# CodeSwitch

Marco Yu, Nick Toldalagi, and Honghao Chen

Department of Electrical and Computer Engineering, Carnegie Mellon University

*Abstract*—**A system capable of handling English/Mandarin code-switching for use in ASR instant messaging applications. Current systems dynamically select between models using a separate LID module which have been shown to be less effective in CS contexts than jointly trained LID and CS models. The system should be plausibly deployable for use in mobile devices. This drives requirements for the end-to-end transcription system's size, speed, and throughput. Evaluated on our limited-size dataset, we were able to achieve a comparable CER of 18.25% in relation to the latest current literature in the field.**

*Index Terms*—**Automatic Speech Recognition, Code-switching, Deep Learning, End-to-end, Instant Messaging, Language Identification, Self-supervised Learning, Speech-to-text, Transformers.**

## I. Introduction

Automatic Speech Recognition has been an area of great focus for machine learning researchers. In the dozen years since 2010, use of the technology has burgeoned from academic experimentation to wide adoption in consumer-facing applications. Thanks to larger labeled speech corpi, better computational resources, and advancements in model architecture, performance metrics of state of the art ASR have practically saturated in many problem domains–surpassing even humans in WER performance.

ASR systems are currently widely available in a range of applications and devices where users are able to harness speech-to-text tools for writing documents, sending text messages, and recording conversations. While the training of these deep models requires significant computational resources, they have seen successful deployment even on mobile and edge devices such as the iPhone through the combined use of a cloud-based computation client and sometimes special native neural hardware accelerators. Though the recognition of dozens of languages are individually supported on these devices, situations in which speakers may switch between languages mid-sentence (code-switching) are not specifically handled.

For conversations and messaging use-cases between multilingual speakers, the accurate handling of code-switching is necessary to allow the participants the ability to express themselves authentically, naturally, and efficiently while capturing their words with precision. This system focuses on handling CS within the context of instant messaging which motivated early design decisions surrounding its architecture, size, latency, and throughput. The less-formal nature of this type of communication allowed us to limit the vocabulary which the model would be required to learn and it is also a use-case in which we expect the real-world prevalence of CS to be high. As the implementation progressed and the system was evaluated, several of these early decisions had to be amended to prioritize creating the most functional prototype over one which could be directly deployed on a mobile or edge device.

When the project began, the goal of this implementation was ultimately to emulate the experience of current monolingual ASR systems. These systems provide ease-of-use, fast response times, and accuracy which make them competitive with a traditional instant messaging experience. Achieving the latter of these high-level features proved to be the hardest challenge during our implementation attempts with our final system falling short of competitiveness with the current state of the art. Though improving accuracy consistently consumed most of our implementation efforts, issues with the quality of our dataset and model training difficulty saturated our achievable performance below expectations. Acknowledging this, a secondary objective of this project has become investigating and explaining precisely where the methods used fell short and what steps could still be taken to reach a competitively high accuracy.

Current implementations such as Apple's text-to-speech feature available in its Messages app perform exceptionally well for day to day use. Though the specifics of this implementation is difficult to find, evidence from experimentation with the feature seems to indicate the use of individually-trained state of the art end-to-end ASR models per language in combination with a small LID model used for choosing the correct model per word [3]. This results in some instances of confused multi-lingual transcriptions in which individually switched words are missed or badly transcribed within the context of the sentence. We planned to overcome these deficiencies by using a jointly-trained LID/ASR architecture which would be able to capture multi-lingual contextual information on both the LID and ASR portions of the transcription problem. A feature extra tor and tunable transformer model front-end which had been pre-trained on multilingual speech were intended to help provide usable language and context information for the downstream modules of the model to make accurate and robust predictions. Though we attempted for several weeks to properly implement and train this architecture, we were ultimately forced to settle on a final implementation which used a multiplexed architecture similar to the one which we suspect is currently used by Apple in an effort to prioritize the delivery of the best performing system for our final demonstration.

## II. Use-Case Requirements

Targeting instant messaging as a use-case demands special requirements be fulfilled by the final system. Speed and accuracy are the first and foremost as they most directly affect the final quality of the user experience. UPL could not be more than 3 seconds, allowing the system to have a similar response time to the iPhone for the first word of an ASR transcription to be returned. The system's accuracy was initially quantitatively measured by the WER (word error rate) metric, but we clarified this to a CER metric since this better captured our system's performance as we decided to do character level transcriptions. As text-to-speech instant messaging was expected to take place in a wide range of audio environments, the system also needed to be robust to background noise, meaning accuracy is maintained even with a lower SNR. For any audio with an SNR greater than 20dB (30dB is considered clean speech), the CER could be no greater than 25% while audio with an SNR as low as 5dB could dbe no greater than 30%.

A vocabulary size of 5000 tokens was estimated to accommodate most day-to-day conversations which we expected in instant messaging. Though we did not expect users to be submitting messages longer than 10 or 20 seconds, we required that the system be capable of handling single messages which correspond with up to a minute of audio. Originally, the throughput requirement for the system focused on matching the average speed a human is capable of typing on a mobile device or ~1.2 seconds/word. The team realized that not only was this too low, but it did not focus on the purpose of meeting a throughput target. Instead, the system only needed to ensure a high enough throughput to create back pressure on the translation pipeline. Therefore, the updated requirement became that a second of raw audio take less than a second to process on average.

Another requirement which we planned to achieve was limiting the size of the model to no more than 20MB prior to compression attempts. This was meant to roughly correspond with the size of Apple's latest mobile chip's system cache (A12). This was excluding the feature extractor/encoder head as they were expected to consist of at least 1GB and would therefore live in a cloud instance in a commercial implementation.

The user interface to the system needed to be simple to use and support multiple browser and audio file types for eventual consumption of the prediction model. To emulate the experience of typing, the interface also had to provide visual feedback to the user by updating in real-time.

## III. Architecture and/or Principle of Operation

The coarse architecture of our system remains relatively unchanged from our original proposal. The system still consists of a browser-based user interface transmitting audio to and receiving transcriptions back from a cloud-hosted backend. Most of the computation of the system resides in the backend. This remains aligned with our intention to focus on creating a functional translation system as our primary objective of the project rather than attempt a full deployment of the system onto a mobile device that is better aligned with our use case. A visual specification of the system can be found in Figure 8 on page 14. It's broken into six software modules. Two reside locally while the other four are hosted and run remotely. Audio is chunked by the frontend to allow for the appearance of continuous audio streaming and translation. Corresponding translation chunks are returned to the UI by the backend through the same API that accepts the audio chunks.

Module 5 consists of the UI of our webapp. It presents the user with the record button that reacts to indicate whether the system is actively recording. This raw audio is fed directly into Module 3. Module 5 is also responsible for sorting and displaying text chunks received back from the backend in case of out-of-order arrival of transcription chunks due to the network jump. These chunks are accumulated and displayed in real time. Once a new "session" has begun by stopping and starting the recording using the provided button, the new transcription text is displayed below the old, mimicking a new message in the conversation. No major changes were made to this model from the original proposed design.

Module 3 is the audio streamer responsible for processing the raw audio recorded into a stream of data used by the downstream system that is capable of giving the appearance of continuous processing to the user. Before sending the audio across the network jump to the backend, it is chunked. The length of these audio chunks was initially a system hyperparameter set statically. However, after experimentation we realized that, by its nature, our translation model performed best when audio segments included entire words or phrases. This led to an architectural addition of a silence detector and partitioner submodule which is capable of splitting raw audio chunks that are dynamically segmented by silence. The threshold for the detection of this silence then became the new hyperparameter for tuning.

