Twitch Clip Recommender

JT Vinolus: jvinolus@scu.edu,
Preeti Kakuru: skakuru@scu.edu,
Abhishek Gowda G R: agejjalagererajanna@scu.edu ,
Brandon Quant: bquant@scu.edu,
& Yash Bhargava: ybhargava@scu.edu

COEN 241
Spring 2022
Professor Choi
June 8, 2022

Abstract

Twitch is one of the biggest leaders in live streaming content: its main focus is providing a platform for video game live streams which includes broadcasts of esports events. Rivaling the likes of YouTube Live, Twitch as of February 2020 had about 3 million broadcasters and 15 million daily active users, generating terabytes of data annually. Every day, broadcasters are generating new and popular clips from a variety of games such as Counter-Strike: Global Offensive, Valorant, and League of Legends: this makes browsing for the most popular clips a hassle. Our Twitch Clip Recommender solves this problem by providing a platform that displays to the users, based on a user selection of categories, the most popular clips of chosen games from a variety of broadcasters, all while without having the user go through the hassle of finding the most popular clips on their own.

Table of Contents

1.0 Introduction

Over the years, as technology has continued to advance and improve along with the emergence of the internet, it has made devices such as smartphones and personal computers cheap and accessible to a wide range of people. People now have actively embraced the use of personal computers and devices as a means of recreation, other than just using them for tedious mathematical computations or work. One of the most emerging fields in this section is the use of PCs for playing video games.

As more and more people have access to PCs, the demand for recreational and competitive video gaming has grown tremendously. People have actively started playing games and have found various means to share their favorite moments online with other like-minded people as well. One of the ways that people have started sharing their content is via live streaming and the recording of clips.

The Leading website in enabling people to share their favorite clips and live streams is Twitch, and as of February 2020 had about 3 million broadcasters and 15 million daily active users, thus generating terabytes of data annually. As there are so many broadcasters for each category of video games like action, racing, sports, and more, it has become a hassle for a user to find the most viewed and popular clips according to the game of their choice, forcing the user to go channel by channel to find clips that appeal to them. Our project aims to provide a solution to this problem.

Our project, Twitch Clip Recommender, enables users to create their own account via the use of an interactive website, thus providing them with login and logout functionality, and setting their preferences for the clips of games they would like to see. After the users set their preferences, the preferences are stored in an online database for future use when a user logs back into the platform. Once users set their preferences, these preferences are sent to our unique recommendation system built using python, which scours the clips stored in Twitch via querying the Twitch API and presents them to the user: this saves the users from the hassle of going through broadcasters and searching for clips on his own, by providing him or her with a curated feed of clips, altered according to his or her personal preferences.

Both the frontend, where the user interacts with the website, and the backend portion, where the information passed by the user is stored and processed, of our project, are hosted on Amazon EC2 instances using docker images, which makes our project versatile and portable.

2.0 Background and Related Work

Our project is inspired by our own experience of using Twitch to browse and watch clips and live streams by various broadcasters. As regular users of twitch, we felt the need to streamline the process of searching for popular clips from various broadcasters and present the clips straight to the user, curated to his own personal preferences. Currently, Twitch does not offer this functionality as it forces the user to manually search the website, which affects the user experience.

During our research, we have concentrated on our own personal user experience while also taking inspiration from Twitch itself and websites such as YouTube, Facebook (Meta), and Instagram which all personalize the content their users view by using the posts they ‘like’, or the channels they ‘subscribe’ to in the case for YouTube. These ‘likes’ and ‘subscriptions’ are linked and saved to the users’ accounts, enabling them to be used every time the user logs in.

Likewise, our project follows a similar methodology, where we provide our users with the functionality of creating their own account, personalizing it based on the games for which

they want to see clips of, and by providing them with a curated selection of clips from our recommendation system. These choices are saved for later use so that when users log back in, they do not need to select their preferred games again. Thus, our project aims to build upon Twitch's own functionality and provides avid users of Twitch with a unique way of getting a curated feed of clips without the hassle of searching for them manually on their own which saves time and effort.

3.0 Approach

3.1 High-Level Architecture

To begin building our project, we started from the very basics. At first, we wanted to have a skeleton of our project ready, which is where the client will interact with our project in order to send and receive information. That is, our team first began with planning and designing the frontend portion of our project. As seen in images 3.1, 3.2, and 3.3, our frontend consists of a dedicated login and signup page, a dedicated game tags selection page, where the user selects the games that he or she wants to see clips of, and finally a dedicated clips display page where the clips that are sent from the Backend's Recommendation system will be displayed for the user to watch. All of these pages were developed using HTML, CSS, and JavaScript.

