ScriptSearch 🔎

ScriptSearch 🔎

Dien Chau, Ethan Cherian, William Harkins, Huy Lai, Byung Jong Min

CSCE 482 Senior Capstone Design, Spring 2024

Department of Computer Science and Engineering

Texas A&M University

Abstract

Retrieving specific information from vast video data on platforms like YouTube presents a formidable challenge due to the limitations of traditional metadata-based methods. These methods often struggle to capture the intricacies of video content accurately. Additionally, the sheer amount of content uploaded to YouTube ensures that a comprehensively nuanced search would take an unreasonable amount of time and effort. This paper addresses the pressing need for more accurate and comprehensive video content retrieval by leveraging transcripts. By storing YouTube transcripts and developing a dedicated website, this solution aims to enhance search experiences, optimizing performance while maintaining a balance between speed and accuracy. The system enables flexible searches with sorting and filtering capabilities, empowering users to efficiently locate videos based on specific words or phrases they recall. Evaluation encompasses rigorous unit testing, integration testing, user acceptance testing, and user studies to ensure both functionality and user satisfaction. Overall, the website provides an effective tool for navigating and accessing video content with precision and ease.

1 Introduction

With the exponential growth of online video platforms like YouTube, the accessibility and consumption of video content have surged dramatically. YouTube has an average of over 500 hours of video content uploaded every minute (GMI Blogger, 2021). Amidst this vast ocean of video data, the challenge of efficiently retrieving specific information within video content has become increasingly pressing.

Traditional methods of video search heavily rely on metadata and manually generated annotations. However, these approaches often fall short in accurately capturing the nuances and details present in video content. Keyword-based search, for instance, may overlook relevant information due to the limitations of metadata and the inability to account for context within the video.

This highlights the importance of accurately searching transcripts, as transcripts provide a textual representation of the spoken content within videos. Unlike metadata, transcripts offer a more comprehensive and detailed overview of the video's content, including dialogue, narration, and other spoken elements. By leveraging transcript data, it becomes possible to enhance the search experience by enabling users to locate specific information with greater accuracy and efficiency.

In light of these challenges and the growing demand for efficient video content retrieval and analysis, there is a compelling need for innovative tools and systems that leverage transcript data to offer a more accurate and comprehensive search experience for users.

1.2 Case Scenarios

Alfred is an older man with memory issues. He is able to remember phrases from a YouTube video he's watched somewhat recently, but he's unable to remember the exact title and image of the video. To find it he'll have to comb through the videos he's watched before and narrow it down, but this would take too much time. It would help to have a search to speed up to queries.

Dhruv Patel is a legendary competitive programmer who watches a lot of videos going over code forces competitions. To prepare for an upcoming contest he wants to practice dynamic programming again. The codeforces videos that often appear only have the competition name and problem letters (CodeForces Round 921 Div 2. A B C D E), so it is difficult to see what approach was used. It would be nice if he had another filter to see what phrases were used in videos.

David Nguyen is trying to get better at the English language. He often watches YouTube videos to try and get the right pronunciation of certain words, but he would like to view a variety of speakers to see if he's truly saying it correctly. It would help if he could search in other videos to see how words/phrases are pronounced to speed up his learning process.

1.3 Goals and Constraints

Goals while creating this app is optimizing search performance, balancing speed and accuracy of search queries. Additionally, searches should be flexible, returning results despite potential user misspellings or partially correct phrases. Ideally, to provide the most functionality, searching should include on-the-fly searching from user provided URL(s) OR word/phrase search YouTube wide. Matched videos need some way of sorting, filtering, and ranking.

One constraint of this project is that it needs to be designed, implemented, and evaluated in a single 15-week semester. During the implementation phase we may need to pivot away from ideas as we run into issues, and during evaluation we may realize we need to scrap entire features entirely. Some other issues related to the cloud architecture include the scalability required to store large volumes of transcript data, along with the cloud services cost tradeoffs. Finally, we'll need to work around the YouTube API, dealing with potential throttling that may occur from calling routes in short bursts.

1.4 Solution Summary

Our solution is to store YouTube transcripts and create a website allowing users to search the stored transcripts. These will make it easy for anyone to use the application in a variety of ways. One of the existing challenges in implementing a system like this is efficiently storing transcripts, which we could solve by using Firebase or another service such as Typesense. Additionally we will allow users to add some filters to narrow the scope of the search.

1.5 Evaluation Summary

To demonstrate the functionality of the system, speeches made by government officials or politicians that are made available on YouTube will be placed within the database. Additionally, various other Youtube channels such as NeetCode or jacksepticeye. Testing will include comprehensive unit testing to ensure individual components work as intended, integration testing to verify the interaction between different modules, and user acceptance testing to evaluate the system's performance in real-world scenarios.

Additionally, usability will be evaluated through user studies conducted on individuals who are unfamiliar with the project. These studies will provide valuable insights into how easily understood and navigable the system is from an outside perspective. This comprehensive approach to testing and evaluation will ensure that the system meets both functional requirements and user expectations.

2 Related Work

YouTube as a platform has been growing tremendously as far back as 2009, when Google themselves released the statistic that 20 hours of video content were uploaded every minute (Alberti et al., 2009). In addition, a 2007 paper by Philippa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti revealed that YouTube experienced 100 million video accesses per day (Gill et al., 2007). Since then, these numbers have only increased, proving the need for an efficient method of searching the vast amounts of content hosted by YouTube.

Numerous efforts have been made to improve video content retrieval and analysis, ranging from traditional keyword-based search systems to more advanced solutions leveraging transcript data. A more prevalent (and relatively recent) system that attempts to remedy this problem is YouTube Keyword Search (YTKS). The website allows users to provide a YouTube channel, playlist, or individual video link, and a query for which to search. We aim to implement a system similar to YTKS and its functionality. However, there are a number of

key differences to make our solution stand out. Most notably, we use a database to store transcripts, allowing us to perform the search even without a user-provided YouTube link. YTKS also performs all searching in the browser, which means search results can take some time to populate, and the browser window becomes extremely bloated. Our solution will implement a functional backend that circumvents many of these issues.

Some other innovative systems have emerged to address the shortcomings of traditional video search methods. For instance, "Filmot" also keeps a database storing YouTube transcripts for queries. Its database is monumental and contains many languages, but there are notable shortcomings. For one, the developer has a separate scraper that must index a YouTube video that runs independently of the searcher. This means that some videos will not be indexable during searching time, and that the database will grow massively without bound. In fact, according to its developer, Filmot costs at least $510 per month to store all the transcript data (Filmot, n.d). This type of cost is something we wish to avoid.

Moreover, projects like YT-DLP have also contributed to the evolution of video content retrieval systems. YT-DLP, a fork of YouTube-DL, introduces improvements in video downloading capabilities, enabling users to access video content for further analysis and processing (github.com/yt-dlp/yt-dlp). This can be leveraged to collect the transcript data for videos without having to create a scraper of our own, which would prove arduous given YouTube's preventative measures to prevent them.

In a paper written by Robert Boyer and J Moore (1977), they discuss an efficient string searching algorithm. The paper delves into a novel fast string searching algorithm designed to efficiently locate substrings within larger strings. It begins by outlining the problem statement of string searching and the importance of optimizing search algorithms, especially in applications like text editors, databases, and search engines. The proposed solution, the fast string searching algorithm, is introduced, highlighting its structure, components, and efficiency compared to other methods. The algorithm's details, including its time complexity and handling of various scenarios and edge cases, are thoroughly discussed with examples and illustrations. Furthermore, the paper evaluates the algorithm's performance and claims its contribution to providing a fast and effective solution for string searching tasks. However, this algorithm only does a simple string search. It does not support additional metadata such as when in the video the string appears.

Typesense is a cloud platform that offers fast and intuitive search experiences across various data types, including text and multimedia content, by providing powerful indexing and querying functionalities (Typesense, n.d.). By storing our transcripts in Typesense, we can quickly and easily search through them and find matches to a given user query. As a bonus, Typesense comes with an easy method to connect to the Google Cloud platform, where

most of our backend is located. Typesense was chosen over other technologies like ElasticSearch or Algolia because of its ease of use and pricing. ElasticSearch has many parameters that would require time fine-tuning in order to maximize performance and results. Algolia, meanwhile, is much easier to learn and tune than ElasticSearch with more features than Typesense; however, the pricing is steep and the search engine is our main product. We cannot handle the cost in the long run.

Despite these advancements, there remains a gap in the existing solutions concerning the comprehensive integration of transcript data into video search systems. While systems like Filmot and Algolia utilize transcripts to enhance search capabilities, there is still room for improvement in terms of accuracy, scalability, and user experience. Additionally, the effectiveness of these systems may vary depending on factors such as the quality of transcripts and the diversity of video content. Therefore, there is a pressing need for novel approaches that address these challenges and offer more robust and reliable solutions for video content retrieval and analysis.

A common problem that our project faces is data storage. With our project needing to search through a list of transcripts there is a potential need to store these transcripts in a compact manner. In the paper by K. Sharma and K. Gupta (2017), the authors tested 3 different compression techniques on a set of different file types. The results of their tests showed the speed of compression for each technique on each of the chosen file types. Since their test included the compression of text files, their work can provide a good insight into what techniques we may want to use to efficiently store our transcripts.

With our project being a search engine, one important factor of the results our project returns is the relevance to the user. Thus we could potentially need a way to be able to determine which search results our project gives is more attuned to what the user was looking for. In a paper by Wadee Alhalabi, Miroslav Kubat, and Moiez Tapia (2007), they develop a new evaluation metric SEREET for ranking search engine results and they explain how the metric is calculated. This information can be a good insight into determining how our search results should be ordered when returning to the user.

Accessibility is a concern when developing software, and search engines are no exception, as evidenced by a series of papers published in the Universal Access in the Information Society journal. The first of these papers, written in 2008 by Barbara Leporini, Patrizia Andronico, Marina Buzzi and Carlos Castillo, details effective practices for developing web applications that support screen-readers, such as placing the most important elements at the top of the source file and alerting users through sound (Leporini et al., 2008). In addition, they developed an accessible alternative to the Google search engine and supported it with a user study of 12 visually impaired users. The knowledge from this paper,

especially the list of effective practices, will allow our team to develop an application that can be effectively used by visually impaired users.

In the second paper, written in 2008 by Myriam Arrue, Markel Vigo and Julio Abascal, the authors bring attention to web accessibility guidelines such as WCAG and present a quantitative statistic for measuring the accessibility of a website (Arrue et al., 2008). They then integrate this statistic into a search engine, known as Evalbot, to test the metric. Knowing about common web accessibility guidelines and being able to test our website with a quantitative metric will allow for our project's functionality to be developed in a more accessible manner.

3 Requirements

The base functionality required of the application is to allow users to input YouTube URLs of single videos, channels or playlists and a search phrase. The application will then return all occurrences of that search term in the specified videos. In the process, users are presented with a streamlined, easy-to-use web interface that facilitates their understanding of the product and its results. Beyond that, additional functionality, such as sorting via video details will permit users greater customization and increase satisfaction.

3.1 User Stories

In the development of our project, we needed to ensure that our end goal was still user satisfaction, rather than merely a technical product. To that end, we employed user stories to determine how end-users might use our application and why. Some of these user stories can be found below.

1. As a student preparing for exams,

I want to search through YouTube transcripts for educational videos on specific topics,

So that I can easily find relevant content to aid in my studies and revision.

2. As a marketing professional,

I want to search through YouTube transcripts to analyze competitor strategies and audience preferences,

So that I can refine our marketing campaigns and content creation strategies for better engagement and conversion.

3. As a digital marketer,

I want to search through YouTube transcripts for trending keywords and phrases related to our industry,

So that I can optimize our video titles, descriptions, and tags for improved visibility and search engine ranking.

4. As a teacher in a diverse classroom with students of varying learning styles,

I want to search through YouTube transcript

So that I can provide alternative access to video content for students who benefit from text-based learning,

5. As a person learning a new language,

I want to search YouTube transcripts for videos with specific phrases

So that I can improve my comprehension, vocabulary, and pronunciation.

6. As a YouTube video creator,

I want to search YouTube transcripts for a specific keyword

So that I can find instances where viewers are discussing my content.

7. As a journalist,

I want to search YouTube transcripts for key phrases related to breaking news

So that I can gather information for my articles.

8. As a gamer,

I want to search YouTube transcripts for game walkthroughs using specific terms

So that I can quickly find strategies for challenging levels.

9. As a job seeker,

I want to search YouTube transcripts for interviews with industry experts

So that I can gain insights and tips for my career.

10. As a cooking enthusiast,

I want to search YouTube transcripts for recipe videos with specific ingredients

So that I can discover new dishes to try.

11. As a user with visual blindness,

I want to search through YouTube transcripts to access video content through text-based formats

So that I can enjoy and engage with the content effectively using assistive technologies like screen readers or braille displays.

12. As a user with visual sensitivity,

I want to search Youtube transcripts while being able to use the site in dark mode

So that I can navigate through the site without potential harm to my eyesight

13. As a user with motor impairments,

I want to search Youtube transcripts while being able to enlarge interactable features on the site

So that I can navigate through the site without difficulty pressing links, the search bar, icons, etc.

14. As a non-native English speaker,

I want to search Youtube transcripts while being able to translate the website to my main language

So that I can utilize the site without difficulty understanding what features are meant to do.

15. As a quality-focused user,

I want to report potential bugs and errors with the application

So that those errors can be remedied by the developers

16. As a user with color blindness,

I want to customize the color contrast and theme of the application's interface

So that I can better read the transcripts and use the application.

17. As a user with hearing impairments,

I want to be able to see the transcripts for each search result

So that I can retrieve the information that I need without having to rely on auditory components

18. As a historian,

I want to be able to search through government videos or other historical content

So that I can find the source of information provided through videos.

19. As a software developer,

I want to search YouTube transcripts for coding tutorials with specific programming languages or frameworks

So that I can enhance my skills regarding these technologies.

20. As a legal professional,

I want to search YouTube transcripts for specific legal terms or case references

So that I can find relevant video content for my research and casework.

3.2 Definition of Success

The user will provide a link to any publicly available video, channel or playlist. The application will then load 250 of the most recent videos to search through. When a search query is made, at least one video will be displayed to the user. The video will have its thumbnail, title, channel name, the number of matches within the video, as well as its upload date. Once a video is clicked, a separate modal view will appear with a list of timestamp links.

The list of timestamps with the line in the transcript where the user's search query appears. The timestamp should also be a link to the actual YouTube video at the given time where the displayed line would be mentioned. The videos will be sorted by upload date by default, with most recent videos appearing first. Other sorting methods can be specified by the user such as video upload, duration, or number of matches. If the query does not find a video, the user interface will display "No results".

The query itself should take no more than 20 seconds to complete.

4 Engineering Standards, Regulations, and Considerations

4.1 Engineering Standards

Our system has no need for any specific industry engineering standards. The frontend is coded following closely with the JavaScript Standard Style. We also use ESLint to debug. The backend is coded following closely with the PEP 8 style guide. Additionally, the documentation present within the code follows the Google Python Style Guide. We invoke the transcript API using standard HTTP requests.

4.2 Applicable Regulations

Privacy regulations, such as General Data Protection Regulation (GDPR), California Consumer Privacy Act (CCPA), and Health Insurance Portability and Accountability Act (HIPAA), are not applicable to our system, as we have no need to store any user information. For user studies, we will take no personal information from subjects and, as such, face no privacy concerns regarding the maintenance of such data. Each user tester will be abstracted into a user persona to ensure that no personally identifiable information is maintained.

4.3 Environmental and Health/Safety Considerations

The only environmental consideration that needs to be taken into account is the large amount of power consumed by the data centers that host the cloud functions we are utilizing for this project. This amount of power consumption creates a carbon footprint that can not be ignored. In addition, these data centers utilize and throw away hardware at a rapid pace, creating a dangerous amount of electrical waste. While these aspects are not connected to our project directly, it is important to consider them in our development process.

4.4 Ethical, Social, and Political Considerations

The content of our application is strictly based off of the same content present on YouTube, including any and all videos that are available on YouTube with transcripts. All content upload restrictions that are followed by YouTube will inherently be followed by our site.

One possible social consideration would be that the english subtitles that are inherently generated by YouTube can be biased. They could change the spelling of some words that

conform more to U.S. English, which may hurt app usage for other English speakers with different dialects (“color” vs. “colour”). Currently, there is not a good way to address this issue without heavy data wrangling and processing of the search query, which would add massive complexity to our app.

One potential ethical concern is that since our site does not collect user information, there is no easy way for our site to restrict content based on a user’s age. While the timestamp that takes a user to a video would restrict them if they are underaged, the potentially explicit contents of the video could be exposed when we show the transcript lines to the user.

5 Design Exploration

Our solution will allow users to search YouTube video transcripts in a manner similar to a search engine. The user will insert their query into a search bar, specifying a channel or playlist link in a separate search bar if they would like to narrow their search. After submitting their search queries, the user will be presented with a list of videos in a grid format. The thumbnails and titles of each video will be displayed. In addition, the timestamps of each occurrence of the search query and the sentence that the occurrence came from will also be displayed. Once the user clicks on a video’s thumbnail, a pop-up window will be displayed and will play the video from the first occurrence of the searched term.

5.1 Comparison of Potential Solutions

The first potential solution we thought of was a website application. Similar to YTKS, the web application would allow users to search through our database of videos for instances of keywords. However our application adds additional control over the scope of videos to search through by providing filters such as Channel Name, Duration of the Video, and Language the video is in. Other filtering options can be added later if the YouTube API allows for it.

The second potential solution we thought of was a Chrome Extension, adding more functionality to YouTube’s actual page. With an efficient cloud backend built out, the same scraping and searching can be done on any page, whether that be a channel video list, playlist, or even a user’s feed or watch history. A YouTube-wide transcript search may not be as feasible with this method, but searching on-the-fly on several pages would be much easier for the user.

The final potential solution we thought of was a mobile application. Similar to the web application, it would allow users to search for videos through a keyword search. However,

due to hardware limitations, much of the search algorithm and filtering would need to be done in the cloud.

Our team has decided to implement the *website application* solution due to its accessibility, flexibility, and familiarity. More specifically, a larger number of users will be able to use a web-based application as opposed to a mobile-based application or Google Chrome extension. Furthermore, many on our team are more familiar with website development compared to the alternatives.

5.2 Lo-fi Prototyping

Before we began writing software to solve the problem, we created a handful of lo-fi prototypes in order to plan and demonstrate the overall structure of the system, as well as potential visual layouts of the website or browser extension.