Module 4 acts as the API for the system's backend, handling the communication to and from the front-end modules. As it receives web-formatted audio data chunks for Module 3, it converts to .wav chunks which are passed on to Module 6 as logically independent transcription requests. The corresponding text transcriptions will pass back through this module before passing directly on to the frontend UI in Module 5. A small optimization we made for accuracy was to add an English spell-checker in this module that can correct for small transcription errors in the English text only. It is not capable of large fixes but with a low latency overhead it provides a quick solution to small mistakes.

Module 6 as a whole did not change significantly from the first architectural design. It encapsulates all of the translation logic of the system with the underlying logic being separately encapsulated by modules 1 and 2. To support scalability of this system, this module is the finest granularity that would be required for replication to handle frontend inputs from many users. Transcription requests arrive as audio chunks from Module 4 and populate an input buffer upon arrival. Since each request is treated as logically independent, requests from several sources could arrive at a Module 6 simultaneously and be handled without interference. Before passing through the translation modules, the audio is passed through the XLSR-53 feature extractor whose embeddings are then passed to both the LID and ASR translators. This is why the feature extractor is considered to reside in this module. Previously, this module was also responsible for the fusion of prediction logits returned by the LID and ASR modules. These were then passed through a CTC module which generated the final sequence of tokens or "text" corresponding to the input audio chunk. Having switched to a multiplexed prediction strategy, the output of the prediction logic is now a collection of Mandarin token chunks and English token chunks returned by the Mandarin and English ASR models, respectively. A single one of these chunks does not correspond to the entire input audio chunk but instead a small segment of it. This is explained further in the architectural section concerning Modules 1 and 2. Module 4 is solely responsible for interleaving and joining these text chunks into a final transcription passed onto Module 4.

Modules 1 and 2 contain the heart of audio to text prediction pipeline. From the beginning, the concept of splitting the problem of CS speech recognition has been separating the task of language identification (LID) and automatic speech recognition (ASR). Module 2 comprises the ASR models while Module 1 contains the LID model. Module 1 has not changed. It still accepts embeddings from the XLSR-53 feature extractor and decoder, returning logits over three tokens: English, Mandarin, and blank (silence). While these would have previously been fused directly with logits from Module 2, they are now fed into Module 2 as a sequence of LID "tags" with each tag representing the most likely frame-level prediction of any of the three tokens. The crux of our multiplexed implementation now takes place in Module 2. Using the boundaries between LID tags, the embeddings from the feature extractors could now be segmented into suspected English and Mandarin segments (with no prediction needed on chunks of silence). These segments are passed through their corresponding ASR models (of which we now have two rather than a single English/Mandarin model). Finally, a collection of English and Mandarin tokens chunks are produced whose union corresponds to a transcription for the original input audio to Module 6. This updated strategy of text prediction increased the critical path through the translation pipeline compared with the joint LID-ASR model we'd originally proposed since LID and ASR prediction must now be done sequentially instead of in parallel. The effect of this extra latency did not prevent us from meeting our latency targets, however, as presented in section VII.

## IV. Design Requirements

The requirements identified for this system's use case early in this document had to be achieved by a further partitioning of these requirements into specific requirements for the various modules and components of its design. Certain requirements could be shared by several modules with the interactions between these modules informing the quantitative requirements that were actually assigned to each module.

Model size was dependent on the sizes of both Module 1 and Module 2. The overall budget of 20MB was split between the two with rough considerations given to the relative complexity of each module. Originally, Module 2 received 11.5MB and Module 1 received 9.5MB. Since the design now uses three distinct models (2 for ASR for LID), the requirement was updated such that Module 2 now received 14MB and Module 1 received 6MB as Module 2 contains an extra model that was also anticipated to be larger than the LID model.

Overall latency was also treated as a budget which could be assigned between each module of the system. The initial budget considered for latency was 3 seconds to correspond with the time roughly measured for Apple's speech-to-text iPhone messaging feature. As development began, this was found to be too lenient and easy to achieve so it was lowered to 1 second. This meant the design requirement had to account for a roundtrip time for an audio chunk to travel the path from Module 5, then 3, then 4, then 6, then back again to be less than a second.

### *A. Modules 1 & 2: LID & ASR Model*

Specifications for the requirements of these modules in early design documentation for this project considered the modules are run in parallel. As a consequence of the updated architecture, their requirements had to be updated to reflect the fact that they now run sequentially, with the outputs of Module 1 (LID) flowing into Module 2 (ASR). The combined latency of this critical path is limited by the latency budget allotted to their parent Module 6 which is .5s.All prediction operations in the ASR and LID modules are GPU accelerated and were therefore anticipated to be very fast on the small inputs that they would receive for each chunk. Any extra work not capable of being performed on the GPU was thus considered the primary factor in how to decide the latency requirements for each module. Of the two, the ASR module has to perform additional logit segmentations not present in the LID module and therefore received more of the .2s budget allotted to Modules 1 and 2. In the end, Module 1 received .05s of latency and Module 2 was allotted .15s.

Accuracy of the overall transcription is related to the accuracy of both the LID and ASR modules. Though the requirement metric CER (measured at the character level) cannot be directly tied to the outputs of these modules (which are token predictions at the audio frame level), some rough calculations can be done to decide on the accuracy needed from each model to reach the system's CER requirement. When we recalculated design-level requirements, Module 1's error was reconsidered to be measured by CER the same as Module 2's rather than with a frame-level classification error to allow for a more correct mathematical combination of the two. When presented with clean audio (greater than 20dB

SNR), the LID module was required to have a CER of no more than 10%. Consequently, the ASR module needed to have a CER or no more than 16% to achieve an overall CER of 25%. This follows from the fact that we'd expect a 75% chance of being successful for each character which would be met with these requirements due to $0.9 \times .84 = .756 > .75$. One can proceed with the same thinking for noisy conditions. A LID error no more than 12% requires an ASR error of no more than 20.3% derived similarly from $.88 \times .797 = .701$. It should be noted that these calculations rely on the weak assumption that the CER of the ASR module is independent of the CER of the LID module which is no longer true due to the multiplexed architecture of the updated system.

### *B. Module 3: Audio Streaming*

As it is responsible for audio chunking and streaming, this module's primary requirement lies in its latency. Originally this module only dealt in 2-second audio chunks. These were not thought to be large for network bandwidth to be a significant bottleneck. This assumption did not change with the updated dynamic chunking based on silence which the system now uses as most chunks would now be even smaller though we would expect some small rise in the rate of chunks being sent to the backend by this module. Scaling the original .2 seconds for a 3 second total latency requirement would result in a new Module 3 latency of ~.066s. However, with the addition of the silence detection and chunking unit, the internal latency of the was expected to rise resulting in a final latency of .08s between the moment the first audio data was fed into the module and the moment the first chunk of that audio was received by Module 4 in the backend. The team recognized that this latency could vary significantly based on the connectivity, but this was accepted as out of the project's scope or control.

### *C. Module 4: Backend API*

Module 4 includes a .wav generator for incoming audio chunks and an English autocorrector for outgoing text translations. The overhead for autocorrection is considered nearly negligible as from testing it completed nearly instantly and most returned transcriptions correspond to less than 2 seconds of audio (a short amount of text) anyway so that latency should not be especially variable. As such, the latency for a request returning from Module 6 to Module 5 was set at .05s. Audio conversion to a .wav took longer and was more directly dependent on the audio chunk size. Compared with our early proposal, a majority of chunks received by Module 4 are now smaller than 2s as they are dynamically sliced on silence. This allowed for an aggressive cutting of Module 4s inbound latency also down to .05s.