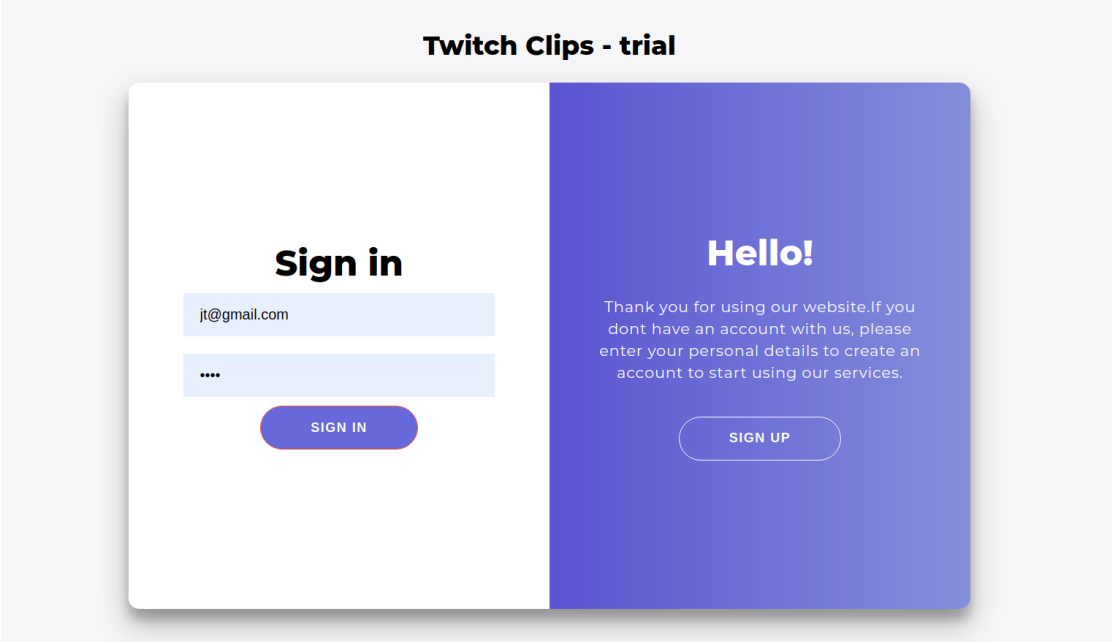


Image 3.1: SignIn/SignUp Page

Once the skeleton of our project was done, we moved on to design the backend of our project. For our backend, we chose to use Python and Flask, due to their flexibility, scalability, and ease of use when developing simple web applications. As for our database, we have incorporated the use of SQLite as it is best suited for use with small to medium-sized websites, which made it perfect for our project. Our backend contains different Python files with different uses; we have a file for user authentication, a file for our database creation and queries to the database, a file for loading Flask variables, Twitch API variables, and refreshing tokens on the application startup, a file for getting clips from Twitch API, and the recommendation system file.

We also have built our own recommendation system, which takes in the user's email and returns Twitch clips to our Frontend. Our recommendation system will first query the database to get the selected tags for the user. For each tag, we will query our database to get the latest clip by timestamp. If the difference between the timestamp of the clip and the current timestamp is larger than 10 minutes, we will query new clips using the Twitch API and send those clips to the frontend. We will also store these new clips in our database, ensuring that we always have new clips for the user to watch. If our timestamp difference is less than 10 minutes, we will instead query the clips from our database and return them to the frontend.

Once each of the components has been built, our team containerized the frontend and backend separately using Docker and deployed the docker images on the AWS EC2 instances.

3.2. Database Schema

Since the platform is storing both user information and Twitch clip information, it needed a schema to organize the data. For our application, we used two tables, one for the user information/profile and one for the clip information as seen in Figure 3.1.

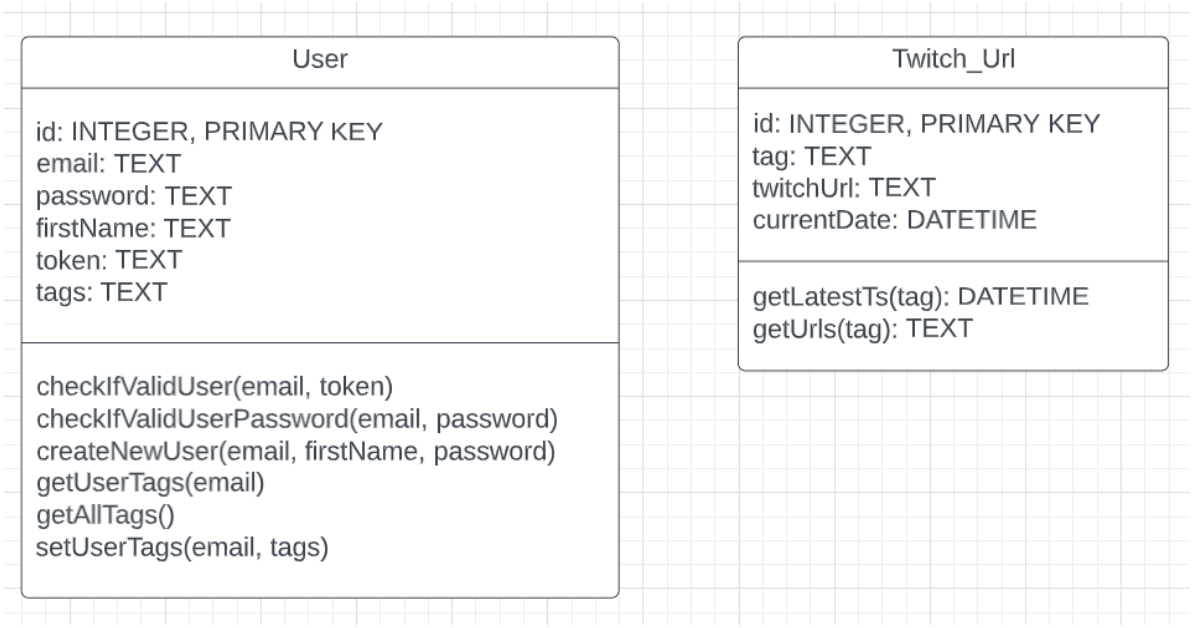


Figure 3.1: Database Schema

The User table consists of user account credentials, where our password is hashed to provide security. In addition, we store the user-specific token and the tags that a user has selected. We have also provided functions to allow for easier user authentication and the update and retrieval of tags. Our TwitchUrl table holds the necessary information for a specific Twitch clip. This table will continuously be updated when running our recommendation system, thus we need the currentDate field to keep track of newer clips. We have also implemented functions in the TwitchUrl table to be used in the recommendation system.