Fig. 5.2(A) is our preliminary system diagram, outlining a rough cloud architecture proposal. The broad idea is to have the client interface with the application via a website where they provide a list of video URLs or a URL to a playlist or channel. From there, a cloud function is triggered, which uses a Pub/Sub model with multiple cloud functions functioning as "consumers" to efficiently scrape the transcripts of the video(s) requested. These transcripts are then stored in a database, of which a read-replica is made on another service that can be searched efficiently. The results of the search are returned to the website for the user to see.

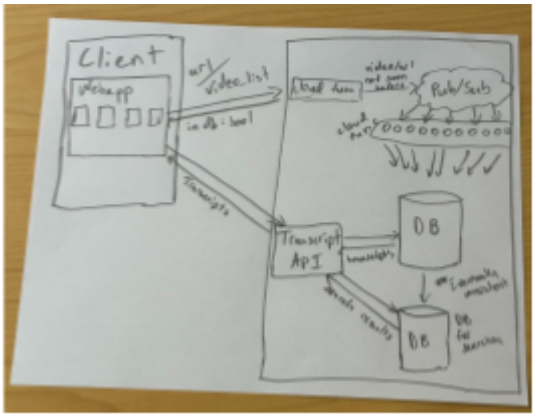


Fig. 5.2(A) - preliminary system design diagram

Fig. 5.2(B) shows one of our potential ideas for the design of the website. In this design, the homepage features a search bar for the user to enter their query and video URLs. After the search concludes, the results are populated immediately below the search bar (on the same page), and the screen may automatically scroll to make this obvious. Each result is placed

into the page as a video thumbnail, with query matches from the transcripts displayed below with timestamps, so users may click on one to be taken to that instant in the video.

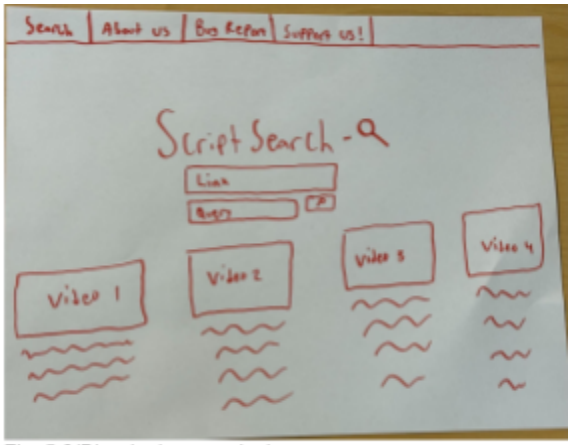


Fig. 5.2(B) - single page design

Fig. 5.2(C) is the second potential design. The first image is the main homepage with the search bar in the middle of the page with a button to display a drop menu with different filters to allow the user to get better results based on their query. Then below will be other text buttons that will take the user to either an *About Us* page, a *Bug Report* form, or a *Support Us* page.

Fig. 5.2(C) - Home Page for Multi-Page Layout

Fig. 5.2(D) shows how the results will be displayed after a user search. The results will show each video in a list whose transcript contained the query searched for by the user. Each video will be shown with the title and thumbnail as shown on YouTube. Then below each video will be a list of timestamps showing the time of when the line that contained the user's query occurred in the video along with the actual line of the transcript next to the timestamp. Furthermore, the timestamp will be a link to the actual YouTube video taking the

user to the exact time where the line show at that timestamp occurred, hence why we colored it blue.

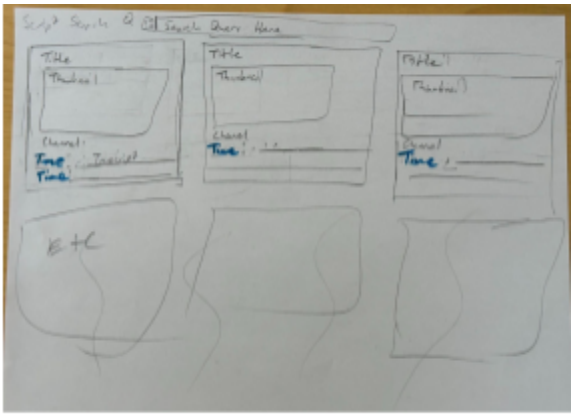


Fig. 5.2(D) - Card View Page of Multi-Page Layout

Finally, Fig. 5.2(E) is a mockup of a potential browser extension. It would effectively act like the web application but limited to the side bar. For example, the YouTube search would be the same as the keyword search within the extension. In the extension, the user can then apply additional filters to the content much in the same way as in the web application.

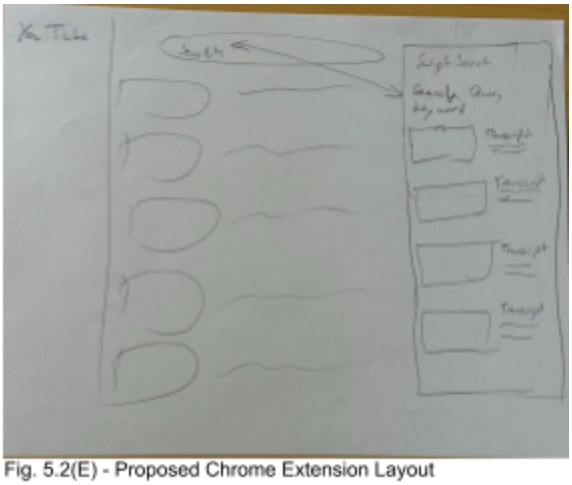


Fig. 5.2(E) - Proposed Chrome Extension Layout

5.3 Pilot Studies

After we created our Lo-Fi Prototypes, we conducted a user study. As part of this study, we showed these prototypes to 10 users and informally discussed the user interface and system structure with them. We received a variety of feedback from users, the most prominent of which is as follows:

- When given a choice between a single-page and multi-page layout, 80% of users preferred the single-page layout.
- Users were confused about the context of the "Link" text box in the single-page layout
- Multiple participants suggested implementing a loading screen to cover for long loading times
- Multiple surveyed groups liked the card view on the multi-page layout
- Users thought implementing the search based on a channel/playlist made the results too narrow and suggested making link optional

After receiving this feedback, our team intends to implement the following design decisions:

- We plan to implement the single-page design while incorporating the card view of the multi-page layout. This is in response to users, who thought the single-page design was more intuitive, but liked the card view of the multi-page view more than the view shown in the single-page design.
- We intend to place descriptive text on the text box that allows the user to enter a link in response to user feedback that it was unclear what to enter in that text box.
- We plan to implement a loading screen after the search is executed to let the user know that the application is processing their request. This is in response to user concern regarding the potentially long loading time of the application.
- We will consider making the link text box optional, though it may not be feasible within the scope of our application. This consideration is in response to user feedback which stated that specifying a channel or playlist made the search too narrow.

5.4 Inclusion, Diversity, Equity, and Accessibility Considerations

As our product is dependent on a very popular social media platform, it's important that our system be designed with accessibility in mind to satisfy as many people as possible.

In that spirit, our design will be simple and clean, as included in our target audience are those with learning disabilities or poor attention spans. Minimizing the number of over-the-top graphics and the like will ensure that users' attention is focused on the critical elements of the design.

In addition, keeping the design straightforward benefits another party: those with visual disabilities and relying on screen readers to navigate the web. Screen readers generally require websites to be tabbable, either manually or automatically, and their jobs are made more difficult when there are extraneous elements on the page. For that reason, we will work to keep the design as uncluttered as possible and to ensure elements on the webpage are tabbable.

6 System Design

6.1 Functional Design

Fig. 6.1(A) shows the overall high level design of the entire project. It starts with the user inputting a search query as the beginning interaction with the frontend. Then using the user's query, the Typesense API will call the backend search algorithm to filter through all videos based on the user's query. Then using the backend's search results it will either call to the database for the transcripts of the searched videos or create any new transcripts for videos that don't have their transcripts stored. Then the database will return a text file of the relevant transcripts and the backend will then parse through the results and return a JSON package to the frontend with the necessary result data.

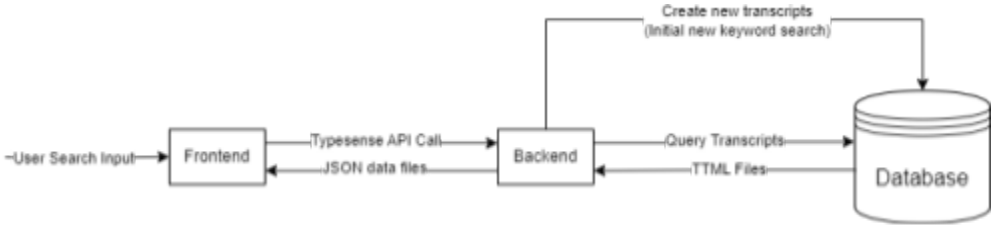


Fig. 6.1(A) - Level 0 Functional Diagram

Figure 6.1(B) is our level 1 diagram outlining the design of our frontend. From the homepage, users will enter their query into the prominently-displayed search bar. If the query returns no

results, the page will update to inform users of such, then allow them to enter a new query. If a query returns results, they will be displayed as a list of videos, which each consist of many clickable timestamp links that redirect the user to that instant of the particular YouTube video. In addition, the user will be able to filter and sort the list of videos on the results page via a collapsible interface.

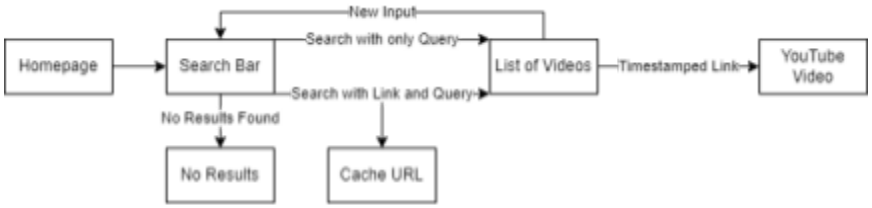


Fig. 6.1(B) - Level 1 Functional Diagram (Frontend)

Figure 6.1(C) is our level 1 functional diagram of the backend. This diagram shows how the user input from the frontend will interact with the backend. When the user makes a query, the backend will route the provided scope link to a cloud function. This cloud function will return a list of 250 video URLs. These URLs are passed to a Pub/Sub which will retrieve the transcripts of all of these videos in parallel to optimize the runtime of the query. These "consumer" cloud functions will then store these transcripts in the Transcript Database for searching. This database is replicated to Typesense, which will then perform the string search through these transcripts for matching instances of the user's search query. These results are then forwarded to the cloud function to translate into JSON packages for the front-end to parse and display to the user.

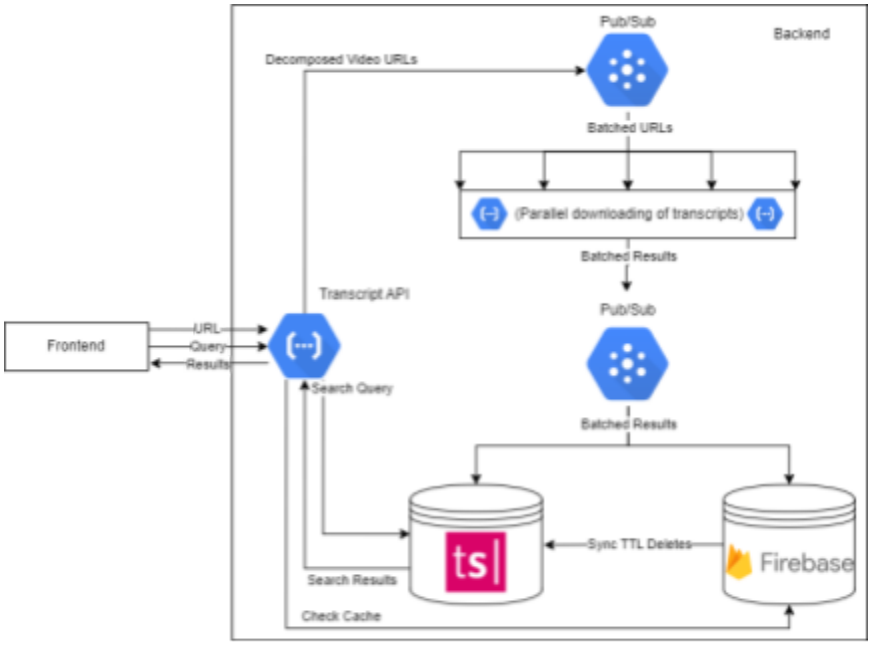


Fig. 6.1(C) - Level 1 Functional Diagram (Backend)

6.2 Data Design

Figure 6.2(A) details our preliminary design for the memory storage of transcript data. The idea is to keep one table (following the best design practices for NoSQL) to allow for scalability. We will use Cloud Firestore from Firebase to achieve this, as it provides a generous pricing plan for reads, writes, and deletes every day and works on a pay as you go basis. The attributes `channel_id` and `duration` will be used to provide additional search filters for the end user and provide more customizability for searching. Transcripts will be stored as a stringified json because Firestore supports up to 1 MB of attribute storage, and transcripts total up to 1 MB (in the worst case 6 hr video). Transcript JSON can be preprocessed even more beforehand to compress the data further and optimize data storage should our needs require it. This table will essentially serve as a "leader" that will serve reads, writes, and deletes primarily. Updates should almost never occur, and deletes would probably function via a TTL. Reads might occur whenever a function needs to check if a video id is in the database or not, to prevent unnecessary downloads and network bandwidth. Typesense will be used as a "follower" or a read replica that will be used to serve complicated search queries Youtube-wide. It should sync up with this table independently and be eventually consistent.

Video	
PK	video_id UUID
	channel_id UUID
	video_id UUID
	duration integer
	upload_time integer
	transcript string[]
	timestamp integer[]

Fig. 6.2(A) - Data Design of TranscriptDB

7 Evaluation

In this section, we will explain how we will evaluate the project. The evaluation will be split into two portions, functionality and usability. The functionality evaluation will be completed through a combination of unit, integration, system and performance testing. The unit tests will be automated while the other tests will require a manual review. The usability evaluation will be conducted solely through interviews with prospective users. During the interviews, no personally identifiable information will be collected about the users themselves. As a result, no extra precautions will be taken. There will be two interviews, one before using the application and after using the application. The results of these tests will follow the description of each test.

7.1 Functionality Evaluation

Our primary functionality goal is that, under any circumstance so long as the website is functioning properly, any search should return results at most a minute after the user's search was submitted.

7.1.1 Evaluation Procedures

Unit Testing:

- All unit tests will be performed automatically through the use of a script that will mock certain JSON objects for our backend functions to simulate requests. For each function we define in the backend we'll define a test for the happy case and a bad case scenario to run through. These unit tests will be later in development as we prototype different implementations of functions and results can change quickly.
- Unit tests must pass within the backend in order for a change can be pushed to the main branch. If each unit test passes we can be sure that the functions within our application work as intended. This will incentivize the user to test more frequently locally to ensure everything works after the changes they implement.
- The system is considered "adequately tested" if it passes all unit test cases.
- We assume that, during unit testing, our database will only contain the video "https://www.youtube.com/watch?v=NpmFbWO6HPU"

Test 1: Get URL type for a video:

The following video's URL will be used as input: Me at the zoo

The expected result of the url type is `URLType::Video`.

Test 2: Get URL type for a playlist:

The following playlist's URL will be used as input: GUESS THE ELO - YouTube

The expected result of the url type is `URLType::Playlist`.

Test 3: Get URL type for a channel:

The following channel's URL will be used as input. jawed - YouTube

The expected result of the url type is `URLType::Channel`.

Test 4: Get URL type for an invalid URL:

The following website's URL will be used as input: Google

The function should raise a `ValueError`

Test 5: Get ID from a video url:

The following video's URL will be used as input: Why Texas A&M?

The expected result is video id 2rkAL9ehPq8.

Test 6: Get list of video IDs from a playlist URL:

The following playlist's URL will be used as input: GUESS THE ELO - YouTube

The expected results is that a list of 72 video IDs

Test 7: Get list of video IDs from a channel URL:

The following channel's URL will be used as input: jacksepticeye - YouTube

The expected result is that the channel ID is "UCYzPXprvl5Y-Sf0g4vX-m6g" and that a list of video IDs is the same length as the max video limit of 250.

Test 8: Get list of video IDs from a channel URL with no shorts:

The following channel's URL will be used as input: OverSimplified - YouTube

The expected result is that channel ID is UCNIuvl7V8zACPpTmmNIqP2A and that a list composed of 32 video IDs is returned.

Test 9: Get a list of video IDs from a channel URL with no videos:

The following channel's URL will be used as input: @huylai2024

The function should raise a `ValueError`

Test 10: Process video URL:

The following playlist's URL will be used as input: Me at the zoo

The expected result is a dictionary consisting of two key-item pairs. The expected value associated with the key "video_ids" in the dictionary is a list of 1 video IDS. The expected value associated with the key "channel_id" is None.

Test 11: Process playlist URL:

The following playlist's URL will be used as input: GUESS THE ELO - YouTube

The expected result is a dictionary consisting of two key-item pairs. The expected value associated with the key "video_ids" in the dictionary is a list of 72 video IDS. The expected value associated with the key "channel_id" is None.

Test 12: Process playlist URL:

The following channel's URL will be used as input: jacksepticeye - YouTube

The expected result is a dictionary consisting of two key-item pairs. The expected value associated with the key "video_ids" in the dictionary is None. The expected value associated with the key "channel_id" is "UCYzPXprvl5Y-Sf0g4vX-m6g".

Test 13: Distribute list of video IDs across multi-search:

The following playlist's URL will be used as input: A list of video IDs

The expected result is a list consisting of five lists. Each of these sublists consists of video IDs with the input list of video IDs evenly distributed across these five lists.

Test 14: Test marking algorithm:

The following strings will be used as input: "game" and "This is a game"

The expected result is the following string: "This is a <mark>game</mark>"

Test 15: Search a phrase that is too long:

The following string will be used as input: "The quick brown fox jumps over the lazy dog"

The function should throw a `ValueError`.

Test 16: Search a single word with no filter:

The following string will be used as input: "game"

The expected result is a list of dictionaries. Each dictionary is expected to have the following keys: "video_id", "channel_id", "title", "channel_name", "matches"

Test 17: Search a phrase with no filter:

The following string will be used as input: "dynamic programming"

The expected result is a list of dictionaries. Each dictionary is expected to have the following keys: "video_id", "channel_id", "title", "channel_name", "matches"

Test 18: Search a single word with channel filter:

The following strings will be used as input: "game" will be used as a query and "UCYzPXprvl5Y-Sf0g4vX-m6g" as channel id.

The expected result is a list of dictionaries. Each dictionary is expected to have the following keys: “video_id”, “channel_id”, “title”, “channel_name”, “matches”

Test 19: Search a phrase with channel filter:

The following strings will be used as input: “dynamic programming” will be used as a query and “UC_mYaQAE6-71rjSN6CeCA-g” as channel id.