### *D. Module 5: Web Frontend*

The frontend UI is the only portion of the system directly visible to users. As such, it must be easy to use and clear in its function. Random hiccups that may occur somewhere in the pipeline must therefore be handled silently and reasonably. To begin with this means that the transcription output must be reasonable. No random symbols should be displayed other than Mandarin, English, or spaces (this is a requirement also shared by Modules 1, 2, and 6). Transcription chunks should also never appear out of order. If the network reorders them, they must be properly organized by this module before they become visible to the user. To keep our web app accessible and technically supported, we targeted the easy to use and widely adopted Chrome browser for compatibility with our UI. Regardless of their device, anyone capable of downloading the latest version of Chrome had to be able to access the app.

### *E. Module 6: Language Model*

This module and its components are nearly entirely responsible for the accuracy of the system and were anticipated to be the largest contributors of UPL latency in the pipeline. Because of this Module 6 was originally granted half of the total latency requirements or 1.5s. With the scaling of the original latency requirement, this fell to .5s. With a longer critical path introduced in the transcription task performed by Modules 1 and 2, they received a larger portion of the module's allowed latency, leaving .3s for the module to perform the other tasks surrounding transcription. These other tasks consist of passing the audio through the feature extractor and joining the collection of Mandarin and English token chunks together into a final transcription.

The translation accuracy requirements for this module are identical to that of the system since it is the combined accuracy of its modules which contribute entirely to this metric.

## V. Design Trade Studies

Over the course of our development, to achieve different the two main design requirements of our project, short end-to-end latency and high transcription accuracy, we experimented with multiple approaches to implement various parts of our system, including our audio stream chunking mechanism and architecture of our transcription and language identification model.

### *A. Audio Chunking Mechanism*

From the start of our development phase, we have been consistently using the technique of sending chunks of ongoing audio recorder streams for transcription to achieve the effect of real-time transcription. segmentation mechanism of input audio stream.

#### *1) Fixed-sized vs. variable-sized chunking*

Our initial approach was to send fixed length chunks. Our hope was that through benchmark testing, we could find an optimal chunk length. If each chunk is too long (> 3000ms), a frame of input audio is sent out only every 3 seconds, meaning that the speaker would see the transcription of what they spoke at least 3 seconds later. If each chunk is too short, a single spoken word is likely to be cut off into different chunks and

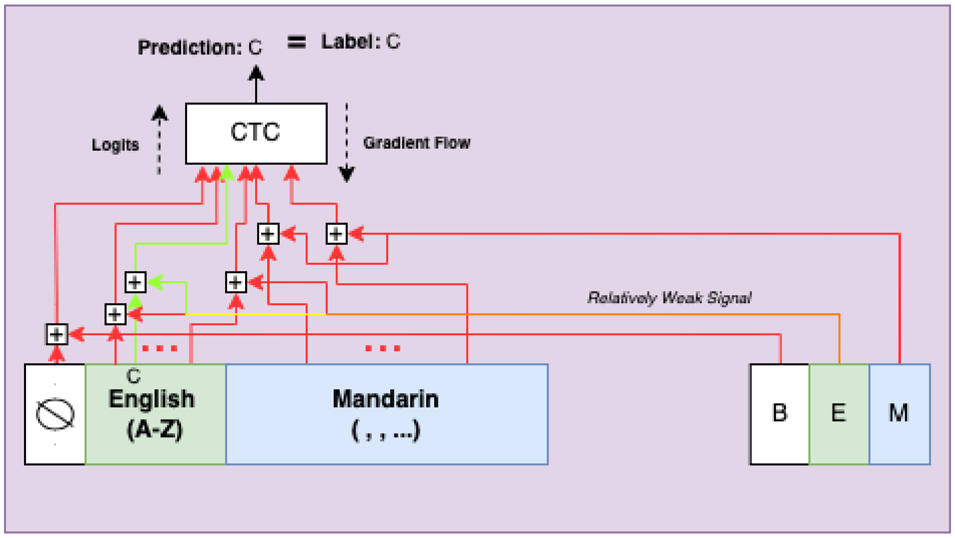
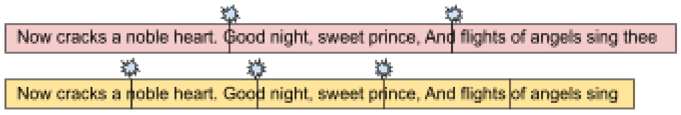


Fig. 1. The weak learning produced by back propagating over a fused LID and ASR CTC loss. Redder gradients are more negative while green ones are positive. Each line shows the gradient contribution of a single LID or ASR token.

fed into the transcription model separately leading to inaccurate transcription for both chunks. However, through rounds of parameter tuning, we realized that the fixed-chunk-size would never be able to fully remove the possibility of cutting into a single word, as illustrated in Fig. 1.



Now cracks a noble heart. Good night, sweet prince, And flights of angels sing thee

Now cracks a noble heart. Good night, sweet prince, And flights of angels sing

Fig. 2. This figure illustrates the problem of fixed-sized chunking. Simply changing the chunk size does not remove the probability of cutting off a spoken word.

Therefore, we began exploring variable-sized chunking mechanisms more capable of preserving the integrity on word level and preferably on phrase level. Inspired by the common speech pattern that pauses often exist between words and between phrases, we decided to use silence detection to identify significant pauses in the input audio stream and use them as boundaries for chunking the stream.

*2) Silence Detection on Frontend vs. on Backend*

We also faced design choices of performing audio chunking on the frontend or on the backend server.

Ideally, where we perform silence detection would not affect any correctness; however, based on our system design and the difference in silence detection tools available for frontend JavaScript and backend Python, there are advantages and drawbacks for each choice.

Performing silence detection on the frontend directly on the recorded audio stream can help limit the number of requests. On the frontend, we know precisely when there is a significant silence gap in the audio stream. Therefore, we only need to make a transcription request only when a silence gap is detected. If silence detection is performed on the backend server, the frontend would not have the direct knowledge of the occurrence of the silence gap until making a request to the server. In this case, the frontend could only make decisions on when to make a request based on some arbitrary semantics other than silence gaps, which could cause many unnecessary requests made to the server and consequently more unnecessary server load and more complexity on managing server responses that may contain overlapping transcriptions.

On the other hand, the drawback of performing silence detection is the limitation of off-the-shelf audio processing tools for JavaScript. Most audio processing packages for JavaScript only support processing an entire audio file. In our system design, audio processing needs to be performed on an audio stream that is still actively growing since the user is expected to see transcription as they continue speaking into the microphone. JavaScript packages that support audio analysis (such as measuring audio amplitude) on active media streams are limited according to our research. Web Audio API is the only package we found that safices our needs. And Web Audio API only supports getting the raw magnitude from the audio stream, so we would have to implement our own silence detector which could be less reliable in terms of accuracy compared to those commonly used audio processing libraries that directly provide silence detection available for Python.

If silence detection is performed on the backend, then we would have to make sacrifices on allowing some extra unnecessary server load and having to handle server responses that might contain transcriptions on overlapping audio portions on the frontend. However, the expected accuracy of silence detection is much higher in this case. The first reason is that when the audio data reaches the backend, it is already in the form of an entire audio file. Therefore, we do not have the problem of having to deal with an actively recording/growing audio stream on the frontend. The second reason is the higher availability of audio processing libraries available in Python. Libraries like pydub [19] and librosa [18] directly provide commonly used and well tested silence detection functions, which could give us more accurate silence detection.

Based on our system design, we decided to perform silence detection on the frontend for our webapp product in order to avoid the complexity of handling server responses that potentially contain transcriptions on overlapped audio portions. For the local version of our system that is purely Python-based, we choose to use librosa library for silence detection.

