DigiClips Media Search Engine

Final Report

Team: sddec21-06

Client: DigiClips, Inc.

Adviser: Dr. Ashfaq Khokhar

Team Members: Tyler Johnson, Samuel Massey, Max Van de Wille, Maxwell Wilson

Email: sddec21-06@iastate.edu

Website: sddec21-06.sd.ece.iastate.edu

Executive Summary

Development Standards & Practices Used

In this project we will not be designing or implementing circuits or hardware, therefore all standards and practices will be software focused. Specifically, for this project we will be using practices such as object-oriented programming as well as the following IEEE standards for software development best practices:

- IEEE Std 1063, Standard for Software User Documentation
- IEEE Std 829 –2008, Standard for Software Test Documentation
- IEEE Std 830-1998, Recommended Practice for Software Requirements Specifications
- IEEE Std 1012, Standard for Software Verification and Validation

Summary of Requirements

- Speech-to-text for television and radio recordings
    - Process recording files to extract text from audio.
    - Store text in searchable keyword/phrase database table(s)
    - Store errors into separate database tables for logging purposes
- Video-to-text for television frames
    - Perform optical character recognition on text in television frames
    - Store text in searchable keyword/phrase database table(s)
    - Store errors into separate database tables for logging purposes

Applicable Courses from Iowa State University Curriculum

- Com S 228 – Data Structures and Algorithms
- Com S 309 – Software Development Practices
- S E 339 – Software Architecture
- Com S 363 – Introduction to Database Management Systems

- S E 417 – Software Testing
- Com S 575 – Computational Perception

New Skills/Knowledge acquired that was not taught in courses

In addition to the listed courses, this project required us to familiarize ourselves with topics such as audio manipulation, signal processing, optical character recognition, and containerization of microservices.

Table of Contents

List of figures/tables/symbols/definitions

Tables:

1 Introduction

1.1 Acknowledgement

Our team would like to express our sincere gratitude to Bob Shapiro, Henry Bremers, and the team over at DigiClips for their continuous support of our project's research and implementation. Their constant encouragement, insightful comments, and hard questions have helped us create a better product that will aid them in providing a more rounded experience for their client base.

We would also like to thank our faculty advisor, Ashfaq Khokhar, for the constant support through the project and for the insightful recommendations. His expertise in data intensive multimedia applications and knowledge of signal and audio processing proved valuable on multiple occasions throughout the tenure of our project.

1.2 Problem and Project Statement

DigiClips, Inc. is a media content analysis company that records and extracts data from diverse types of media, such as television and radio, and stores this information in a searchable format. It aims to provide its clients a user interface that would facilitate searching of the database for keywords or phrases of user's interest uttered in audio or video clips. For example, a client may be interested in finding if their company name has been mentioned (along with its frequency) on television or radio within a given time frame.

General problem statement - The data currently being extracted from the television recordings is from the television network-provided closed captions only. This closed captions data often misses words or phrases spoken within the broadcast, causing a disconnect between the actual content of the broadcast and the searchable content provided. In addition to missed audio, closed caption data does not provide any means of searching the content of broadcasted frames themselves, where there is often visible text that denotes the current segment of news, breaking stories, etc. This information is currently lost within hours of recordings and extremely difficult to perform searches on.

Proposed Solution - This project will investigate existing solutions and develop efficient speech-to-text and video-to-text modules that will take television and radio recordings as its inputs and record the timestamp-location of keywords and phrases of interest in these recordings. The outputs of these modules will be organized for ease of use in a database system. The speech-to-text and video-to-text extraction capability will give the company an edge in the industry by granting them access to data that is currently not being tracked, providing more opportunities for clients to find any and all mentions of their desired keywords. The focus will be to develop near-real-time solutions that can scale with the number of audio and video recording streams. These modules will be integrated with other components in the system that include signal processing applications, databases, query and retrieval frameworks, and user-interfaces.

1.3 Operational Environment

The operational environment of our project is simply a computer based in an office environment. The specific computer that our application will be running on is a custom-built system running Ubuntu 18.04. In terms of processing power, this computer boasts a very powerful AMD Ryzen 9 5900X CPU alongside an NVIDIA GeForce 210 GPU for minor graphics-based computations. The computer running our code will not be exposed to extreme temperatures or conditions thus the project will not be considering preventative measures for these anomalies.

1.4 Requirements

- Functional Requirements
    - Speech-to-text system
        - Must be able to convert audio streams of spoken words into plain text.
        - Must accept mono and stereo audio recordings as input, processing all streams into their own results feed.
    - Video-to-text
        - System must detect multiple fonts/styles of text in video frames.
        - System must process text located within the entirety of the recording's frames.
    - Output formatting
        - All system results must be processed to check for correct grammar and spelling.
        - Results should be indexed via their corresponding timestamp in the video/recording.
        - System errors should be traceable and identifiable for maintainability.
        - System errors must be caught and returned to the original requesting service
- Non-Functional Requirements
    - System shall be built without utilizing any costly APIs/cloud resources.
    - System shall be built with documentation to explain usage and integration.
    - System should scale with the assumed amount of data present.
    - System should reliably output results within a reasonable timeframe.

1.5 Engineering Constraints

- Cannot utilize paid APIs for speech-to-text or optical character recognition
- Developed program must be able to run on an underpowered computer
- System must reliably output within the timespan of the input audio/video

1.6 Intended Users and Uses

The intended users of this project are Bob Shapiro, Henry Bremers, and the DigiClips Media Search Engine. Bob Shapiro is the chairman of DigiClips Media Incorporated. Henry Bremers is the Senior Software Engineer in charge of managing DigiClips software. The DigiClips Media Search Engine is a front-end application that will be operated by DigiClips customers and clients and will be utilizing data produced by the project system.

1.7 Assumptions and Limitations

Assumptions:

- The program will only be processing up to 10 broadcast television channels.
- The program will be operating on new recordings as they are recorded rather than previously recorded broadcasts.

- The inputted television recordings will be of high resolution and high enough audio quality for accurate processing and output.
- The resulting program output will match the same database schema as the currently stored closed captions.

Limitations:

- Due to budget constraints, we are not able to utilize certain paid APIs for speech-to-text or optical character recognition.
- Our developed system must be computationally efficient and able to run on a relatively underpowered computer.
- The program must be able to operate quickly enough for customers to query data within 24 hours of recording.

1.8 Expected End Product and Deliverables

The expected end product is a pipeline through which television and radio recordings will be processed to extract searchable data in the form of keywords and phrases spoken or visible through text in the video frames. As part of this end product, this pipeline must include a speech-to-text system and what we refer to as a video-to-text system which extracts words, letters, and numbers from the image frames of the recording.

The speech-to-text system must process audio identified as human speech into plain text phrases and keywords that, along with the timestamp of the spoken phrase, will be made searchable as part of a database. The system must be able to work with audio that comes from television and/or radio recordings performed by the current DigiClips recording backend.

Similarly, the video-to-text system will identify and process any words, standalone letters, and numbers that may be present within a given frame of the recording into searchable plain text words/phrases associated with the timestamp they were displayed in the recording to a searchable database to be used by the DigiClips front-end search engine service. The provided input to this system will be video recordings of television locally stored or frames of television passed through the pipeline as they are being captured/recorded.

These deliverables that make up the end product will be developed, thoroughly tested, and ready for integration into the existing DigiClips television recording backend by the project end date in December 2021.

2 Project Plan

2.1 Task Decomposition

- **Build speech-to-text system.**
    - Planning
        - Determine what speech-to-text functionality is already present within the DigiClips codebase.

