follows:

Should the architecture be monolith or microservices?

The first problem we faced is whether PeerWatch requires a microservice-based architecture or a monolith architecture is sufficient. To decide on an option, we listed out several criterias and verified how each option perform in the criterias through a table like so:

Criteria	Monolith	Microservices
Development complexity	The overall development is simpler and more straightforward in many ways such as in its architectural design, in its deployment and in the communication between its components.	Introduces extra baggage in the form of complexity and initial overhead into the application. These baggage comes in the form of establishing communication, designing architecture and deploy the application
Maintainability	Every developer must work on a single large codebase which will slow down the development as the application grows in size	Simpler to maintain as the codebase of the application is effectively separated in a component by component basis. Each component can be worked on independently without dependency on other components
Scalability	Difficult to scale efficiently as we cannot scale each component of a monolith architecture independently	Easy to scale efficiently as we can just scale up the number of microservices that are responsible for the component in question
Extendability	Requires a long-term commitment to a technology stack which can limit the possible features that PeerWatch can have in the future	Since each microservice runs in a container, it is loosely coupled enough for us to swap out the technology/libraries used for each component. This flexibility will make the extending effort much easier

Even though the maintainability, scalability, and extendability of a microservice architecture is higher, it is unnecessary for an application like PeerWatch. Since the sole purpose of this application is to allow users to watch YouTube videos together, its feature set will remain relatively static and not be extended for the foreseeable future. This also means that the application's codebase will not be expanding from the current manageable size. On the other hand, as the user base of PeerWatch is relatively small (as estimated in [NF8.1](#)), the need to scale up is also minimal. Therefore, we decided to go with a monolith architecture as we can reap the benefit of a much more simpler development process.

Which monolith architecture to use?

The second problem we faced is which monolithic architecture to base on. To decide on an option, we listed out several criterias and verified how each option perform in the criterias through a table like so:

Criteria	Layer-based	MVC	SPA MVC
Potential coupling	Splits the system into 1 or more layers whereby each layer is coupled to 1 or 2 other layers	Split the system into three sections that are only coupled to 1 other section	Similar to MVC, but the client also follows the MVC pattern to reduce coupling further
Scalability	Can be scaled on a layer-by-layer basis	Can be scaled by duplicating Models or Controllers	Same as MVC
Maintainability	Codebase may become a mess when the client side of the application becomes increasingly complex	Same as Layer-based	Due to the client having its own nested MVC pattern, the codebase will be maintainable even if the application grows

Although the scalability among these three architectures are relatively similar, their potential coupling leveling does not. SPA MVC not only has at most one coupling on each of its overall components, but also has at most 1 coupling on the components in its client side. The maintainability of SPA MVC also benefits from this reduced coupling as the codebase can be more clearly divided between server and client and also between the model, view, and controller at the client's side.

4.2 Devops design

4.2.1 AWS Elastic Compute Cloud (EC2)

In the early stages of the development, we deploy our application to an AWS EC2 instance with the aim of making sure the dependencies and libraries are compatible in the instance's environment. Since AWS Elastic Beanstalk uses EC2 instances internally, we figured that it will be an effective way of testing our application.

Such an instance serves as a virtual machine whereby we need to set up a production environment for deployment. To do this, we had to SSH into the instance to install the required software (Git, PM2, NodeJS) and set up the necessary environment variables. Manual deployments can be tiresome, as we need to SSH into the EC2 instance to redeploy the application each time a change is made. Therefore, we set up scripts on Github Actions and AWS CodeDeploy pipeline to automate deployment.

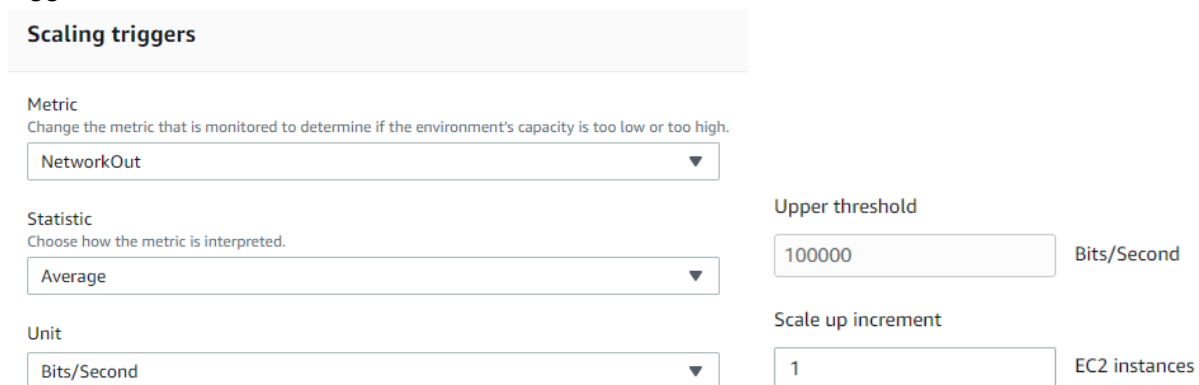
Additionally, to ensure that the application stays online all the time, we utilized the PM2 library, a process manager that helps to monitor PeerWatch and restarts the application if it went down for unforeseen reasons.

4.2.2 AWS Elastic Beanstalk (ELB)

Once we verified that PeerWatch's core features were functional on an EC2 instance, we migrated the production environment onto ELB. This provides a layer of abstraction through AWS Application Load Balancer whereby it helps us to load balance, auto scale, and deploy. Unlike a basic EC2 instance where we needed to SSH into the EC2 instance to perform manual configurations to the production environment, these configurations can be easily performed on the ELB console.

Scaling Triggers

To adhere with the scaling requirement stated in [NF8.1](#), we have given the load balancer an upper threshold of 1Mb/s of NetworkOut traffic for each instance. Whenever an instance exceeds this limit, a new EC2 instance will be created to handle the excess traffic. Conversely, the application will scale down by one EC2 instance when the NetworkOut traffic falls below a lower threshold of 0.2Mb/s. The following screenshot showcase the configuration of our scaling trigger:



The screenshot displays the 'Scaling triggers' configuration interface. It includes three dropdown menus on the left: 'Metric' set to 'NetworkOut', 'Statistic' set to 'Average', and 'Unit' set to 'Bits/Second'. On the right, there are two input fields: 'Upper threshold' set to '100000' with the unit 'Bits/Second', and 'Scale up increment' set to '1' with the unit 'EC2 instances'.

Configuration Field	Value	Unit
Metric	NetworkOut	
Statistic	Average	
Unit	Bits/Second	
Upper threshold	100000	Bits/Second
Scale up increment	1	EC2 instances

Sticky Sessions

As will be later discussed in [4.4.1 Feature 1 - Video Synchronization \(Server\)](#), we have to consider the case where multiple clients are in the same room, but are doing so from servers in separate EC2 instances. The message transmission between them will be handled through a Redis mediator. However, we have to ensure that the client needs to be consistently directed to the server where it joins the room as it has the Socket.IO session that they belong to. If the client got routed to another server, that server has no knowledge about the room that the client is in. Therefore, it is paramount that we provide a means to ensure a client is consistently routed towards the EC2 instance it joined initially.

To do this, we enable sticky sessions feature for the EC2 instances through ELB's configuration as follow:

Sessions

The following settings let you control whether the load balancer routes requests for the same session to the Amazon EC2 instance with the smallest load, or consistently to the same instance.

Stickiness policy enabled

☒ Stickiness policy enabled

Cookie duration

Lifetime of the sticky session cookie between an Amazon EC2 instance and the load balancer.

604800

4.2.3 Continuous Integration / Continuous Delivery (CI/CD)

For the project, we set up a workflow using Github Actions to automate our testing and deployment whenever there are changes made to the codebase. Here is how the workflow works:

1. Listen for commits pushed to the **master** branch.
2. Run test cases
3. Install dependencies and build the project
4. Prepare for deployment, by converting the project package into a ZIP file
5. Upload the project to AWS S3 Bucket
6. Deploy project stored in S3 Bucket to the production environment

4.2.4 AWS Relational Database Service (RDS)

Similar to ELB, AWS RDS serves as a layer of abstraction to provide availability and scalability but for our database. It is also reasonably easy to set up and configure. Therefore, we decided to run our mySQL database within this service.

Daily Backups:

For our application's database, we configured RDS such that daily backups occur at 5am. This is so that under unforeseen circumstances, we have the option to roll back the database to its previous day's state.

