CSE 546 — MONTAGE WIZARD

Group 22: Sainath Latkar, Ryan Kittle, Sharanya Banerjee

INTRODUCTION

Twitch is one of the leading live-streaming platforms for E-Sports today. An individual streamer creates around 10 hours of video game footage daily. In a week around 70-80 hours of gameplay is streamed and uploaded on Twitch. Since the average attention span of a viewer is not more than a couple minutes, the streamer needs to spend hours on end editing and clipping his videos. These clips are called ***Montages*** *and include game highlights like high intensity battles, clutch plays and ace plays. Almost every popular video-game steamer has a team of content creators who spend their time cropping the 10-hour long videos and extracting shorter clips. Then they edit and apply transitions to it. This is a tedious task and thousands of dollars are spent behind the scenes. Using Machine Learning, this process can be automated, thereby saving the time spent by content creators on watching the entire video to find quality content. This is a novel idea whose solution does not exist today, and we believe this will greatly benefit a large community of Streamers.*

BACKGROUND

In this 21st century, the E-Sports and Content Creation industry is booming. Gen-Z and Gen-Alpha are migrating by the day, to Social Media and are inclined to share their lives and activities with other people, on the Internet. Also with the advent of high performance systems, their affinity to the virtual world is increasing exponentially. As a consequence, E-Sports today is becoming a career for many people and those who excel and show tremendous skill in the field are rewarded very handsomely. Video-game streaming today is a full-time job for many. People play many competitive games like Call of Duty Black Ops: Cold War, Fifa 21, Defense of the Ancients (DOTA 2), Apex Legends, and stream their gameplay on platforms like Youtube (Google), Twitch (Amazon), Mixer (Microsoft). Twitch, a subsidiary of Amazon, is a video live-streaming service (Streaming as a Service) that focuses on video-game streaming and streaming of E-Sport events.[1] Viewers flock to those streams which are not only entertaining, but also engaging and fast paced. Some notable video-game Streamers of today are , TSM_ImperialHal[2], TSM_Beaulo, BDS_Shaiiko, GodlyNoob, SSG_ThinkingNade and many many more. These Streamers have grossed tens of millions of views on Youtube and Twitch and some have gone pro with many E-Sport teams like Team SoloMid(TSM), Rogue, DarkZero, G2 Esports, SpaceStation Gaming, etc.

Game Streamers stream gameplay from their game(s) of choice for 10-12 hours everyday. The revenue on their videos comes from running ads, sponsorships, and on a per-view basis. Twitch Streamers stream hours of content on Twitch and then take highlights from these videos, edit them and stitch them into a fast-paced, action packed, short video (in the order of minutes). These edited clips are called **Montages.** Montages are then re-uploaded on Twitch and/or Youtube for a greater spectrum of audience. However, culling such hour-long videos into Montages is a very tedious and resource heavy job. For this purpose, many established Streamers hire Content-editing teams who go through hours and hours of content at end to find those juicy clips which would appeal the most to their viewers. Not only is this a costly process, but also requires a lot of man hours. On an average, it takes more than 48 hours to cull a 12 hour long stream and edit the montages into Highlight videos.

Our project, **Montage Wizard** helps to fix this issue by automating this entire process of having to sit through hours at end and cull videos. It takes hour long videos and culls them meticulously to find those tempting Montages. It takes intricate clips where:

1. when a player wins a lot of battles in quick succession
2. when a player shows extreme reflexes and clutch plays
3. when there is interesting chatter in the in-game voice chat or funny clips

The entire model has been deployed on Google Cloud and processes videos on the cloud to deliver Montage videos to the user efficiently. This process is completely lossless and the Montages are of the same resolution and clarity as the input stream. Using this model, Streamers can save money that would have otherwise been spent on editing; and content creators can save hours and days of their time.