The expected result is a list of dictionaries. Each dictionary is expected to have the following keys: “video_id”, “channel_id”, “title”, “channel_name”, “matches”

Test 20: Search a single word with playlist filter:

The following strings will be used as input: “game” will be used as a query and a list of videos ID for the filter.

The expected result is a list of dictionaries. Each dictionary is expected to have the following keys: “video_id”, “channel_id”, “title”, “channel_name”, “matches”

Test 21: Search phrase with playlist filter:

The following strings will be used as input: “dynamic programming” will be used as a query and a list of videos ID for the filter.

The expected result is a list of dictionaries. Each dictionary is expected to have the following keys: “video_id”, “channel_id”, “title”, “channel_name”, “matches”

Test 22: Multi-search with playlist filter.

The following strings will be used as input: The URL of this playlist GUESS THE ELO - YouTube will be used as the filter and the query will be “chess”.

The expected result is a list of dictionaries. Each dictionary is expected to have the following keys: “video_id”, “channel_id”, “title”, “channel_name”, “matches”

Integration and System Testing:

- Our overall application consists of a frontend, backend API, Pub/Sub, subscriber functions, and a database, and the interactions between all of these components have to be tested. Specifically, the interaction between the frontend and the backend as well as the interaction between backend components should be covered. Our team has conducted this testing already throughout the development phase of the project. As very little of our project’s functionality is dependent on the

interactions between individual components, a lot of our integration testing was combined with system testing (using the system already defined in the cloud).

- Integration/System testing has been done manually during development by entering a query on the frontend and receiving a response on the backend. In addition, interaction between the backend components was also tested manually by sending fake data from our transcript API to the Pub/Sub functions.
- Our expected result for a query with no link is a list of all videos in our database that contain the query in their transcripts. If the user provides a link, the 250 most recent videos from the provided channel/playlist will be added to the database, and a series of videos from that channel/playlist that contain the user's query will be returned with timestamps of the query's appearance.
- This method of returning the results of a query almost directly matches what was outlined in our definition of success. Assuming that our application returns videos according to the expected results, it will be fulfilling our definition of success.
- The system will be considered "adequately tested" if it passes all of the integration/system test cases.
- We can use the results of our tests to figure out which components within our backend are bottlenecks and address them accordingly. Some more considerations will have to be made in certain cases such as the yt-url-consumer and the firestore-typesense-syncer, as multiple instances of these can be spun up extremely quickly. This means that even though they may run in under a second, more ms level optimizations have a massive impact.

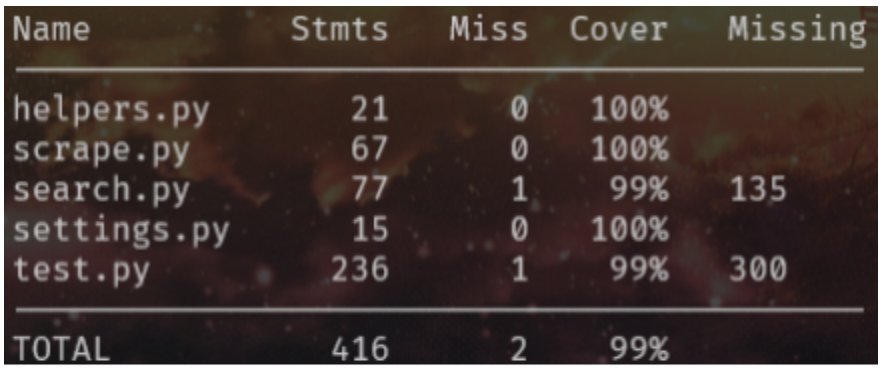
Performance Testing:

- We test individual components in the system leveraging the metric graphs and logs from each of the backend functions. This testing was done manually so far, but ideally this would be done automatically to test out more users or concurrent users. This includes the components for the transcript-api (yt-dlp and search), go-publish-ids (handling checking db and publishing ids), yt-url-consumer (ingesting urls into database), and firestore-typesense-syncer. The amount of time taken and memory per instance will be measured per instance for each function to figure out what bottlenecks exist in our current system.
- Throughout the Integration/System tests that we define for our test environment we'll be able to measure the logs and metrics for execution time and the memory. Using this we will be able to find out which functions are taking the longest and iteratively address them to make the overall backend application faster.
- From these tests we expect results that indicate the system runs under 20 seconds. However, as our system has developed we are moderately confident that we can

optimize the backend even further to allow for scraping and testing to both happen under 4 seconds. The tests will assume that 10 seconds is the threshold as we work to achieve this goal.

7.1.2 Evaluation Results and Discussion

For our unit tests, Figure 7.1.2(a) shows our overall code coverage percent when running Transcript API unit tests described above. The high coverage rate is an indication of the thoroughness of our unit tests, which mean our functionality is extremely likely to succeed in any given input scenario.



Name	Stmts	Miss	Cover	Missing
helpers.py	21	0	100%	
scrape.py	67	0	100%	
search.py	77	1	99%	135
settings.py	15	0	100%	
test.py	236	1	99%	300
TOTAL	416	2	99%	

Fig. 7.1.2(a) - Code Coverage

The actual speed of the application varied greatly depending on the current state of the user input. When it comes to a pure search of our database, there are three possibilities: searching without a filter, searching with a playlist filter, and searching with a channel filter. The average runtimes of each of these cases can be seen in Figure 7.1.2(b). Evidently, searching without a filter takes significantly longer than including either of the two possible filters, though it's still well below our 20-second desired threshold.

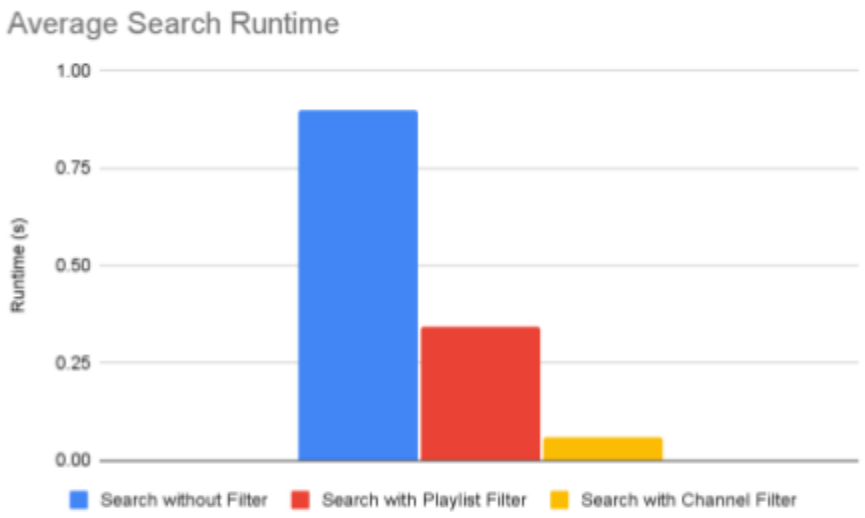


Fig. 7.1.2(b) - Average Search Runtime

The overall runtime of the API, including searching and postprocessing, similarly depends on the state of user input. The average of each case can be seen in Figure 7.1.2(c), and the trend broadly mirrors that of the average search runtime above. Searching without a filter is still slower but nevertheless low enough to meet our 20-second execution goal.

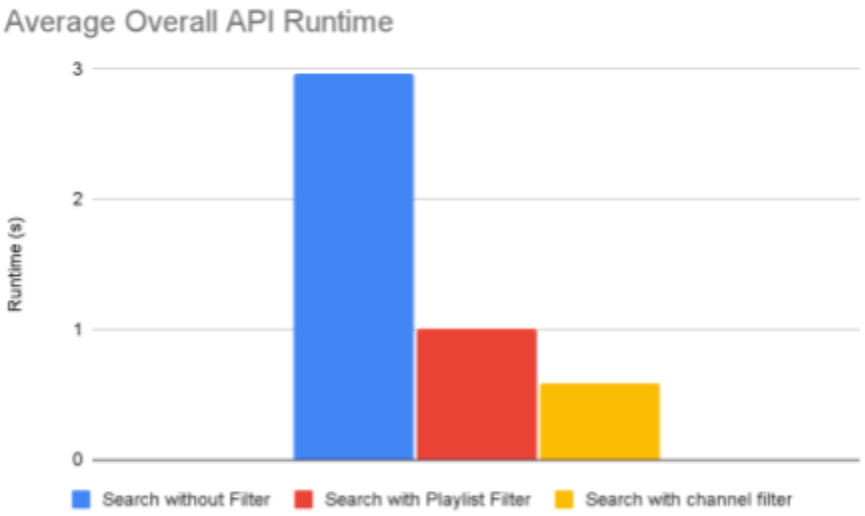
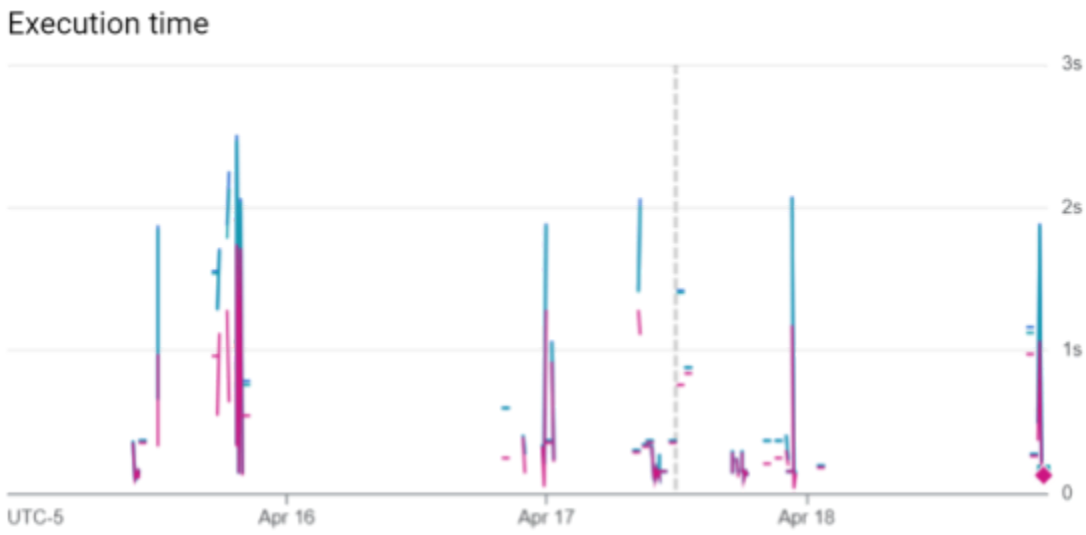


Fig. 7.1.2(c) - Average Overall API Runtime

If we look at the logs for our cloud functions within the scraping architecture, we see that is well under our 20-second threshold. Each instance runs in parallel, so it is bottlenecked only by the slowest instance. These video urls can be scraped on average within 1.5 seconds (50% of instances are 1.5 seconds, median), with a cloud function startup time of around

100-500ms (which can be calculated by considering the POST time which sometimes has to startup the instance - the handler time which is the function execution). Memory is much lower than 256MB even though we're batching urls and spawning many different goroutines, meaning that this process is cost-efficient too.

Figure 7.1.2(d) below shows the execution time of our backend scraping function and how the metrics above were calculated. Figure 7.1.2(e) below shows the memory used up by the instance, showing that the batches of 25 urls is incredibly memory efficient.



2024-04-18 20:33:07.462 CST	POST	200	612 B	1.8 s	APIs-Google; (+https://developers.goo…	h
2024-04-18 20:33:07.470 CST	POST	200	612 B	1.8 s	APIs-Google; (+https://developers.goo…	h
2024-04-18 20:33:07.477 CST	POST	200	612 B	1.1 s	APIs-Google; (+https://developers.goo…	h
2024-04-18 20:33:07.478 CST	POST	200	611 B	1 s	APIs-Google; (+https://developers.goo…	htt
2024-04-18 20:33:07.525 CST	POST	200	612 B	1 s	APIs-Google; (+https://developers.goo…	htt
2024-04-18 20:33:07.530 CST	POST	200	611 B	1 s	APIs-Google; (+https://developers.goo…	htt

2024-04-18 20:33:08.577 CST	2024/04/19 02:33:08 handler took 599.911413ms
2024-04-18 20:33:08.956 CST	2024/04/19 02:33:08 handler took 956.291221ms
2024-04-18 20:33:08.957 CST	2024/04/19 02:33:08 handler took 952.450488ms
2024-04-18 20:33:08.978 CST	2024/04/19 02:33:08 handler took 973.435978ms
2024-04-18 20:33:09.012 CST	2024/04/19 02:33:09 handler took 1.014043297s
2024-04-18 20:33:09.286 CST	2024/04/19 02:33:09 handler took 1.817928056s
2024-04-18 20:33:09.287 CST	2024/04/19 02:33:09 handler took 1.816690624s

Fig. 7.1.2(d) - Scraping Execution Time

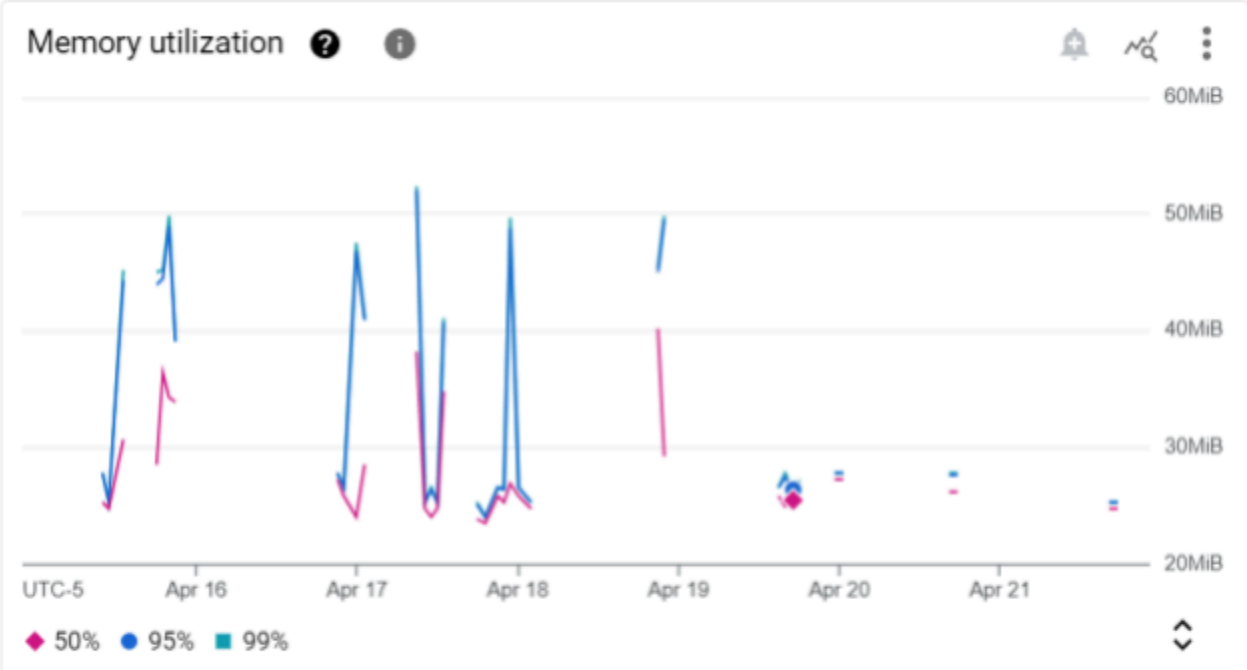


Fig. 7.1.2(e) - Scraping Memory Utilization

Another key component of the architecture deals with inserting the results into our databases and indexing the results for searching. While we aren't sure how long it takes Typesense to index the results, it is assumed to be quite fast (C++ used under the hood). What we do measure is how long the execution time for the Typesense batch upsert function as well as its memory utilization, both of which look to be good. Over the course of 7 days the execution time in the worst case still falls under 1.5 seconds, with a cold start of around 100ms-500ms as well. Memory utilization falls well below under the threshold of 256MB and could be decreased in order to be more cost effective. These results can be seen in fig. 7.1.2(f) and fig.7.1.2(g).