## B. *Joint LID-ASR Model vs. Multiplexed Architecture*

With the originally proposed "combined" model that used separate networks for LID and ASR tasks, this model's logits were supposed to be fused directly with the ASR logits and softmaxed for inference. The difficulty with this combination was the fact that the gradient flow was not optimal. When combined, the LID model's E (English) logit was added to all of the ASR model's English logits (A-Z, e.g.) and likewise with the M (mandarin) logit and B (blank) logit. Because of this, on any given prediction the LID model's efforts were punished. If the true output token was C, and the ASR model predicted C and the LID model predicted E, the ASR model's weights would be adjusted to favor C and disfavor all other logits given the current context–the correct behavior. Meanwhile, the LID model's weights would be updated with negative feedback on all of its logits since a majority of the E prediction negatively affected the final prediction. The learning signal here becomes very weak. This concept is illustrated in TABLE I. To avoid this, a second loss geared only towards the LID model could be devised using a classification scheme with Cross-Entropy loss (CEL). Fusing these losses would take a hyperparameter λ which would balance these effects so both models would learn. The subtle difficulty in this is that CEL requires a frame-wise labeling which is not immediately extractable from the audio (as this is the very task we are attempting to solve). The team had planned to use forced aligner software which would be capable of aligning a given sequence of labels to an audio vector, but the biggest issue was finding one capable of handling CS speech. All libraries we could find used speech models only meant for a single language. At this point, the team could have committed further effort into surmounting this issue using a more clever, complicated, and risky forced alignment procedure that would some combination of silence detection and multiplexed English and Mandarin forced aligners, but we decided we needed to prioritize a working but simpler end-to-end system.

TABLE I. COMPARATIVE MODEL PERFORMANCE

| *Architecture* | *CER** | *WER** |
|---|---|---|
| Single_v1 | .36 | 1.00 |
| Single_v2 | .23 | .64 |
| Combined_v1 | .22 | .59 |
| Combined_v2 | .22 | .58 |
| Muxed_v1 | .18 | - |

a. *Performance on the entire evaluation set.*

Comparison of attempted model architectures compared to key metrics. CER became the target objective once it was realized WER was not accurately representing the performance of the system on CS speech.

## VI. SYSTEM IMPLEMENTATION

### A. *Module 1: LID Model*

The LID model converts context representations (or features) created by the pretrained XLSR-53 feature extractor and encoder. It has an output of dimension three corresponding to the logits values for a Mandarin, English, and blank token. These logit values represent the relative probabilities that the model believes should be assigned each of the three output classes. Due to the time constraints the team was under by the time we decided to switch architectures, a simpler model architecture was used than was initially planned for the final stage of the model. A single BLSTM layer sandwiched between summarization and prediction FC layers comprises the final portion of the LID model. Though this grows the network and increases inference and training times, it provides the network with important contextual information about the sequence of audio over which it is run so it does not have to make "instantaneous" predictions about what language a single contextual embedding of the audio represents. The team hypothesized that recognizing switches between the languages would be an especially important part of accurate LID performance.

After updating the architecture, the LID model's job did not change, training it now became an entirely separate procedure from the overall model. Force alignment using standard libraries still couldn't be used, but labels could still be generated using a custom model to perform the forced alignment. Using an earlier single end-to-end CS model the team had developed during early experiments, the force alignment procedure could still be used. With this older model, the training data for the ASR model was preprocessed such that the new LID model was now fed solely frame-level LID labels and trained using CEL. After training, the best performing iteration of the new LID model was combined with the ASR models in a new multiplexed architecture.

### B. *Module 2: ASR Model*

There are two ASR models in our system: one for Mandarin and one for English. The English ASR model is taken from [21]. It is originally from the base model XLSR-53, a large model pretrained in 53 languages released by Facebook, and fine-tuned on English using the Common Voice corpus, which contains almost 3000 hours of labeled English speech. The English model achieves a WER of 19.06% and a CER of 7.69%. The Mandarin ASR model is trained by us using a Mandarin corpus consisting of over 70 hours of labeled speech. It achieves a CER of 24.09%.

### *C. Module 3: Audio Streaming*

This module needs to be capable of two primary tasks: identifying the silence gaps in the audio stream and dividing the stream into chunks at the silence gaps to send separately to the server for transcription.

To identify silence gaps on frontend, we utilized Web Audio API available for JavaScript. Specifically, we used the .getFloatTimeData method from AnalyzerNode class [22] to measure the amplitude of each sample frame of the audio stream. Through testing, we found the optimal amplitude threshold to be considered as a silence frame is below $10*10^{-6}$. The audio is sampled every 100ms. Any frame below this value would be considered as a silence frame. In order to avoid oversensitive chunking, we defined a silence gap to be at least 3 consecutive silence frames, which equals 300ms of silence.

To divide an ongoing recording audio stream into chunks by silence gaps to send to the server, we applied the mediaRecorder API available for JavaScript, which provides an audio recorder interface and supports periodic callbacks for newly recorded audio data. The period is a custom parameter. Using the API, every 500ms, a callback along with the new audio data recorded in the last 500ms. In the callback, we check if there is a silence gap within the last 300ms, by checking if the count of consecutive silence frames at the moment exceeds 3. If this callback is detected to happen in a silence gap, a transcription request to the server is made that sends the entire audio stream recorded so far along with a quantity *l*, that represents the length of audio data when the last transcription request was made. The mechanism is illustrated in Fig. 3. The server receiving the request will retrieve the most recent chunk that has not been transcribed from the audio data using *L*. The detailed mechanism to retrieve the most recent chunk will be discussed in the next section.

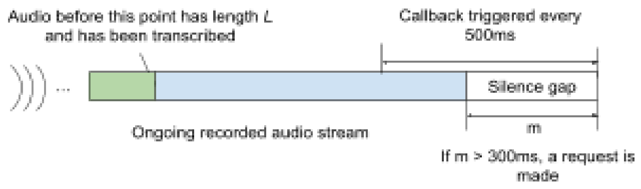

Audio before this point has length L and has been transcribed
Callback triggered every 500ms
Silence gap
Ongoing recorded audio stream
m
If m > 300ms, a request is made

Fig. 3. This figure illustrates how silence detection is used to control the trigger of a transcription request.

### *D. Module 4: Backend API*

This module needs to be capable of four primary tasks: identification of the untranscribed chunk from input audio buffer, .wav generation, transcription autocorrection, and interfacing with the language model. It uses the Django framework and routes http request bodies to a .wav generator function within views.py. The model is loaded and stored in a Python runtime instance when the server is first booted. When future transcription requests arrive, the model is ready by directly referring to the existing runtime instance.

When the server receives a transcription request, the FILES field of the request is converted into a .webm file which is finally converted to a .wav file using the ffmpeg toolkit. As requests are received, their sequence IDs and expected .wav local paths are also stored into a Python data structure. Then the variable *L* is retrieved from the request, which represents how long the audio stream was when the last transcription request was made. In other words, *L* means the length of the portion of the audio that has been transcribed. Therefore, the portion after length *L* is untranscribed and should be extracted and fed into our model.

The output transcriptions from our ASR models are run through the autocorrect library also provided by Python [15] to correct very small errors. The final transcriptions are packed in an http response body along with the appropriate sequence ID which is then sent to UI. Files are deleted from the server to prevent memory starvation. The main changes in the module from our original design were the addition of retrieval of the untranscribed portion and the auto correction on the output.