<input type="checkbox"/>	rds:peerwatchdb-2021-11-07-05-02	November 07, 2021, 5:02:15 AM UTC	Available	Automated
<input type="checkbox"/>	rds:peerwatchdb-2021-11-08-05-07	November 08, 2021, 5:07:50 AM UTC	Available	Automated

Database Availability and Failover:

RDS also offers a Multi-Availability Zone configuration, allowing us to set up a synchronous standby replica in a different Availability Zone. The primary DB instance is then synchronously replicated across Availability Zones to a standby replica to provide data redundancy, eliminate I/O freezes, and minimize latency spikes during system backups.

Availability & durability

Multi-AZ deployment
[Info](#)

☒ Create a standby instance (recommended for production usage)
Creates a standby in a different Availability Zone (AZ) to provide data redundancy, eliminate I/O freezes, and minimize latency spikes during system backups.

☐ Do not create a standby instance

4.3 Database design

In this section, we will be discussing the database system used, its schema design and the setup process for local development and for production.

4.3.1 Choice of Database System

Initially, we plan to use MongoDB due to the team's familiarity with the technology. However, after a suggestion from our mentor to consider other types of databases, we started to look into other databases and a relational database management system (RDBMS), MySQL has caught our attention. To determine which database to opt for, we decided to measure how suitable they are for PeerWatch based on the [CAP theorem](#).

As the primary purpose of the PeerWatch is for users to stream and watch video in sync, it is important for the database to guarantee all queries will be responded with the latest known accurate information. This means that we lean towards a database that provide consistency and availability:

Criteria	MySQL	MongoDB
Consistency – Every read is the latest write	High consistency as it is a single node system, hence operations are only executed in that single node.	Similar to MySQL, operations can only be performed on the primary node
Availability – Every request receives response	High availability, no network partition as it is a single node system	Low availability, does not allow write operations when primary node becomes unavailable
Partition-Tolerance – System continues to operate when nodes are disconnected	Low partition-tolerance, single node system which means single point of failure	High partition-tolerance, able to elect a new primary node if the current primary node becomes unavailable

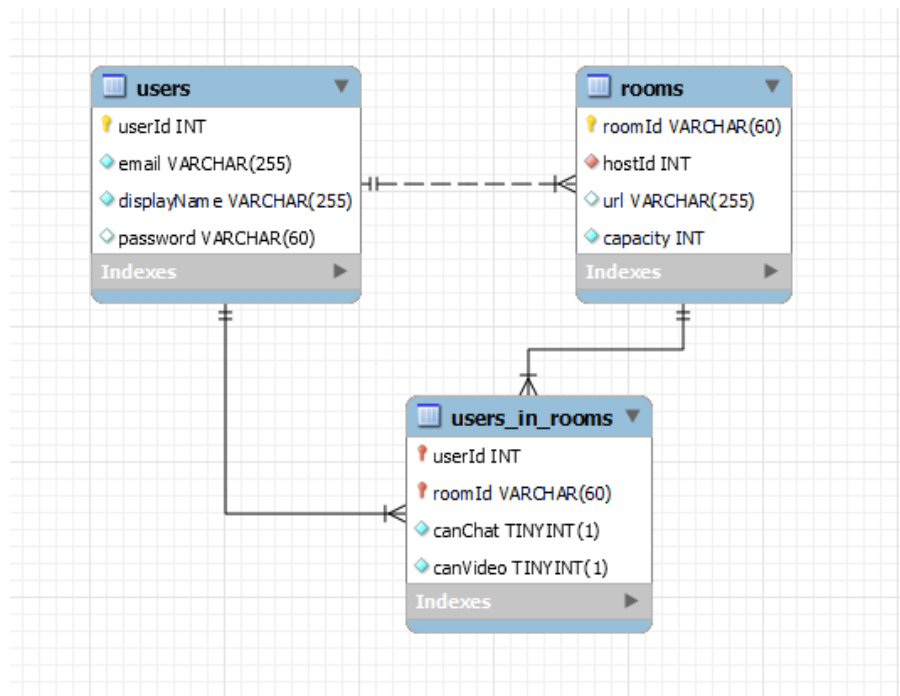
Even though MySQL does not provide a high amount of partition tolerance, it is able to provide PeerWatch with consistency and availability. This lack of partition tolerance can be mitigated

through setting up redundancy which was mentioned in [4.2.4 AWS Relational Database Service \(RDS\)](#). Therefore, we opted to use MySQL as our database system.

4.3.2 Database schema

The database consists of three tables:

Table	Description
Users	Stores information related to a user's credential and account like the user's email, password and display name
Rooms	Stores the information of a room like its ID, URL of the currently playing video and its maximum capacity
Users_in_rooms	Stores the existence of users in a particular room and the permission that each user has for live text chat or video player in that room.



Column type and constraints of each table:

The following tables details the columns in each table and provide description for them:

Users table			
Columns	Type	Constraint	Description
userId	Integer, auto-increment	Primary key	Only used internally by the database and server. Acts as a foreign key for rooms and users table

email	Characters, max length of 255	Unique	Email of the user, used for authentication and account recovery
displayName	Characters, max length of 255	Not null	Display name of the user to identify users when they join a room
password	Characters, length of 60	Not null	Account password for authentication, stored as a hash using bcrypt password-hashing function

Rooms table			
Columns	Type	Constraint	Description
roomId	Characters, max length of 60	Primary key	ID generated by socket.IO to uniquely identify each room
hostId	Integer	Foreign key to users.userId	ID of the room's host
capacity	Integer	Not null, max value of 15	Display name of the user to identify users when they join a room
url	Characters, max length of 255	Can be null	Youtube link that is currently played in the room

Users_in_rooms table			
Columns	Type	Constraint	Description
userId	Integer	Primary key Foreign key to users.roomId	ID of the users in the room
roomId	Characters, max length of 60	Primary key Foreign key to rooms.roomId	ID of the room
canChat	Tinyint/boolean	Not null, default to true	Permission to chat in the room
canVideo	Tinyint/boolean	Not null, default to true	Permission to insert the youtube link for the room

4.3.3 Setting up and accessing the database

To adhere to [NF6.1](#), we provide scripts based on the data definition languages (DDL) statements for the 3 tables (located in the “<project_root>/sql” directory). These scripts can be run on a local or production MySQL database to generate the database needed for PeerWatching. These scripts must be executed in the following order:

Script	Purpose
peerwatch_users.sql	Creates the Users table
peerwatch_rooms.sql	Creates the Rooms table that is dependent on the Users table
peerwatch_users_in_rooms.sql	Creates the Users_in_rooms table that is dependent on both the Users and Rooms table

For development, each developer will set up the database individually on a local MySQL server. The database can only be accessed locally by the developer using MySQL client (eg. MySQL workbench) with credentials setup by themselves.

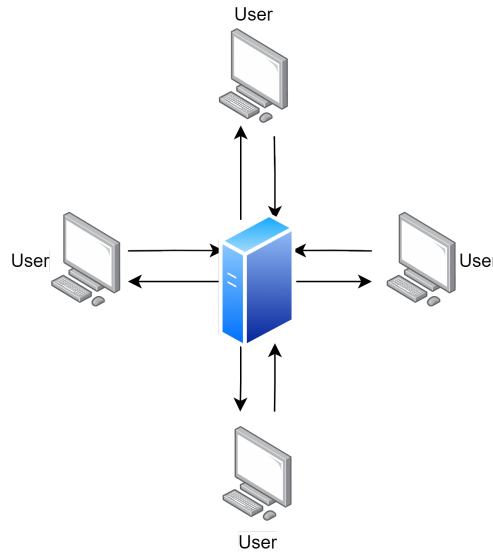
For the production environment, the database is hosted on the AWS RDS mentioned in [4.2.4](#). To ensure data integrity in the production environment, the access is only limited to the developer in charge of DevOps.