ScriptSearch 🔎


```
2024-04-18 20:33:10.372 CST  POST  200  612 B  1.1 s  API
2024-04-18 20:33:10.384 CST  POST  200  611 B  1 s  APIs-
2024-04-19 13:54:26.953 CST  POST  200  667 B  693 ms  AP
2024-04-19 14:07:12.272 CST  POST  200  611 B  206 ms  AP
2024-04-19 14:40:53.580 CST  POST  200  667 B  460 ms  AP
2024-04-18 20:33:11.980 CST  bulk import took 549.196106ms
2024-04-18 20:33:12.789 CST  bulk import took 754.82861ms
2024-04-19 13:54:27.810 CST  bulk import took 346.081447ms
2024-04-19 14:07:12.494 CST  bulk import took 200.263307ms
2024-04-19 14:40:54.182 CST  bulk import took 228.082769ms
```

Fig. 7.1.2(f) - Typesense Batch Insert Execution Time

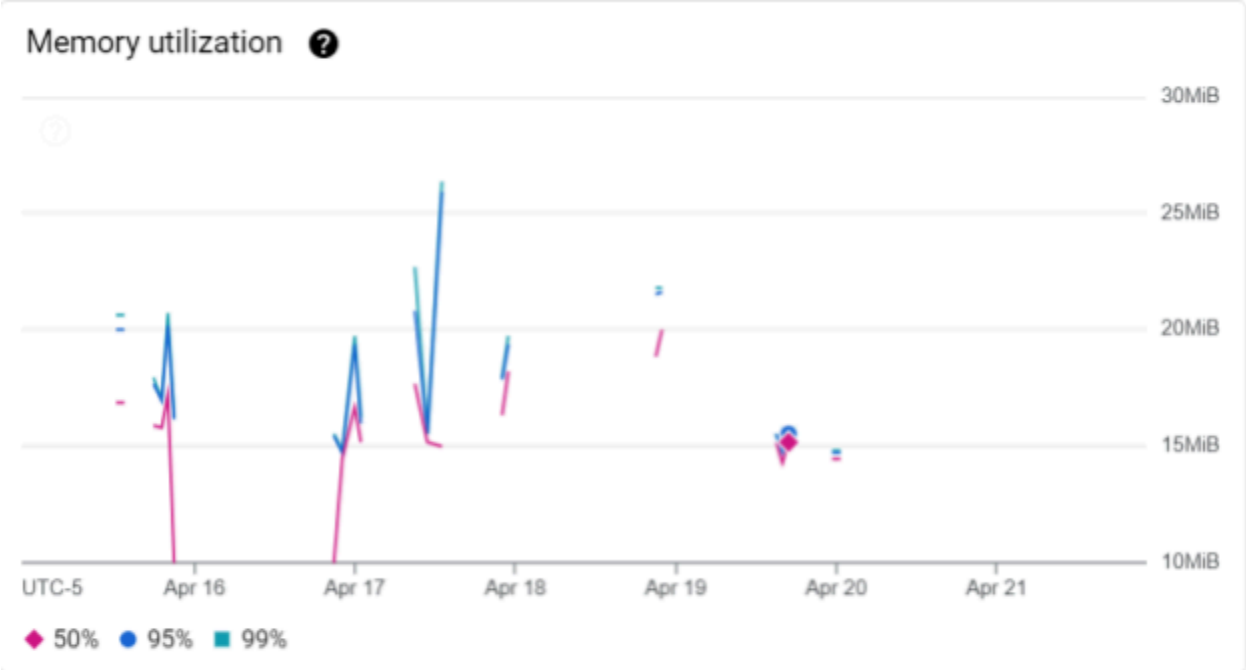


Fig. 7.1.2(g) - Typesense Batch Insert Memory Usage

Finally, the overall time taken between the user searching using a link not in our database and receiving results is less than the 20-second threshold even in the worst case. Overall time taken can be calculated by the time the transcript-api function finishes processing a url to when the last url batch is upserted through the typesense batch upsert function. This isn't automatically measured, but one can manually calculate this from the function logs (see Fig. 7.1.2(h))

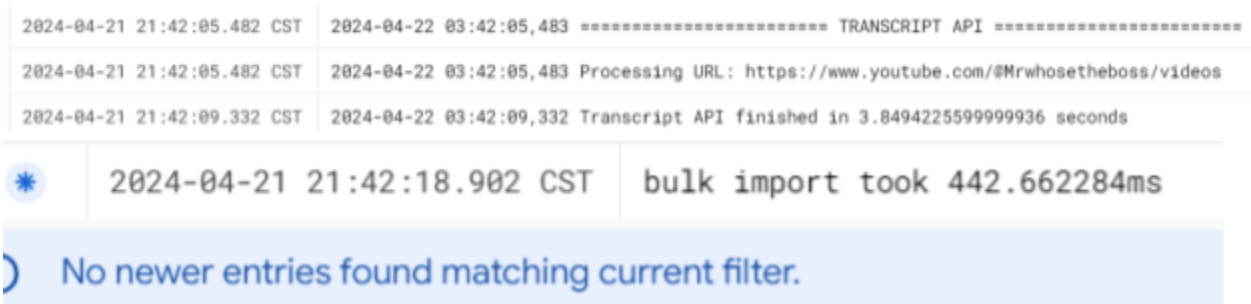


Fig. 7.1.2 (h) - Overall Time Processing + Upserting + Indexing

7.2 Usability Evaluation

We will also conduct user studies with at least 20 participants, to make sure end-users can easily use our system from top to bottom, without facing serious confusion. To that end, we plan to conduct mostly interviews with individual users, with a focus group thrown in for good measure.

7.2.1 Evaluation Procedures

Our user study target group will be fellow A&M students, which we believe more than encompasses the system's target audience. Script Search was designed to be used by anyone who uses YouTube, so a population of college students is as representative as any other. We will, of course, exclude those under 18 years of age and any students who do not speak English, as well as anyone without a functional knowledge of YouTube and the Internet.

To contact participants, we will email our personal friends, coworkers, and others we know at the university. The content of the emails will be of the following form:

> Howdy {name}!
>
> I am reaching out today to invite you to participate in a user study for my CSCE 482: Capstone Design project. The purpose of the study is to evaluate the usability of our application, ScriptSearch, which is a web application for searching YouTube transcripts.
>
> To be eligible for the study, we need you to:
> - Be fluent in English
> - Be 18 years of age or older
> - Have some familiarity with YouTube as a service
>
> As participants in the study, all you will need to do is interact with our application while I (or one of my team members) supervise and answer any questions you may have. Following the study itself, we have a short survey for you to complete.
>
> The study should take no longer than 15 minutes to complete, and, though we would prefer participation in-person, we are willing to conduct it virtually.
>
> Don't hesitate to contact me if you have any questions, and please let me know if this sounds interesting to you!

> Thanks in advance,
> {one of our names}

Our user studies will be *primarily one-on-one interviews*, which we will do our best to conduct in-person, wherever is most convenient. We will conduct a post-study survey to get a better idea of our respondents' quantitative and qualitative thoughts on our product. During the study itself, the participant will be encouraged to verbalize their thoughts as they explore the system. We don't want to explain too much, as we want to determine how self-explanatory our system is; with that said, we will probably explain a little about the nature of the system to mitigate initial confusion.

Given the nature of our application, we have no need to collect any identifying information about respondents themselves, so there are no privacy or confidentiality concerns. The only potential discomfort a user could experience would be related to the YouTube videos returned by their search query. We will, however, want to collect data regarding the time it took for the user to accomplish certain key tasks, such as reading the instructions, as well as the total time their search(es) took. In addition, we will synthesize qualitative data about the areas that gave them trouble and why. This data will be stored in a Google Sheets file and will be de-identified by only storing an anonymous "interview ID".

Our planned post-survey questions include:

- How easy was it to use the application overall? [1-5, very difficult - very easy]
- How fast did you perceive the search to be? [1-5, very slow - very fast]
- What limitations on the search, if any, did you find frustrating? [free response]
- How did you feel about the feedback provided by the website? [1-5, not helpful - very helpful]
- How did you feel about the overall look and feel of the website? [1-5, not understandable - very understandable]
- What, if anything, would you change about the application? [free response]
- How satisfied were you with the application overall? [1-5, deeply unsatisfied - very satisfied]
- Any other comments or feedback you'd like to share with us? [free response]

7.2.2 Evaluation Results and Discussion

The results received from conducting our usability evaluation are chronicled and analyzed below. Each response helps us build toward our definition of (usability) success: users should be able to easily understand intended actions on the website, complete their queries

quickly, and leave the website overall satisfied. We conducted relatively informal, unstructured interviews, followed by a post-study survey: the results of both phases can be found in this section.

Our interviews gave us much insight into how users interact with our system when given minimal instruction or explanation. Some common threads of the feedback given by these interviewees and the notes we took of their experiences are as follows:

- Many users experienced an initial period of confusion when interacting with our site: the URL input proved especially confounding at first.
- Some users found the visuals hard to look at, especially in dark mode. In particular, users expressed that the contrast between the red we use and the black background occasionally made elements difficult to see.
- A major complaint users had was that searches didn't return expected results: they'd put in a channel and search for a phrase they know is said on the channel, but our search would come up empty.
    - The cause of this is the inconsistency and inaccuracy of YouTube's automatic transcripts, which struggle to understand words when spoken quickly or with an accent, which happens a fair amount.
- Most users were quite impressed with the speed and visual polish of the website, especially our animations, and the aesthetics of our card view.
- Users were generally in agreement that the problem ScriptSearch solves is a good one, with genuine applications to their everyday lives.

The most important result from this portion of our evaluation was the overall satisfaction users experienced after interacting with our system. From our post-study survey, we found that most users were satisfied with the website in general. Figure 7.2.2(a) shows the distribution of responses to a question to that effect. As this question was very general in nature, we further probed users about a handful of specific areas to determine which parts may have left them less than satisfied.

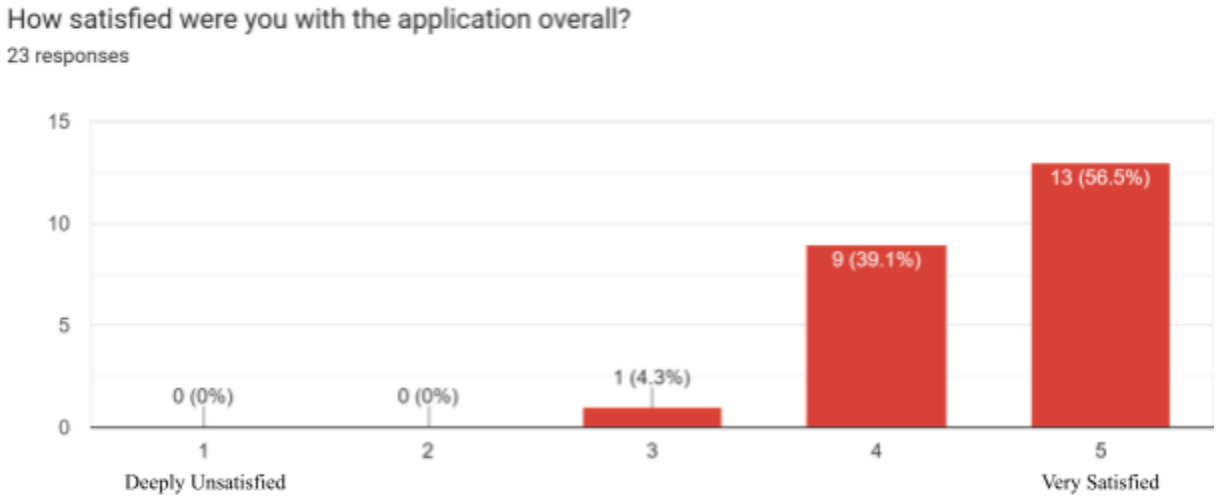


Fig. 7.2.2(a) - User Satisfaction

It was also in our best interest to make our system simple to use and understand. Figure 7.2.2(b) shows how easy interviewees found navigating and using our website. Overall, people found it quite easy to use, with UI elements being relatively self-explanatory and obvious in their intended usage. Additionally, our on-screen explanations and instructions were apparently helpful in understanding the intended application flow.

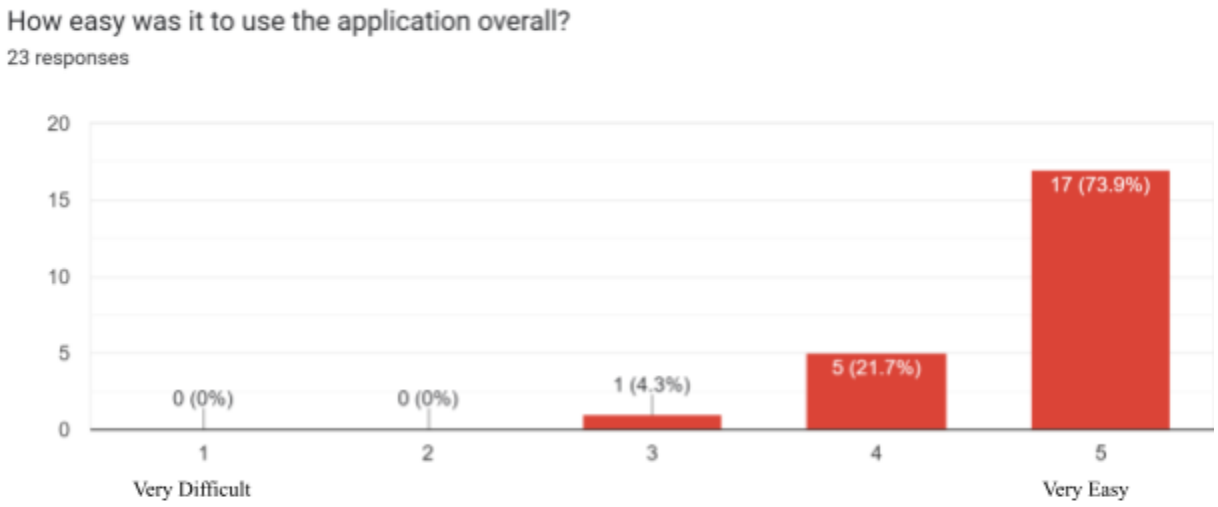


Fig. 7.2.2(b) - Ease of Use

When developing the project we theorized that perceived speed, a subjective statistic separate from actual speed, would be vital. This is because despite the actual speed of the application, if the application does not communicate progress appropriately with the user, the perceived speed would be immensely slow. However, with sufficient communication, the user would perceive the speed of the application to be quick. Figure 7.2.2(c) below shows the perceived speed of the application.

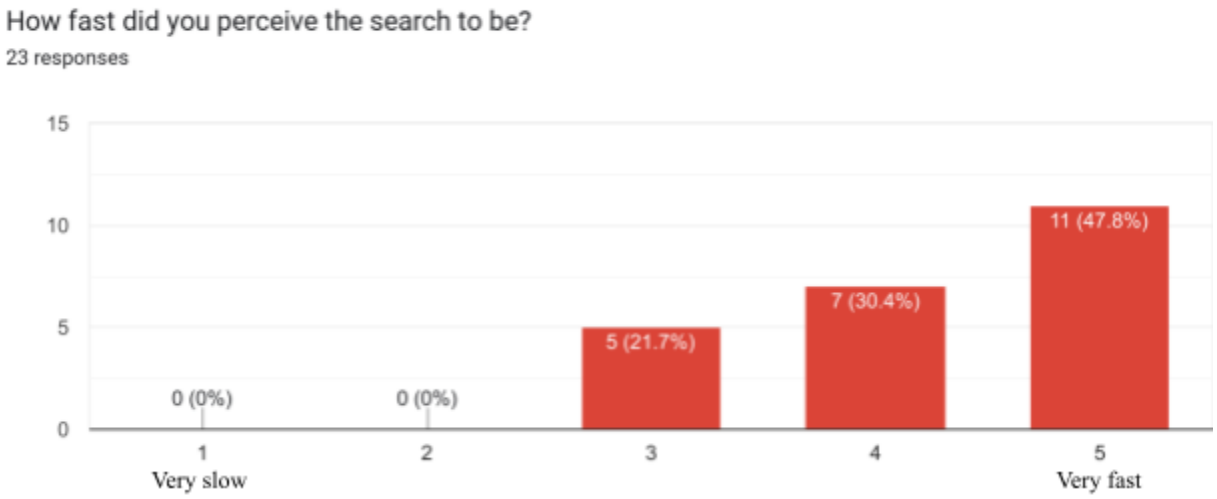


Fig. 7.2.2(c) - Perceived Speed

We wanted to ensure that the website effectively communicated its state to the user, as well as any errors and the like that occurred. We gathered user opinions and synthesized them into Figure 7.2.2(d) seen below. It seems that some users found feedback provided by the website lacking, but we believe that this was in large part due to the question being vague. When asked directly about the system's communication to the user (i.e. dynamic loading text, error messages, etc.), users actually seemed quite pleased; however, we believe the question on the survey was unclear and users understood it to be asking something else entirely.

Fig. 7.2.2(d) - Website Feedback

Users were also satisfied with the aesthetics of the website: they didn't find our UI distracting or hard to look at, demonstrated in Figure 7.2.2(e). With that said, some interviewees found certain portions of our UI (especially the red color we use quite a bit) did not contrast very well with our background at times, leading to some disapproval.

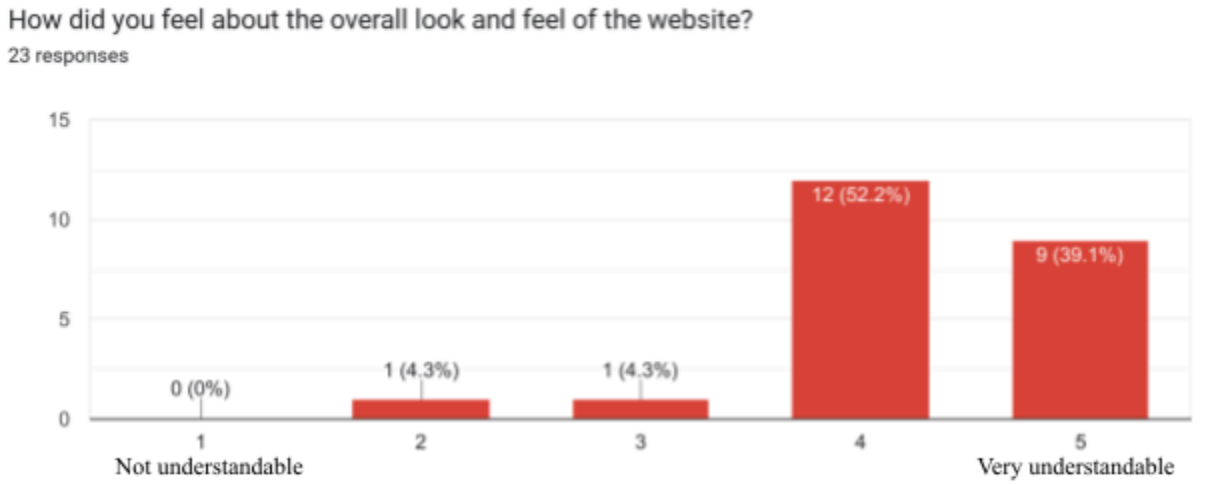


Fig. 7.2.2(e) - Aesthetics

Our free response questions gave us a lot of interesting, actionable feedback. For instance, many of our initial interviewees expressed a desire to have a greater amount of control

when it comes to the results shown, such as sorting and filtering options. Though we deemed filtering results out of scope, we did add a simple option to sort results, giving users more control over the order that videos are displayed after performing a search query. In addition, we made many changes to the "card" view based on what users said they would like to see for each video: we added upload date, duration, and the number of transcript matches found for the user's query. Other responses indicated that users wanted the instructions to be displayed in a more interesting way, as reading an entire paragraph of text was unappealing: to that end, we added "tooltips" that display relevant information about an input when the corresponding element enters focus.

Despite our attempts to make the user interface as clear as possible, some users failed to discover some features of our application that we thought were obvious. For instance, some users did not realize they could click on the cards to bring up more information regarding the transcript matches for that particular video. On top of that, some users found it unclear that the blue timestamp present in the "hits" view was a timestamped link to the video in question. Users also had very mixed reactions to the concept of providing a URL to the website: some thought it was required, while others believed it had to be done alone before a query could be performed.

8 Discussion

Our decision to use the Agile methodology proved invaluable, enabling us to dynamically adjust our project scope and goals while maintaining a working prototype throughout development. Early prioritization of integration-related tasks simplified major challenges faced by other teams, and focusing on core functionality in initial sprints allowed us to address edge cases in the final stages, resulting in a robust product. Communication and documentation were key, though challenges arose due to developers specializing too deeply in specific areas, which sometimes obscured bug origins.

The initial ideation phase of the project was also very important and useful. It forced us to consider a wide variety of potential solutions, then narrow down our scope to that which we believed to be the most feasible, interesting, and useful. Getting feedback on our initial prototypes also gave us an early understanding of what users may expect from our interface. All of these elements aided our development significantly, as we had a better grasp of the fundamental expectations and assumptions underlying our solution.