        - Research possible techniques for implementing speech-to-text using existing software.
    - Design
        - Outline possible speech-to-text system structure using planning and research.
        - Collectively decide which outline is the most effective at meeting the problem requirements.
    - Development
        - Using the selected design develop the speech-to-text system as outlined in the Design phase.
    - Testing
        - Test the implementation to ensure the given requirements have been met.
        - Have our intended users test and report feedback on the project implementation.
- **Build video-to-text system.**
    - Planning
        - Determine what video-to-text functionality currently exists in DigiClips software.
        - Research video-to-text processing to find pre-processing techniques and libraries ideal for the project
    - Design
        - Build rough outlines of possible system structures.
        - Decide on which structure solves the problem most effectively.
    - Development
        - Using the chosen design, build the implementation according to the proposed structure.
    - Testing
        - Rigorous testing of the implementation to ensure requirements have been met.
        - Receive feedback from our intended users on the state of the implementation and make sure their requirements are met.
- **Integrate speech-to-text and video-to-text.**
    - Take completed speech-to-text and video-to-text systems and integrate them together to ensure ease of use for the user.

2.2 Risks And Risk Management/Mitigation

- **Speech-to-text task**
    - Speech-to-text processing will not be accurate enough to provide substantial value to the DigiClips business.
        - Probability: 0.2
        - Risk Mitigation Plan:
            - To mitigate this risk, we put extra care into researching our speech-to-text system. We have seen good accuracy results using Mozilla's DeepSpeech engine which we utilized for our speech-to-text processing.

- **Video-to-text task**
  - Video-to-text system will be too processor intensive to be a realistic solution to DigiClips' problem.
    - Probability: 0.5
    - Risk mitigation plan:
      - Extra time will be spent in the planning and design phases of this task to ensure that we consider the limitations of processing large amounts of video data. In particular, our designed system only processes a certain number of frames from each input video, i.e. processes a frame once every 0.5 seconds since the likelihood of text being visible for mere fractions of a second is unlikely.
  - System misidentifies words making it too inaccurate to be useful.
    - Probability: 0.5
    - Risk mitigation plan:
      - To mitigate this risk, we set up multiple output filtering layers that help to ensure that the data captured by the video-to-text service is useful data. These steps are detailed further in Section 5: Implementation.
- **Integration task**
  - The primary risk is that the speech-to-text and video-to-text systems won't integrate easily when being developed separately.
    - Probability: 0.5
    - Risk mitigation plan:
      - During the design phase we made the decision to approach containerizing our different services, which lets their integration go much smoother. This essentially allows each service to work on its own, and the only integration needed between the two processing services is a driver microservice to connect them, which can be performed with HTTP requests to also alleviate integration issues.

2.3 Project Proposed Milestones, Metrics, and Evaluation Criteria

Milestones:

- Complete the speech-to-text system.
- Complete the video-to-text system.
- Integrate speech-to-text and video-to-text on one complete program.
- Document and test software to prepare for integration on DigiClips' systems.

Evaluation criteria:

- Achieve 80% accuracy on speech recognition and 95% coverage with microservice unit testing.
- Achieve 70% accuracy on video text recognition and 95% coverage with microservice unit testing.
- Process the speech-to-text for a video file within 75% of the file's length.
- Process the video-to-text for a video file within the length file.

2.4 Project Timeline/Schedule

Proposed Project Schedule:

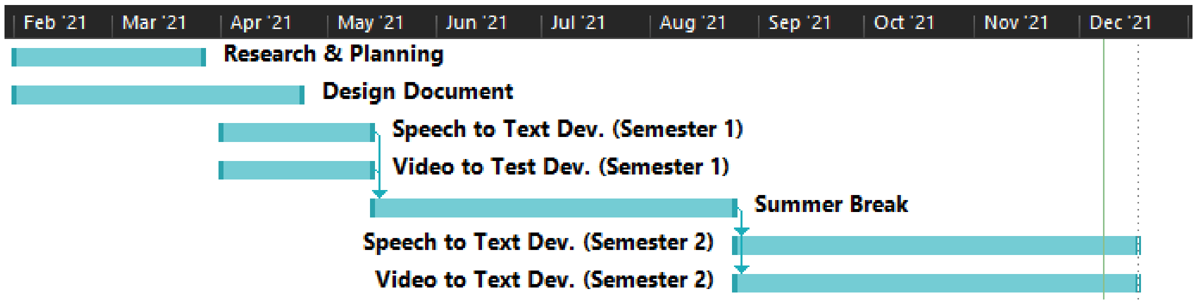


Figure 1: Project Plan

2.5 Project Tracking Procedures

We plan to use multiple methods of tracking our project progress. Our main source of tracking progress will be GitHub as that is where the current codebase resides, and so when pushing to our repository we can see what was added, removed, or modified. We also plan to use communication channels like group messaging to ensure everyone is up to date with the current state of the project. Finally, we plan to have constant communication with our group as a whole and taking notes to have a better understanding of what should be expected.

2.6 Personnel Effort Requirements

Table 1: Estimated Time

Task	Estimated person-hours required
Speech-to-text system	70 hours
Video-to-text system	100 hours
System integration	30 hours

2.7 Other Resource Requirements

We will not require any physical resources to complete this project. In terms of digital resources, we will need access to the existing code that DigiClips has pertaining to speech-to-text and video-to-text. We will also need access to the DigiClips systems and backend code. This will ensure that we are able to integrate our solution with the rest of the DigiClips Media Search Engine.

2.8 Financial Requirements

The only financial requirement we have for this project is that our application should not use paid libraries or APIs. Our project will make use of free, open-source software.

3 Design

3.1 Previous Work And Literature

The idea of speech-to-text and video-to-text are not novel ideas, and many libraries and interfaces exist to perform these actions on multimedia recordings. Additionally, other groups have created similar applications for recording and pulling text specifically from television and radio stations.

Detailed in his Forbes article, Kalev Leetaru describes the process of using Google's speech-to-text API alongside natural language processing on television recordings to thematically analyze the segments of television. In this project, Leetaru mentions using the Google Cloud Speech-to-Text transcription to automatically generate a transcript for the video (Leetaru, 2019). In our research on the available APIs for speech-to-text we discovered that Google's implementation is considered as one of the top-of-the-line speech-to-text solutions currently available, especially when you consider its built-in grammar and spell checking alongside many other pre- and post-processing features that make the output into clear, accurate sentences and phrases. Unfortunately for our project, Google's speech-to-text solution requires payment, leaving us to try and implement a lot of its functionality on our own. Our project aims to differentiate from Leetaru and Google's work primarily through the application of the resulting data. Rather than post-processing the data through natural language processing, we will be focused on formatting the resulting data to make indexed, query-able phrases stored in a database table.

Similar to Leetaru's implementation of speech-to-text for television, other groups have considered and possibly developed solutions for extracting speech data from multimedia sources. As seen in their granted U.S. Patent, Daniel Barcy and Charles Statkus propose an implementation for extracting human-readable captioning for television and radio streams (United States of America Patent No. US6542200B1, 2001). In their design, an application runs using a live feed of television or radio audio and processes the audio input for gain control, audio filters, and finally into a speech-to-text converter that performs the data extraction. Their design also makes considerations for language translation processing, which they would perform on the direct output of the speech-to-text converter. A key difference between our proposed implementation and the one shown in the patent is the source of the audio input. In their design, the input is processed as a live feed, whereas in our implementation, we will be processing audio files after they have been recorded into a standard video format. This will integrate more smoothly with DigiClips existing television and radio recording systems and require less interference with their current applications.

Regarding video-to-text, one notable previous implementation comes from a team of researchers at University of Novi Sad, Serbia, discuss their implementation of using optical character recognition (OCR) on frames of television recordings. Their implemented system receives a frame as part of a live television feed, performs some image pre-processing including locating potential text, then processes those possible regions using an OCR solution. The output from the OCR solution is then used to verify the functionality of the television set as a means of testing hardware or software faults within the device. This implementation of OCR technology is very similar to our proposed design, since we will be attempting to locate text within each frame prior to OCR processing, which will greatly speed up our application and reduce the overall workload for our system. However, a few key differences in the implementation again stem from the input type and output formatting. In our application, we will be accessing frames of a television recording rather than grabbing frames from a live feed. Additionally, our application will be focused on making timestamp-indexed, query-able, phrases stored in a database table.