4.4 Server design

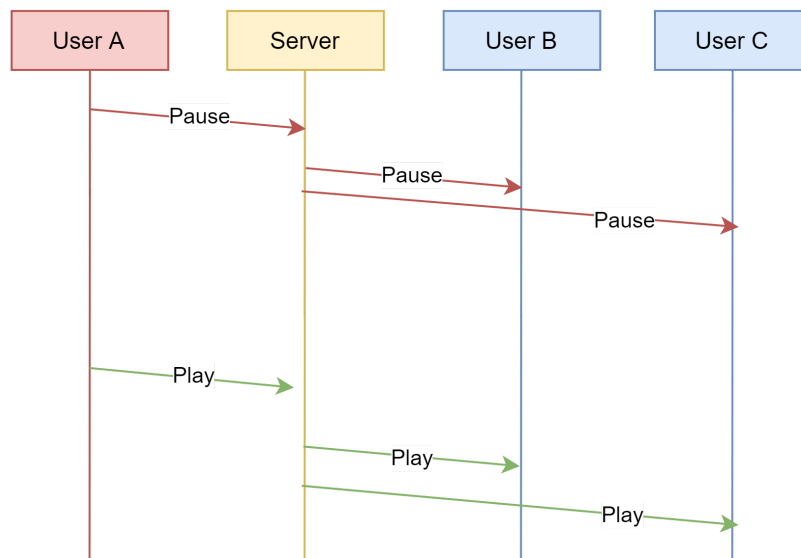
In this section, we will be discussing the server aspect of the application. To better examine this, we will discuss this in the context of the features within PeerWatch.

4.4.1 Feature 1 - Video Synchronization (Server)

This first key feature is the timing synchronization of the YouTube videos for multiple users. To achieve [F1.1](#), we made use of websockets to connect users to the server and have the users send messages to one another with the server being their mediator. These messages include information like the latest timing to synchronise to, a notification to pause the video and a notification that the host has left the room. The message flow between users in a 1 server environment will look as follows:



For simple messages like a notification to pause a video, we make use of Asynchronous Message Passing to pass from the user that initiates the pause to the rest of the users in the room. Here is a diagram that depicts how that message passing happens:



However, problems arise when we need to fulfill [F1.4](#) and perform transmission across servers. These requirements require a more sophisticated form of message transmission and we will highlight three issues:

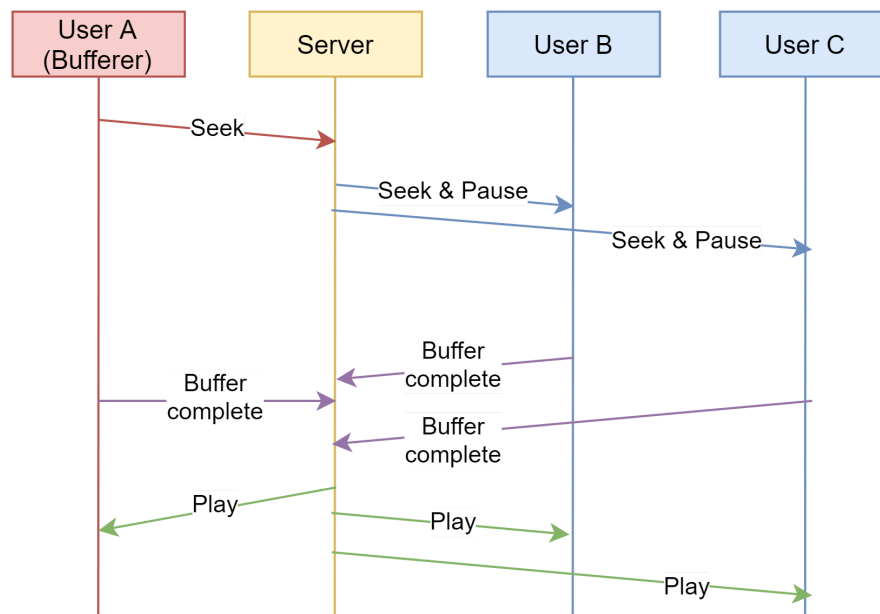
Video Buffering

A video can be buffering due to reasons such as a user having slow internet connection or user initiates a Seek event (Skipping to another playback timing). Regular message passing is insufficient here as a Seek event requires all users to buffer and we need to ensure that all

users complete their buffering before resuming the playback. Otherwise, some users will de-synchronise.

To ensure that, we implemented a form of one to many handshake whereby the user that is buffering (bufferer) sends a message to the server to announce its need to buffer. Rather than passing along that message, the server broadcasts a command message to request all other users in the room to notify the server once they finish buffering. When the server receives a response from all users in the room, the server will broadcast a message for all to resume their playback. Only then, everyone resumes the video and desynchronization does not happen.

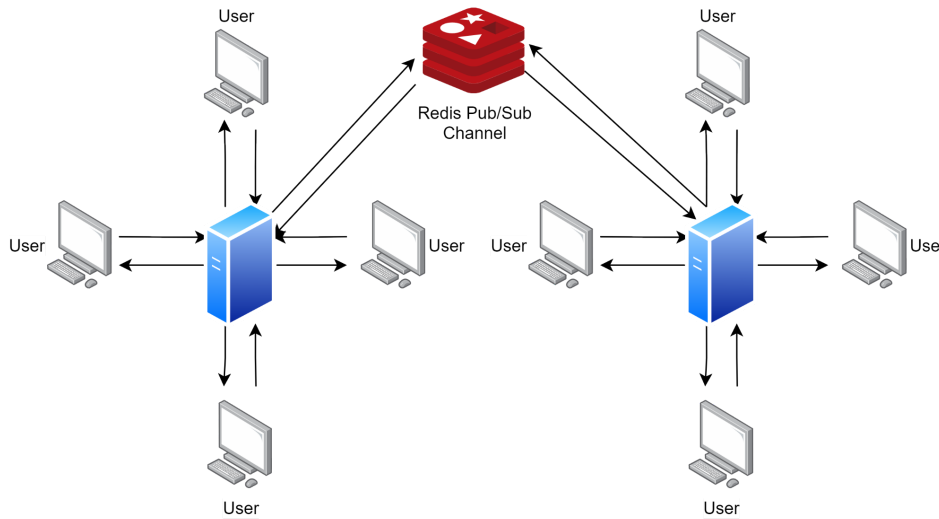
Here is a diagram to depict the one-to-many handshake performed when a video buffers:



Synchronization across servers

When an instance experiences a high amount of network traffic, our loadbalancer can create another instance to help bear the traffic overflow. This means that there is a possibility whereby two users from the same room will be accessing two different servers in two different instances. The message transmission will now require another mediator and that can be done by Redis. Using its Publish-Subscribe (Pub-Sub) channels, we can have the server subscribe to a common channel where they can send and receive messages with other servers.

Here is a diagram to depict how the message passing happen across servers:



As mentioned earlier, users need to go through a handshake process whenever a user buffers. This process requires the server to keep count on how many users have responded thus far and such a count may be trivial to do in a 1-server environment but it becomes an issue when we need to count across servers.

This counter is stored within our Redis mediator and requires an asynchronous access to retrieve it, increment it and set it. If multiple users interleave their retrieval and incrementation of the counter, it can lead to the counter undercounting the users. This can then cause all users to permanently have their playback stuck in pause mode. As such, we must treat the data like a critical section and prevent multiple users from accessing it at the same time.

To do that, we have made use of Redis's ability to run Lua script within the server to construct a special type of Transaction. A transaction is an object that contains a series of Redis commands that will be processed together synchronously with the guarantee that no other commands can interleave them. If a transaction is constructed through Lua script, we can perform data manipulation between the Redis commands as well.

```

releaseReady.lua
11
12 local isEntryFound = redis.call("exists", bufferReadys_buffererId)
13
14 if isEntryFound == 1 then
15     local entryStr = redis.call("get", bufferReadys_buffererId)
16     local entry = cJSON.decode(entryStr)
17     local newEntry = entry
18     table.insert(newEntry.readys, socketId)
19
20 if #newEntry.readys >= newEntry.target then -- # of readys has reached target
21     redis.call("del", bufferReadys_buffererId)
22     redis.call("del", roomHolders_roomId)
23     return ENTRYFOUND_REACHED;
24 else -- Insufficient # of readys
25     local newEntryStr = cJSON.encode(newEntry)
26     redis.call("set", bufferReadys_buffererId, newEntryStr)
27     return ENTRYFOUND_INSUFFICIENT
28 end
29 else

```

This allows any user that finished their buffering to retrieve, increment and set the counter within a synchronous action, eliminating any possible interleaving. Therefore, this eliminates the problem of multiple users stuck in pause mode during buffering.

Handling user disconnections

As mentioned in [4.3.2 Database schema](#), a user's existence in a room is captured within the *Users In Rooms* table in the MySQL database. Whenever the server detects a user disconnecting its websocket, we will remove an entry from the table. However, when multiple users disconnect within a short interval, it causes a deadlock in the database and some users are not removed from that table. As such, when those users attempt to re-enter the room, they will be detected as already in the room and be rejected.