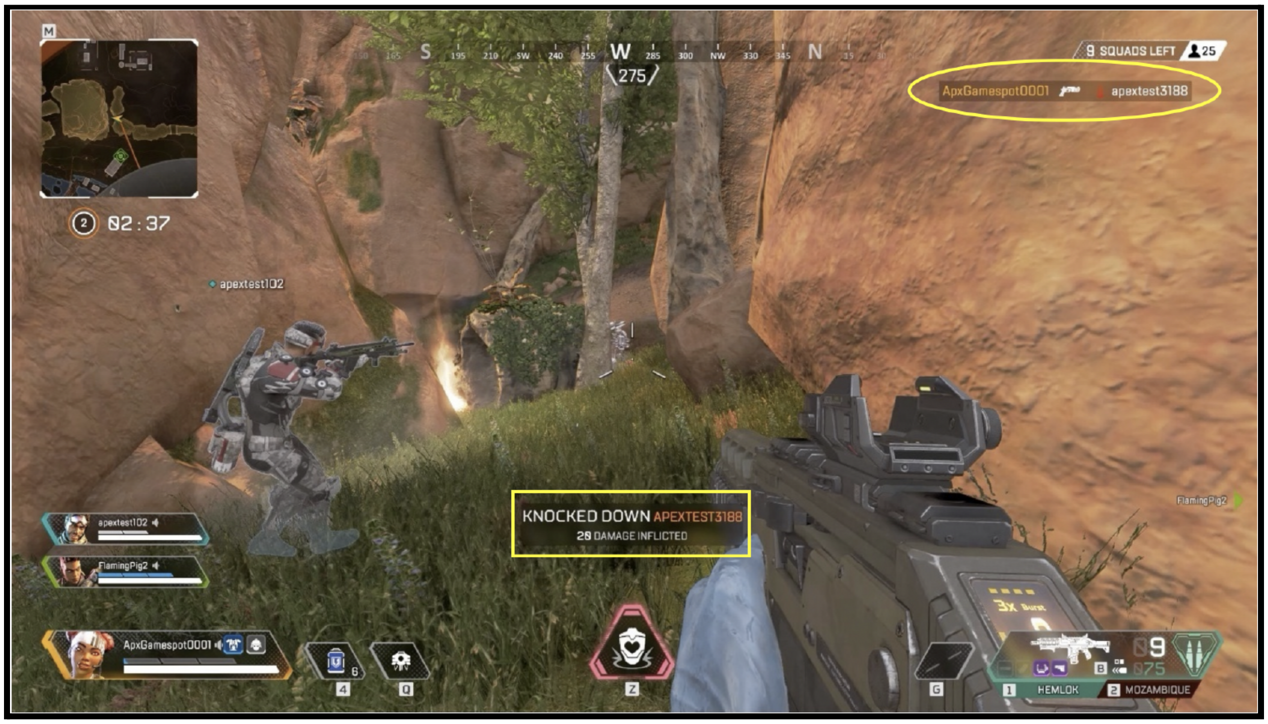

Fig. 0: Still frame from livestream

DESIGN AND IMPLEMENTATION

Architecture Diagram-

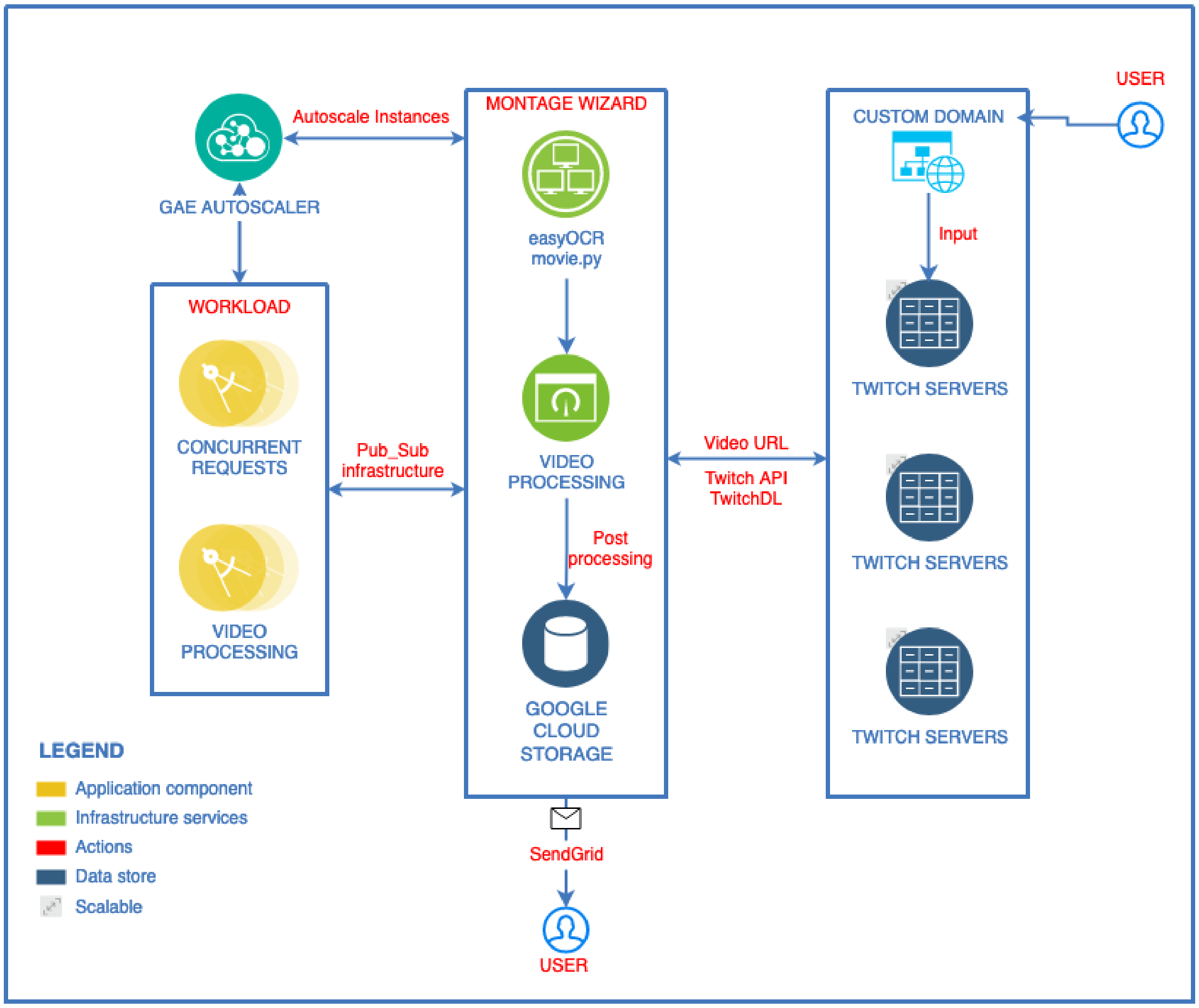

Fig. 1: Architecture Diagram

Module Description -

1. **Website (Custom Domain):** The website has a text input field where the user can input the name of the Streamer he is looking for. Upon pressing submit, all the videos of the Streamer that he has uploaded onto his Twitch account will be visible to the user. Upon clicking any one of the videos, processing will start and a dialogue box will appear on the screen stating the same.
2. **Twitch API:** Used to search for Streamers who stream on Twitch and collect URLs of the videos for each Streamer that the user searches for. Once the URL is retrieved, it will be passed as a parameter to the next module for processing.[3]
3. **PUB sub Infrastructure:** This module generates a Publisher-Subscriber queue between the Website and the model. All messages between the user and the system are passed through this infrastructure.

4. **TwitchDL Package:** It is an open source package used to download the videos that the user has selected from the URL sources from Twitch API, for processing.[4]
5. **OpenCV module:** Once the video is downloaded, the OpenCV module disintegrates the video frame by frame and gets a localized square block image from each frame using PIL: Python Image Library.
6. **EasyOCR module:** EacyOCR (Optical Character Recognition) is an open source python library which scans the frames and identifies the text blocks (if any) within the frame. If it finds texts such as "KNOCKDOWN" or "KILLED" or "0,1,2….9" identified as damage dealt, it selects the frame for capture.[6]
7. **Movie.py module:** Once frames are captured, videos are clipped from the original input video based on timestamps of the frames captured, using the Movie.py module.[5]
8. **Google Cloud Storage bucket:** Once clips are generated, they are uploaded to the custom storage bucket and their public URLs are captured and stored in a buffer memory.
9. **Send Email module:** Upon completion, the public URLs are delivered to the user's inbox via email, along with a custom completion message. This is performed using SendGrid library.[7]

Cloud Services used -

1. **Google App Engine:** The entire implementation is hosted on Google App Engine. It consists of all the modules starting from the Custom Domain to the Send Email module. All processing is done on GAE and outputs are delivered to the user. To handle excess load or more number of concurrent requests, GAE autoscales itself.
2. **Autoscaler:** The model deployed on GAE autoscales itself consistently. It upscales itself for more number of concurrent requests and terminates idle instances when there is less load. This feature makes Montage Wizard very efficient.
3. **PUB sub Infrastructure:** This module generates a Publisher-Subscriber queue between the Website and the model. All messages between the user and the system are passed through this infrastructure.
4. **Google Cloud Storage bucket:** Once clips are generated, they are uploaded to the custom storage bucket and their public URLs are captured and stored in a buffer memory.