3.3. Software Architecture

Our software architecture can be easily understood with the help of the following diagrams: Figure 3.2 and Figure 3.3. Our software architecture diagram shows how the different software components in our system communicate with one another. Given that our frontend and

backend are hosted on separate Amazon EC2 instances, the two communicate with each other using HTTP POST and GET requests, where messages are token-based. This allows for more security since we do not pass sensitive user information in our REST API calls. The backbone of our backend, Flask, communicates with our database, SQLite, and recommendation system. Flask is able to query and insert into SQLite for user account creation and authentication. Additionally, Flask is able to call the recommendation system, which returns a list of Twitch URLs to be displayed on the Frontend. The recommendation system is able to query and store clip information into our database as well as query newer clips using the Twitch API. This allows users to continuously see new clips during every user session. Figure 3.3 gives a detailed explanation of the flow of our application and more information on how REST calls were implemented can be found in the appendices under *Description 3.1: REST API Calls Between The Frontend and Backend.*

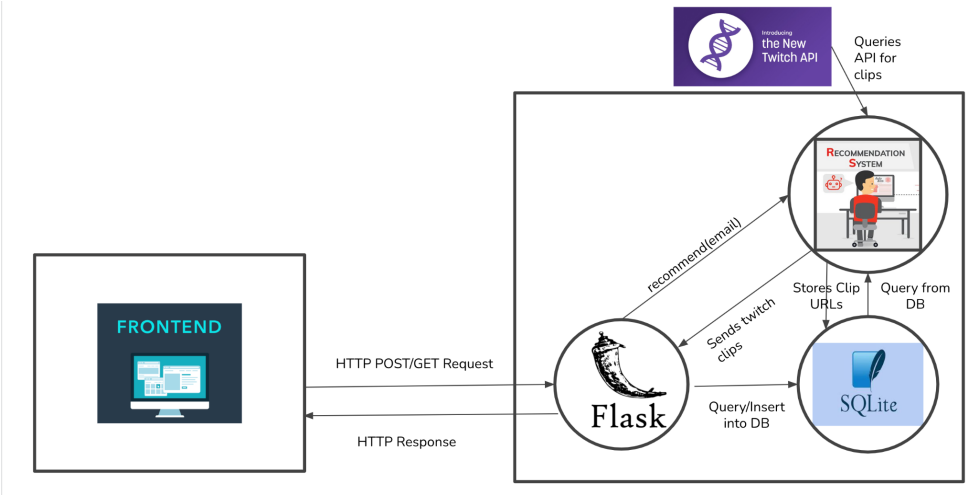


Figure 3.2: Software Architecture

3.4. Software Architecture Flow

When the application is first loaded on the browser, the user is directed to the User Login/Sign Up page. In the software architecture flow, we have two separate flows: one for the creation of a new user, and the other for an existing user.

For new users, they will sign up by entering their email, name, and password. This information is passed to the backend via a POST request, where the Flask application will insert the user credentials into the database. The user will then be redirected to the tag selection page. On this page, the user is able to select the tags they wish to view clips for. Once the tags have been selected, the tags are added to the user's profile in the database. Then, Flask will call the recommendation system for a list of Twitch clip URLs. For each tag, the recommendation system will first query the clip with the latest timestamp. If there is a large enough time difference between the clip timestamp and the current timestamp, the recommendation system will query for new clips using the Twitch API. If the time difference is small enough, the recommendation system will query the database to return the list of Twitch clip URLs. These clips will be passed to the Frontend to be displayed on the clips display page.

For an existing user, the user will login using their email and password. The user credentials are passed to the backend and will be authenticated by comparing with the user information in our database. Once the user has been authenticated, Flask will query our database to get the previously selected tags from a previous session. These tags are then passed back to the frontend and will be preselected when the user is redirected to the tag selection page. As an existing user, the user is able to add, remove, and keep existing tags. The selected tags will be updated in the user's profile in SQLite. Then the recommendation system will be called and will use the same logic as the scenario with a new user to return a list of Twitch clip URLs; these clips are sent to the frontend and displayed on the clips display page.