Lastly, some previous speech-to-text and video-to-text work has been made by other senior design teams for DigiClips. After some research and analysis of these previous implementations, we have found that the currently existing speech-to-text system is suffering from low accuracy while the currently existing video-to-text system suffers from extremely high processing times as well as low accuracy and consistency. Nevertheless, this existing work has the benefit of providing a point that we can work off rather than building from the ground up, as some parts of this project will already have a basic implementation. However, given our proposed architectural changes and our choice of processing libraries, for the most part we feel that this previously existing solution will not lend us much help, especially since it is not well documented.

3.2 Design Thinking

We had performed decision making when considering and defining what data DigiClips is currently missing from their recorded channels. After discussing with Bob and Henry, who pointed out that a lot of data gets lost from text that is not necessarily spoken but appears on screen as a scrolling bar or other display, users would almost certainly want to be able to search for that text as it contains news updates, so we decided to add the video-to-text into our scope.

In addition to the speech-to-text and video-to-text, we had also generated the idea of lip-reading to text, where watching lip patterns to provide text as well. In the case of lip-reading to text we felt that this task was not very feasible due to technological limitations. Getting accurate text results from reading lips in a video is an immensely challenging task that likely would have accuracy issues. Rather than attempting to do something like this we have decided to focus on speech-to-text because it is a simpler task that fulfills the same goal. Having a good speech-to-text system would replace the need for a lip-reading system.

During the ideate phase of our design thinking process we came up with some design decisions that we ended up not going with due to limitations. Primarily, using paid APIs and services to achieve our goals. For example, Google has a high-quality speech recognition API that would work great for our use case. However, this is a service that requires payment. For the standard speech-to-text service it is $0.006 / 15 seconds of audio. Running eight television channels through this constantly for a month would rack up charges close to $8,600. This kind of pricing model is not feasible for DigiClips to consider due to how much television and radio they constantly record.

Another decision we made during our design thinking process was related to programming languages. Initially, we thought that C or C++ might be the best for this kind of application. However, upon further research, it became clear that Python seems to have the most up-to-date and well-maintained libraries when doing optical character recognition and speech recognition. Using C or C++ would likely provide us with faster processing times, but libraries to perform this kind of analysis aren't as available as Python and would require much more strenuous integration.

3.3 Proposed Design

The first method of solving this problem that we came up with was to use C or C++ to develop an app that performs Optical Character Recognition (OCR) and speech recognition on video files. We initially thought to use C or C++ because of the speed that those languages offer. Having extra speed and efficiency would give our application an edge since DigiClips has a massive amount of data to analyze. Eventually, we discovered that there are not many libraries for OCR and speech recognition written for C and C++. There are some options, but they are difficult to use, not well supported, and not user-friendly.

Once we realized that C or C++ might not be a viable choice, we investigated possibly using Java. We were a bit hesitant to try Java because it is known to be much slower than other languages due to its built-in garbage collection and inefficient data access. However, there is a well-supported speech recognition library, CMUSphinx, designed to be used in Java. We found that using this library to perform speech recognition was effective at fulfilling the speech-to-text part of our project's requirements.

At this point, we had already found that Java was not a good choice for the video-to-text portion of our project, so we began to consider using a microservice-based architecture. Since speech-to-text implementation in Java worked well, we could run speech-to-text in Java and video-to-text in another language. This design would utilize one of the primary benefits of using microservices, simple communication between different programming languages.

Next, we investigated using Python since our team had already done some work using Python's popular OCR software, Tesseract. Python seemed like a viable choice for doing the video-to-text portion of our project. In our research, we saw a speech recognition library developed by Mozilla called DeepSpeech. This library seemed like a viable choice because it is open-source and well maintained, so we set it up and tried it out with some of our sample data. After some initial testing, we realized that DeepSpeech had a similar level of accuracy when compared to CMUSphinx and was over twice as fast. This discovery made DeepSpeech the obvious choice to ensure our application is as efficient as possible.

At this point, we have decided on using Python both for our video-to-text system and for our speech-to-text system. These two systems must be as efficient as we can make them, so we must take care when designing their functionality and their interactions. We have also decided that a microservice architecture will be an effective solution to this problem because it is flexible and resilient to crashes that could hamper DigiClips' business model. Also, using existing REST API technology allows for requests to be made to each service in parallel. Parallel processing using REST API technology will enable DigiClips to keep up with incoming data without complex process management systems.

This design for our application will have three different services that will interact with one another: a central driver service, a speech-to-text service, and a video-to-text service. First, the central driver service will process each request as they come in, sending the audio to the speech-to-text service and the video to the video-to-text service. Then, the speech-to-text service will process the audio of the video clip to detect speech while the video-to-text service will process frames of the video clip to find text. Lastly, once the driver receives the results from both sub-services, the data will be returned to the service that made the request.

This project design will capture all the necessary processing into one application to streamline extracting data from the video files that DigiClips collects.

3.4 Technology Considerations

The current limitations are mainly on the computers used to run the programs that will be written. Video-to-text is a taxing process, and the CPU can only handle so much, so checking the entire video frame every frame would severely limit performance. In addition, the CPU must also handle the recording of the normal speech-to-text which also is rather intensive, though not as much as video-to-text. Performing both would be too much to handle, and so limiting frames and frame sizes will be needed.

Possible alternatives would be to attempt code optimization and reduce the code as much as possible while still performing the same job. Removing unnecessary functions and rewriting inefficient code may help lessen the burden these processes carry. As well, it may show that the current computer strength is limited, and in need of an upgrade. A newer computer, or adding another device, would help to split the load. Otherwise, we will need to decide how to limit our code to run smoothly.

We know that the computer that our application will likely run on has a Ryzen 9 5900X 12-core CPU. This is a very high-end CPU and will give us a substantial amount of processing power to work with. That computer also has a Nvidia GeForce 210 GPU. This is unfortunate because this GPU is low-end and does not offer much processing power. GPU processing is something that can make a significant difference in the kind of calculations we will be doing in both speech-to-text and video-to-text. Thus, we must take the lack of an effective GPU into consideration when designing our application.

3.5 Design Analysis

At the completion of our project, we find that our proposed design has been reasonably successful at fulfilling our speech-to-text and video-to-text goals. The proposed microservice architecture has been completely implemented at this point along with additional containerization features, and the video-to-text and speech-to-text systems are functioning as standalone services that can be bridged using the developed driver microservice. Now, we are focused around finding the best way to perform the speech-to-text and video-to-text.

In terms of the speech-to-text system, we have opted to split each audio file into chunks and process each chunk separately as an option to increase the efficiency and accuracy. Also, we are utilizing several different grammar and punctuation libraries to ensure our speech-to-text output is as readable as possible.

We maintain that the proposed iteration of our design is effective for several reasons. Firstly, the design is functional and capable of producing the results we wish to acquire, as seen throughout our rounds of design testing where we implemented a barebones system and compared the output to our expected result. Secondly, the design is maintainable. We have documented the code itself, usage procedures, and ensured that as much information as possible is given about our program so that DigiClips lead Software Engineer or other student groups can easily and quickly modify or implement our solution. Our design also makes use of API calls which are easy to swap in and out should a better API solution be developed. Finally, the components which we are still evaluating design effectiveness on are reliability and feasibility. In terms of reliability, we are confident that our system is able to handle the expected amount of traffic based on our extensive testing of program runtime, however we have been unable to generate the same amount of traffic our system will be potentially expected to maintain as we do not have full access to the television and radio recording devices at DigiClips. In terms of feasibility, our initial concern with the design was primarily focusing on the selection of a speech-to-text and optical character recognition API. While accurate, free, open-source models are difficult to come by, we believe that our choices of DeepSpeech and Tesseract OCR have proven to be accurate and reliable options for the data-intensive processing that our solution requires.