During development, we faced a number of technical challenges, including modifying the YT-DLP Python module to better suit our needs, enhancing our understanding of YouTube's internal API. This led to a significant backend shift from Python to Golang to utilize its

concurrency features and the "Innertube" API, which avoided IP blocks and rate limits. We also optimized our scraping architecture by having each cloud function handle multiple URLs simultaneously using Goroutines, significantly improving runtime and reducing cold starts.

Memory limitations in our cloud functions, especially those handling large transcript data, posed significant hurdles. We managed significant reductions in data payloads by pruning unnecessary details, though challenges persisted due to the initial choice of search framework. If repeated, we would opt for Elasticsearch over Typesense to handle these large data volumes more efficiently and would establish a more effective database scheme from the project's onset.

User feedback, gathered through interviews and surveys, was crucial though limited by the project's timeline. Though it was overall extremely productive, future iterations would benefit from recruiting even more interviewees to glean more actionable feedback. Our functionality testing was also more difficult than anticipated, primarily due to the complexity of performing integration and performance tests automatically. Setting up the testing architecture earlier on will certainly prevent these challenges and potentially allow us to make debugging even easier.

In summary, continuous research, community engagement, and flexibility in our development approach played pivotal roles in overcoming obstacles and enhancing the project's success. Future projects would benefit from maintaining these strengths while improving documentation, bug tracing, and database management to prevent feature creep and ensure a more streamlined development process. In addition, we believe that our desire to optimize our speed slowed our development at times, as did our tendency to constantly research alternatives rather than commit to a particular path.

9 Future Work

In the future, there are a number of features that we would like to add or improve.

First, we would like to improve the mobile view of our application. As of right now it is just a simple, static scaling of our desktop application that still looks bad when the device is too small. We would like to implement a dynamically scaling solution in the future.

An improvement on an existing feature that we would like to implement is for the ability to process multiple URLs simultaneously. This is referring to the allowing the user the ability to input multiple URLs in the URLs search box on the frontend and then searching through all videos in the respective playlist/channels that are listed.

One of the most requested features of our application during user testing was the ability to sort the results of their query. Currently the sorting algorithm only sorts after getting results. We could change this to sort the results on the backend during the search itself, as this would better align with user expectations about such a sorting feature. We would also like to add filtering to the search results, additionally allowing the user to remove content from the results if uninterested.

Next, we would like to create a Google Chrome extension for our web application. When searching through transcripts, navigating to a separate website can be an inconvenience to the user. The Chrome extension would allow the user to search through the current video they are watching on YouTube itself, additionally, it would allow the user to search through their list of recommended videos.

The last main feature we'd want to add in the future is integration with a speech-to-text artificial intelligence model, such as Whisper AI. Currently, our URL processing is limited only to videos with transcripts enabled, and, while this is the majority of them, we'd like the ability to search videos that have no generated transcripts on YouTube. Integrating a model like Whisper will give us the ability to generate transcripts ourselves, increasing the depth of the application's usefulness.

10 Conclusion

This paper outlines the process by which we created a YouTube transcript search engine, allowing users to easily search for a phrase within a group of YouTube transcripts. In essence, end-users are able to quickly and easily find a YouTube video based on a particular word or phrase mentioned in the video, along with a timestamped link to the particular moment where it was spoken with the goal being to help users recall the specific video that contains the information they remember. In the process, we developed a robust service, making use of cloud technologies and web development frameworks to better deliver the URL population and searching interface. Our evaluation, which included interviews with users and unit/integration tests, revealed that our application was functionally complete and easy to use, according to the interviewees.

References

Alberti, C., & Bacchiani, M. (2009, December 4). *Automatic Captioning in YouTube*.

Blog.research.google.

https://blog.research.google/2009/12/automatic-captioning-in-youtube.html

Alhalabi, W. S., Kubat, M., & Tapia, M. (2007). Search Engine Ranking Efficiency Evaluation Tool. ACM SIGCSE Bulletin, 39(2), 97–101. https://doi.org/10.1145/1272848.1272894

Arrue, M., Vigo, M., & Abascal, J. (2007). Web accessibility awareness in search engine results. *Universal Access in the Information Society*, 7(1-2), 103–116. https://doi.org/10.1007/s10209-007-0106-8

Banon, Shay. *Elasticsearch: The Official Distributed Search & Analytics Engine.* (2014, February 12). Elastic. https://www.elastic.co/elasticsearch

Boyer, R. S., & Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, *20*(10), 762–772. https://doi.org/10.1145/359842.359859.

Dessaigne, N., & Lemoine, J. (2012, October 10). *Algolia*. Algolia. https://www.algolia.com/

Gill, P., Arlitt, M., Li, Z., & Mahanti, A. (2007). Youtube traffic characterization. *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement - IMC '07*. https://doi.org/10.1145/1298306.1298310

GMI Blogger. (2021, October 4). *YouTube User Statistics 2020*. Official GMI Blog. https://www.globalmediainsight.com/blog/youtube-users-statistics/

Jopik. (2021, July 2). *Filmot - Subtitle and Video Metadata Search Engine*. Filmot.com. https://filmot.com/

Karim, J., Hurley, C., & Chen, S. (2005, February 14). *YouTube*. YouTube. https://www.youtube.com/

King, M. (2023, August 2). Generate a YouTube transcription index w/ Whisper. Algolia. https://www.algolia.com/blog/engineering/generate-a-transcription-index-for-youtube-using-whisper/

K. Sharma and K. Gupta, "Lossless data compression techniques and their performance," 2017 International Conference on Computing, Communication and Automation (ICCCA), Greater Noida, India, 2017, pp. 256-261, https://doi.org/10.1109/CCAA.2017.8229810

Leporini, B., Andronico, P., Buzzi, M., & Castillo, C. (2008). Evaluating a modified Google user interface via screen reader. *Universal Access in the Information Society*, 7(3), 155–175. https://doi.org/10.1007/s10209-007-0111-y

Strobel, B. (n.d.). *YTKS - Youtube Keyword Search*. Ytks.app. https://ytks.app/

Typesense. (n.d.). Typesense.org. https://typesense.org/

YouTube transcript search. (n.d.). https://chromewebstore.google.com/detail/youtube-transcript-search/eeebipnojmgobognppffkenhdoidendi

YouTube-DL. (2021, December 17). GitHub. https://github.com/ytdl-org/youtube-dl

YT-DLP. (2023, December 30). GitHub. https://github.com/yt-dlp/yt-dlp

Appendix A: Project Management

A.1 Team Agreement

Team Agreement

A.2 Software Development Methodology

We decided that the best, most effective way to tackle a project of this magnitude is by using the Agile methodology, which breaks the project down into phases (sprints) and emphasizes continuous collaboration and improvement. At the end of each of the four 2-week sprints, we will aim to have a minimum viable product (MVP) to showcase the additional features implemented during that sprint. By the end of development, we hope to have created a fully functional system, as well as potentially adding additional features not originally included in the scope of the project (stretch goals).

User stories and product/sprint backlogs will be handled through a Jira project management interface, which unifies and simplifies the management process. Scrum meetings will be held virtually via Discord at times agreed upon by team members, as well as during class meeting times.

By following the Agile methodology, we are able to be user-focused throughout our development and responsive to feedback. This is done by conducting user studies at the end of each sprint to determine what current features may need to be improved and other features that users may want that were not considered.

A.3 Implementation Schedule

The tentative implementation schedule for our project is shown below, including necessary and desired content for the Minimum Viable Product (MVP) at the end of each sprint.

Sprint 1: Feb 5th - Feb 19th

- Desired Functionality:
    - Simple website [need]
    - Cloud backend communicating with website [need]
    - Simple CI/CD prototype [need]
- No transcript searching or storage yet
- Mainly focused on setting up frontend/cloud environments
    - Implement integration to understand how frontend/backend communication will work

Sprint 2: Feb 19th - Mar 4th

- Desired Functionality:
    - Frontend search bar finalized [need]
    - Transcripts can be accessed and stored via cloud [need]
    - Frontend/backend integrated via simple API [need]
    - Return single transcript search results [want]
    - CI/CD pipeline mostly implemented [need]
        - Want unit testing as soon as possible
- Broadly, want to be able to test searching on a single, known transcript for simplicity

Sprint 3: Mar 4th - Mar 25th

- Desired Functionality:
    - Allow searching of multiple transcripts simultaneously [need]
    - Implement timestamp links in search results [need]
    - CI/CD pipeline finalized and testing integrated [need]
- Expand Sprint 2's goal into being able to search arbitrary transcripts

Sprint 4: Mar 25th - Apr 8th

- Desired Functionality:
    - Searching transcripts fully implemented [need]
    - Frontend results page fully implemented and displaying search results as desired [need]
    - Ensure overall polish is acceptable [need]
- Want to finalize overall system and ensure tests meet desired standards

A.4 Software Development Artifacts

As we elected to use the Agile methodology, we determined our product backlog in advance, which contains every individual task we must complete in order to create a functional system overall and is seen in Figure A.4(a).

Fig A.4(a) - Product Backlog

Our overall product burndown chart, showing our forward development of the project, is seen here in Figure A.4(b).

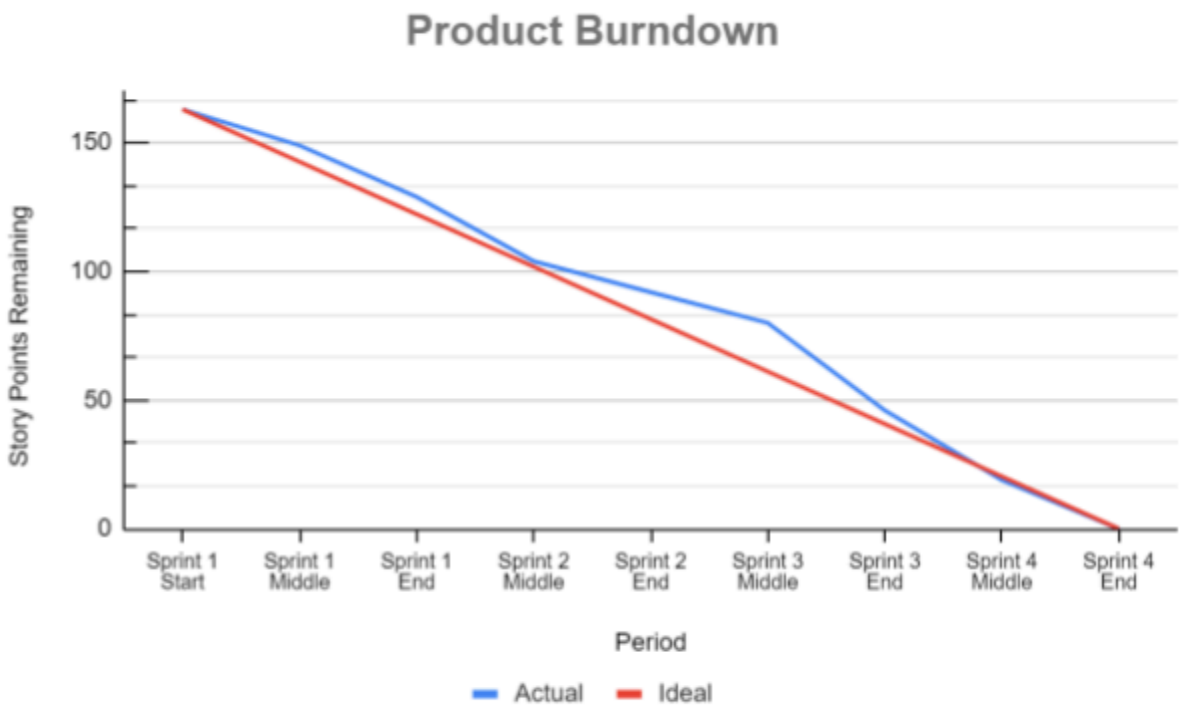


Figure A.4(b) - Product Burndown

In addition, we have the backlogs for each individual script and the corresponding burndown charts that show our progress over time in each sprint. These backlogs are shown below in Figures A.4(c)-(f), and burndowns are in Figures A.4(g)-(j).