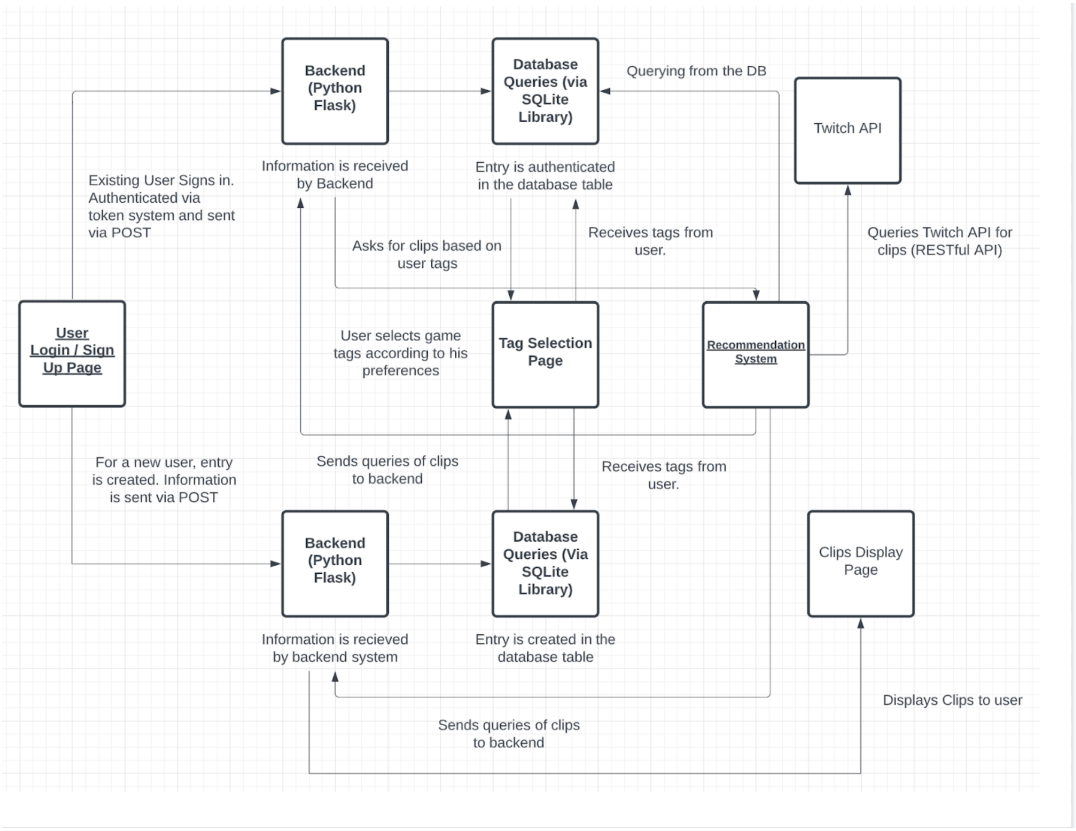


Figure 3.3: Software Architecture Flow

3.5. Routing

Since Twitch requires a site that wants to display its clips as video embeds to have a domain name and be HTTPS encrypted, traffic could not be passed simply unencrypted from one server to another. To tackle this, our platform was divided into two hosted zones. As seen on the left-hand portion of Figure 3.5 in the Appendices, the Frontend Docker Container traffic is routed using Nginx on the AWS instance: this essentially allows the traffic to be SSL encrypted as Nginx works in tandem with Let' Encrypted, a software that provides free open SLL certificated. After traffic is encrypted using Nginx, it is sent to the AWS Route 53 Hosted Zone which allows essentially translates traffic to and from the EC2 instance's IP address and the domain name, "Clipz.ml". A similar process occurs in the Backend Hosted Zone.

4.0 Outcome and Performance

To examine the platform's performance, our team measured the end-to-end response time between both EC2 instances. To accomplish this, our group developed a script that ran on the Frontend and sent REST requests to the 3 main URL routes on the Backend. Each request being sent was timed and stored in milliseconds in an excel file. Furthermore, one of the biggest factors that affected the end-to-end response time between the two servers, was the user-selected categories. This had a big effect for two reasons. First, the more categories the system was given meant that the system had to make more queries to the database. Second, when the database was empty, the more categories the system was given meant that the system had to make more requests to the Twitch API. Both of these reasons would increase the amount of time it took to retrieve clips and because of that had the greatest impact on our platform's performance.

Moreover, to test category selection factors, the script was run 3 times with 3 different options when selecting categories. The first option, One Category, involved only selecting 1 category to be sent to the Backend from the list of all available categories. The second option, All Categories, involved sending all the categories from the list of all available categories to the Backend. The third option, Sequential Category, involved selecting one category from all available categories, sending it to the Backend, and removing it from the list so that when the request was called again the script would select another category to be sent from a list of currently available categories. Once the list of currently available categories ran out, it would be replenished with the original list of all available categories. More information on how the script runs can be found in the appendices under *Description 4.1: Script Logic*.

The synthesized results from the script can be seen in Tables 4.1, 4.2, and 4.3. In general, the All Categories option performed overall the worse with the largest response time of over 14,000 milliseconds while Sequential Category performed the second best and One Category achieved the best results.

	Sign In	Selection	Query
Average	89.9402	117.72	90.5504
Max	106.93	1,392.87	99.42
Min	87.13	89.09	87.9

Table 4.1: One Category (ms)

	Sign In	Selection	Query
Average	90.4428	1,528.36	1,266.10
Max	100.18	14,315.94	1,524.20
Min	87.47	998.46	961.34

Table 4.2: All Categories (ms)

	Sign In	Selection	Query
Average	90.4732	696.66	363.2042
Max	101.39	5,268.25	3919.82
Min	86.75	92.00	87.66

Table 4.3: Sequential Category(ms)

When examining the results more closely, the end-to-end performance for the request being made to the SignIn route stayed consistent throughout each option as seen in Figure 4.1. This is not surprising as the different category options should not have affected the SignIn portion of the script. Not only that but each initial signUp request (request 1) had a greater response time than most preceding requests. However, interestingly, all 3 options, when we don't include the initial request, had a local response time maximum between requests 17 and 19. A possible reason for this is that the Backend Flask server felt overloaded at those points and decided to dedicate more workers to alleviate the incoming traffic; however, more investigation needs to be done to understand this phenomenon.