Autoscaling in Google App Engine - We have tested the auto scaling feature extensively. We bombarded the custom domain and the Pub_Sub module with 80000 requests from a Workload Generator. Upon receiving an increasing number of incoming requests, GAE upscaled by spawning more instances. Once the requests were satisfied, it began terminating idle instances. Along with that, we have tried processing very heavy videos (HD, of the order of 6-8 hours long → 20GB+ size) and even then Montage Wizard was able to process them successfully, even over a longer period of time. Therefore, we can say that the model autoscaled and handled load with consistency. The only bottleneck we saw was that GAE takes more time to terminate idle instances than in spawning them, but this we believe is a limitation of GAE itself. Also processing large videos takes a considerable amount of time due to network limitations.

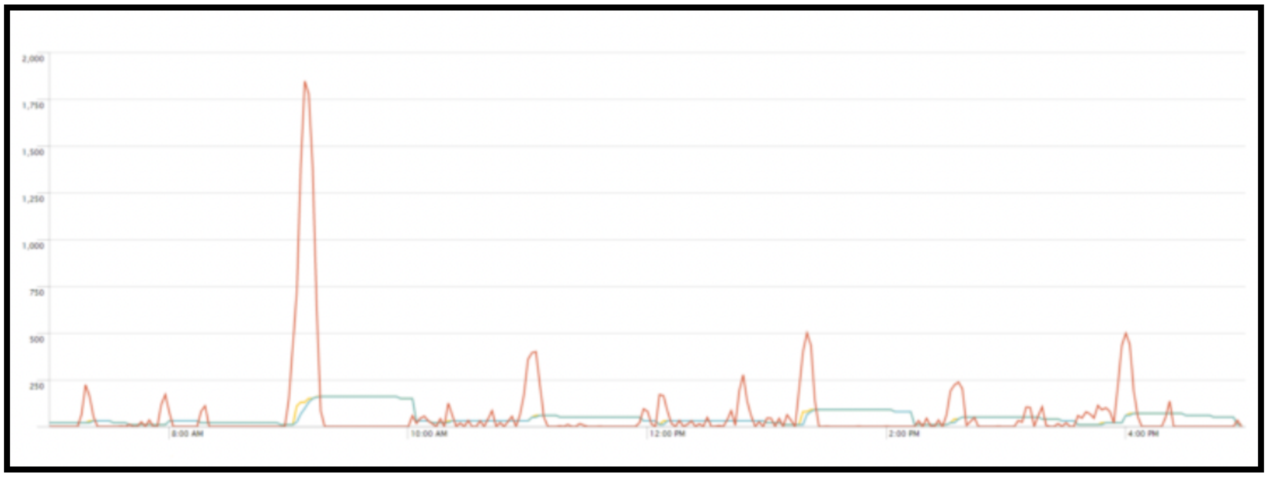

Fig. 2: Snippet of autoscaled instances

Using Montage Wizard, the resources spent on footage generation from hour-long streams will reduce drastically. Such a service does not exist currently and streamers currently have to cull images manually and stitch them by hand before post processing and applying transitions. Streamers can push their videos as soon as they get off of their stream and in a few hours, they shall receive an email in their inbox as soon as the processing is done; and they would be good to go to applying transitions to their montages. This will allow content creators to devote more time and resources in content creation, rather than culling. Therefore, Montage Wizard is significantly better than any state-of-the art solution currently existing in this sphere.