To mitigate this, we make use of recursion to retry the deletion continuously up to 3 times. By then, the transactions that blocked this deletion would have been completed and this deletion will be executed.

4.4.2 Feature 2 - Authentication

There are two different authentication requirements for our application, the **App** and **Reset** Authentication. These authentication are used for the Peerwatch to authorize users to perform certain actions. The **App** Authentication authenticates users that have successfully logged in their account. The **Reset** Authentication authenticates users that have requested an account password reset.

Both of this authentication utilizes Json Web Token (JWT) to authenticate users. The following is a brief overview of how a user is authenticated with a JWT:

1. The user hands over a JWT that is signed by the server with a secret key.
2. The server validates the JWT with the secret key.
3. The server authenticates the user if the JWT is valid.

In order for a user to be authenticated, the user must receive a JWT token signed by the server with the secret key. The following are the description of the API necessary to allow users to receive a JWT and to be authenticated:

API	Description
Login API	This API validates the account credentials entered by the user with reference to the registered accounts stored in the database. If the account credentials match an account in the stored database, the server will sign a JWT with its secret key and the account credentials as its payload, and return this JWT to the user.
Register API	<p>This API ensures account credentials entered by the user on the account registration form meets our specification. The following must be satisfied for the credentials to be valid:</p> <ol style="list-style-type: none">a. The display name entered must contain at least 1 character.b. The email entered must be of a standard email format.c. The email does not exist in the database.d. The password entered must contain at least 8 characters, and a mix of digits and letters. <p>The above specification ensures that users will be able to identify each other with a name of choice, users have entered a valid email that has not been used as it is required for account password resets, and the password is of a decent complexity for account security.</p> <p>Before this account is added into the database, the password is salted and hashed for the reason in F5.2.</p> <p>After the account has been added, the server will sign a JWT similar as in the Login API and return this JWT to the user.</p>
App Authentication API	This API validates the JWT given by the user. The server checks if the JWT is signed with the server's secret key. This secret key is 64 characters long and generated using <code>randomBytes</code> function from the crypto library. The server then indicates whether the user is authenticated depending on whether the JWT is signed by the server.
Password Recovery API	This API checks whether the email address entered by the user matches an email address for an account in the database and sends

	<p>a unique password reset link to the user's email address. This unique password reset link consists of two important parts, the reset ID and the JWT.</p> <p>The reset ID is 16 characters long generated using the same <code>randomBytes</code> function as before, and is mapped to the account's email address. This mapping is stored in Redis to account for the scaling of the application. The JWT is signed using the account's current hashed password as the secret and the email address as its payload. These two parts are then appended to the application's password reset endpoint as such http://peerwatch.ap-southeast-1.elasticbeanstalk.com/reset/resetID/JWT.</p>
Reset Authentication API	<p>This API receives a JWT and resetID from the user and validates this JWT. The server first retrieves the email address from the resetID mapping and the account's current hashed password. The server then checks if the JWT is signed using the account's current hashed password with the correct email address as its payload. The server then indicates whether the user is authenticated.</p>

4.4.3 Design decisions

Throughout the design process of PeerWatch's server, there are mainly two design problems we faced and they are as follows:

Location of the counter

The first problem we faced is where should the counter used during handshake be stored at. The possible locations are the client side, server side and within the Redis database. To decide on an option, we listed out several criterias and verified how each option perform in the criterias through a table like so:

Criteria	Client	Server	Database (Redis)
Implementation complexity	Easy. No matter whether the users are all in the same server or spread across servers, we just have to send the counter and the incremented back and forth like a message	Similar to User, the counter and the incremental counter just have to be send back and forth between servers	Hard to implement as we need to ensure that the access and modification of the counter are done synchronously without other users interfering
Performance stability	Unstable as any transmission of the counter can go	More stable than Client as any transmission of the	Stable as every user will be accessing the same endpoint to

	through an dynamic number of jumps	counter between users of the same room has a static 2 jumps (User -> Server -> User)	access and modify the counter (User -> Server -> DB -> Server -> User)
Handling disconnections	Need to pass the counter to another user and broadcast to all users in the room on who owns the master copy of the counter	Need to pass the counter to another server if all users from a server have disconnected. After that, the server has to broadcast to all servers on who owns the master copy of the counter	Immune to user disconnections as the master copy of the counter will always be in the database

Although having the counter stored on the client or server can be easily implemented, the tradeoff comes in its performance stability and the ability to handle disconnections efficiently. Both require a passing of the counter whenever a client or server disconnects. This will not only take up unnecessary bandwidth but is also an extra effort to implement. Additionally, the number of transmissions we make in either solution may vary in each passing of counter. If the number of transmission increases, it may increase the time taken for the counter to reach the destination and eventually lead to a slower handshake process. On the other hand, the database solution requires no form of pass counter when disconnections happen and the number of transmissions remain static. Therefore, we determine that storing the counter in the Redis database is our best option.

Access Tokens

The second problem we faced was the decision on the type of access token to be used for PeerWatch. The following are the two possible libraries for access tokens we have looked at. Similar to the previous decision, we have listed out several criteria in the table below:

Criteria	Json Web Token (JWT)	Platform-Agnostic SEcurity Tokens (PASETO)
Ease of usage	API provided is easy to use for signing and verifying tokens. Have experience with it from CS3219 OTOT Assignment.	Similar to JWT, the API provided is easy to use for signing and verifying tokens.
Security	JWT's current set of encryption algorithms has known security vulnerabilities. Often misused by developers which allows their applications	Uses a different set of encryption algorithms from JWT that prevents those known security vulnerabilities.

	to be vulnerable to security attacks.	
Library support	Widely known and used by developers for node.js. Library maintained by Auth0 and support for the library is likely to remain for a long time.	Not as widely used by developers for node.js. Library maintained by a single person. This library may lack long term support.

Since both libraries are relatively easy to use, we looked at the tradeoffs between security of the access tokens and its library support. Security of access tokens are important to prevent malicious attacks from happening and PASETO beats JWT in this criteria as it was developed to eliminate the security flaws present in JWT. However, the library for JWT is widely used and maintained by a group in Auth0 and not the same can be said for PASETO. The lack of support of a library for access tokens could expose our application vulnerable to security attacks. Hence, we chose JWT as the access token for our application as we can decide on the encryption algorithm that does not have the security flaws and be confident in preventing our application from security attacks with the long term support of the library.

4.5 Frontend design

In this section, we will be discussing how the architectural design from [4.1 Architectural Design](#) affects the frontend's code structure. We will then discuss frontend's design in the context of the features within PeerWatch.

4.5.1 Code structure

Following the MVC pattern in [4.1.2 Nested MVC](#), the components of frontend can be categorized into Model, View and Controllers. The following table lists a few examples of components in each category:

Model	View	Controller
User context (UserContext.js)	Chatbox message display (ChatContent.js)	Landing page (Landing.js)
Any usage of useState hook (eg. JoinRoomPanel.js, LoginPanel.js, and RecoveryPanel.js)	User list (Watchmates.js)	Room page (Room.js)
	Logout button (LogoutButton.js)	Video player (VdeoPlayer.js)

As long as we ensure that any interaction from Model components and View components goes through the Controller components, we can reap the benefits of the MVC pattern. This helps our application to maintain Separation of Concerns and components from different groups are

modular from one another. Additionally, it becomes easy for us to extend and build new features on PeerWatch.

An example of an extension is [F5.4](#) where we introduce the password reset feature onto our Landing page. We implemented a RecoveryPanel component to capture the inputs necessary to reset an account's password. This component can just be added to the Landing page and has its input submitted by a separate button component.

4.5.2 Feature 1 - Video Synchronization (Client)

As a client in the video synchronization process, it has five responsibilities:

- 1) Connecting user to the appropriate socket and room
- 2) Retrieving a room's status and settings
- 3) Keep playback time in sync with the host's playback time
- 4) Handle incoming messages while updating other clients
- 5) Cleanup after the user disconnects

For responsibility (1), (2), (3), and (5), they can be handled by making Axios requests to the server's API or via the message transmission process mentioned in [4.4.1 Feature 1 - Video client Synchronization \(Server\)](#). However, we need to handle responsibility (4) explicitly.