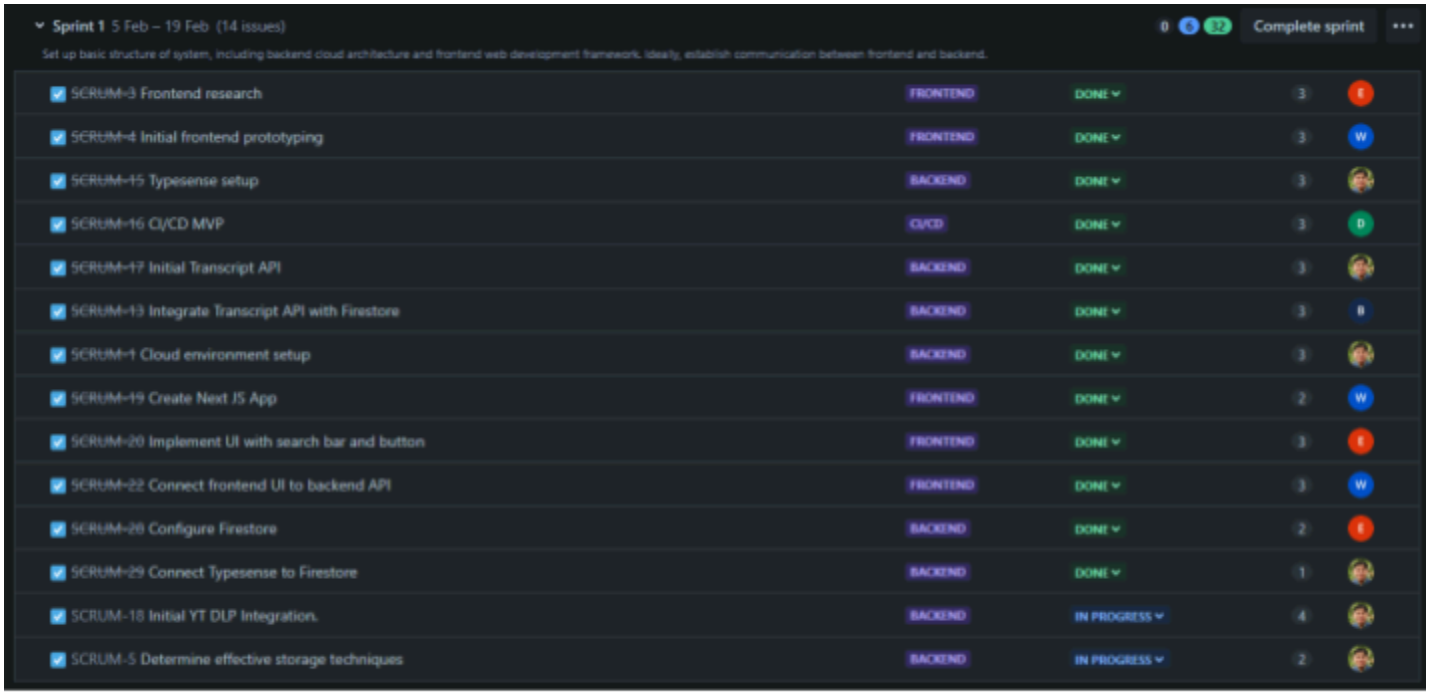


Fig A.4(c) - Sprint 1 Backlog

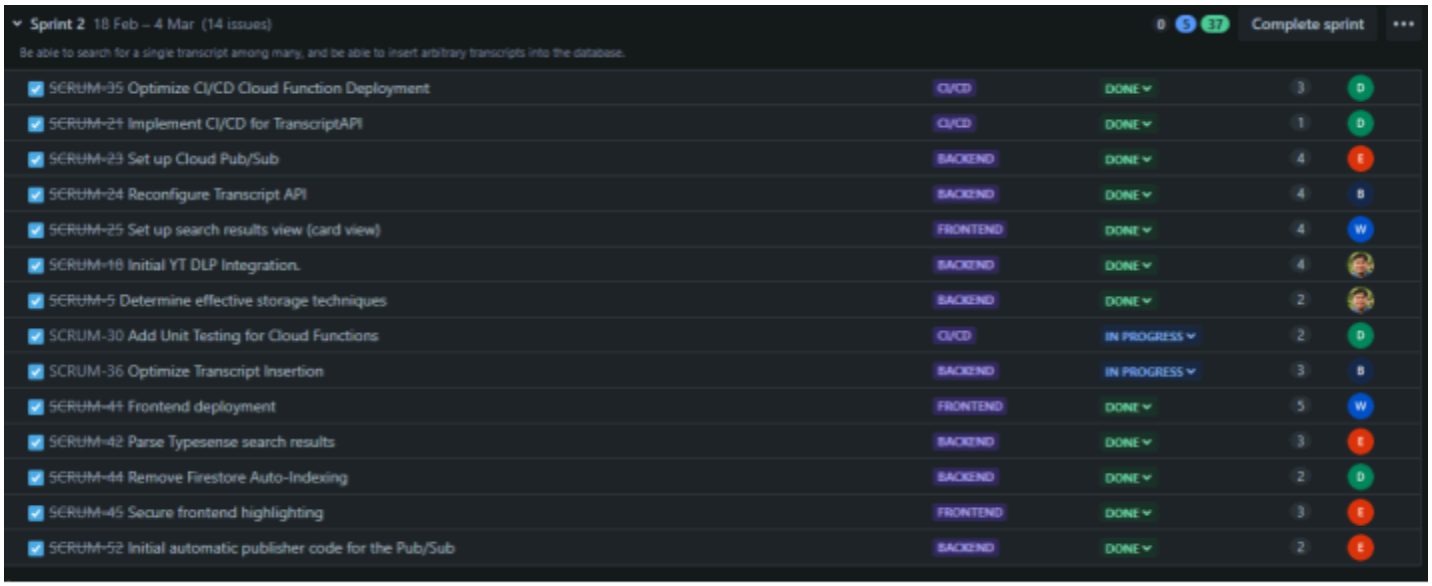


Fig A.4(d) - Sprint 2 Backlog

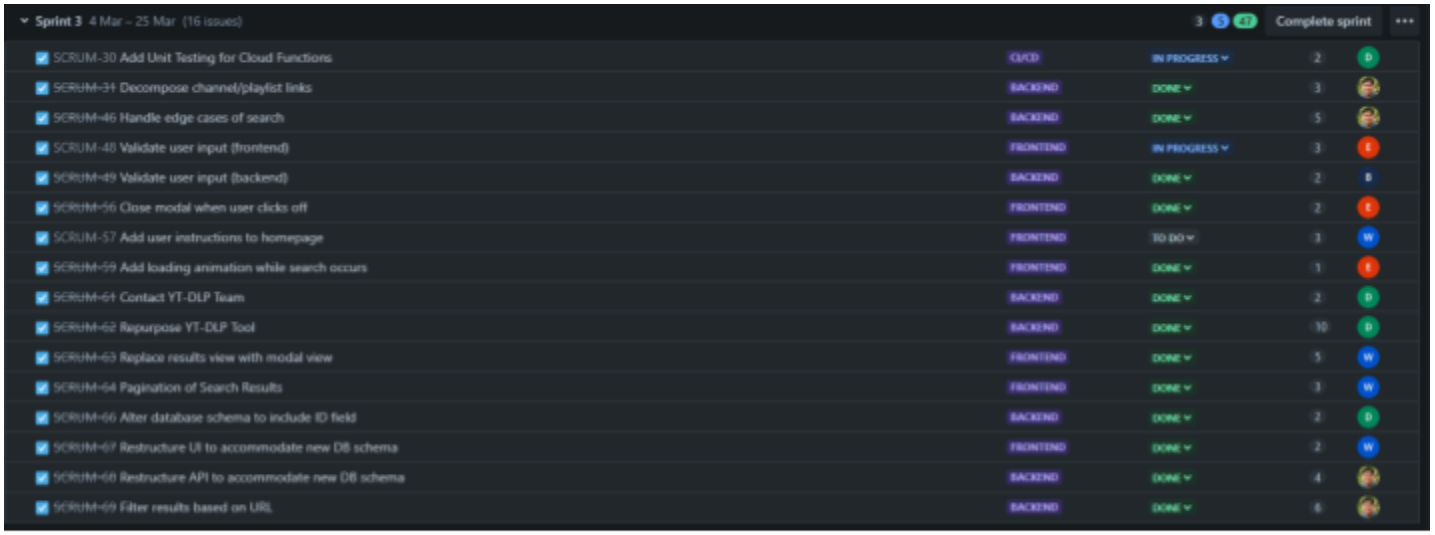


Fig A.4(e) - Sprint 3 Backlog

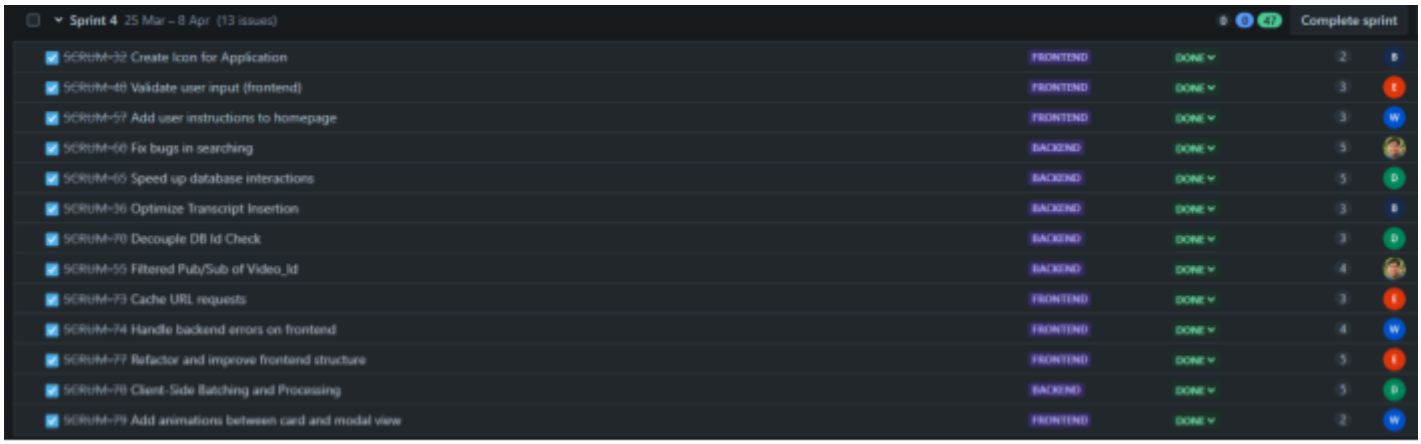


Fig A.4(f) - Sprint 4 Backlog

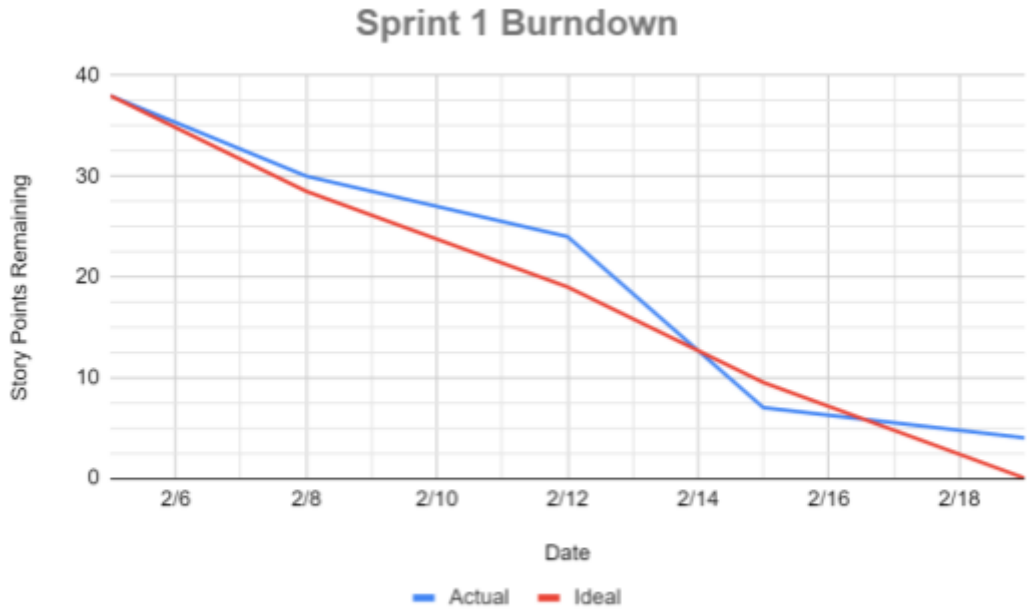


Fig A.4(g) - Sprint 1 Burndown

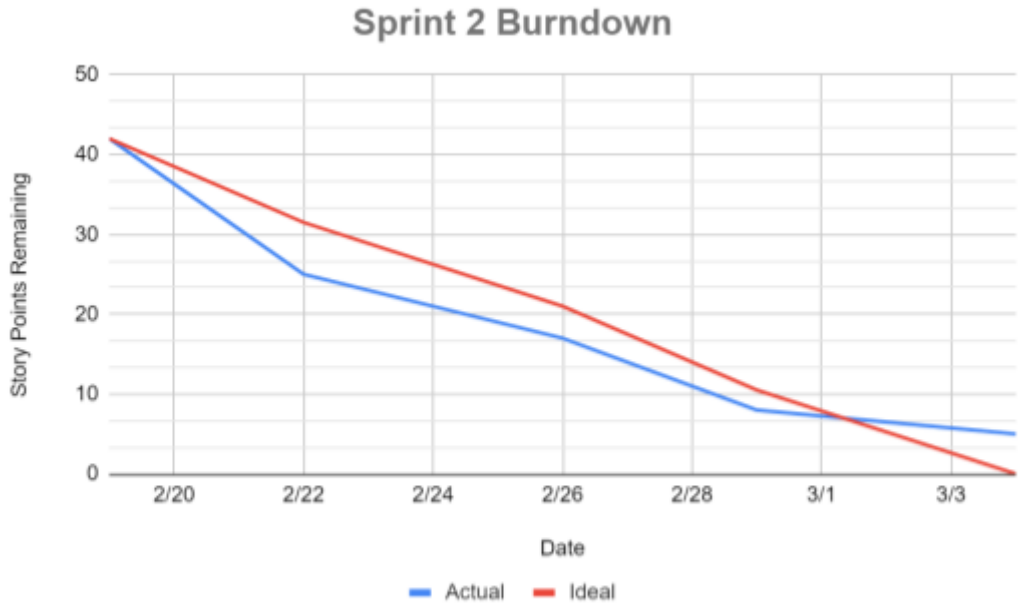


Fig A.4(h) - Sprint 2 Burndown

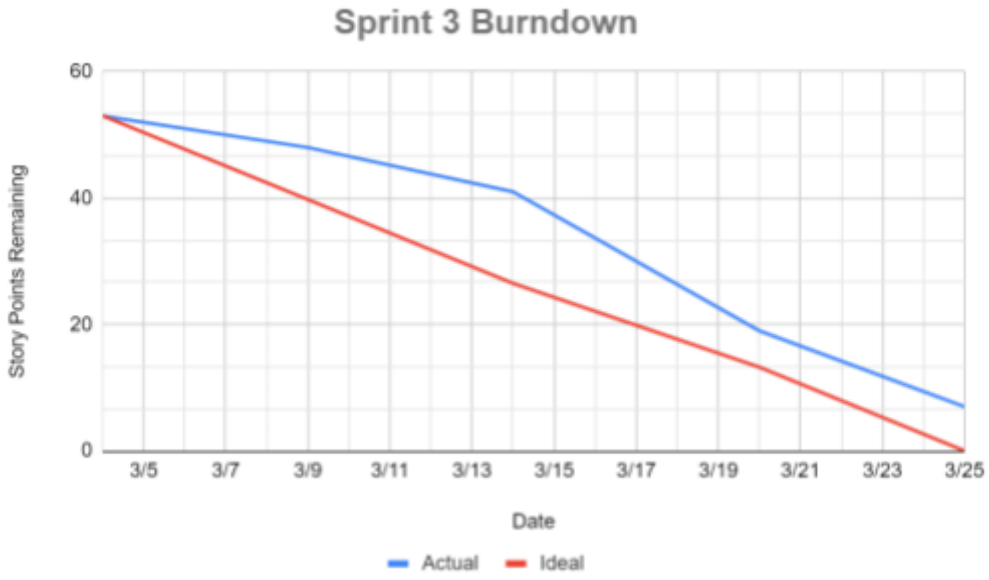


Fig A.4(i) - Sprint 3 Burndown

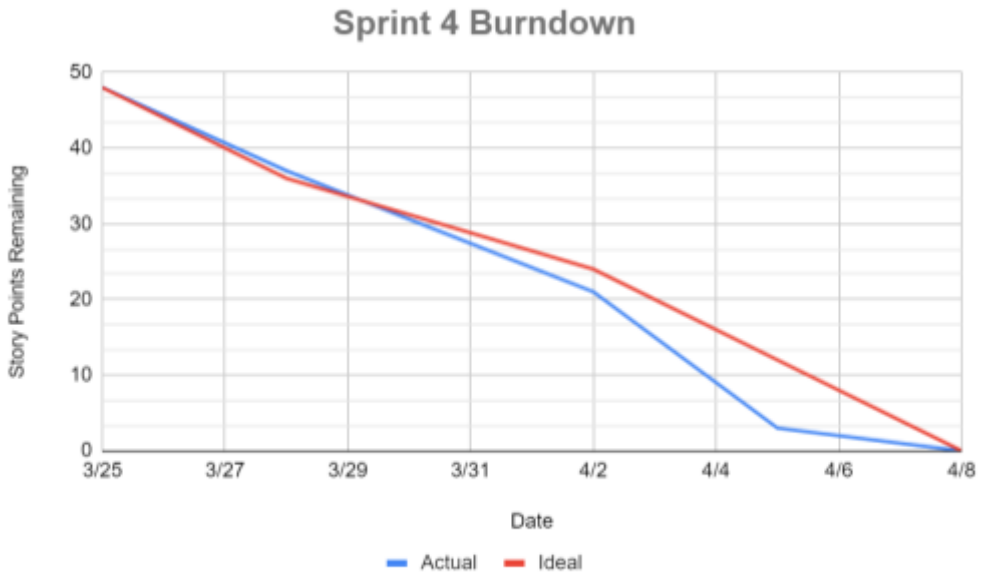


Fig A.4(j) - Sprint 4 Burndown

We also held weekly scrum meetings, in accordance with Agile best practices. The summaries of these meetings can be found attached below.

Team Weekly Report #1

Team Weekly Report #2

Team Weekly Report #3

Team Weekly Report #4

Team Weekly Report #5

Team Weekly Report #6

Team Weekly Report #7

Team Weekly Report #8

Team Weekly Report #9

Team Weekly Report #10

Team Weekly Report #11

A.5 Budget

The below table summarizes services our project requires, as well as estimated potential costs. Our use of Google Cloud Platform (GCP) services, in particular, should remain within the free tier, but may end up requiring funds if we greatly exceed expected usage.

Service/Software	**Purpose**	**Estimated Costs**
GitHub Pages	Deploy website	$0.00
Typesense	Searching transcripts	$130.00
GCP Functions	Triggering separate parts of backend, individual transcript downloaders	$0.00
GCP Firestore	Storing transcripts	$0.00
GCP Pub/Sub	Coordinating parallel transcript download	$0.00

Appendix B: Implementation Details

The details of our implementation can be found on our wiki here (see frontend/backend implementation):

https://github.com/Script-Search/Frontend/wiki

The contents of these pages are included below for reference.

Backend Implementation

Backend is split between two parts:

- Scraping and indexing YouTube videos and transcript
- Searching through transcripts

To accomplish this, we use a centralized API that routes requests from the frontend.

This way, we are able to distinguish between these two cases properly and support both functionality seamlessly.

This centralized API uses Python and uses a modified version of YT-DLP that enables faster meta-data extraction of videos so that their ids can be piped for potential scraping.

Scraping

Transcript API

When a user enters a URL in the frontend (video, playlist, channel), it kickstarts the scraping pipeline within the backend.

The `transcript-api` first determines the relevant info to gather from the URL, either extracting the id from the video or decomposing the playlist/channel videos to create a list of video ids.

Once this is gathered, `transcript-api` pushes the entire list to a Pub/Sub topic and returns a response to the user immediately.

Cache Checking

Another Cloud Function is triggered to read from the Pub/Sub topic in order to decouple the cache checking **for now**.

Golang is used within this part as the client libraries support more endpoints, allowing us to check multiple ids in just one query AND also allows us to merely check for id existence, saving on precious memory.

For the ids not present in our Firestore database (which acts like our cache), it pushes lists to another Pub/Sub topic for another cloud function.

During this step, we implement client-side batching to reduce the overall network bandwidth of our system and also to leverage bulk upsert endpoints later down the line.

This client-side batching takes an overall list and splits it into batch lengths of `BATCH_SIZE` to publish to the topic.

In the future, this step could be combined with the transcript-api again as we leverage the firestore's REST API route better.

YT-URL-Consumer

This Cloud Function is triggered from the Pub/Sub topic from the above in order to be completely decoupled and asynchronous.

This function leverages Golang's concurrency with goroutines in order to spin up lightweight threads that will handle the abundant I/O bound tasks.

For each video id, we must make a request to get some metadata information about the video (using Innertube API).

Then, after retrieving this information we can extract the transcript link and make a request to this.

Finally, we process and sanitize the responses from both requests in order to build out a more readable data schema for our Typesense database.

The results of each batch are published to a final Pub/Sub topic that handles the database syncing.

Upserting To Databases

Two separate Cloud Functions are triggered from a Pub/Sub topic in this step, each of them handling different database upsertions.

This is to increase the overall fault-tolerance of the system and keep both components decoupled.

Both of the Cloud Functions preprocess the incoming data minimally before using their respective client library (Firestore or Typesense) in order to do a bulk upsert, saving more cpu and network bandwidth instead of one at a time.

This syncs both databases essentially in parallel and pre-populates the results in the database before searching occurs.

As of writing this in 4/5/2024, this entire process takes ~6-7 seconds for 250 videos

Searching

The searching algorithm relies heavily upon Typesense as the basis.

The system is structured with the following components:

- Typesense Client: Responsible for initialization and communication with the Typesense API.
- Search Typesense: Executes search queries and processes search results.
- Find Indexes: Identifies indexes of words or phrases within transcript data.
- Mark Word: Marks occurrences of words within sentences using `<mark>` tags.
- Initialization Functions: Initializes the Typesense client and ensures readiness for search operations.

Individual Components

1. Typesense Client Initialization

   The `init_typesense()` function initializes the Typesense client with the necessary connection parameters, such as the host URL, port, and API key.

   It ensures that the client is created only once and is ready for subsequent operations.

2. Search Operation

   The `search_typesense()` function executes search queries on the transcript data stored in Typesense.

   It constructs the search query based on provided parameters and retrieves search results.

3. Find Indexes

   The `find_indexes()` function identifies the indexes of specific words or phrases within the transcript data.

   It distinguishes between single-word queries and multi-word queries and employs different strategies for each.

4. Mark Word

   The `mark_word()` function marks occurrences of a word within a sentence by wrapping them in `<mark>` tags.

   It employs regular expressions with word boundaries to find and replace occurrences, ensuring case insensitivity.

Frontend Implementation

The web app side (frontend) of the system mainly consists of querying the Transcript API (one or more times), then parsing and displaying the results to the user. The queries themselves, as well as the specifics of the results displaying process and additional features, are explained in more detail below.

API Queries

We split our connections with the backend into (at most) two requests: a URL request, and a search request. This is to easily cover all possible cases wherein the user provides one, both, or neither inputs. In particular,

- if the user provides *both* a URL and a search query, we make a URL request, then include the response in the search query that follows.
- if the user provides *no* URL, but a search query, only a search request is made, and the results are populated on-screen.
- if the user provides a URL, but *no* search query, only a URL request is made, and an on-screen message informs the user of its completion.
- if the user provides *neither* a URL nor a search query, no request is made, and an error message is displayed.

ScriptSearch 🔎

URL Requests

The URL request simply sends the user-provided URL to the Transcript API, receives a response, and waits a particular amount of time for the relevant database insertions to finish. The response provided by the API contains a field that is used to filter a search request, when present.