### *E. Module 5: Web Frontend*

Nothing changed in the design of this module, so its requirements did not change either as it was not considered a source of overall system latency. The in Module 3 for sending audio chunks to the cloud backend is also responsible for listening for and receiving requests from the server. The text is received with a sequence ID that is identical to one which was assigned to its corresponding audio chunk when it was first sent to the backend API by Module 3. Internally, the software is required to maintain a record of the sequence IDs it has received and which ones it has displayed so that it doesn't append text to the UI out of order. This is intended to deal with the weak ordering semantics of the transmission protocol from the backend. Using a strictly ordered protocol would unnecessarily decrease the system's throughput.

### *F. Module 6: Language Model*

The formulation of the language transcription pipeline was reworked several times as the team identified and tackled difficulties encountered during implementation. Rather than fusing the predictions of separate LID and ASR models mathematically, text predictions are now generated in Module 2 by either the English or Mandarin ASR models. The job of Module 6 is now to ensure the correct ordering of prediction chunks as they are returned from Module 2. When the LID model segments contextual representations into Mandarin and English chunks, they are assigned sequence numbers representing their order within the context of a single transcription request originally fed into this module. Using these ids, the prediction chunks are reordered and joined into a final text prediction corresponding to transcription request whose own sequence ID will be used in the system frontend to correctly order and display the transcription in real time. A further discussion of the weaknesses and difficulties of the original implementation plan for Module 6 are discussed further in section V. In contrast with the originally proposed implementation, this system was also exclusively trained on the ASCEND [20] dataset rather than the much larger and more comprehensive SEAME [8] dataset. This was due to serious resource limitations encountered when trying to preprocess and train the system on the SEAME dataset. The team's best hypothesis about why this occurred was due to the fact that individual training samples (an audio and transcription pair) proved extremely large. A single audio file provided could contain hundreds of individual samples, so loading a random collection of samples actually resulted in the loading of hundreds of MBs quickly into and out of GPU RAM which seemed to overwhelm the system during training. Careful processing of the dataset could have split the training samples further into smaller audio files and label pairs to allow for a better granularity for moving data into and out of GPU memory.

## VII. Test, Verification and Validation

### *A. Results for End-to-end Latency & Throughput*

In order for our system to operate smoothly and efficiently, we used two metrics to measure the performance of our system. The first is end-to-end latency, which measures the time it takes for the transcription of a segment of audio to the user interface. For this metric, our expectation is for the system to return the transcription of the first audio segment in no more than 1 second. We tested this metric by measuring the time difference between request to AWS instance and the returned result through multiple trials. The system, on average, took 800 milliseconds to respond, which achieved our initial expectation.

The second metric we used is throughput, which measures the processing power of the system. Ideally, we want our system to process audio faster than audio input. More specifically, the system needs to at least process 1 second of audio within 1 second. Throughput was measured by testing the average processing time between input and output on AWS instance. The final result we achieved was 0.7s/s, which means that it takes the system 0.7 seconds to process 1 second of audio input.

### *B. Results for Real-time Text Output*

We want our users to feel like the system is quick and responsive, not clunky and laggy. Thus, it is crucial for the system to process audio as quickly as possible while maintaining a reasonable accuracy. The results we achieved guarantees two things: the system responds quickly (end-to-end latency) and it does not clog up due to the amount of audio input (throughput). Consequently, the system allows a user to actively speak at a reasonable rate (of around 100 words per minute) while returning transcriptions in a comparable amount of time as typing.

### *C. Results for Character Error Rate*

Character error rate is the most crucial metric for our system, since it governs the accuracy of the transcription. As a reference, the best result we've found for a Mandarin-English code-switching ASR is released by Tencent [17]. Their model, exceeding in size, is trained with over 2000 hours of speech. Tencent's model is able to achieve a CER of 7.69%. In our case, we have about 80 hours of speech. Thus, for this metric, we set a goal of a CER lower than 25%. It is measured by testing our system on multiple corpuses and computing the average character error rate across all the samples in the dataset. For convenience, the metric is calculated using HuggingFace's cer tool [18]. On a code-switch dataset, the system achieved 22.94% CER. On a dataset containing English, Mandarin, and code-switching speech, it achieved 18.25% CER.

### *D. Results for Reasonable Output & Vocab Recognition*

We want our system to output reasonable results, which means that it should not output gibberish and should recognize vocabularies used in daily conversations in both English and Mandarin. For the most part, this was achieved. We saw that in most daily conversations the system is able to pickup most of the vocabulary. However, due to the fact that our system outputs at a character level, it sometimes outputs words that sound the same phonetically, but aren't actually English vocabs. This has both advantages and disadvantages. The advantage is that the system is not limited by vocabulary, meaning that even if it sees a word that is not in the dictionary, it is able to "sound it out". On the flip side, it will make mistakes on some of the common words where the sound is not exactly as it spells.

### *E. Results for Noise Tolerance*

The system should be resistant to some noise and still output reasonable transcriptions. This metric is measured by testing on a dataset with samples of low signal-to-noise ratio audio segments. Our goal is to maintain the CER under 30% when signal-to-noise ratio is lower than 20dB, which means that the output is still relatively intelligible. Unfortunately, we were unable to achieve the target, ending with only 31.28%.

### *F. Results for Robustness in Environment*

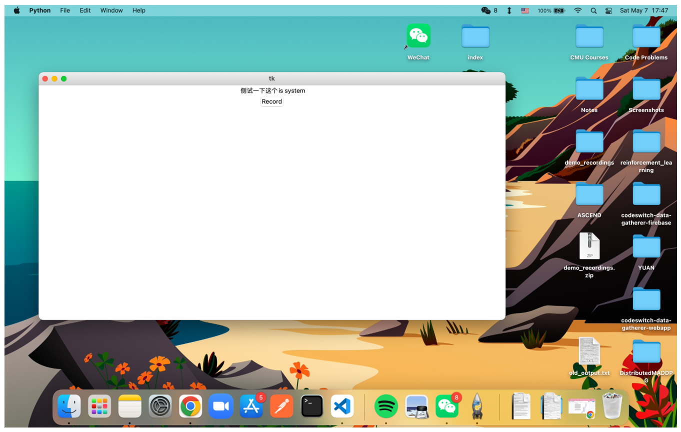
Our intent was for our system to be able to function in various environments, including ones that are noisy. However, our system was shown to be not effective in dealing with noise. The noise impacts two aspects of our system. First, the silence chunking mechanism cannot detect silence properly since it depends on the average amplitude of the input audio. Second, the ASR models cannot decipher the audio inputs properly due to the added noise. The combination of these results in the system that is both slow in terms of real-time output and inaccurate in terms of error rate.

### *G. Results for Supporting Environment*

One of our design requirements is for the system to support different environments and usages. We implemented two versions of the system: one running locally using Python and one allowing user to interact with Chrome browser. For testing, the system is run on Python local environment and through Chrome browser on both MacOS and Windows. The local Python script is able to run on any OS, provided that Python 3.8 is installed and a valid audio support is available. For the Chrome version, the system runs smoothly on both MacOS and Windows.

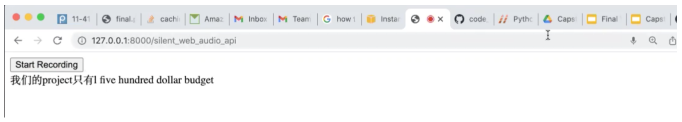
### H. *Results for Cross-device Support*

We want our system to be accessible from all devices. Since we support the application on Chrome, any laptop with a Chrome browser will be able to access the website (assuming the production is deployed). For the Python version, anyone can clone the GitHub repository, install the required dependencies, and use the system.