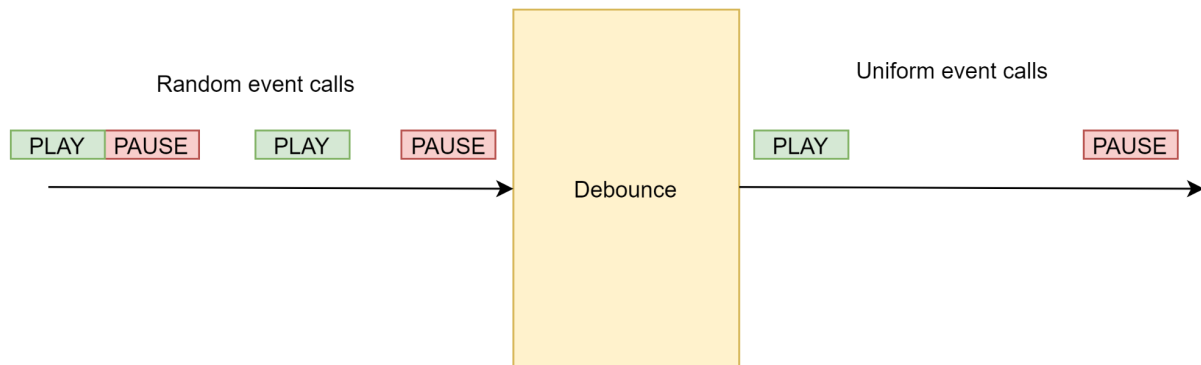
Handle incoming messages while updating others

Incoming messages can serve many purposes such as to pause the video, resume the video, or to receive a chat message etc. These actions often require updating of local or global state for the client and results in a re-rendering of the UI. Such re-rendering can be expensive and thus, state updates and re-rendering are done asynchronously.

This asynchronous nature can lead to clients being in different playback states for a short interval. This opens up a sequence of events that can lead to desynchronisation. Here is one such sequence:

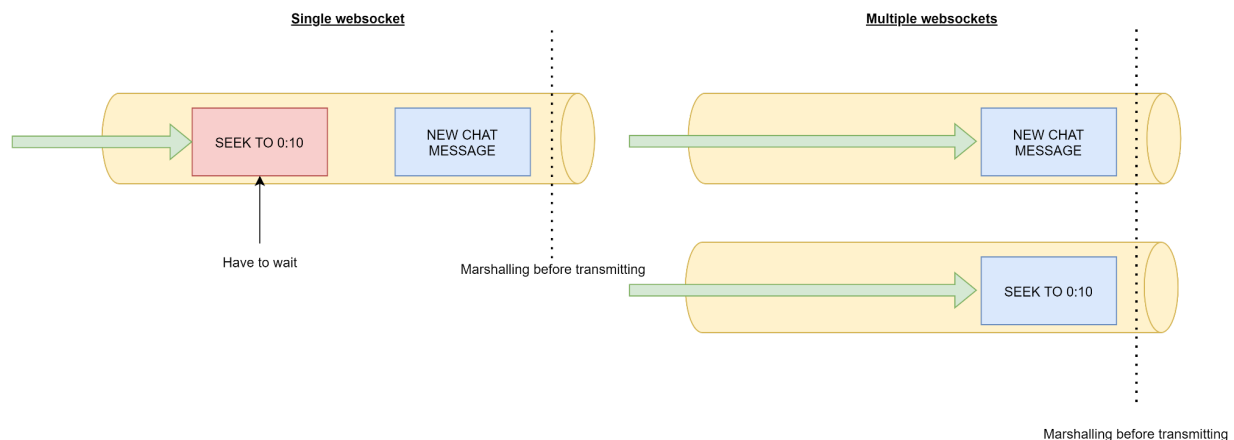
1. There are two clients A and B in the same room
2. Client A pauses the video and the message to pause other client's videos gets sent
3. While the message is sending, Client A resumes the video again and another message to resume other client's videos get sent
4. The two messages can interleave and cause client B to resume and pause in the order
5. So, client A is now playing while client B is pause and thus, both client has desynchronised

To mitigate this, we opted to use a JavaScript pattern called Debounce. This pattern essentially allows us to filter out the repeating call of a function within a time interval and only execute the final call. In this case, the initial pause will not result in a pause message being sent and only the result message will be sent. As such, a quick succession of pausing and playing will just become playing. Here is a diagram to illustrate how that pattern will work if a user continuously plays and pause the video for a direction::



4.5.3 Feature 2 - Live text chat

For live text chat, we made use of the same setup mentioned in [4.4.1 Feature 1 - Video Synchronization \(Server\)](#) and we shared the same Redis instance to transmit messages. However, the message transmission for live text chat and room management are done in a separate socket and socket.io room. This is to ensure that video synchronization can get its own separate connection to the server. If it shares its connection with other types of messages, it will be possible for those messages to block the synchronization message and this can affect the user's viewing experience. The following diagram depicts how a single connection can get messages block while multiple connections will not get block:



4.5.4 Feature 3 - Authentication

The client has the responsibility of authorizing users to view certain pages of Peerwatch depending on the authentication status of the user. This is to prevent users from having unauthorized access to certain actions. The following are pages that requires the client's authorization:

1. Landing Page
2. Room Page
3. AccountReset Page

For the clients to perform the above actions, it is handled by making Axios requests to the APIs as mentioned in [Section 4.4.2](#). The following actions will be performed for each page and panel to allow users to be authenticated and authorized:

Page/Panel	Description	Actions Performed
Login Panel	Contains a form for users to enter their registered account credentials.	An Axios request to the Login API with the account credentials entered by the user. If the API indicates that the user's account is valid, the JWT received will be stored in the client's local storage.
Register Panel	Contains a form for users to enter the account credentials they wish to register with.	An Axios request to the Register API with the account credentials entered by the user. If the API indicates that the user's account credentials meet the specifications, the JWT received will be stored in the client's local storage.
Recovery Panel	Contains a form for users to enter their email address that was registered with their account to request for a reset link	An Axios request to the Password Recovery API with the email address entered by the user. An email with the password reset link will be sent to the user if the email address is determined to be valid by the API.
Landing Page	Allows users to create or join a room.	The JWT will be retrieved from the local storage and an Axios request to the App Authentication API with the JWT. If the user is authenticated, the user is authorized to create a room and join a room. If not, the Login Panel will be displayed.
Room Page	Allows users to input Youtube video links, watch Youtube videos together, and chat with each other via live text chat.	Similar to the actions done in the Landing Page. If the user is authenticated, the user is authorized to view the Room Page. If not, the user will be redirected to the PleaseLogin Page.
AccountReset Page	Contains a form for users to enter their new password for their account.	The resetID and JWT will be extracted from the password reset link and an Axios request will be made to the Reset Authentication API with those information. If the user is authenticated, the user is authorized to access the AccountReset Page. If not, the user will be redirected to the NotFound Page.

4.5.5 Design decisions

Throughout the design process of PeerWatch's frontend, there are mainly two design problems we faced and they are as follows:

Broadcast timing by interval or by action

The first problem we faced is how often should the latest timing of a video be broadcasted from the host to the other users. The host can broadcast the timing once per time interval or once per user event (eg. User forwarding the video). To decide on an option, we listed out three criterias and analyze how each option perform through a table like so:

Criteria	By interval	By event
Potential to desynchronize	Lower potential as we can fallback to comparing the current playback time with the latest received timing to detect a desynchronisation and recover from there	Higher potential as we cannot implement any fallback mechanism without having a constant source of latest timing
Implementation complexity	Lower complexity as we just need to continuously broadcast until the video is paused or completed	Higher complexity as we have to detect the received event requires a synchronisation or not
Performance	Worse performance as the constant stream of timing messages from all the hosts can flood the socket and cause delays before the timing reaches the destination	Better performance as it reduces the amount of network traffic and the occurrence of delays before the timing reaches other clients

Despite the better performance by sending timing on an event-by-event basis, the lack of fallback mechanism may make fulfilling [NF1.1](#) difficult. The implementation of this option is also more complex and maybe fragile if more events are introduced to the synchronization process. Since we have implemented [NF1.4](#), the amount of traffic we can expect per socket should be a manageable amount and the delay will not be significant enough to be noticeable. Therefore, our broadcasting of playback timing is done on a time interval basis

Single socket or Multi socket connection

The second problem we faced is whether the client should be connected to a room by one or more sockets. The client can connect to a room with one socket where the messages for video synchronization, live text chat, and room management are all transmitted through it. The client can also opt to send those messages through multiple sockets but with one socket solely for video synchronization. To decide on an option, we listed out three criterias and analyze how each option perform through a table like so:

Criteria	Single socket	Two Sockets
Implementation complexity	Easy to implement as every client just need to establish one websocket connection in a room with the server	Harder to implement as every client need to establish two websocket connections with each in a different room with the server and messages need to be send through the right websocket
Performance	Worse performance as performance critical messages (video synchronization related) can get delayed by any other message in the same socket due to marshalling and unmarshalling	Better performance as performance critical messages (video synchronization related) will not be delayed by any other message in the same socket due to marshalling and unmarshalling
Separation of Concerns	Weak as the messages meant for different features are all transmitted through a singular socket	Stronger as the messages for video synchronization are kept separated on a socket while the other messages on another socket

Even though a one socket setup is easy to implement, its performance takes a toll due to its potential marshalling delay caused by other messages onto performance critical messages for video synchronization. Additionally, it is natural to separate these performance critical messages to its own separate socket to gain a stronger separation of concerns. Therefore, we went with a two socket setup where each client will make two websockets connections with the server anytime it connects to a room.