Search Requests

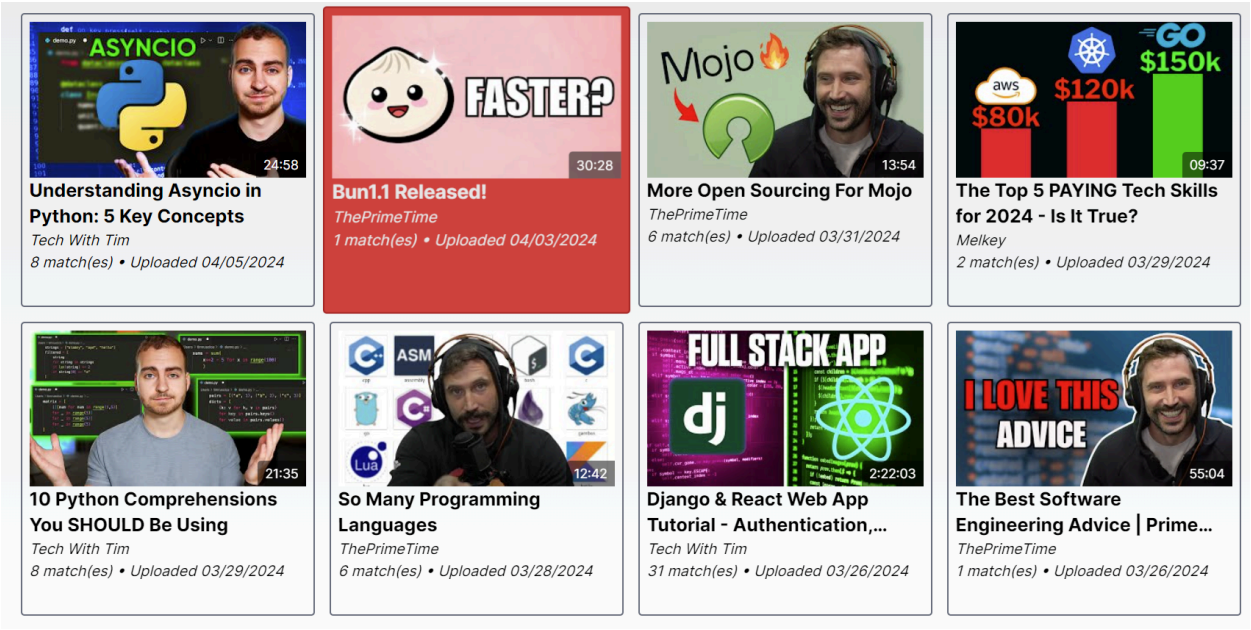
The search request sends the user-provided search query (word or phrase) to the API and receives a response containing all results, in a format that can be easily parsed into the card view (discussed below). If this request was preceded by a URL request, the response from that request is passed along back to the API, which will filter the search to only those transcripts associated with the link accordingly. Otherwise, no filter is applied, and our entire database is searched.

Results

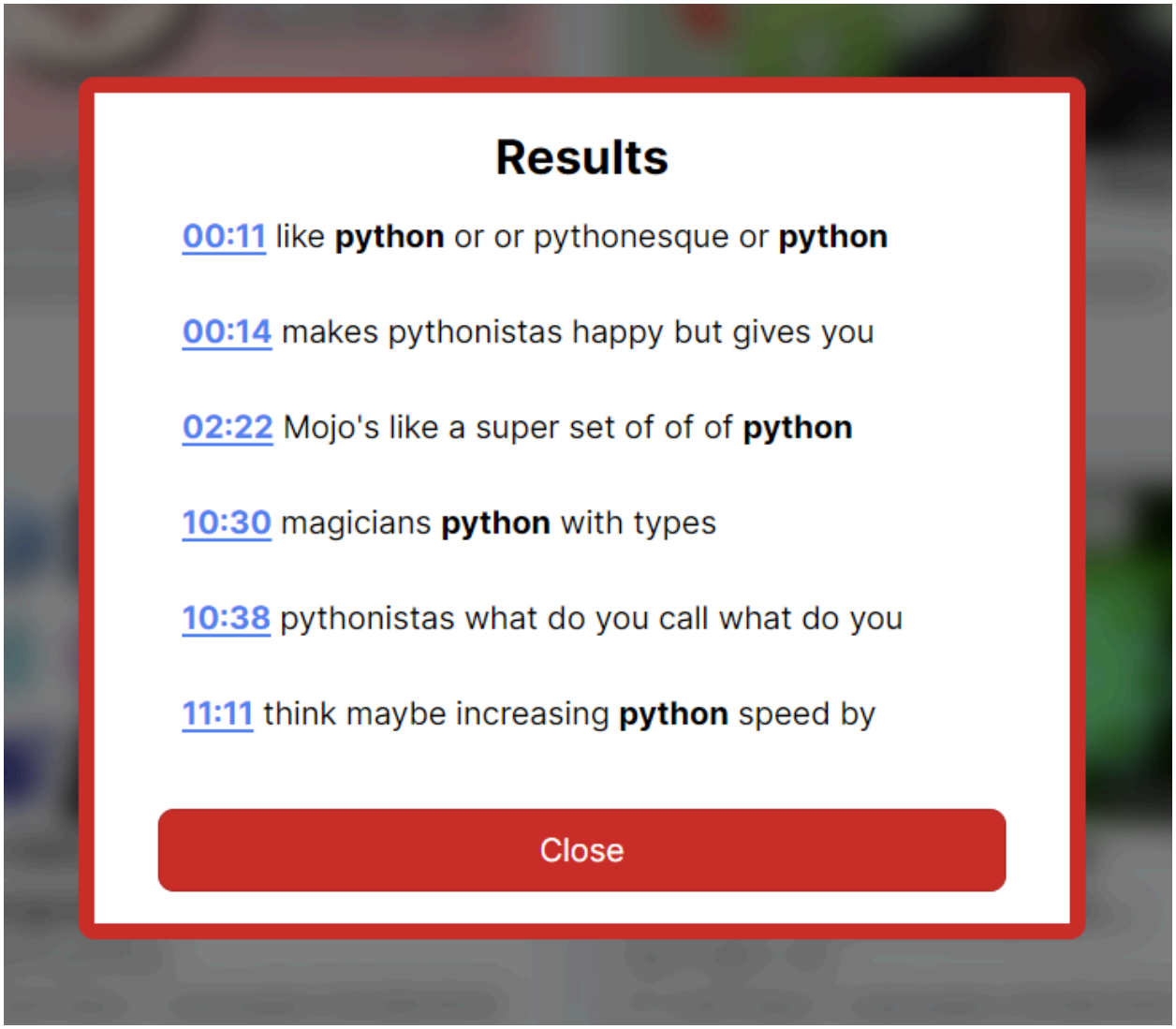
Search results are stored via an interface for each video containing necessary information: video_id, title, channel_id, channel_name, upload_date, duration, and a list of all query matches, with transcript snippets and relevant timestamps. The results from a search request to the API are guaranteed to be mappable directly onto a list of this interface, which can then be displayed to the user via our *card* view. Each video (element of the result list) becomes its own card, which can be clicked on to reveal the specific transcript matches in a "modal" (pop-up window). Using the video_id, each card can turn the timestamps of matches into appropriately timestamped links. Minor processing is also done to make timestamps and upload dates more readable.

For instance, searching for "Python" will give a card view that looks like this:

ScriptSearch 🔎



Clicking on a video will bring up an in-browser window that looks like this:



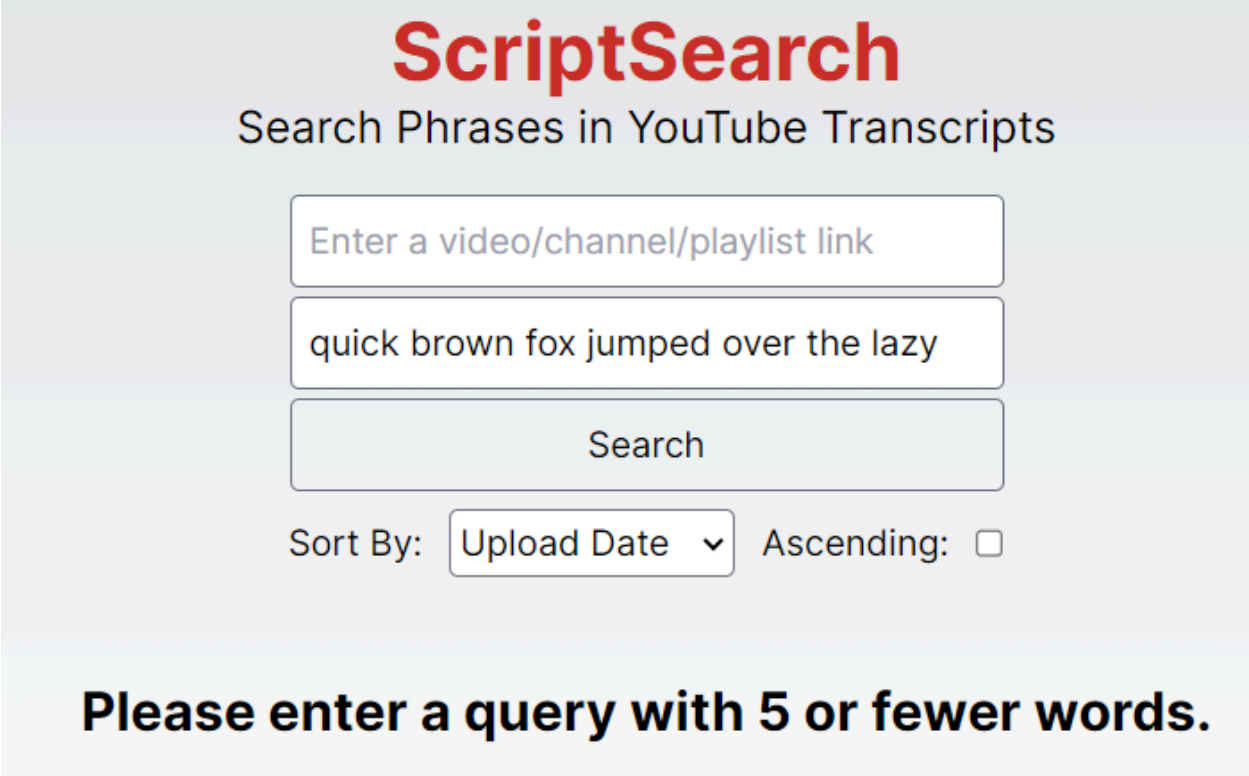
Additional Features

Error Handling & Validation

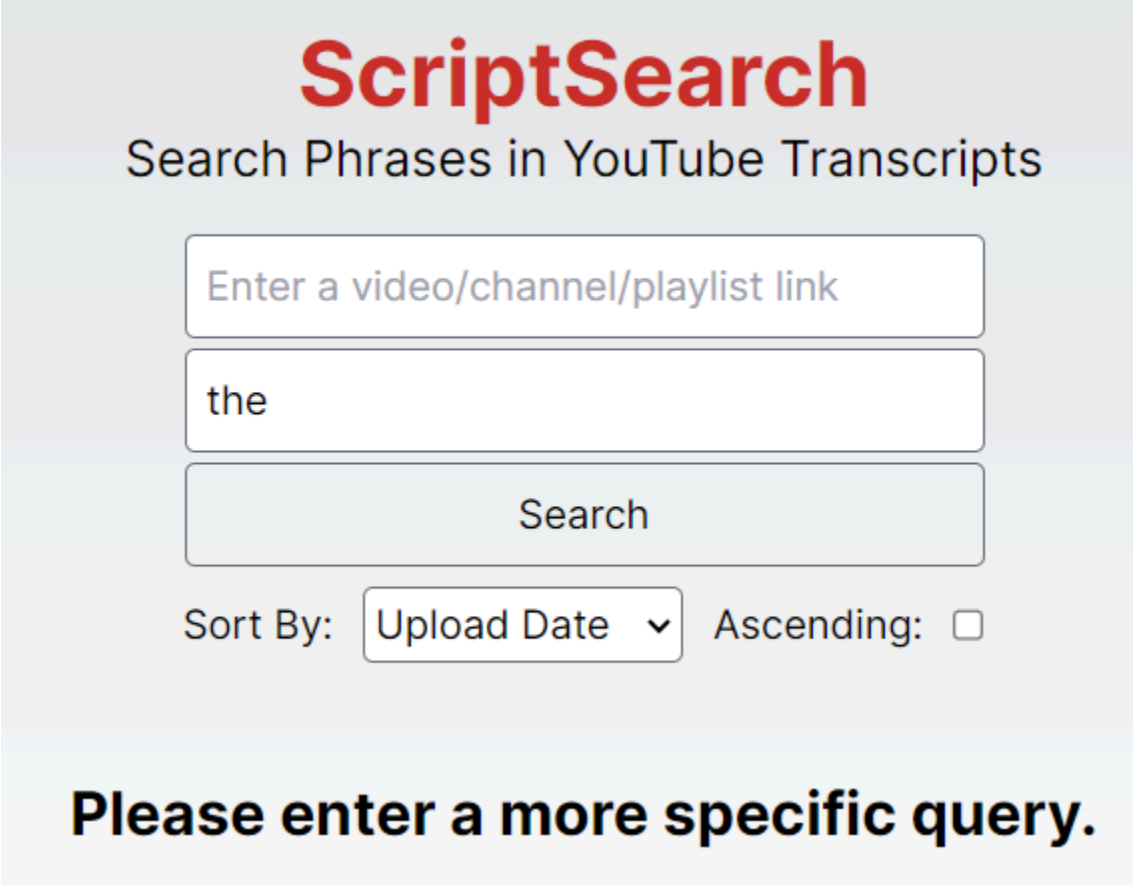
To make our searching feasible, we decided on limitations on word and character lengths for the query, and also opted to ban certain common words and ignore special characters. If the user enters a query breaking any of these rules, a message is displayed on-screen, such as in the following examples:

Entering a search query like "the quick brown fox jumps over the lazy dog" (longer than 5 words) results in the following:

ScriptSearch 🔎

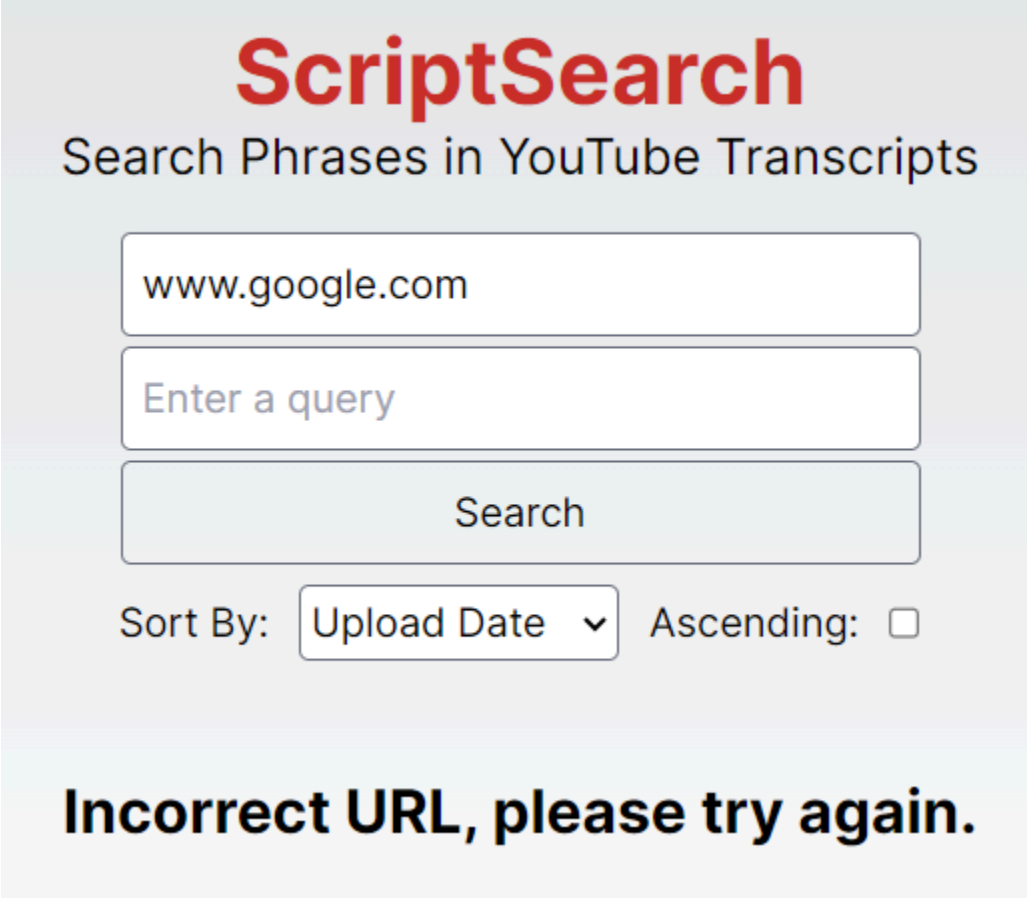


Entering a search query like "the" (too common of a word) results in the following:



In addition, any errors returned by the backend are handled similarly.

Entering a URL like "www.google.com" results in the following:



Caching

To improve usability, we implement a simple in-memory cache to store the five most recent URL requests. This is done by simply storing the URLs and the respective response given by the API (which should be fairly small).

When the user enters a URL already contained in the cache, we skip the URL request step entirely, cutting out an extra API request and (more importantly) 5-10 seconds of waiting.

NOTE: This cache is stored server-side, in the script's memory rather than the browser's local storage. As a result, refreshing the page causes the cache to empty, and URL requests will have to be completed at least once more before being cached.

Sorting

The user has the option to sort search results by one of the following fields: video title, channel name, upload date, duration, or number of matches. This is implemented simply by applying a custom sort to the search results, then (re)displaying them on the page. Sorting may be performed before or after a search has occurred, and the settings selected will apply

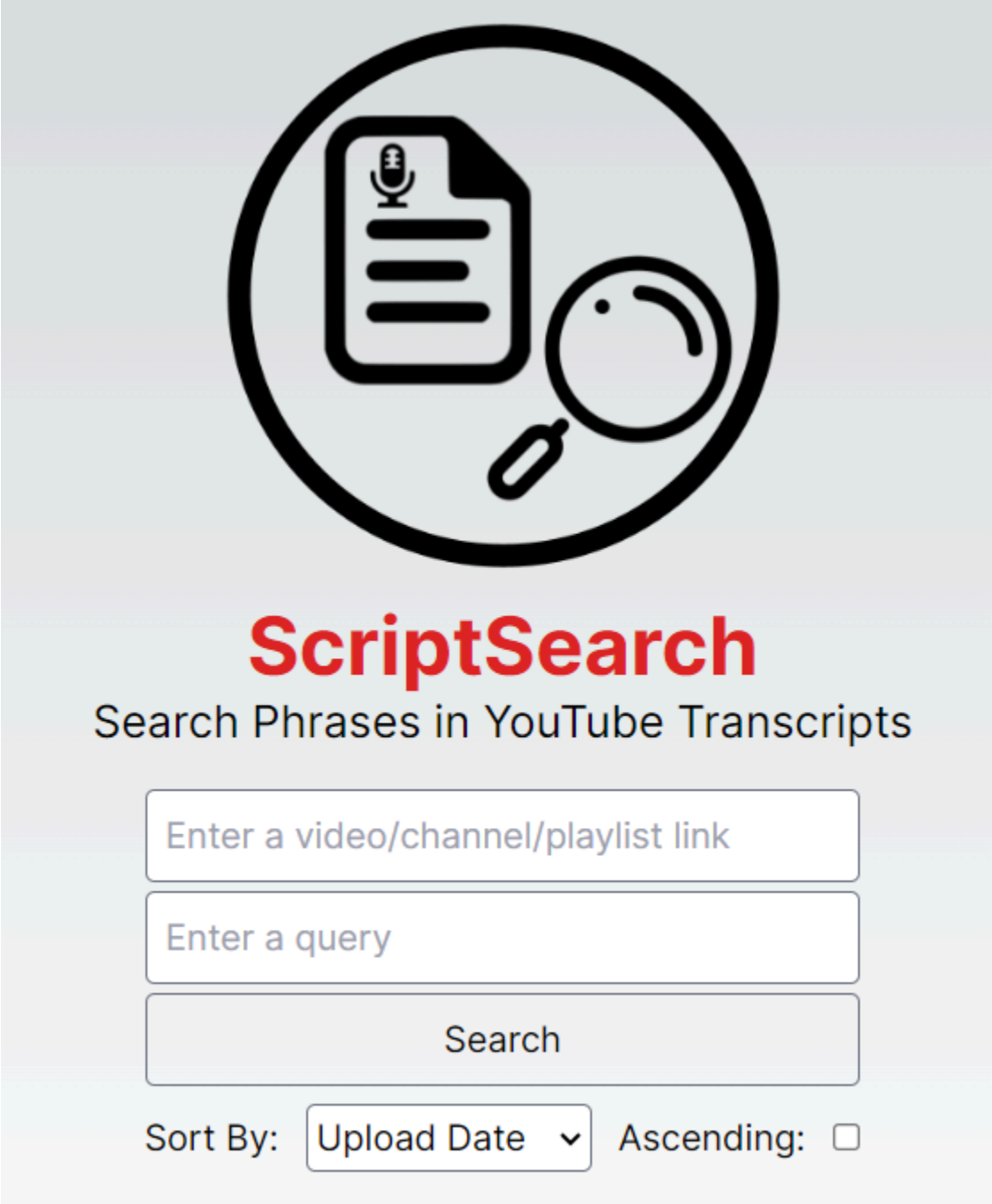
immediately. By default, results are sorted by upload date descending (newer videos appear first), as this is how our search is conducted in general.