Cost Analysis - Here is an estimation of how much we believe Streamers would save on a monthly basis using Montage Wizard, in contrast to having an Editing Team manually cull videos:

Average hourly pay of a video editor in the United states: $27.50

Total Cost to cull a 10 hour video to generate clips: around $300

Total Cost borne by a streamer to edit 25 videos a month: $7,500

Cost of our subscription model: **$100 per video** (Pay per use model)

Average cost per subscriber per month @ 25 videos a month: $2,500

Gross income for a streamer from 25 montage videos a month: $17,500 (for 1mil. Views per video)

Net income if streamers manually cull their videos: **$10,000**

Net income if streamers use Montage Wizard: **$15,000**

Net profit percentage: **50% increase in income**
(All data has been taken from google and estimated)

TESTING AND EVALUATION

We stress-tested Montage Wizard in multivarious ways. Based on the modules that have been listed in the Design segment above, we ran the following tests:

Table 1: Testbench description

MODULE NAME	TESTS RUN	TEST OUTCOME
Client_Website	Used workload generator to test request-response routes Bombarded 80000 requests to test autoscaling	PASS The model autoscaled (spawned new VMs) the App-tier to handle incoming requests seamlessly
Twitch_API	Used Postman to test video parsing for incoming requests	PASS Videos parsed successfully
Video_Downloader	Passed concurrent requests to download videos	PASS Videos downloaded successfully
Machine Learning model	Processing videos concurrently in real-time	PASS Videos processed successfully
Montage_Uploader	Uploaded dummy videos with no highlights	PASS Dummy videos return no clips
Send_Email_SendGrid	Sent a large number of emails from server to client, post completion of processing	PASS All emails sent/received successfully

Montage Wizard fared really well under all the conditions mentioned above and passed all Stress tests. We bombarded the App Engine with 80000 requests and it autoscaled (upscaled and downscaled) within minutes and with consistency. We tried to parse very big videos (>100GB size) and even though it took a considerable amount of time to process them, it returned good clips from those videos. Along with that, we passed some dummy videos with no prospective clips to see how the system would react. In that situation, the model did not give any wasteful returns. An email was sent to the user's inbox as soon as processing was completed and all output clips were outputted every time to the assigned output bucket.

CODE DESCRIPTION

Our front-end UI is generated using two files, main.py and index.html, which communicate through the use of Flask. Index.html displays a field for the user to input a streamer's name. Upon entering a name and clicking the submit button, the html file sends the streamer's name as a string to the python method *LoadVodList*. This function calculates the streamer's *USER_ID* and calls the Twitch

API, which returns a JSON representing the 10 most recent stream videos and 10 most recent clips. These videos and clips are then sorted by date, and sent back to the html file to be displayed for the user. Once the videos are displayed, a user can click a video's thumbnail to begin processing.

Once a user clicks on the thumbnail of the video, a message with a link to the twitch stream is sent on the Google Pub-Sub messaging queue. This message is received in the VMInstanceHandler.py file. This file calls download.sh file with the link as input parameter. Download.sh file passes the link to twitchdl module and twitchdl module downloads a 720p stream at 30FPS using multiple concurrent threads. Once the stream is downloaded locally, download.sh file calls main.py file which implements the main machine learning model that clips the video. Once the main.py file comes into execution, it looks at the download directory and starts processing all stream files with .mp4 or .mkv extension. For processing each video, main.py file first reads the video using the opencv library and starts processing each frame. For each individual frame using the Python Image Library also known as PIL, we crop the section of the frame using pixel indices where we expect to see messages like "Knocked down", "Eliminated", "You are the new Kill Leader". This cropped image is then provided to easy ocr module, easyocr library looks at these cropped frames and determines whether the image contains desired text. Once it encounters a frame with the desired message, say "Knocked down", it notes down the time of the stream. If there are no consecutive knockdowns further in the threshold time which is 30 seconds, it generates a clip from the input stream which starts at 20 seconds before knockdown and 15 seconds after knockdown. These clips are saved locally using moviepy library. Once all the clips for a stream are generated, we upload those clips to google cloud storage and send an email to the streamer with the links to download those videos using SendGrid API.

CONCLUSION

Using Montage Wizard, the resources spent on footage generation from hour long streams will reduce drastically. Streamers can push their videos as soon as they get off of their stream and in a few hours, they shall receive an email in their inbox as soon as the processing is done; and they would be good to go to applying transitions to their montages. This will allow content creators to devote more time and resources in content creation, rather than culling.

Future Work-

We believe that Montage Wizard can take the content creation world by storm. This model can be scaled, published on a subscription basis or as a pay-per-use basis, or be put in a licensing deal for creators or game developers. When scaled for all competitive games, this model can cull videos in real-time. It can be used to generate montages in real-time from livestreams, as highlights. This would enable any viewer to see any highlights they might have missed. Also, this model may be used to automatically keep score in E-Sport competitions in real-time; something that is currently done manually by an administrator. These features, if and when deployed, would change the face of Esports, for ages to come.