4.6 Technical choices

In this section, we will be listing the different technical and tool choices we made throughout the planning and development of PeerWatch, providing justification when the choice is significant to the application. These choices include what tech stack was used, what tools and libraries were used and what code standards we follow.

4.6.1 Tech stack

The stack used for PeerWatch is based on the MERN stack but with the difference in the choice of database.

Frontend	React
Backend	Express.js

Database	MySQL, Redis
Deployment	AWS Elastic Beanstalk, AWS RDS, AWS EC2
Pub-Sub Messaging	Socket.io, Redis
Cloud Providers	AWS
CI/CD	Github Actions
Monitoring	PM2
Orchestration Service	N.A
Project Management Tools	Github Issues

Since some of our team members have previous experience with MERN stack, that is what we went with for PeerWatch. The adjustment we made is to replace MongoDB with MySQL database and as discussed in [4.3 Database design](#), it not only provides a more applicable relational nature between data, but also provides the data consistency and availability we need in an application like PeerWatch.

AWS is our choice of deployment, scaling, and availability support for PeerWatch due to how integrated these services are. Once we deploy to Elastic Beanstalk, we only have to adjust the configuration to customize and achieve the scaling behavior we are looking for. It can also be linked with other AWS services like AWS RDS that manages our database instances while providing backups and redundancy. If we seek to set this up by ourselves, it will definitely take up a large section of our development time.

4.6.2 Libraries and Packages

Here is a list of tools, library and packages used for the development of PeerWatch:

Libraries / Packages	Purpose
mysql2	A library to connect a Node.JS application to a local or external MySQL database. Necessary for us to use MySQL database in PeerWatch
redis	A library to create connections from a Node.JS application to a local or external Redis storage. Necessary for us to provide F1.2 across servers after scaling up
axios	A library that allows Node.JS applications to make HTTP requests towards a server. Essential for PeerWatch client to interact with the server and database.
socket.io	A library that allows Node.JS applications to communicate with other applications via websockets through a server. Required for

	PeerWatch to fulfill F1.2 and F3.1
jsonwebtoken	A library that allows Node.JS applications to encapsulate data into a JWT token for transmission. Essential to fulfill NF5.1
bcryptjs	A library that allows Node.JS applications to encrypt critical data such as account passwords. Helps to encrypt an account's password before storing into our database and thus, fulfilling NF5.2
uuid	A library that allows Node.JS applications to generate universally unique identifiers according to the RFC4122 standards. Helps to generate unique IDs that are used for F2.5 and F5.4
nodemailer	A library that provides Node.JS applications with the ability to send emails. Essential to PeerWatch for fulfilling F5.3
mui	A library that provides React applications with pre-made and customization UI components. Allow us to quickly construct UI for requirements like F4.1 , F4.2 , and F4.3

4.6.3 Coding standards

Instead of going with any formal code standards like Google JavaScript Style Guide, we have decided to adopt a less formal style used by an auto formatter called Prettier. Our team believe that the benefits it provided far outweighs its con and here are two main reasons we identified:

Presentable codebase

As our team does not have much experience in a JavaScript based project, our codebase may not be as presentable as an experienced team even if we put the time into following a code style guide. Through auto-formatting, our codebase will not only be more presentable but also consistent.

Accelerate development

To properly adopt and adhere to a coding standard, a team has to spend time and mental energy to maintain their codebase. Through Prettier, by using keyboard shortcuts, we can quickly reformat to adhere to Prettier's style guide. This not only allows our team to put more focus on the actual development of the program, but also ensures we do not miss out any formatting mistakes due to human errors.

4.7 Development process

4.7.1 Project plan

We have decided to use a mix of depth-first and breadth-first iterative development cycle where we split the development into six iterations of a week long. In the earlier iterations, we concentrated on developing a minimal viable product of PeerWatch. We continued with depth-first iterations to complete the Video Synchronization feature. Afterwards, we return back

to breadth-first iterations to finish up the remaining features. We have indicated the functional requirements for the feature in each iteration that we will have to fulfill so that we can easily delegate tasks for every member.

4.7.2 Development schedule

Iteration	Member	Requirements	Task
1	Qin Liang	F1.1 , F2.1 , F2.2 , F4.1	<ol style="list-style-type: none"> 1. Setup base project 2. Create UI mock up for PeerWatch 3. Create room page (F2.2, F4.1) 4. Create button to join a room (F2.1) 5. Implement playing of Youtube video with link in the room (F1.1)
	Yeou Tzer	F3.1 , F3.3	<ol style="list-style-type: none"> 1. Learn socket.io 2. Implement live text chat in rooms using socket.io (F3.1) 3. Implement the announcement of user joining a room (F3.3) 4. Setup mocha and chai test tools for project
	Marcus		<ol style="list-style-type: none"> 1. Design database schema required for Peerwatch
	Zen		<ol style="list-style-type: none"> 1. Learn AWS EC2 deployment 2. Deploy the base project onto AWS EC2
2	Qin Liang	F1.2 , F1.4 , F1.5 , F1.6 , F4.2 , F4.3	<ol style="list-style-type: none"> 1. Implement syncing of Youtube videos with users in the room using socket.io (F1.2, F1.4, F1.5, F1.6) 2. Create page for creating and joining rooms (F4.2) 3. Create page for logging in and registering an account (F4.2) 4. Create page to change account password (F4.3)
	Yeou Tzer	F2.8	<ol style="list-style-type: none"> 1. Implement viewing of users in the room using socket.io (F2.8) 2. Setup CI for tests
	Marcus	F2.1 , F2.2 , F2.4	<ol style="list-style-type: none"> 1. Create MySQL script for database schema 2. Create APIs for room management (F2.1, F2.2, F2.4)
	Zen		<ol style="list-style-type: none"> 1. Setup AWS RDS for MySQL database 2. Setup CD to deploy project onto AWS EC2 3. Setup PM2
3	Qin Liang	F1.2 , F1.3 ,	<ol style="list-style-type: none"> 1. Continue with implementation of sync

		F1.4 , F1.5 , F1.6 , F4.4 ,	<p>Youtube videos with users in the room (F1.2, F1.4, F1.5, F1.6)</p> <ol style="list-style-type: none"> 2. Implement retrieval of room's playing video (F1.3) 3. Implement ability to navigate to previous page (F4.4)
	Yeou Tzer	F5.1 , F5.2 , F5.3	<ol style="list-style-type: none"> 1. Create and integrate account creation API with display name, email, and password validation (F5.1) 2. Create and integrate account login API with account validation (F5.2) 3. Create API for account password recovery via email (F5.3) 4. Create API for account password reset (F5.3) 5. Update CI to include MySQL in github actions
	Marcus	F2.1 , F2.2 , F2.4	<ol style="list-style-type: none"> 1. Setup AWS RDS database with schema 2. Continue with creation of APIs for room management (F2.1, F2.2, F2.4)
	Zen		<ol style="list-style-type: none"> 1. Learn to deploy project onto AWS Elastic beanstalk
4	Qin Liang	F4.3 , F4.5 , F4.6 , F4.7 , F4.8 , F2.1 , F2.2 , F2.3 , F2.4 , F2.5 , F2.6 , F2.7	<ol style="list-style-type: none"> 1. Create account recovery page (F4.3) 2. Create room full fallback page (F4.5) 3. Create room not found fallback page (F4.6) 4. Create not logged in fallback page (F4.7) 5. Create page not found fallback page (F4.8) 6. Implement creation of room with API (F2.1) 7. Implement joining of room with API (F2.2, F2.6, 2.7) 8. Implement leaving of room with API (F2.3, F2.4)
	Yeou Tzer	F5.3	<ol style="list-style-type: none"> 1. Integrate account password recovery and reset API with frontend (F5.3) 2. Start writing tests for APIs with mocha and chai
	Marcus	F3.2 , F5.1 , F5.2 , F5.3	<ol style="list-style-type: none"> 1. Implement identification of user in text chat via display name (F3.2) 2. Integrate database with current account management API (F5.1, F5.2, F5.3)
	Zen		<ol style="list-style-type: none"> 1. Setup CD of project onto AWS Elastic beanstalk 2. Set scaling configuration for AWS Elastic beanstalk