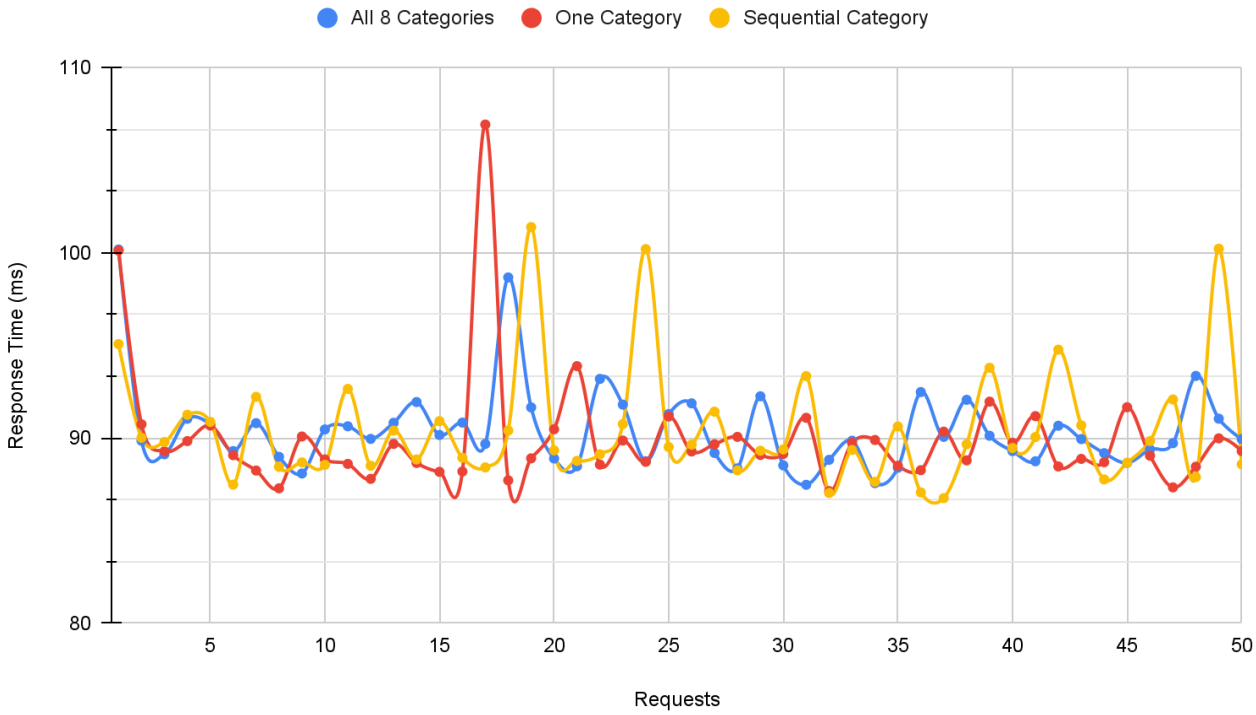


Figure 4.1: Sign In/ Sign Up Response Time Versus Request

As expected, the selection page's end-to-end performance had widely different results for each category selection scenario. As seen in Figures 4.2 and 4.3, the Response Times over a period of 50 Requests for both graphs are almost identical; however, the All Categories graph is 10 times larger than that of the One Category graph. This is reasonable because when testing the platform's performance, all categories contained 8 categories making the recommendation query 8 times larger than that of one category: the aforementioned reason and discoveries made in the next section explain the consistent double-digit factor increase between the two scenarios.