3.6 Development Process

We attempted to utilize Agile programming techniques to work on our project. To further this, we strove to work on our project using Feature Driven Development (FDD) techniques. We wanted to work in small pieces, have those pieces working as intended, then work on using those pieces to create larger parts and connect them all together. This made sure that the basic components worked, and that connecting the components was made easier. As an example of this FDD, in developing the speech-to-text engine we went through several stages to ensure that we can extract audio from a given video file, then utilizing this extraction process to split the audio into chunks before passing into DeepSpeech, and so on. This allowed us to work on our application piece by piece and more quickly detect issues with newly written code, since many of the previous functions could be assumed to be working given that they had been previously tested.

3.7 Design Plan

We ended up having a few different services that interact with each other to perform the necessary functions. Here is the proposed microservice structure:

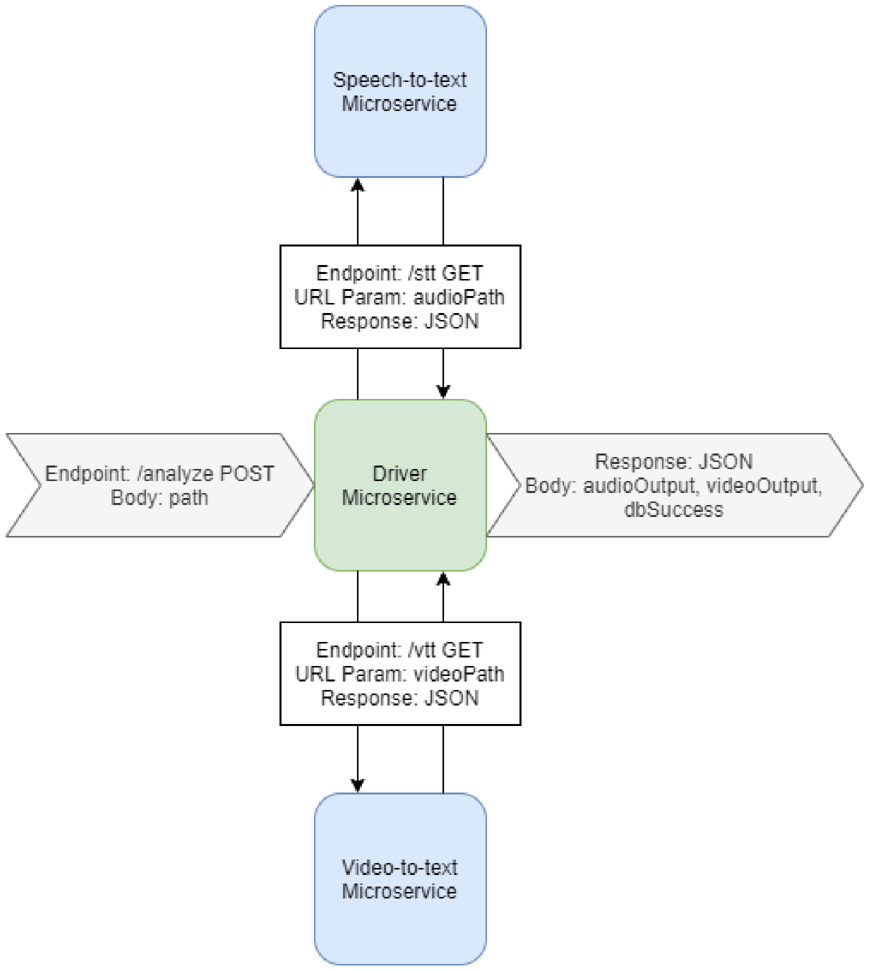


Figure 2: Microservice Structure

Driver Microservice

The entry point to this pipeline is through the Driver Microservice. HTTP POST Requests are made to an endpoint of this service. A file path is needed in the body of a request to supply a file to analyze. The Driver Microservice will process the file at the given path to prepare it for processing by the Speech-to-text Microservice and the Video-to-text Microservice, including verifying that the file exists, is of the correct type, and more. The Microservice is responsible for delegating the provided file to the sub-services based on its file type. For example, a video file can be passed to both sub-services while an audio file can only be passed to the speech-to-text service since there is no image data to process for the video-to-text. Finally, the path to the input file will be passed to the Speech-to-text Microservice and Video-to-text Microservice, at which point the Driver Microservice will wait for the results of both sub-services before responding to the original POST request.

Speech-to-text Microservice

Requests made to the Speech-to-text Microservice will be GET requests with a URL parameter containing the path to the audio file. The microservice will process this file to extract audio from the given file, split the resulting audio into chunks, then process text from the audio chunks. The extracted text will be formatted with JSON containing tags that identify the timestamp for each piece of text. This JSON will be sent back to the Driver Microservice.

Video-to-text Microservice

Requests made to the Video-to-text Microservice will be GET requests with a URL parameter containing the path to the video file that needs to be analyzed. The microservice will analyze frames of the given video, looking for text on the screen. This text will be extracted from the image frames. The microservice will return the extracted text within formatted JSON containing tags that identify the timestamp for each piece of text located in the video. The resulting processed JSON will be checked for grammar inconsistencies, then returned to the Driver Microservice to package and return to the original POST request.

4 Testing

Testing is an integral part of any project, as results from testing provide good feedback for improvement and give metrics to base future decisions off of. Through our use of testing, we are able to understand and refine how our program works.

4.1 Unit Testing

Unit tests were originally used in the determination of what type of speech-to-text and video-to-text libraries or tools to use for this project. Simple tests were created using single words or phrases, or single frames of shows, in order to determine which program was most accurate in transcribing. Once the decisions were made as to which program should be used for these transcriptions, unit testing then was lessened and attention was turned towards acceptance testing.

4.2 Interface Testing

Interface testing was to be used in making sure the microservice architecture we had created would be smoothly ran on the DigiClips machines without issue. This was made infinitely easier upon the use of Docker containers, which did a lot of the heavy lifting in assuring compatibility. By switching to a Docker container, we can rest easy knowing that there will be compatibility between our machines and theirs. Not only does this make maintainability much easier, but it also means that there needs to be little in ways of interface testing, since as long as Docker is able to be ran on their machine, then they can run our programs without any issue, as all dependencies, services, and other compatibility points are taken care of within the Docker container.

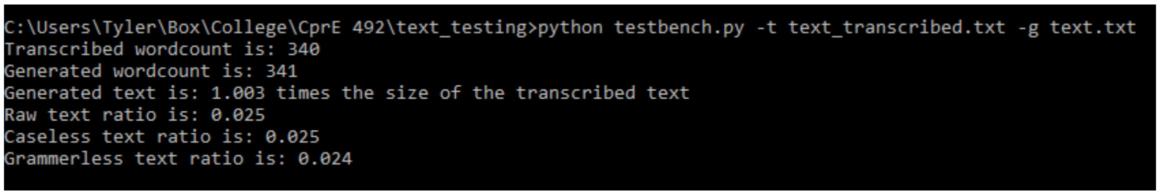
4.3 Acceptance Testing

The acceptance testing then became the primary concern, as once the basic decisions have been made and the path has been laid out, it comes down to making sure that the metrics are within what is expected of our program. In order to provide these metrics, a testbench has been set up to compare a human-transcribed text document with the programmatically-generated text. The main comparison points to look at are the wordcounts of each, the unaltered accuracy ratio, the case-less ratio, the punctuation-less ratio, and the overall speed of the generated transcription. These metrics will help to demonstrate any weak points of the program, by being able to compare the unaltered score with scores that ignore capitalization or punctuation it can show problems with the algorithm that determines the text. In addition, knowing other metrics such as the speed of transcription and the wordcount will provide insight into if there are any problems with the generation of text itself.