NOTE: The sorting logic is contained entirely in the frontend and has no functional connection with the database itself. Sorting only rearranges whatever results the backend returned; it does not give new or different results based on the sorting criteria. That is, the backend will return some set of videos for the particular query made by the user, and applying a sort does not change anything about that set, other than the order in which elements are displayed.

Appendix C: User's Manual

The Homepage

When opening our homepage, the use should see the following at the center of the browser:



ScriptSearch 🔎

There are two textboxes for user input:

- The first textbox will hold the URL that the user provides.
- The second textbox will hold the query the user wants to search for.

Providing a URL

Supported types of URLs:

- Video Links
- Playlist Links
- Channel Links

When the user provides one of these links, the application will load the transcripts from these videos into its internal database.

Restrictions

If the user provides an unsupported URL, the application will display an error message. This ensures that only searchable videos are present within the database.

Valid Video URL Formats

The following is a list of video URL formats that will be accepted.

```
https://www.youtube.com/watch?v=jNQXAC9IVRw&feature=featured
https://www.youtube.com/watch?v=jNQXAC9IVRw
http://www.youtube.com/watch?v=jNQXAC9IVRw
//www.youtube.com/watch?v=jNQXAC9IVRw
www.youtube.com/watch?v=jNQXAC9IVRw
https://youtube.com/watch?v=jNQXAC9IVRw
http://youtube.com/watch?v=jNQXAC9IVRw
//youtube.com/watch?v=jNQXAC9IVRw
youtube.com/watch?v=jNQXAC9IVRw
https://m.youtube.com/watch?v=jNQXAC9IVRw
http://m.youtube.com/watch?v=jNQXAC9IVRw
//m.youtube.com/watch?v=jNQXAC9IVRw
m.youtube.com/watch?v=jNQXAC9IVRw
https://www.youtube.com/v/jNQXAC9IVRw?fs=1&hl=en_US
http://www.youtube.com/v/jNQXAC9IVRw?fs=1&hl=en_US
//www.youtube.com/v/jNQXAC9IVRw?fs=1&hl=en_US
www.youtube.com/v/jNQXAC9IVRw?fs=1&hl=en_US
youtube.com/v/jNQXAC9IVRw?fs=1&hl=en_US
```

ScriptSearch 🔎

```
https://www.youtube.com/embed/jNQXAC9IVRw?autoplay=1
https://www.youtube.com/embed/jNQXAC9IVRw
http://www.youtube.com/embed/jNQXAC9IVRw
//www.youtube.com/embed/jNQXAC9IVRw
www.youtube.com/embed/jNQXAC9IVRw
https://youtube.com/embed/jNQXAC9IVRw
http://youtube.com/embed/jNQXAC9IVRw
//youtube.com/embed/jNQXAC9IVRw
youtube.com/embed/jNQXAC9IVRw
https://youtu.be/jNQXAC9IVRw?t=1
https://youtu.be/jNQXAC9IVRw
http://youtu.be/jNQXAC9IVRw
//youtu.be/jNQXAC9IVRw
youtu.be/jNQXAC9IVRw
```

Valid Playlist URL Formats

```
https://www.youtube.com/playlist?list=PLBRObSmbZluRiGDWMKtOTJiLy3q0zIfd7
```

Valid Channel URL Formats

```
https://www.youtube.com/@jawed
https://www.youtube.com/@jawed/videos
https://www.youtube.com/@jawed/featured
https://www.youtube.com/c/jawed
https://www.youtube.com/channel/UC4QobU6STFB0P71PMvOGN5A
```

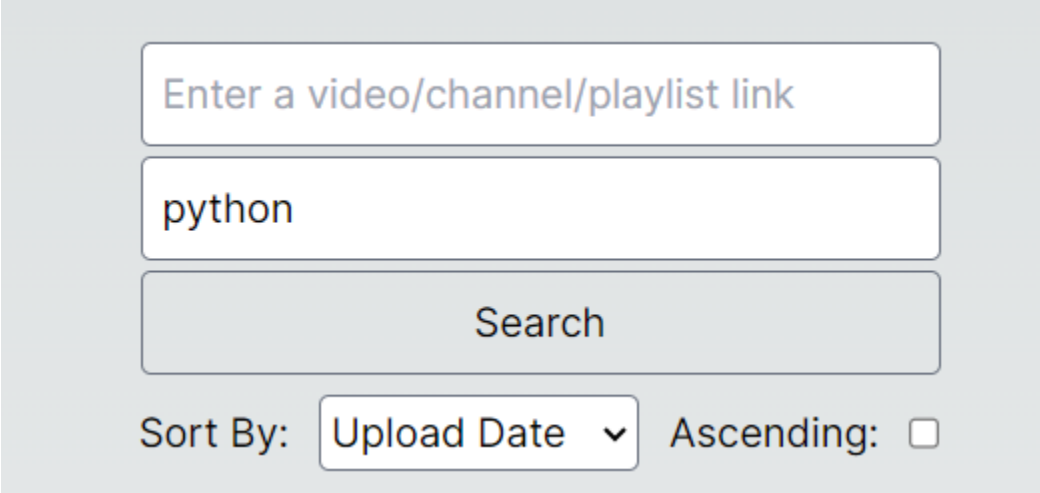
Performing a Query

In order to perform a query, the user must type text into one or both of the provided text boxes. There are two specific searches that the user can perform:

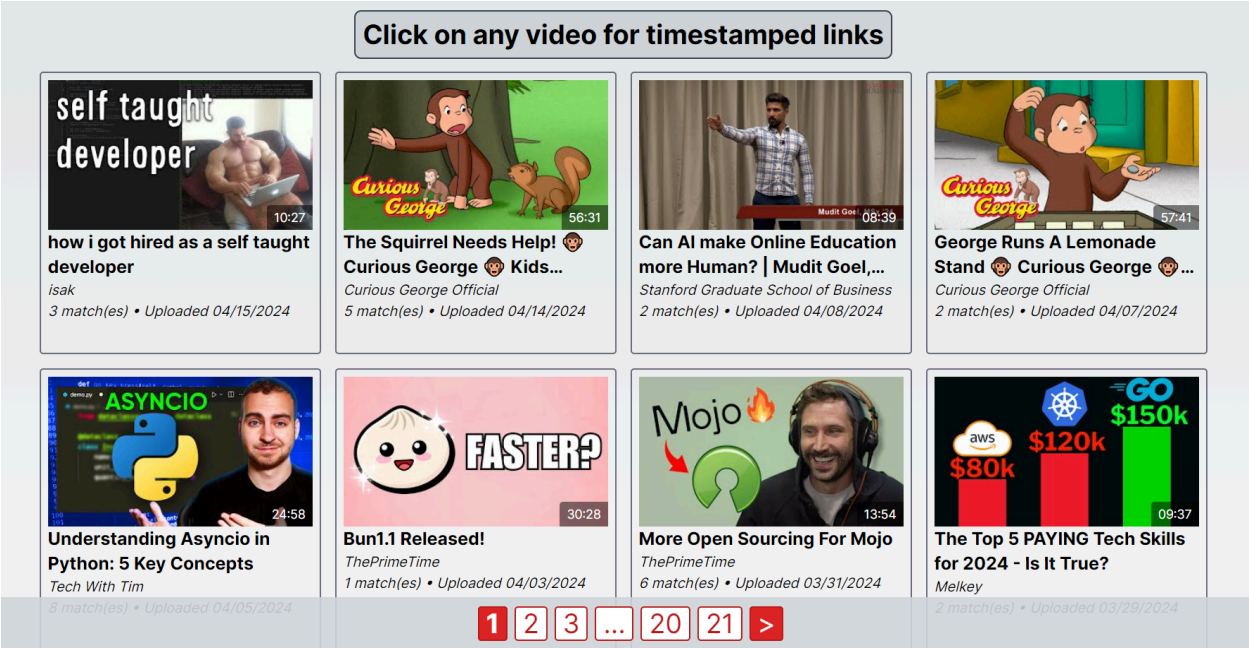
1. If the user enters only a query, our entire database of videos will be searched for that query.

Query:

ScriptSearch 🔎



Results:

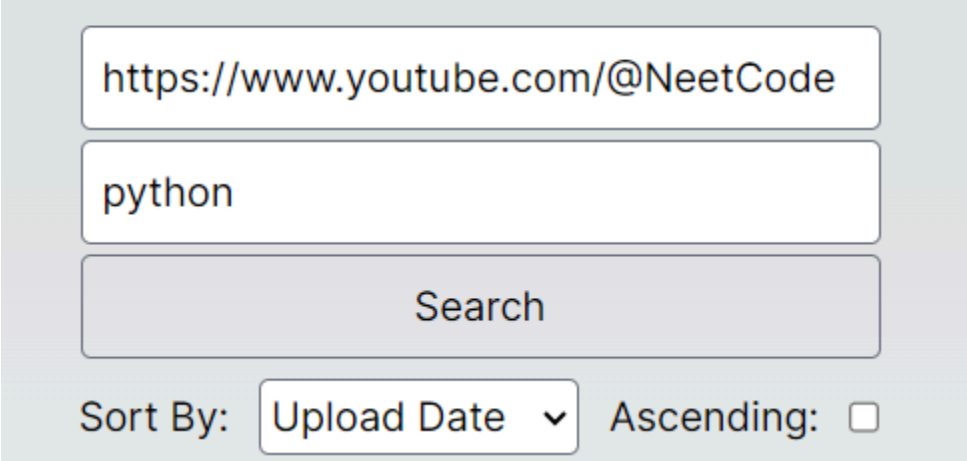


---

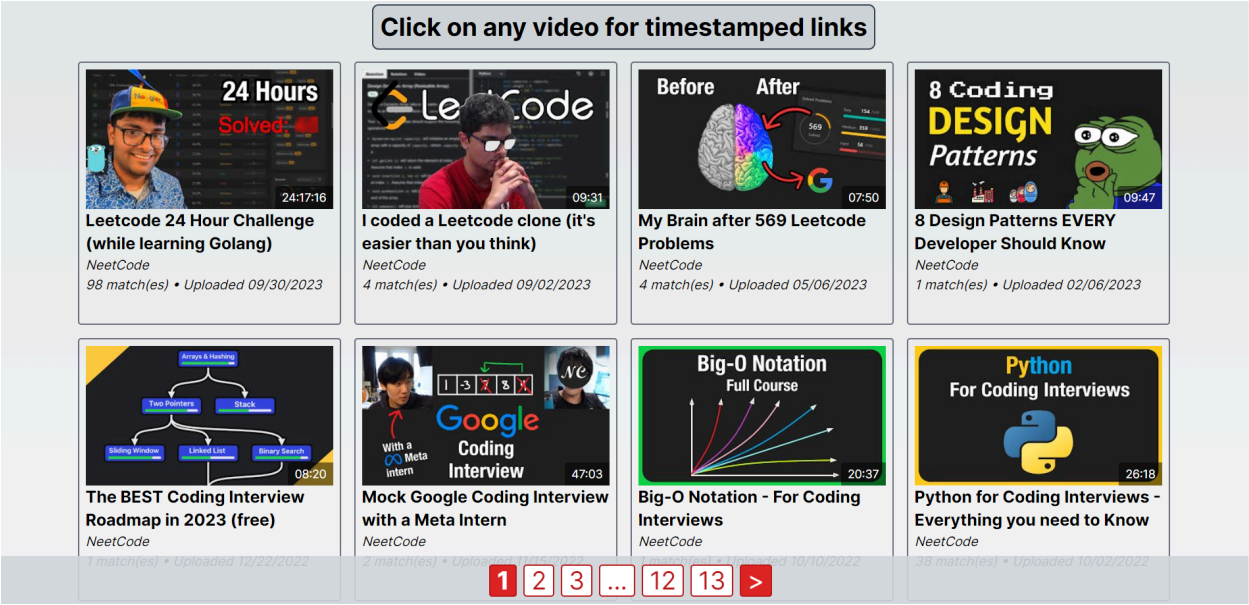
2. If the user enters both a link and a query, the videos from the channel/playlist specified will be added to our database and searched for the user's query.

Query:

ScriptSearch 🔎

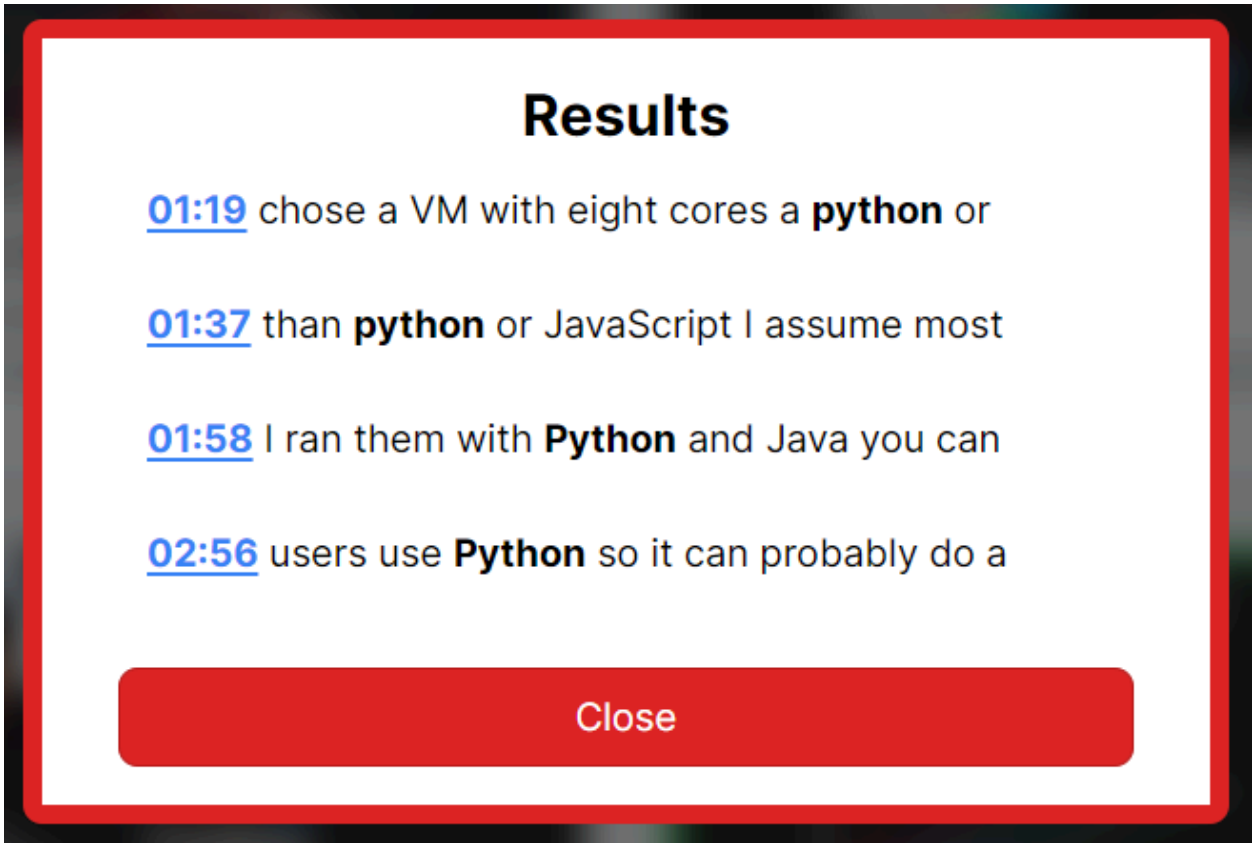


Results:



As shown above, each of the cards displayed contains the thumbnail of the video, its title and its uploader.

Once the user clicks on the card the following will be displayed:



This view shows a list of occurrences within the video where the search query appears. Each list item represents one occurrence of the query in the video. It consists of a timestamped YouTube link and a snippet from the transcript at that timestamp, with the queried word bolded. Clicking on a timestamped link will redirect the user directly to the video at that timestamp.

To close out of this menu, the user can press the "Close" button, click outside of the popup, or press the "Escape" key.

Restrictions

- Our search ignores cases, and queries are matched exactly with portions of the transcript. (i.e. entering the query **the quick brown fox** will search for **the** followed by **quick** followed by **brown** followed by **fox**.) This ensures that phrase order is preserved.
- When providing a channel or playlist link, it must be a direct YouTube link (i.e. 'youtube.com' or 'youtu.be', NOT tinyurls or shortened links).

ScriptSearch 🔎

- We only search the 250 most recent videos in the link provided, and our search will return at most 250 results.
- Only English language transcripts are searched by our application.
- Queries must be 5 words or less and shorter than 75 characters as too many words or characters would cause the search algorithm to slow dramatically.
- Searching common words such as 'the' or 'a' is not allowed.
- Special characters in a query will be ignored except for apostrophes (') and the plus symbol (+) will be converted to the word 'plus' if provided in a query.