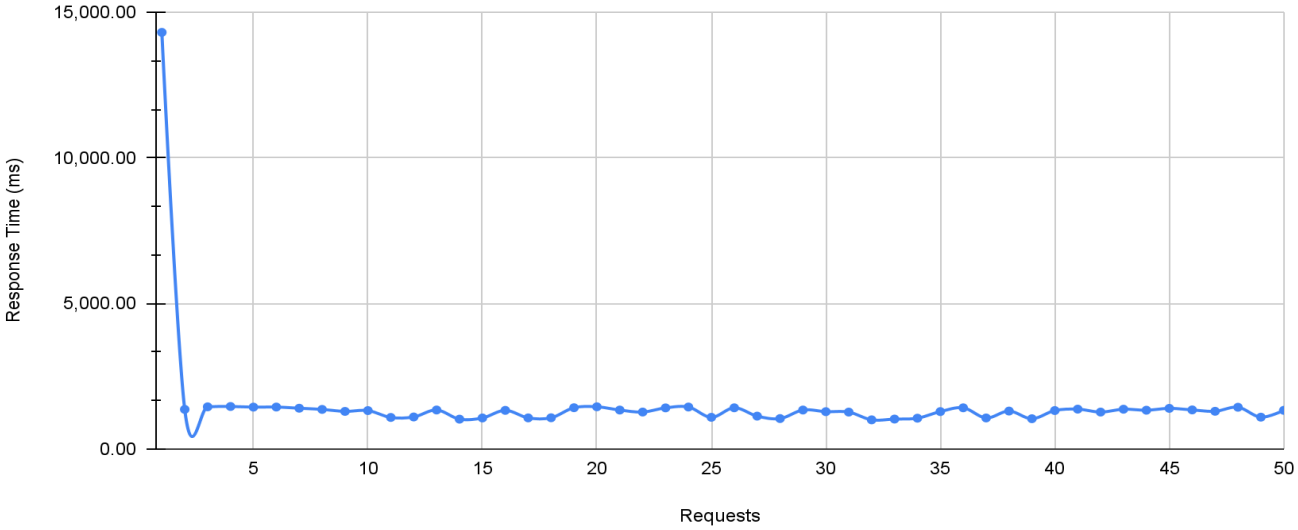


Figure 4.2: Selection Response Time Vs Request: All Categories

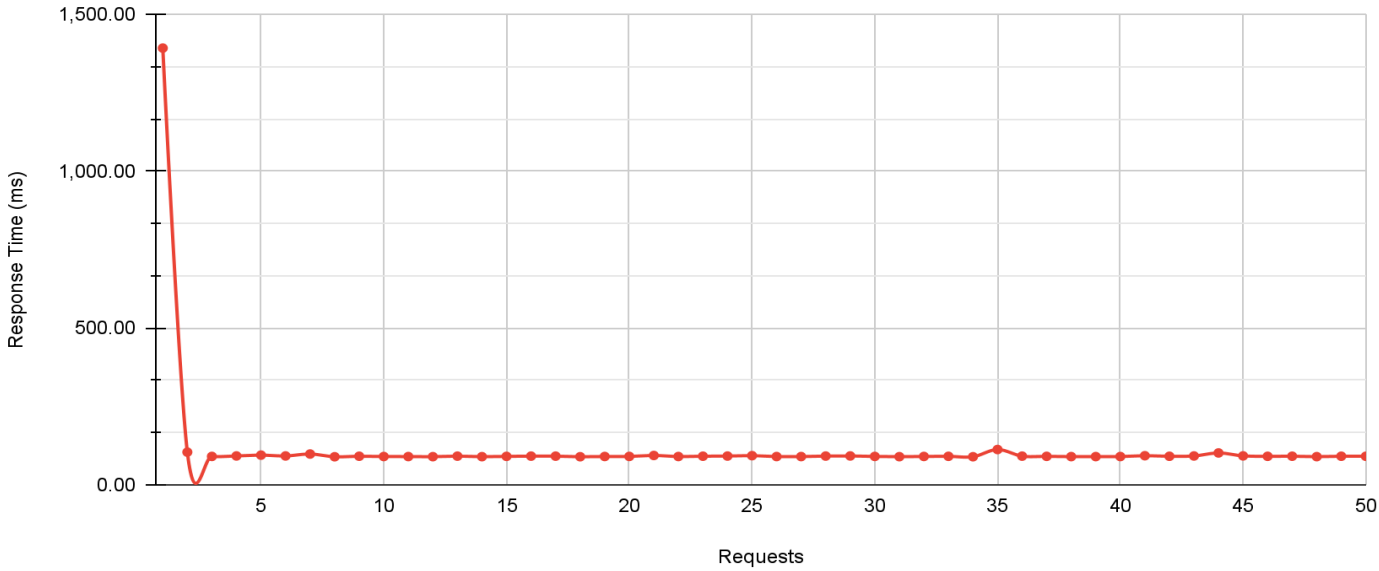


Figure 4.3: Selection Response Time Vs Request: One Category

Moreover, when we examine the sequential category option for the selection page we can see widely differing results as seen in Figure 4.4. The sequential category scenario starts off as expected with a large response time because there are no entries for the categories in the database: this means the recommendation system must pull from the Twitch API for the first 8 categories which can be clearly seen in Figure 4.4; however, interestingly when one examines request 14, 22, 30, 38, and 46, one can see that their response times are abnormally higher than that of other requests after request 8. Since these requests all have a difference of 8, this means that one category has a greater response time than the others, or more specifically the category "League of Legends" has a far greater response time. This is reasonable, however, because

“League of Legends” is by far the most popular category in our list which means it most likely has the most amount of clips to query. To combat this decrease in performance, our team should look into possibly limiting the query size while also asynchronously limiting the number of clips being held in the database through the removal of older clips.

Since the only difference between the query route and the selection route is that the selection stores the new categories in the database and then calls the recommendation system while the query route only calls the recommendation system, the end-to-end performance between the two should be very similar. As seen in Figures 4.5 and 4.6, which can be found in the appendices, the All Category and One Category options stay relatively consistent with requests seen after request 8 in Figures 4.3 and 4.4 respectively. This is logical because clips have already been added to the database, therefore, the query route avoids the initial cold start problem found on the selection page. Also as seen before in Figure 4.7, the Sequential Category for the query route is identical to that of the selection page with one category, League of Legends, having an abnormally large response time.

5.0 Analysis and Future Work

While accomplishing most of our set goals during the making of this project, we also noticed a few shortcomings as listed below:

- The Amazon EC2 hosting was unstable at times. Given more time, our team could research, and learn more about how to make the servers more stable.
- Currently, in our project, there is no way to store or ‘like’ clips and bookmark them for future use. In the future, we could research how to incorporate this functionality into our project.

Thus, as future work for our project, our team can implement the following ideas:

- Using more clip information so as to better build a profile for our user.
- Allow tags to be dynamically pulled from Twitch.
- Incorporating machine learning models to improve our recommendation system.
- Converting SQLite database to DynamoDB or Amazon S3 for better robustness, scalability, and flexibility.

6.0 Conclusion

Improvements in data speeds and broadband costs sparked an explosion of first-generation video streaming services in the mid-2000s. Twitch was launched in 2011 as a spin-off from its predecessor, Justin.tv. Twitch is now the world's largest live-streaming network, broadcasting anything from sporting events to video games and e-sports. The live streaming market is growing rapidly and the competition is also very stiff. Users have become very selective when accessing the content which makes it very important for live streaming platforms to cater to the content as per the user's interests. As shown, our team was successfully able to build and implement a twitch clip recommendation platform that would show users a personalized curated feed of clips tailored according to their preferences. During the implementation and building of this project, we were successfully able to learn more about cloud technologies such as Docker and Amazon EC2 and incorporate their use in our project. We were also able to expand our knowledge of web development frameworks and languages such as Python Flask, and JavaScript. We were also able to explore the concepts of network routing more deeply which immensely helped us during the making of our project. Overall the project was a success and the code for the Frontend and Backend can be found at *Github.com/jayteaftw/Twitch-Clips-FE* and *Github.com/jayteaftw/Twitch-Clips-BE* respectively.

7.0 References

[1] *https://en.wikipedia.org/wiki/Twitch_(service)*
[2] Frontend Github: *Github.com/jayteaftw/Twitch-Clips-FE*
[3] Backend Github: *Github.com/jayteaftw/Twitch-Clips-BE*