侧试一下这个is system

Fig. 4. This figure shows the Python version running locally on MacOS.



我们的project只有1 five hundred dollar budget

Fig. 5. This figure shows the webapp interface on a Chrome Browser.

### I. *Results for Max Data per Input*

One design requirement is that we want our system to be able to take in a long segment of audio. This ensures that the system will not break with long, uninterrupted speech. For this metric, we set the objective as the system should be able to transcribe a maximum of 1 minute of audio in a single input instance from a user. The testing is straightforward: testing if audio segments longer than 1 minute will return with reasonable outputs. In general, our system is able to handle 1 minute long audio. However, this is dependent on the memory of the GPU (if the models are run on GPU) or the size of the RAM (if the models are run on CPU). Any input size that exceeds the memory size will result in the system crashing. Nonetheless, when testing with Nvidia's RTX 3060 Ti GPU, the longest possible audio segment exceeds 90 seconds.

### J. *Results for Reasonable Speech Length*

The results from our test above ensures that the system has the capability to transcribe a useful quantity of speech in one shot. In scenarios such as conferences with long speeches, the system is able to process these kinds of audio without any interruptions.

## VIII. Project Management

### A. Schedule

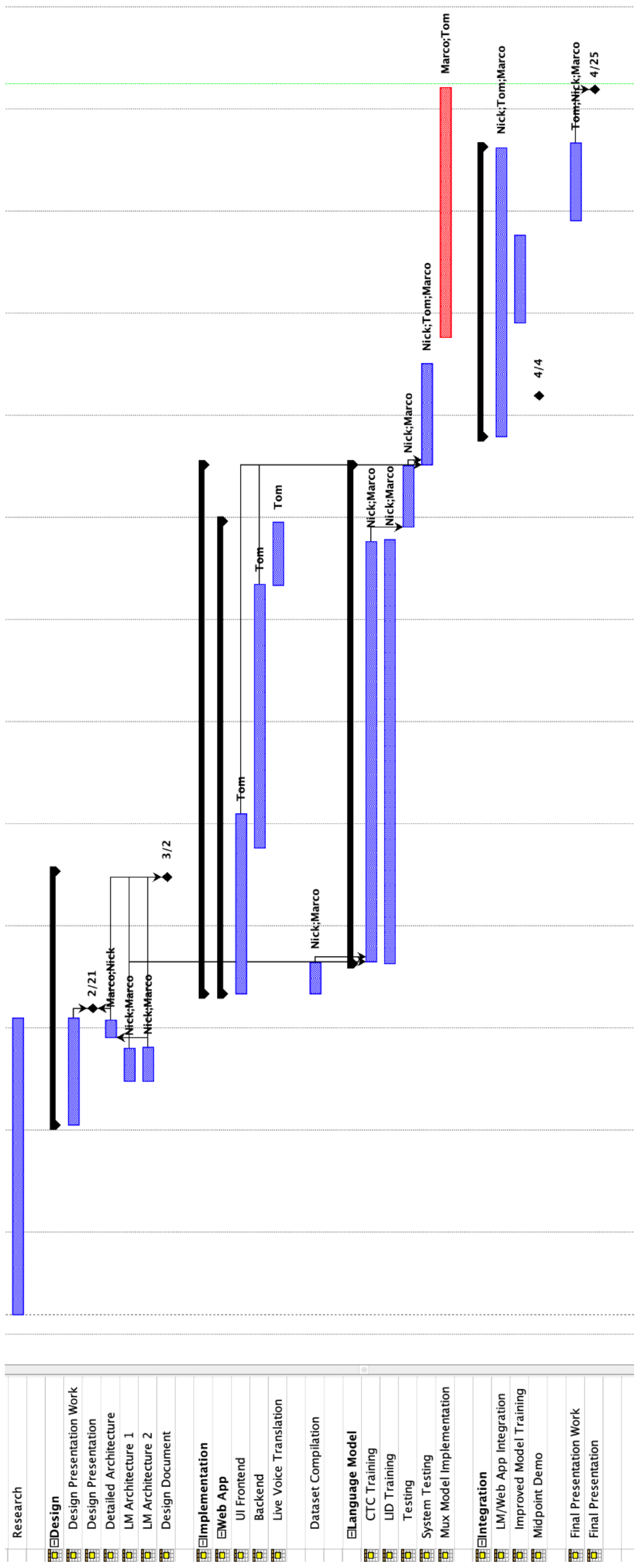


Fig. 6. This figure shows the schedule of our project progress during the entire semester.

The only change in our schedule is the addition (marked red) of the implementation of our multiplex architecture that uses our LID model to segment an audio buffer into chunks with the same language tag and feeds each chunk into the ASR model corresponding to their language.

### B. Team Member Responsibilities

The division of the team member responsibilities has not changed since our design review.

Nick focused on training and designing the LID model while Marco focused on implementing our Python-based local system and the multiplex architecture.

Tom was responsible for the implementation of all front-end and back-end web app modules. This included designing a front-end UI for accepting raw speech audio, audio detection on audio stream on the frontend, backend retrieval of untranscribed chunks from the audio data and auto correction of the transcription output of the ASR model.

The detailed team member responsibilities are labeled in Fig. 6.

### C. Bill of Materials and Budget

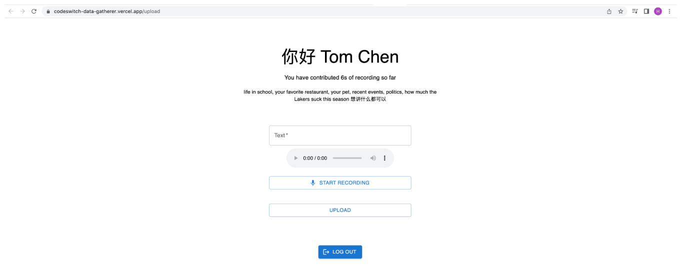
All of our system's components are purely digital. The budget is used exclusively for cloud hosting costs of server instances as well as compute instances needed for model inference and training as well as data storage. A detailed breakdown may be found in the Bill of Materials and Budget on page 15, Table II.

### D. AWS Usage

AWS instances were used in both training and the deployment of the system. 3 g4dn.xlarge instances totalling \$265.10 were used for training the ASR models. 1 p2.xlarge instance totalling \$64.80 was used for app testing and deployment. \$19.50 of gp2 storage was used to operate the instances. A special thanks to AWS for providing the credits and GPU usage.

### E. Risk Management

There are several areas of the project that posed risks to the successful implementation of the system. Though we have group members who are experienced with implementing and training deep models–even in the sequence-to-sequence context–model training is inherently uncertain in several aspects and especially with a limited time frame. Data is always a worry in speech recognition tasks since we require a large quantity of high-quality recordings and corresponding labels. Although we had access to the large SEAME Mandarin-English Code-Switching in South-East Asia dataset, there is still the potential that this dataset met not be large enough to train the model robustly on transcribing English and Mandarin words since it is a more difficult problem than LID and the dataset is somewhat biased towards Mandarin recordings and is made up of Singaporean and Malaysian speakers. If validation indicates that it is necessary, we have additional datasets with which we could augment SEAME such as the LibriSpeech ASR Corpus (Eng.) [9] or Aishell (Mand.) [10]. In addition, we built a data collection website (https://codeswitch-data-gatherer.vercel.app/upload) to crowdsource additional audio recording and transcript from our Mandarin-English bilingual friends and families.


codeswitch-data-gatherer.vercel.app/upload

你好 Tom Chen

You have contributed 6s of recording so far

life in school, your favorite restaurant, your pet, recent events, politics, how much the Lakers suck this season 想讲什么都可以

Fig. 7. This figure shows the interface of our data collection site.