4.4 Results

For the results of this, the main focus will be on the speech-to-text system as this is an easier metric to demonstrate, however, similar improvements are reflected upon the video-to-text system as well.

Initially, the untrained and unoptimized program chosen was good at creating small words and phrases, however for longer clips it failed to hold up over the timespan.

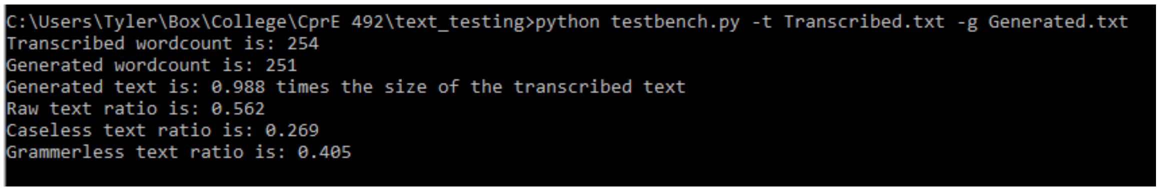


```
C:\Users\Tyler\Box\College\CprE 492\text_testing>python testbench.py -t text_transcribed.txt -g text.txt
Transcribed wordcount is: 340
Generated wordcount is: 341
Generated text is: 1.003 times the size of the transcribed text
Raw text ratio is: 0.025
Caseless text ratio is: 0.025
Grammerless text ratio is: 0.024
```

Figure 3: Initial Testing Accuracy

This initial accuracy is not very good, with a raw accuracy of only 2.5% being true to the human-transcribed text. It can also be noted that the values that ignore case or punctuation are similarly low. This was the first trial of the program, and there was much to be done to improve this to the expected accuracy.

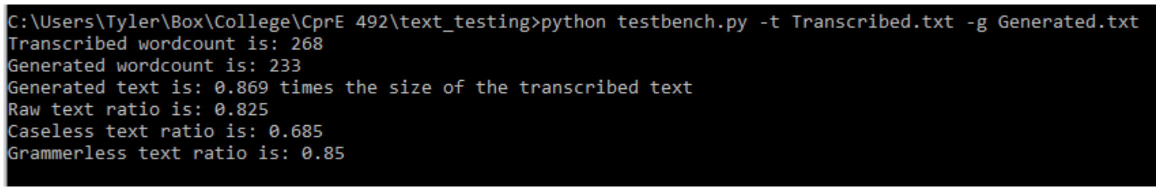
After a few months of work, notable improvements had been made to allow more acceptable results. These improvements are discussed primarily in Section 5: Implementation, however, these improvements bring the raw accuracy up to 56.2%, a large improvement. Note that in this situation, there were major issues with the caseless rate in particular, leading to the belief that there are issues in the correct casing of the text received.



```
C:\Users\Tyler\Box\College\CprE 492\text_testing>python testbench.py -t Transcribed.txt -g Generated.txt
Transcribed wordcount is: 254
Generated wordcount is: 251
Generated text is: 0.988 times the size of the transcribed text
Raw text ratio is: 0.562
Caseless text ratio is: 0.269
Grammerless text ratio is: 0.405
```

Figure 4: Improved Testing Accuracy

Finally, as the project nears its end, the last few parts have been tuned and provide what can be considered an acceptable set of results. These results have received a slight improvement, up to 82.5% accuracy raw. There are still slight drops when ignoring case, which is something that will need to be considered in the overall implementation of the program. However, overall this program does reach rather high levels of accuracy and is able to consistently provide text that is searchable, the overarching goal of the Digiclips team.



```
C:\Users\Tyler\Box\College\CprE 492\text_testing>python testbench.py -t Transcribed.txt -g Generated.txt
Transcribed wordcount is: 268
Generated wordcount is: 233
Generated text is: 0.869 times the size of the transcribed text
Raw text ratio is: 0.825
Caseless text ratio is: 0.685
Grammerless text ratio is: 0.85
```

Figure 5: Final Testing Accuracy

As stated at the beginning of this document, the goal for speech-to-text was to achieve at least 80% accuracy, which has been met by this criteria. Similar successes have been seen for video-to-text as well. After discussing both of these with the DigiClips team, they have approved of these metrics and as such we can consider acceptance testing for our system to be completed. After discussing with DigiClips about the low caseless ratio results, we were informed that most searches will not rely on having perfect case, and searches will be using different algorithms (such as fuzzy search) therefore having a lower case ratio is not critical and an acceptable loss.

5 Implementation

5.1 General Implementation

Our implementation uses Python to build the microservices that will make up our app. Python enabled us to use cutting-edge OCR tools and speech recognition models. While it may not be as fast as other languages like C or C++, we think Python is a great solution because of the variety of machine learning applications available.

To build our REST API microservices, we used the popular Python framework FastAPI. FastAPI offers a straightforward interface for building both simple and complex web APIs. There is minimal configuration and boilerplate needed to get a FastAPI web service up and running, which keeps our services slim and straightforward. The REST API structure enables our services to process requests simultaneously. Parallel processing helps our services process even more data within a given time. With FastAPI, the programmer doesn't have to create complex process management systems to facilitate parallel processing. The framework will handle everything automatically.

5.2 Speech-to-text

Our speech-to-text system uses the DeepSpeech library, developed by Mozilla using Tensorflow, a Python machine-learning library. Mozilla has gone to extreme lengths collecting thousands of hours of crowd-sourced voice recordings with their project Common Voice. They then use all this data to train their DeepSpeech recognition model used in our application. Using these existing libraries and systems enables our team to create an efficient and accurate application without collecting mountains of data and building custom machine learning models.

An important consideration for our speech-to-text service is how data is sent into the DeepSpeech model. Sending in an entire 30+ minute audio file can strain the computer the service runs on and takes a long time. Our solution to this problem is to split audio input into roughly 20-second chunks before passing it into the recognition model. Through testing, we have discovered that DeepSpeech doesn’t work well when a lot of audio is passed in at once. It seems that 20 seconds is a good middle ground between having too many chunks to process and not losing data by inputting too much audio at once. Also, the chunks have a second of overlap on each side to ensure we don't slice the audio in the middle of an important word.

Another benefit of breaking the audio into pieces is that we can process each piece in parallel. Using the Pool data structure in Python's multiprocessing library, each audio chunk is processed in parallel instead of looping through them sequentially. We sped up our speech-to-text processing by nearly 20% using this method.

One downside to using DeepSpeech is that we don't get any data on when a certain word was detected in the given audio clip. The model takes in audio clips and returns only a detected text string. Since our client wants to search through the output and find where a word occurs in a certain video, it would be great to link a word to a timestamp in the video it came from. We found that linking an individual word wasn't very feasible; however, using the chunking method described above, we can package each piece of text with some metadata to describe the file the text came from and the time frame within that file. Since we split audio into roughly 20-second chunks, we can associate each chunk of text with a 20-second time window in a given file.

The generated data is formatted JSON, packaged with the video text recognition data, and returned to the user.

5.3 Video-to-text

The video-to-text portion of our system utilizes OpenCV image processing library alongside Google's TesseractOCR to successfully extract visual text from individual frames of a news broadcast video. Our final implementation exists as a full-fledged API able to receive a video file and output a JSON-formatted list of extracted text along with their corresponding timestamps. The system is primarily a pipeline that opens the given file path, verifies that the file is a video, then proceeds to extract the necessary frames from the video before passing them through to the text detection functions. To limit the processing time and resources necessary for image processing tasks, we have opted to only perform image processing and optical character recognition on certain frames of the video. After considering many sample videos, we concluded that the likelihood that text would appear on screen for only a fraction of a second is extremely minimal and based on this conclusion we decided to only scan one frame for every second of video. This delay between processed frames is configurable to the user, so if our client chooses in the future that they would rather have every single frame processed, they can.