INDIVIDUAL CONTRIBUTION

Sainath Sharad Latkar: My name is Sainath Latkar (ASU ID: 1219483096). Below is listed my individual contribution to the project, with respect to Design, Implementation and Testing of the system.

Individual Contribution :: Design - With respect to the design of Montage Wizard, I contributed to designing the Machine Learning model. I built the PyTorch scripts containing EasyOCR, moviepy and TwitchDL. As soon as a user clicks on a video, the TwitchAPI will fire and help to access the URL of the selected video. Once the URL of the target video has been sourced, it will send the URL to the downloader script. Using TwitchDL, the downloader will download the video from the source URL and begin processing. It will strip the video down to its frames and localize a portion of each frame. Then the EasyOCR library will kick in, trying to find character prompts within the localized portion. It will look at each frame, one at a time. If it finds character prompts, it captures the frames based on their timestamps. Then the moviepy library helps to cull the video into clips based on the timestamps captured by the EasyOCR module. Once the entire video has been scanned and clips culled, the generated clips are sent to the uploader, to be uploaded to storage buckets.

Individual Contribution :: Implementation - Once I run the system, it will wait for user input. Once the user selects a video of his/her choice, the URL of the selected video will be sourced from the TwitchAPI and pushed to the downloader containing TwitchDL library. The video will download and based on the number of concurrent requests incoming from the users, and based on the size of the videos, the system will autoscale. Once the video has been downloaded, the OpenCV module will strip the video using Python Image Library (PIL) down to its frames and localize a portion of each frame. Then the EasyOCR library will kick in, trying to find character prompts like "KNOCKDOWN" or "KILLED" or "0,1,2….9" identified as damage dealt, within the localized portion. It will look at each frame, one by one. If it finds character prompts, it captures the frames based on their timestamps. Then the moviepy library helps to cull the video into short clips (~30-40 seconds) based on the timestamps captured by the EasyOCR module. These clips would be called "Montages". Once the entire video has been scanned and clips culled, the generated clips are sent to the uploader, to be uploaded to storage buckets.

Individual Contribution :: Testing - I carried out testing of the entire Machine Learning model. I gave dummy tests to each module both while Training the models as well as Testing them. The model has been trained on a large variety of full-length videos as well as shorter clips. Post testing, we are confident that the model is robust and consistent in its processing. I ran the following tests on the model:

Twitch_API	Used Postman to test video parsing for incoming requests	PASS Videos parsed successfully
Video_Downloader	Passed concurrent requests to download videos	PASS Videos downloaded successfully
Machine Learning model	Processing videos concurrently in real-time	PASS Videos processed successfully

Table 2: Tests run by Sainath Latkar

Ryan Kittle: My name is Ryan Kittle (ASU ID: 1212804526). Below is listed my individual contribution to the project, with respect to Design, Implementation and Testing of the system.

With regards to the design, I primarily worked towards creating a satisfying experience for the user. In the early stages for our project, we were originally planning on taking user input and displaying the output on the same webpage. This would be similar to the design of the first project, where the output was displayed as a list for the user. However, I realized it would not be realistic to apply this design to this project; our ML model could take hours to process a 10-hour video. It wouldn't make sense to require that the user stays on the same webpage while the entire stream video is processing. Instead, our design would take a user's email, and send them the completed montage clips. This would allow the user to simply select a video to process and go about their day as usual, without the fear of their browser crashing and having to restart the process.

Moving on to the implementation, I spent the majority of my time creating the front-end UI, sending an email to the user with the processed clips, and creating a workload generator. The front-end was created with two files which communicate through Flask. This was pretty straightforward, and I basically followed the structure of the front-end in Project 1. The website was fairly simple in its design; it would only need to take a streamer's name and display their most recent videos. From there, a user can select a video to be sent to the back-end ML model for processing. This was all made possible with the Twitch developer API, which proved simple to implement and reliable upon receiving many requests. Once the front-end was up and running, I began working on the code to send the user their output upon the completion of the video processing. Again, this was made possible through the use of an API, this time utilizing SendGrid. It would allow us to automatically send an email with a link to the generated clips, which the user could click to view and/or download their clips for the montage. Once the user was able to view their output through the link in the email, I began work on creating a workload generator to test our design and implementation. Creating the workload generator was simple enough, utilizing a multitude of threads to send requests to our app on the App Engine. There were a couple programs available for download online that would be able to generate a workload automatically, but creating the code myself seemed simpler than creating an environment to download the program from online. In addition, it would give me a greater understanding of how it functions, making changes easier to implement.