Model accuracy issues could also arise as a function of time constraints or model-size. Since full training epochs are anticipated to take a substantial amount of time (a dozen hours perhaps), we may find that the model does not appear to be converging in performance fast enough. To tackle this we could launch a second training instance with a smaller vocabulary size, simplifying the problem and leading to a short time till model convergence. Whichever model results in better WER will be selected. We did not encounter a significant lack of time issue for model training with the help of powerful instances provided by AWS EC2.

Finally, achieving our throughput targets remains an ambitious and central goal in order for our system to be viable for real deployment in our messaging use-case. Major sources of latency will be the network jump to and from the web application backend (which are difficult to predict day to day) as well as inference time through the model. We achieved our desired end-to-end latency by reducing unnecessary server requests by performing silence detection on the frontend and only making transcription requests when there are significant silence gaps detected. And we chose to deploy our app using p2.xlarge instance type which has a GPU and enough memory to run our web app and LID and ASR models.

## IX. Ethical Issues

As this project does not concern a life or safety-critical product, the ethical issues it presents are more concerned with the subtler implications it would have to its users.

The first of these would be an issue of privacy. Instant messaging today is considered personal and private whether it technically could be or not (encrypted or not encrypted). As such, a user of our system would not expect their messages or audio to be stored anywhere other than within their message history or a personal profile account like iCloud. They would also expect that their identity be obfuscated enough to make it impossible for someone administering this service to identify them from stored metadata associated with their audio input or text transcription. A commercial implementation of this system could address these issues by using secure communication protocols between the frontend and backend, encrypt data in transit, and never associate location metadata with a transcription request (which the system does currently).

Another area of concern that is more unique to our application space is the usability of our system by a wide range of users. There are over a billion people in the world who speak either Mandarin or English or both and they represent a wide range of accents and colloquial vocabularies. To achieve the best user experience, a system like this would strive to consider all of these diversities. This presents a serious technical challenge to a system like this as the underlying models can only be as robust as their training sets are diverse. CS is the hardest aspect of our ASR problem but is also the one with the most limited number of datasets which themselves are biased since they usually only cover a population of individuals from a specific region (such as Taiwan, e.g.). With more time and by combining multiple datasets during training, we could work to overcome this challenge by pushing our model to generalize further. A commercial implementation could also use several versions of the backend LID and ASR models which have each been tuned to specific regions and could be chosen automatically based on a user's location or explicit preference. Still, this is an issue that persists with the current implementation. During testing, the system presented several specific examples in which the pronunciation of a word clearly confused the system resulting in either a cross-misprediction (English to Mandarin, e.g.) or a misprediction between English words.

Once the system was even observed to produce unintentionally explicit transcriptions whose pronunciations were somewhat similar to the input audio but were clearly erroneous. With additional training this behavior has not been observed again, but it is not outright prevented. The use of a fixed prediction vocabulary would help with this by preventing the production of any words outside of the specified vocabulary. In this sense, the use of a character-level prediction schema in this system was probably not optimal as it does allow for potentially explicit language to be produced in English transcriptions. Adding a post-processing step to remove these transcriptions could work but it would also be limited by the size of its own vocabulary to detect unwanted words or phrases.

## X. Related Work

Building multilingual ASR systems is an area of ongoing research in the machine learning community. The primary guidance and inspiration for this project was from a paper published in October, 2021 titled Mandarin-English Code-switching Speech Recognition with Self-supervised Speech Representation Models. This paper aimed to build a joint ASR/LID model for end-to-end prediction and training for audio to text transcription. As with our original architecture, one LID model was optimized on the classification task of labeling audio frames as English/Mandarin/blank. These logits were then combined with predictions from an ASR model with a large multilingual vocabulary that used standard characters for Mandarin and sub-word units for English (a complication and improvement missing in this system's formulation). It also solved the problem of generating frame-wise training labels for  the LID model using the Montreal Forced-Aligner (MFA). This was a subtle but technical feature of the implementation since the team was not able to find a MFA model intended for English and Mandarin use.

From what the team's been able to infer from Apple technical publications [6], the newer version of the implemented system follows a similar strategy of multiplexing between language models as their current voice to text feature available on the Messages application. The publications would imply that they use a very small version of a contextual LID model (using BLSTMs) to quickly predict the required model for transcription which is then selected and utilized for prediction in the cloud.

## XI. Summary

The responsiveness and useability of the final system fell above the team's expectations. Latency and throughput proved not to be an issue with both the quantitative and validation experiments of the system indicating a very reasonable and responsive transcription system. As the team had anticipated, the most challenging part of the system implementation was achieving the desired accuracy while balancing the requirement for a real-time experience. The biggest limits identified originated from the fact that the system was trained on a smaller than anticipated dataset with a very limited diversity of speakers and topics and therefore vocabulary. It struggled to generalize to pronunciations and contextual information beyond that which it had already seen. More complex neural architectures with larger BLSTM layers could also have improved performance. Within the overall design of the current system, the accuracy is also strictly limited by the accuracy of the LID model that is the model responsible for choosing whether English or Mandarin will be returned for a particular segment of audio. This was also the model with the poorest labels and the weakest procedure for creating them. Without ground-truth input for solving this classification problem and poor training characteristics associated with fusing its gradients with those returned by the ASR's CTC loss, the model performance suffered.

### *A. Future work*

There is some interest by members of the group to continue pushing the performance of the system by fixing and updating several aspects of the system. The first step would be to retry training using a version of the SEAME dataset that has been better processed for memory efficiency. The team would anticipate a nearly automatic boost in accuracy simply from better and more data. Retraining the model to predict on the word-level for English would also probably help by limiting the output of the model to a fixed vocabulary, resulting in more reasonable transcriptions without explicit and misspelled words. It's also hypothesized that this would make LID easier by not requiring that the correct language be identified at the granularity of the English letter.

Finally, a more involved reformulation of CTC loss is a possibility that the team has discussed. Although currently the use of a dual CTC and CEL loss seems to be well founded theoretically, the generation of LID labels remains. What future work could entail is implementing a system that is capable of generating labels on the fly by utilizing information from within the CTC loss function. As ground-truth labels are aligned with frame-level token predictions, a separate frame-wise label could be generated for LID of either M, E, or B (one of the three LID labels described previously). For example, the token 'c' would then serve as the label for the ASR model and also be transformed into the token E as a label for the LID model, removing the need for any forced alignment procedure. The details and complexities of this implementation would require significant extra thought, but the approach seems sound based on the team's experience this semester.

There is still room for improvement in the formulation of solutions to this problem and this project has helped to further characterize the areas in which future effort must be focused.

## Glossary of Acronyms

| | |
|---|---|
| ASR | Automatic Speech Recognition |
| BLSTM | Bidirectional long short-term memory |
| CER | Character Error Rate |
| WER | Word Error Rate |
| CS | Code-switching |
| CTC | Connectionist temporal classification |
| FC | Fully-connected |
| LID | Language Identification |
| SNR | Signal to noise ratio |
| SSL | Self-supervised Learning |
| UPL | User perceived latency |

## References

[1] Peng, Zilun, et al. "Shrinking Bigfoot: Reducing WAV2VEC 2.0 Footprint." *ArXiv.org*, 1 Apr. 2021, https://arxiv.org/abs/2103.15760.

[2] Aalto University. "Smartphone typing speeds catching up with keyboards." ScienceDaily. ScienceDaily, 2 October 2019. <www.sciencedaily.com/releases/2019/10/191002075925.htm>.