The system reads in a video file using OpenCV's VideoCapture object and begins to step through the video frame-by-frame. As each frame is passed, a counter keeps track of the frame's location relative to the beginning of the video which, along with the framerate of the video, is used to determine the frame's timestamp to be paired with the text output. Some image preprocessing is performed on each frame such as binary thresholding, image dilation using a kernel optimal for conjoining nearby text characters, then finally filtering remaining contours in the image based on their size and aspect ratio to determine which objects in the frame are most likely to be text. This serves to highlight the contrast between text and background and in the vast majority of cases removes noisy objects from the image that make text detection troublesome and isolates the text within the frame for easier processing. After pre-processing, the frame is passed into Google's

TesseractOCR, a free and open-source optical character recognition library, which extracts text from the image and parses it into a string.

Once the Tesseract output is received, some minor post-processing is performed to trim whitespaces, eliminate excessive punctuation, and check words for spelling errors. Finally, the frames are run through a duplicate filtering algorithm that attempts to identify neighbor frames as one "segment" of a frame being visible. If, for example, the same text is visible on screen for five consecutive frames, the client would rather store that text as a single instance with a generated start timestamp and end timestamp rather than five instances of single-frame timestamps. To do this, the program utilizes an algorithm that compares a frame to its predecessor, using Damerau–Levenshtein distance to determine the likeness of the two strings. The algorithm continues comparing subsequent frames until it reaches a frame that does not meet the likeness requirements specified in the configuration files.

After completing duplicate filtering, the array of timestamp-indexed string objects is returned in a JSON-formatted output to the driver microservice, which packages its contents alongside the output from the speech-to-text microservice and passes the results to the original requestor.

5.4 Driver Microservice

The last microservice we created was the driver microservice. This microservice helps to simplify the use of our application. Instead of making requests to the speech-to-text and video-to-text services individually, the user only must send a request to the driver microservice. The driver will make requests to both the speech service and video service in parallel, package the resulting data together and return it to the user. This also helps with our application's security because the users don't have direct access to the speech or video services. This provides a layer of separation between the user and the main processes of our application.

5.5 Docker

Another important aspect of our project implementation is our use of Docker. Since we have three different microservices described above that make up our application, things could get messy trying to ensure they are all running, and all dependencies are installed to operate each service effectively. We also need to ensure they can all communicate effectively while ensuring users cannot directly access the speech and video services. Docker solves both problems incredibly effectively.

First, Docker makes the whole application portable. All the dependencies and setup for each service are contained within the Dockerfile. Thus, the only software a user would have to install to run our code is Docker. Then, each service can be built into a Docker image using the provided Dockerfile and then run within a container. Instead of installing DeepSpeech, OpenCV, Tesseract, and all the other dependencies our project requires, we use Docker to automatically install all of that within a container. Using Docker, we got our code running on our client's machine with only a few console commands and no messy dependency installation.

Docker also solves our networking problem. We want the speech and video services to be only accessible by the driver service. Using Docker Networks, we put all three containers into their a network so they are effectively in a bubble where they can't be accessed by something outside the network. The driver microservice has its port 5000 mapped to port 5000 on the host machine, which serves as our one access point to the application.

Overall, Docker reduces many of the pain points of setting up an application like this to nothing more than a few console commands. Once all the containers are running, they can be paused and resumed with ease, and none of the software installed in each container affects the other programs running on our client's computer. If for whatever reason, someone wants to stop the application and uninstall it, the containers need to be deleted, and it will be as if they were never there.

6 Closing Material

6.1 Conclusion

Our team has worked closely with DigiClips to create and develop an initial plan to complete the project we have been assigned. Our goal is to create an element of speech-to-text and video-to-text that will use self-developed and open-source software, which will in the end be incorporated into the DigiClips overall system, allowing them to translate any television recordings into something that is searchable using their own search engine. Currently, after doing research and different testing on each element, the best plan of action that we currently have is to develop each element individually, test the element that was created, then incorporate them into one product which can be used by the system in place at DigiClips. This solution, using python with different open-source software, will be the most reliable option with the given requirements.

6.2 References

Barcy, D. J., & Statkus, W. C. (2001, August 14). *United States of America Patent No. US6542200B1.*

Kastelan, I., Kukolj, S., Pekovic, V., Marinkovic, V., & Marceta, Z. (2012, September 22). Extraction of text on TV screen using optical character recognition. *2012 IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics*.

Leetaru, K. (2019, June 8). *Using Google's Speech Recognition And Natural Language APIs To Thematically Analyze Television*. Retrieved from Forbes: https://www.forbes.com/sites/kalevleetaru/2019/06/08/using-googles-speech-recognition-and-natural-language-apis-to-thematically-analyze-television/?sh=460cbd87783b

randall. (2017, December 17). *FM2TXT: Automatically Perform Speech To Text on FM Signals*. Retrieved from RTL-SDR: https://www.rtl-sdr.com/fm2txt-automatically-perform-speech-to-text-on-fm-signals/

6.3 Appendices

Appendix I: Operation Manual

Docker

Docker can be installed here: https://docs.docker.com/get-docker/

Docker is used to package each application into it's own mini virtual machine. This ensures that each application will be isolated from the host as much as possible while running. This project uses Docker to vastly simplify the process of getting the applications running with all appropriate dependencies installed.

An up to date install of Docker is all that is needed to run the applications.

Docker is not absolutely necessary to run the applications, but it does simplify the process. If a python 3.7 environment is available with all necessary dependencies installed the applications should be able to run fine using either uvicorn or the default FastAPI server.

Setting up the speech-to-text application

To setup the speech-to-text (stt) application first install Docker locally using the link above. Once Docker is installed pull the latest commit of this repository to your local machine and you will be ready to build the application.

First, we need to setup the network that our containers will use to talk to each other. To do this open a terminal or command prompt window and run the following command:

docker network create toText

This command creates a network called toText. This name can be whatever you want as long as it remains consistent throughout the rest of the setup.

Next, navigate to the main repository folder and then into the stt folder. There are some files that need to be downlaoded into the models folder of the repository. To use Deepspeech we need two files, the model file and the scorer file. Those files can be downloaded here:

Model: https://github.com/mozilla/DeepSpeech/releases/download/v0.9.3/deepspeech-0.9.3-models.pbmm

Scorer: https://github.com/mozilla/DeepSpeech/releases/download/v0.9.3/deepspeech-0.9.3-models.scorer

Download these two files and put them into the models folder in the stt folder.

We also need the model file for the punctuator. This file can be accessed in a Google Drive hosted by the library's creator at this link:
https://drive.google.com/drive/folders/0B7BsN5f2F1fZQnFsbzJ3TWxxMms?resourcekey=0-6yhuY9FOeITBBWWNdyG2aw

The file to download is named Demo-Europarl-EN.pcl. Place this file in the models folder too.

Now, run the following command:

```
docker build -t stt .
```

This command builds a docker image using the Dockerfile in the stt directory. The Dockerfile provides the instructions to build the right container for the app. Also, in the above command the -t flag denotes the name of the image we are creating. So if you don't want the name to be 'stt' then change it to whatever you like. But keep in mind you'll have to remember this name later.

Now that we have the image built we are ready to run the image in a container. To start it up run the following command (with a couple changes based on your setup):

```
docker run—net toText --mount type=bind,source=/path/to/files,target=/test/audio --security-opt seccomp=unconfined --name stt -d stt
```

This command creates a new container named 'stt' and runs the image 'stt' inside it. The --name flag set the name of the new container that is being created. Here we set it to 'stt' the same as the image name.