Once my workload generator was complete, this would allow me to test multiple aspects of our design with a singular command. More requests would be generated the longer the program was run, but when looking through the gcloud console, it appears as though we maxed out at around 100 requests/second. I was satisfied with this, and it appeared as though our application was scaling by approximately the same amount every time I would run the workload generator. In addition to the auto scaling feature, the workload generator would also test the APIs we used. The Twitch API had a maximum amount of requests/minute, so with the workload generator we were able to see how quickly we would reach this maximum. This also led us to create fail-safes for when we maxed out our requests, ensuring it did not cause our program to crash. The workload generator worked similarly for the SendGrid API, which had a maximum amount of emails we could send in a given time. Again, our testing led us to creating a backup plan for when too many emails are being sent at once.

Sharanya Banerjee: My name is Sharanya Banerjee (ASU ID: 1219403952). Below is listed my individual contribution to the project, with respect to Design, Implementation and Testing of the system.

Contributions in Design - With respect to the Design aspect of our project, I contributed to designing the Storage architecture of the model. Once the videos would be processed, they would be uploaded onto a storage bucket made in Google Cloud Storage. Once clips are generated, they are uploaded to the custom storage bucket using the uploader script and their public URLs are captured and stored in a buffer memory. The URLs are then sent to the user's inbox via email. This would allow the user to simply select a video to process and go about their day as usual, without the fear of their browser crashing and having to restart the process. Along with the storage, I also worked on the Publisher Subscriber infrastructure. As soon as the user selects a video, it's source URL is accessed and put on a Messaging Queue. This URL is then pushed to the downloader to begin processing. I tried to make this process as seamless and automatic as possible while mitigating the need for user interference.

Contributions to Implementation - I drafted the architecture diagram for our model. After having made the Design for the storage buckets, I created them on the Cloud Storage service. I integrated the Bucket IDs into the code. The buckets would be empty when there are no clips to be stored. Once clips are generated, they are uploaded to the custom storage bucket using the uploader script and their public URLs are captured and stored in a buffer memory. The URLs are then sent to the user's inbox via email. This would allow the user to simply select a video to process and go about their day as usual, without the fear of their browser crashing and having to restart the process. I integrated the bucket architecture with the SendGrid email service which Ryan had incorporated into the system, as an effort to automate the process. Once the user has downloaded his clips, the bucket would be auto-cleaned in order to prevent interference with future clips and make the system more robust and efficient. I also implemented the Pub-Sub service for our model. As soon as the user selects a video, it's source URL is accessed and put on a Messaging Queue. For this purpose, I made use of a FIFO queue infrastructure with a high throughput. This service would be robust to a large number of incoming requests and hence would adjust itself to autoscaling the system. The requested URL is then pushed to the downloader to begin processing. By automating this process, I eliminated the need for human computer interaction.

Contributions to Testing - While we stress tested the model, I tested its robust nature. I developed a testbench to upload a large number of files to the Storage buckets. The system fared pretty well and was able to upload files losslessly and with consistency. Once I was done with the buckets, I tested the accuracy of our Video processing model. I pushed some dummy videos which do not have any prospective clips within. The model performed as expected and did not return any dummy clips. We also pushed some videos concurrently to the model to test how it would adapt. Consistent to our architecture, the model processed all videos successfully and returned clips subsequently; even though having taken more time to do it owing to the sheer size of the videos being processed. Once all the videos were processed and clips were generated successfully, I also tested the downloader to see if it uploads all the clips to the bucket, or not. Montage Wizard passed all tests, in a blink.

REFERENCES

[1] https://en.wikipedia.org/wiki/Twitch_(service)

[2] https://www.twitch.tv/tsm_imperialhal

[3] https://dev.twitch.tv/docs/api/

[4] https://github.com/ihabunek/twitch-dl

[5] https://pypi.org/project/moviepy

[6] https://github.com/JaidedAI/EasyOCR

[7] https://sendgrid.com/