8.0 Appendiecies

Image 3.2: Selection Page

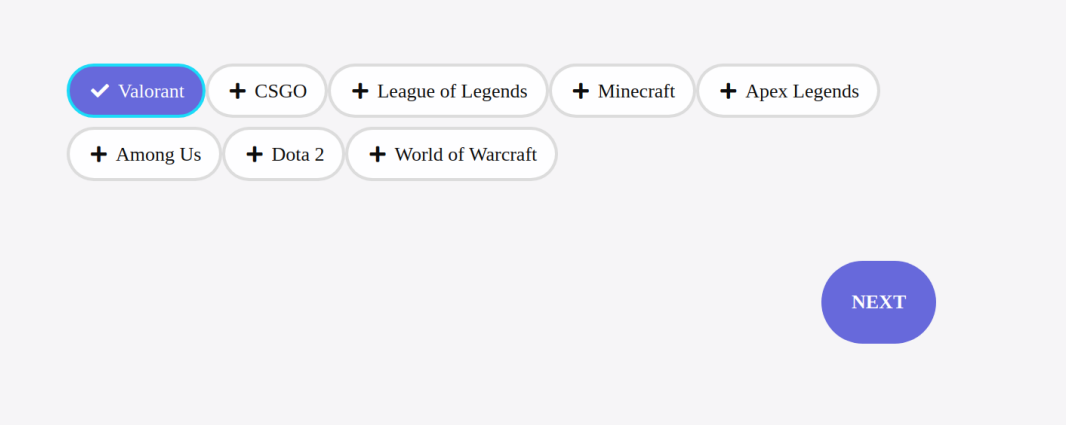
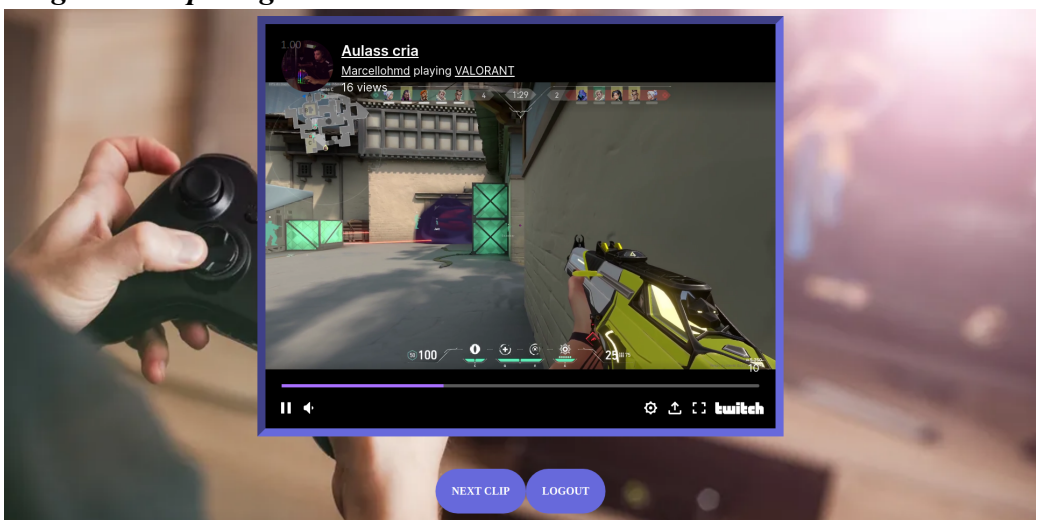


Image 3.3: Clips Page



Description 3.1: REST API Calls Between The Frontend and Backend

When the user credentials are passed to the backend via a REST API call, Flask will call either the signIn() or signUp() based on if we have an existing user and backend. The format of the POST request from the frontend is as follows:

```
{
 method: 'POST'
 headers: {'Content-Type': 'application/json'}
 body: {'Email': email, 'Password': password}
}
```

In signIn(), the user credentials is authenticated and if we have an entry for that specific user, we will query the database for the user-selected categories and send the following information in JSON format:

```
{"token": user_token, "checked": user_selected_tags, "categories":
all_categories_stored_in_DB}, 200
```

The format of the REST call shown above is also used in signUp(), where the "checked" field will be empty since the new user has not preselected any tags. Once the information has been passed to the frontend, we have the necessary information to be loaded on the tag selection page.

Once the user has selected their tags, the POST request containing the selected tags is sent to the backend, where the format of the request is as follows:

```
{
 method: 'POST'
 headers: {'Content-Type': 'application/json'}
 body: {'Categories': user_selected_tags, 'Token': user_token, 'Email': email}
}
```

The backend will call selection() and will parse the request to get the selected tags. If we have an existing user, we will first validate the user using their email and token, update the selected tags in our database, and then call recommend(). The recommendation system will return the list of Twitch URLs and will be sent to the frontend. If we have a new user, we will validate the user using their email and token, call the recommendation system and return the list of URLs to the frontend using the following format:

```
{"links": list_of_twitch_urls,}, 200
```

The information will then be displayed on our clips display page. If the user has watched all the clips that have been loaded on the page, the frontend will send another POST request to the backend to retrieve new clips using the following format:

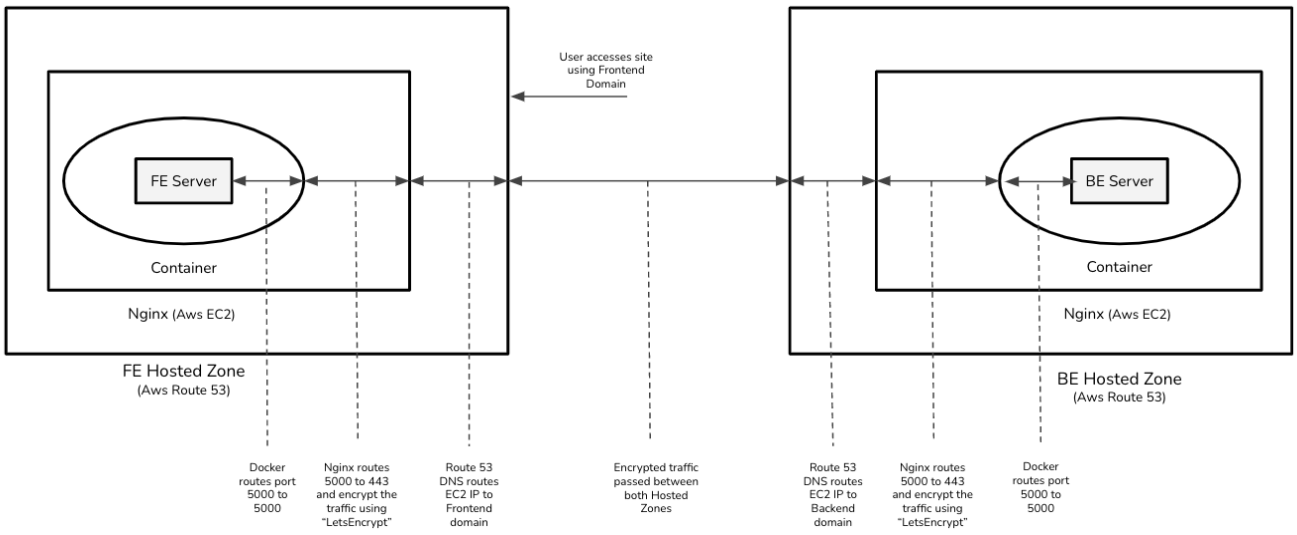
```
{
```

```
  method: 'POST'
  headers: {'Content-Type': 'application/json'}
  body: {'Email': email, 'Token': token }
}
```