The --mount flag creates a link between a folder on the host (the computer docker is running on) and the container file system. This is useful since all the Digiclips recordings are stored on the linux filesystem. This path will be different depending on where the files you want to process are on your computer. The target is the location in the container's file system that we will link to.

The –net flag links this container to the network we created earlier. This will ensure all the containers can communicate within their own network.

The --security-opt flag is a little more complicated. Essentially, Docker enables a security option by default that protects against Spectre attacks. This security option causes performance issues with CPU-intensive processes like our application. This option disables that security option in the container so the performance is not affected. This does increase the applications vulnerability to Spectre attacks but at this time there is no other solution without affecting performance.

Now the speech-to-text service is up and running! We can let it run while we setup the other services. Next we will run the video-to-text service.

Change your terminal working directory to the vtt folder and run the following command:

```
docker build -t vtt .
```

This command will build the video-to-text service similar to the speech-to-text service above.

```
docker run --net toText --mount type=bind,source=/path/to/files,target=/test/audio --security-opt seccomp=unconfined --name vtt -d vtt
```

This command runs the video-to-text service the same way we ran the speech-to-text service. Now two out of three services should be running.

Change your terminal working directory to the driver folder and run the following command:

```
docker build -t driver .
```

Now we build the driver microservice.

```
docker run -p 5000:5000 –net toText--name driver -d driver
```

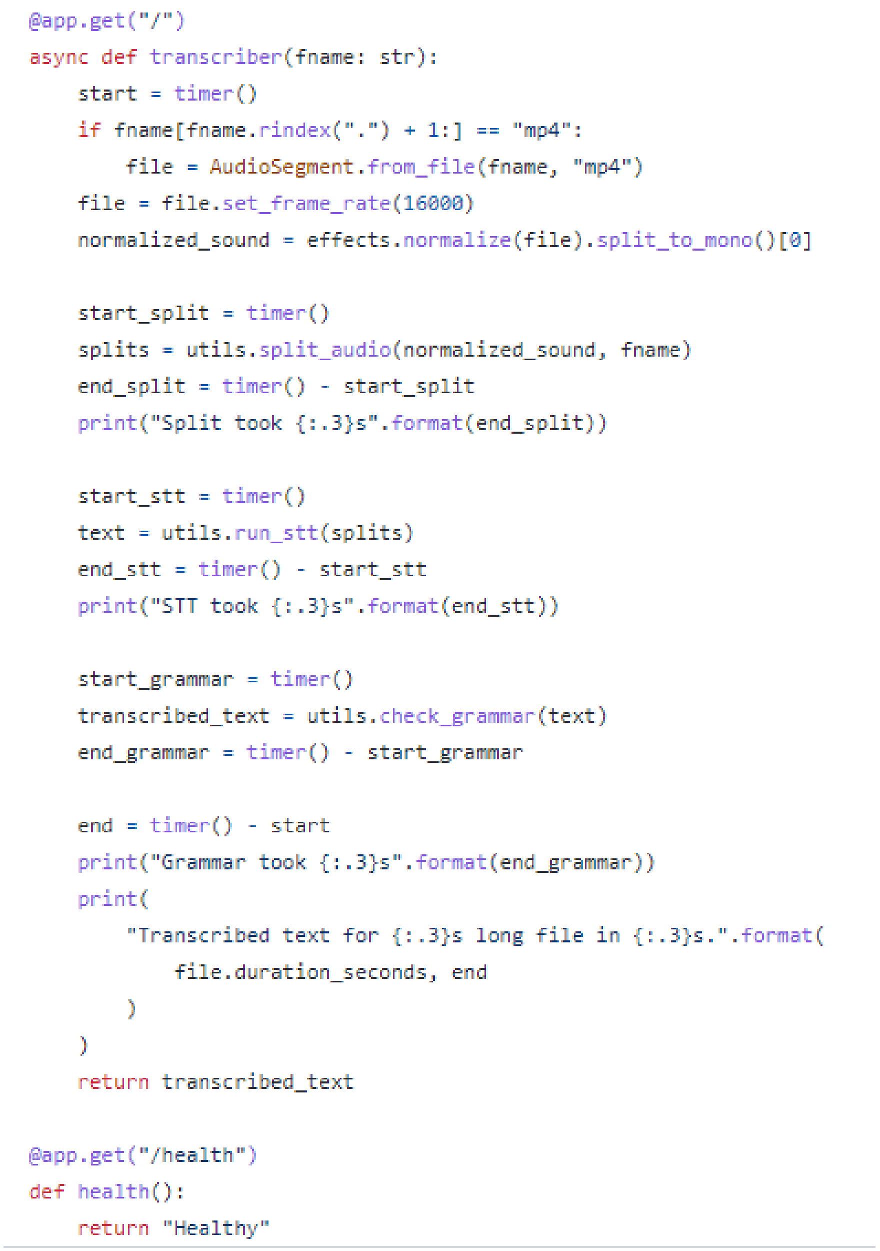
And we run the driver microservice. This service doesn't need access to files and isn't processor intensive so we don't need as many flags. The flag -p links a port on the host machine to a port on the container. The Python API inside the container uses port 5000 so for simplicities sake we will link port 5000 on the host to 5000 on the container. All this really means is when we make requests to the application we will direct them to port 5000.

Now that all three services are running, using either Postman, curl or some other method of making HTTP requests, make GET requests to http://localhost:5000/. You will also need a single URL param called 'fname' that points to the file to be processed. This will be '../test/audio' + the name of the file. A complete request would look like the following:

```
http://localhost:5000/?fname=../test/audio/test_file.mp4
```

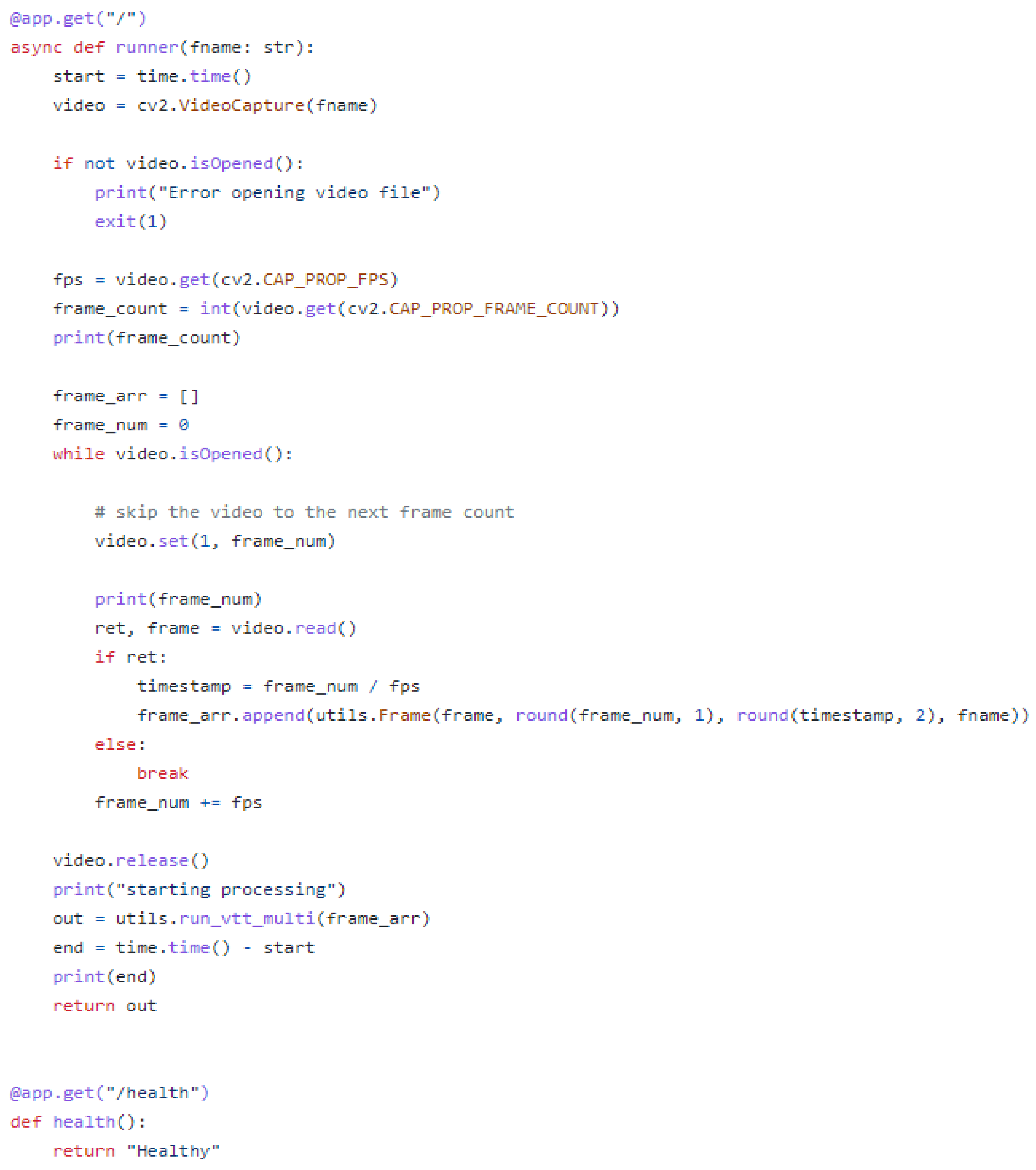
The application will process the request and response once it has finished.