5	Qin Liang	F2.9 , F2.10 , F2.11 , F2.12	<ol style="list-style-type: none"> 1. Implement room settings configuration for host (F2.9, F2.10, F2.11, F2.12) 2. Refactor code to include socket.io redis due to scaling
	Yeou Tzer		<ol style="list-style-type: none"> 1. Refactor code to include redis due to scaling 2. Finish writing tests for APIs with mocha and chai 3. Update CI to include redis in github actions
	Marcus		<ol style="list-style-type: none"> 1. Update database schema to remove unused fields.
	Zen		<ol style="list-style-type: none"> 1. Finish configuring the appropriate settings on AWS to satisfy NFR
6	Qin Liang		<ol style="list-style-type: none"> 1. Optimize application to satisfy NFR 2. Report writing 3. Bug fixes
	Yeou Tzer		<ol style="list-style-type: none"> 1. Report writing 2. Bug fixes
	Marcus		<ol style="list-style-type: none"> 1. Report writing 2. Bug fixes
	Zen		<ol style="list-style-type: none"> 1. Report writing

4.8 Project management

As mentioned in [4.7.1 Project plan](#), our development process is weekly iteration based. In every iteration, we set out the tasks to be completed by the next iteration and raised them as issues in GitHub. These issues are then managed via GitHub Project to allow team members to easily understand how far we are to completing an iteration. Bugs found were tracked as issues and through the GitHub Project interface, any team member with extra bandwidth can pick up the issue and attempt to resolve them.

5 Possible improvements and enhancements

With the limited development time for PeerWatch, there are many aspects that we desire to improve or enhance upon and here are three of them:

Availability and scalability of Redis instance

As we expect our user base to be quite small, we deem it to be sufficient to run only one Redis instance for message transmission. However, improving its availability and scalability will definitely help PeerWatch's operation in the long run. For availability, we can make use of the existing Redis Sentinel feature to maintain several replicas that can serve as failover if the master replica broke down. For scalability, we can scale the Redis instance with the same

method described in [4.2.2 AWS Elastic Beanstalk \(ELB\)](#) where we use Elastic Beanstalk with network traffic as the scaling trigger.

Implement HTTPS

By implementing HTTPS, the HTTP requests and responses from PeerWatch will go through a SSL/TLS connection which encrypts the payload. This allows the transmission of user confidential data to be more secure.

Support more YouTube features

There are several features that the YouTube player provides but PeerWatch has yet to provide synchronization for. These features include playing a video playlist and playing suggested video from the player. Since playing a playlist involves additional playback actions like switching between videos from the playlist, we had to pass on implementing this within our development cycle. When a user plays a suggested video, we are unable to detect the URL change and therefore, cannot request the other users in the room to play that video. However, it may be possible using the lower level functionality of YouTube API and is definitely worth looking into.

6 Reflections and Learning points

Although our development cycle for PeerWatch was relatively short, it taught us the importance of performing prior research and time management:

Performing prior research

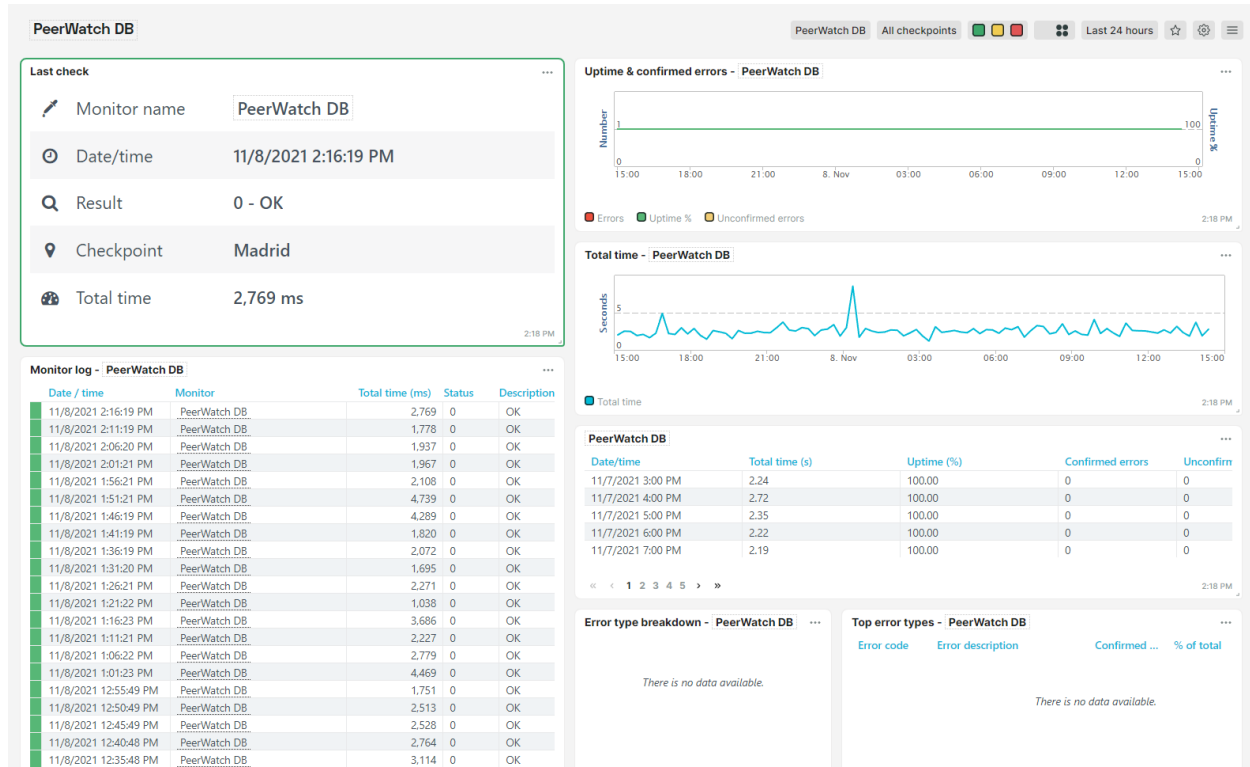
In our prior research, we place a priority on FR related topics instead of NFR. This combined with how we only focus on NFR in the later iterations (refer to [4.7.2 Development schedule](#)), resulted in us not recognising the issues that scalability can introduce. When we have to transmit messages across all the servers due to scaling, we did not plan on having a separate Redis instance as the message mediator. However, it was necessary and it required us to ensure the server's interaction with it was synchronous. These changes were made relatively late in development and consume our buffer time that is supposed to be for debugging and improving UI. Therefore, in future projects, we need to have a wider approach towards researching for an application.