[3] Tseng, Liang-Hsuan, et al. "Mandarin-English Code-Switching Speech Recognition with Self-Supervised Speech Representation Models." *ArXiv.org*, 7 Oct. 2021, https://arxiv.org/abs/2110.03504.

[4] "Montreal Forced Aligner Documentation¶." *Montreal Forced Aligner Documentation - Montreal Forced Aligner 2.0.0 Documentation*, https://montreal-forced-aligner.readthedocs.io/en/latest/.

[5] "Hey Siri: An on-Device DNN-Powered Voice Trigger for Apple's Personal Assistant." *Apple Machine Learning Research*, https://machinelearning.apple.com/research/hey-siri.

[6] "Language Identification from Very Short Strings." *Apple Machine Learning Research*,

https://machinelearning.apple.com/research/language-identification-from-very-short-strings.

[7] "Jiwer" *PyPI*, https://pypi.org/project/jiwer/.

[8] Nanyang Technological University, and Universiti Sains Malaysia. Mandarin-English Code-Switching in South-East Asia LDC2015S04. Web Download. Philadelphia: Linguistic Data Consortium, 2015.

[9] Panzayotov, Vassil, et al. *LIBRISPEECH: An ASR Corpus Based on Public Domain Audio Books*. https://www.danielpovey.com/files/2015_icassp_librispeech.pdf.

[10] Fu, Yihui, et al. "AISHELL-4: An Open Source Dataset for Speech Enhancement, Separation, Recognition and Speaker Diarization in Conference Scenario." *ArXiv.org*, 10 Aug. 2021, https://arxiv.org/abs/2104.03603.

[11] Velardo, Valerio, Deploying the Speech Recognition System with uWSGI, *youtube.com*, 13 Apr, 2020, https://www.youtube.com/watch?v=7vWuoci8nUk

[12] musikalkemist, Deep-Learning-Audio-Application-From-Design-to-Deployment, *github.com*, 13 Apr 2020, https://github.com/musikalkemist/Deep-Learning-Audio-Application-From-Design-to-Deployment/tree/master/6-%20Deploying%20the%20Speech%20Recognition%20System%20with%20uWSGI/code

[13] Conneau, Alexis, et al. "Unsupervised Cross-Lingual Representation Learning for Speech Recognition." *ArXiv.org*, 15 Dec. 2020, https://arxiv.org/abs/2006.13979.

[14] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 48–53, Minneapolis, Minnesota. Association for Computational Linguistics.

[15] filyp, autocorrect, *github.com*, 4 Dec 2021, https://github.com/filyp/autocorrect

[16] McFee, Brian, Colin Raffel, Dawen Liang, Daniel PW Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. "librosa: Audio and music signal analysis in python." In Proceedings of the 14th python in science conference, pp. 18-25. 2015.

[17] huggingface, datasets, *github.com*, 28 Mar 2022, https://github.com/huggingface/datasets/tree/master/metrics/cer

[18] Shan, Changhao, et al. "Investigating End-to-End Speech Recognition for Mandarin-English Code-Switching." *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2019, https://doi.org/10.1109/icassp.2019.8682850.

[19] jiaaro, pydub, *pypi.org*, 09 Mar 2021, https://pypi.org/project/pydub/

[20] Lovenia, Holy, et al. "Ascend: A Spontaneous Chinese-English Dataset for Code-Switching in Multi-Turn Conversation." *ArXiv.org*, 3 May 2022, https://arxiv.org/abs/2112.06223.

[21] Grosman, Jonatas, XLSR Wav2Vec2 English by Jonatas Grosman, *Hugging Face*, 13 Apr 2021, https://huggingface.co/jonatasgrosman/wav2vec2-large-xlsr-53-english

[22] *AnalyserNode.getFloatTimeDomainData() - web apis: MDN*. Web APIs | MDN. (n.d.). Retrieved May 7, 2022, from https://developer.mozilla.org/en-US/docs/Web/API/AnalyserNode/getFloatTimeDomainData

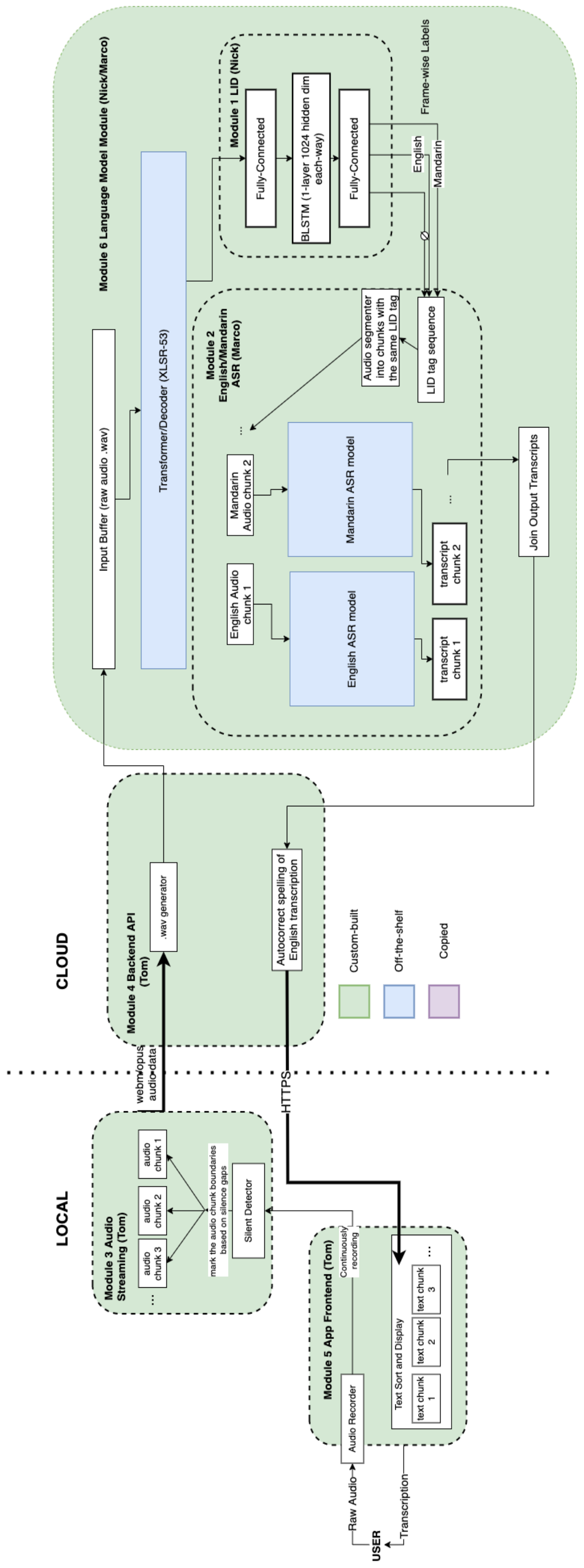


Fig. 8. The current system architecture.

**Table II. Bill of Materials**

Table I. Bill of Materials

| **Description** | **Model #** | **Manufacturer** | **Quantity** | **Cost @** | **Total** |
|---|---|---|---|---|---|
| AWS GPU Instance for LID Module, CTC Module, each instance estimated 7 full days of runtime | g4dn.xlarge | AWS | 3 | $.526/hr | $265.10 |
| AWS GPU Instance for app testing and deployment, each instance estimated 3 full days of runtime | p2.xlarge | AWS | 1 | $.90/hr | $64.80 |
| AWS EBS 1.5 Months SSD Storage | gp2 | AWS | 1 | $.1/GB-Month | $19.50 |
| | | | | **Grand Total** | **$349.40** |