Appendix II: Code



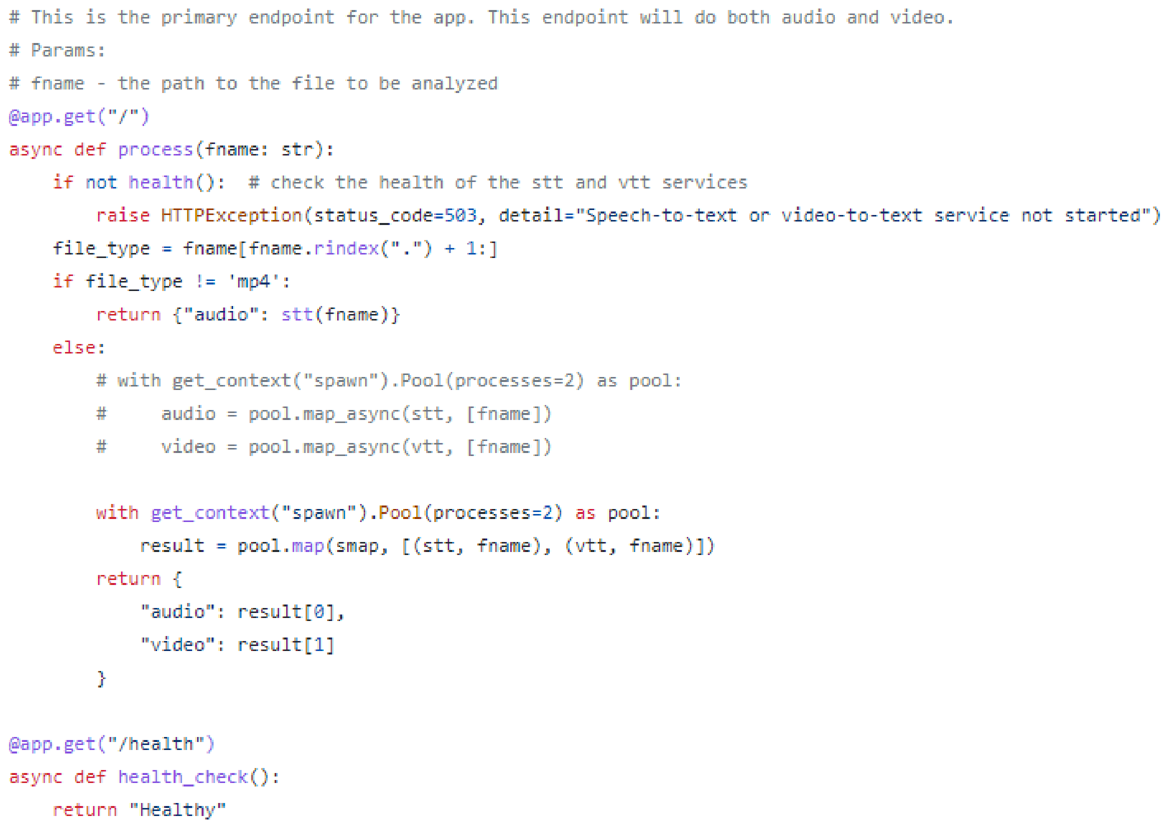
```python
@app.get("/")
async def transcriber(fname: str):
    start = timer()
    if fname[fname.rindex(".") + 1:] == "mp4":
        file = AudioSegment.from_file(fname, "mp4")
    file = file.set_frame_rate(16000)
    normalized_sound = effects.normalize(file).split_to_mono()[0]

    start_split = timer()
    splits = utils.split_audio(normalized_sound, fname)
    end_split = timer() - start_split
    print("Split took {:.3}s".format(end_split))

    start_stt = timer()
    text = utils.run_stt(splits)
    end_stt = timer() - start_stt
    print("STT took {:.3}s".format(end_stt))

    start_grammar = timer()
    transcribed_text = utils.check_grammar(text)
    end_grammar = timer() - start_grammar

    end = timer() - start
    print("Grammar took {:.3}s".format(end_grammar))
    print(
        "Transcribed text for {:.3}s long file in {:.3}s.".format(
            file.duration_seconds, end
        )
    )
    return transcribed_text

@app.get("/health")
def health():
    return "Healthy"
```

Figure 6: Speech-to-text Endpoints



```python
@app.get("/")
async def runner(fname: str):
    start = time.time()
    video = cv2.VideoCapture(fname)

    if not video.isOpened():
        print("Error opening video file")
        exit(1)

    fps = video.get(cv2.CAP_PROP_FPS)
    frame_count = int(video.get(cv2.CAP_PROP_FRAME_COUNT))
    print(frame_count)

    frame_arr = []
    frame_num = 0
    while video.isOpened():

        # skip the video to the next frame count
        video.set(1, frame_num)

        print(frame_num)
        ret, frame = video.read()
        if ret:
            timestamp = frame_num / fps
            frame_arr.append(utils.Frame(frame, round(frame_num, 1), round(timestamp, 2), fname))
        else:
            break
        frame_num += fps

    video.release()
    print("starting processing")
    out = utils.run_vtt_multi(frame_arr)
    end = time.time() - start
    print(end)
    return out


@app.get("/health")
def health():
    return "Healthy"
```

Figure 7: Video-to-text Endpoints



```python
# This is the primary endpoint for the app. This endpoint will do both audio and video.
# Params:
# fname - the path to the file to be analyzed
@app.get("/")
async def process(fname: str):
    if not health():  # check the health of the stt and vtt services
        raise HTTPException(status_code=503, detail="Speech-to-text or video-to-text service not started")
    file_type = fname[fname.rindex(".") + 1:]
    if file_type != 'mp4':
        return {"audio": stt(fname)}
    else:
        # with get_context("spawn").Pool(processes=2) as pool:
        #     audio = pool.map_async(stt, [fname])
        #     video = pool.map_async(vtt, [fname])

        with get_context("spawn").Pool(processes=2) as pool:
            result = pool.map(smap, [(stt, fname), (vtt, fname)])
        return {
            "audio": result[0],
            "video": result[1]
        }

@app.get("/health")
async def health_check():
    return "Healthy"
```

Figure 5: Driver Endpoints