Then the recommendation system will run again, and return a new list of Twitch clip URLs, using the following format:

```
{"links": list_of_twitch_urls,}, 200
```

Figure 3.5: Routing in Depth



Description 4.1: Script Logic

To go more in-depth on how the script works, first, a post request is sent to the sign-up URL: this is because when the Backend powered by Docker is rebuilt, there are no entries in the database. On further iterations, the sign-In URL is used because there already exists a user in the system. Once the token, categories list, and categories checked are received, the script executes one of the 3 category options dictated by the programmer as mentioned before. After which, the post request with the same category option would be sent to the query URL. To get an accurate view of the performance, these 3 requests are contained in a block of code and repeated and recorded 50 times per script run. Furthermore, each time the script was run, the Backend's docker container was rebuilt to simulate how a new system would perform.

Figure 4.4: Selection Response Time Vs Request: Sequential Category

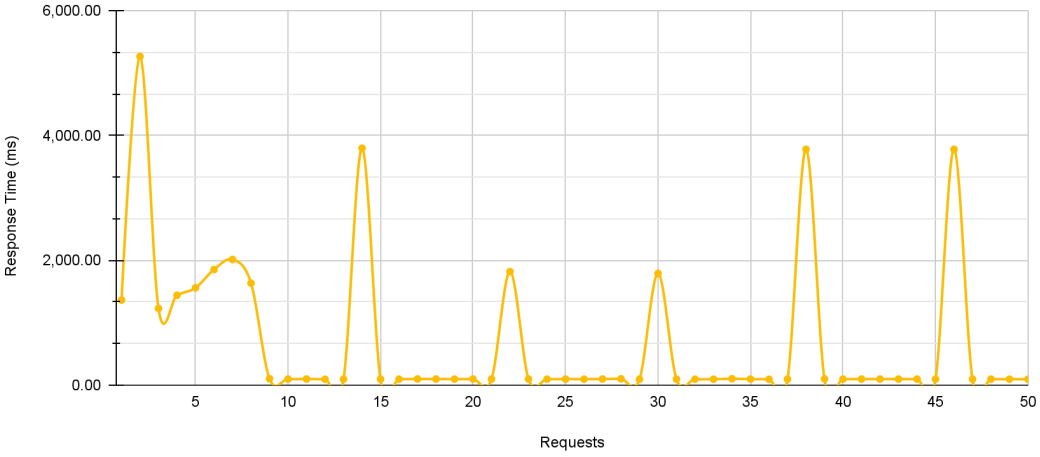


Figure 4.5: Query Response Time Vs Request: One Category

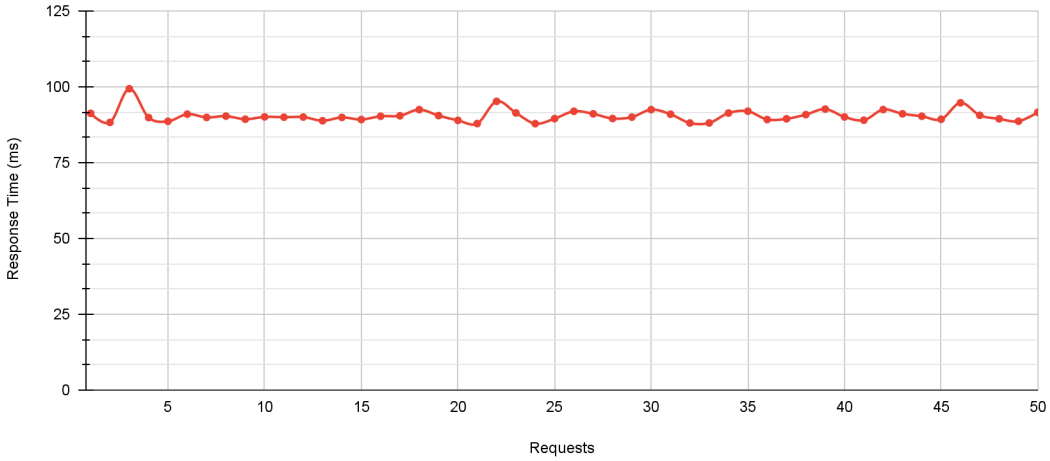


Figure 4.6: Query Response Time Vs Request: All Categories

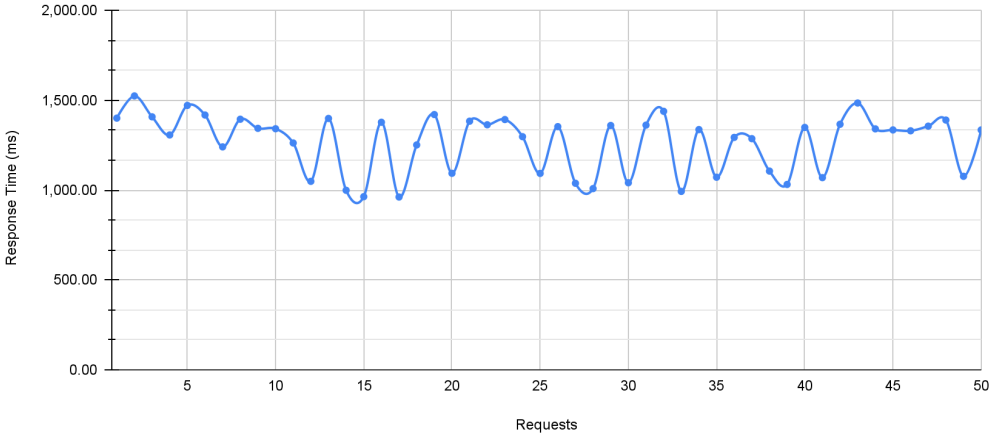


Figure 4.7: Query Response Time Vs Request: Sequential Categories