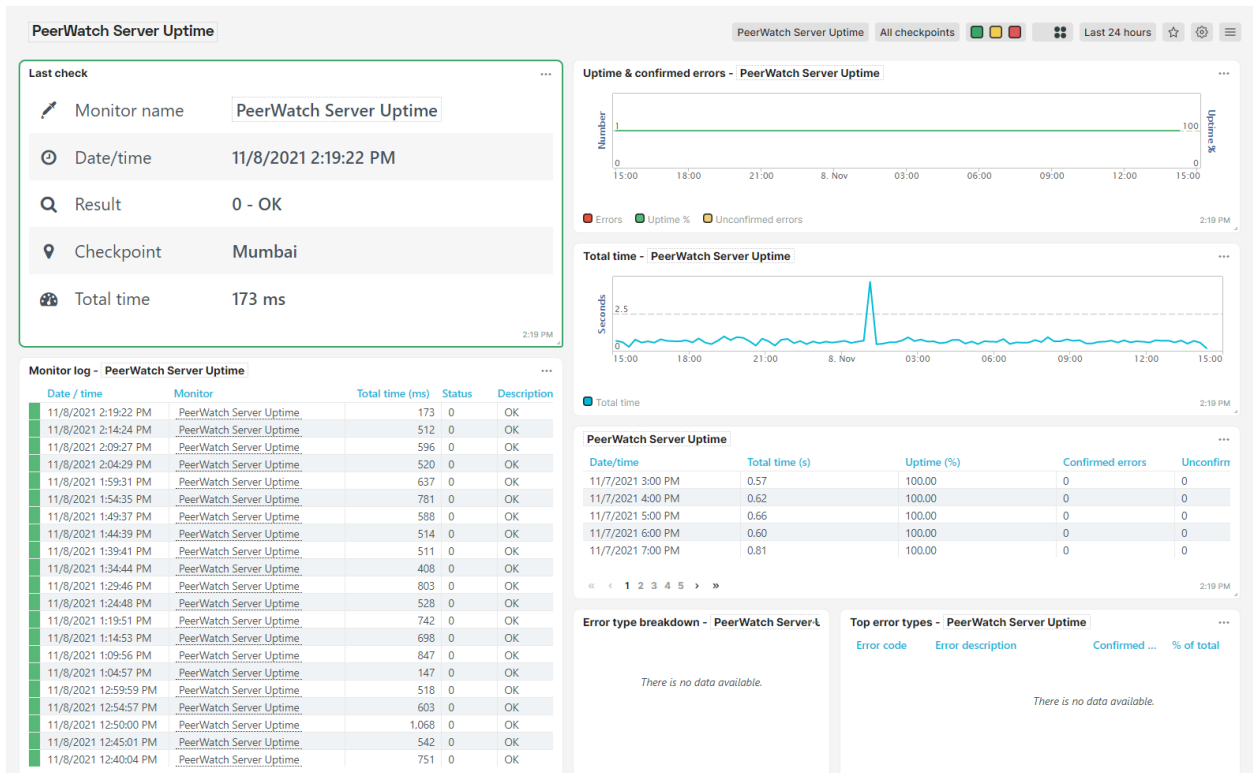
Importance of time management

When we were planning out the iterations, we did not fully consider the level of commitment that each team member can make as our workloads were different. This combined with us having to pick up new technologies like AWS, Socket.io, and Redis, we ended up having to not only delay features into later iterations, but also remove some good-to-have features. We may be able to improve this in the future through better scoping of the project and also better understanding of workload differences.

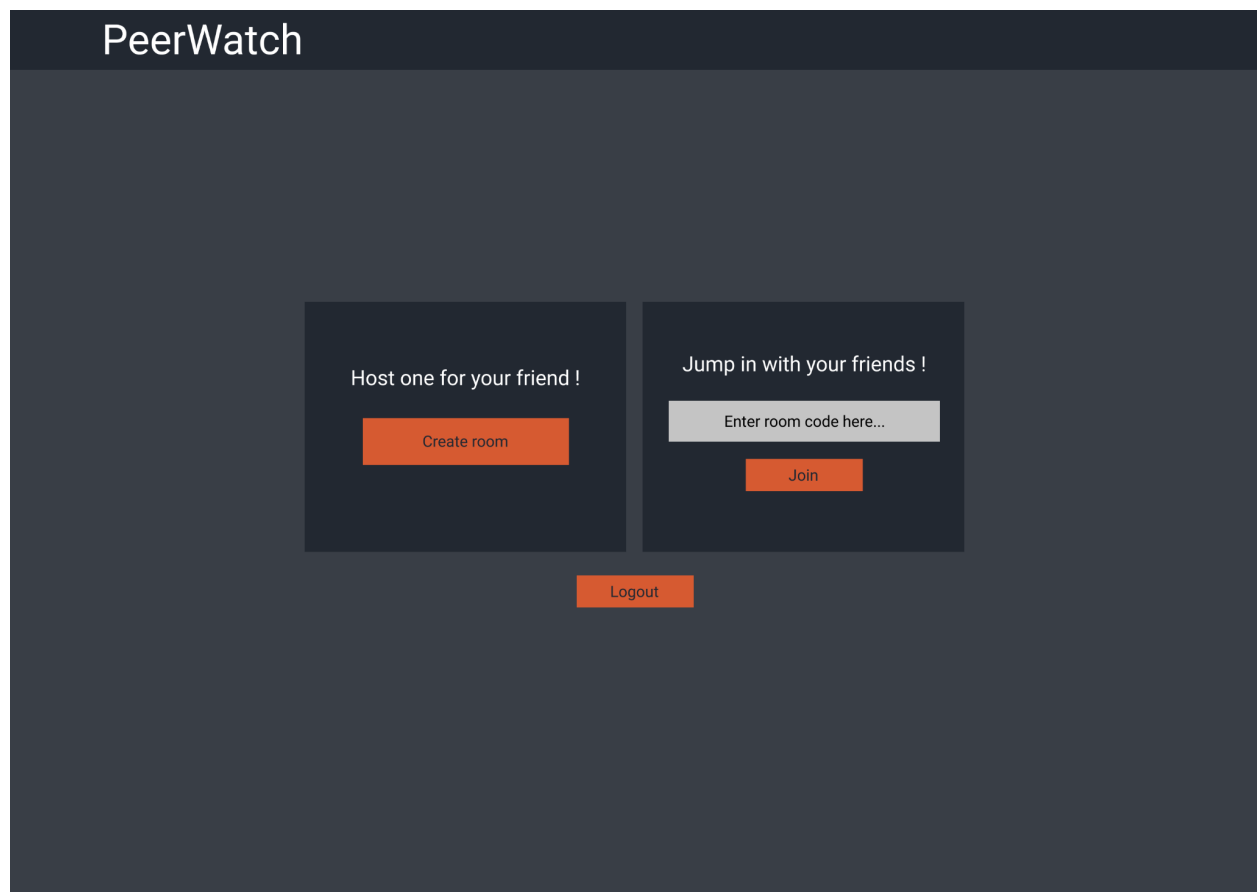
6 Appendix

6.1 PeerWatch's uptime through Uptrends



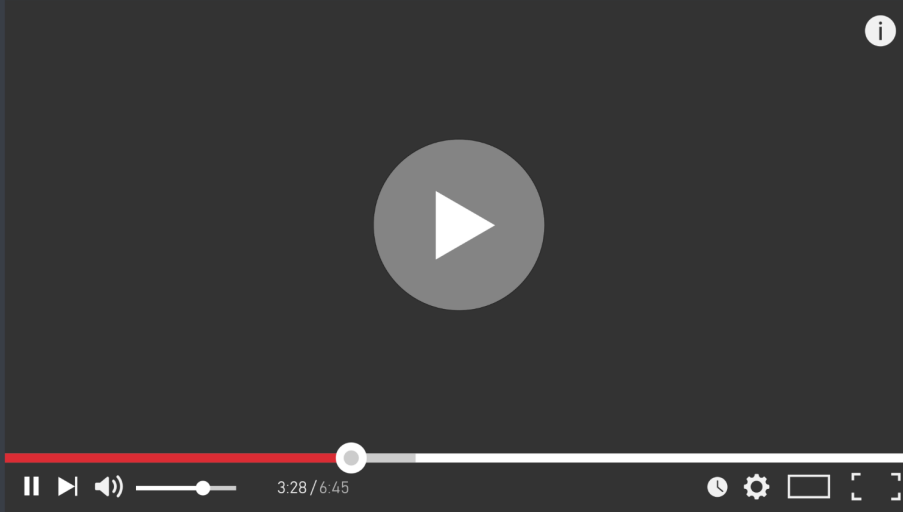


6.2 UI Mockup



< Back

PeerWatch



Enter video link here ...

Watchmates (4)

User1 (Host)

User2

User3

User4

User1 (Host): XXXXXX

User2: XXXXXXXXXXXXXXXXXXXXXXXX

User3: XXXXXXXXXXXX

User4: XXXXXXXXXXXXXXXXXXXXXXXX

Chat here...

Submit

Room Settings

Only shows up for host
and opens a popup

Looks like you have forgotten your password?

Don't worry XXX, we got you covered! Just enter the new password you want to have and we are ready to go.

New password

Re-enter password

Confirm

Cancel