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.**

***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 copri, better computational resources, and advancements in model architecture, performance metrics of state of the art ASR systems have practically saturated–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 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 special native neural hardware. Though the recognition of dozens of languages are individually supported on these devices, situations in which speakers may switch between languages mid-sentence are not specifically handled.

For conversation and messaging use-cases between multilingual speakers, the accurate handling of code-switching is a must to allow 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 has motivated our design decisions surrounding its architecture, size, latency, and throughput. The less-formal nature of these communications allows us to limit the vocabulary which the model will need to learn. It is also a context in which we expect the real-world prevalence of CS to be high.

The goal of this implementation is ultimately to emulate the experience of current monolingual ASR systems which provide the kind of easy-of-use, fast response times, and accuracy which make them competitive with a traditional instant messaging experience.

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 its 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 plan to overcome these deficiencies by using a jointly-trained LID/ASR architecture which is able to capture multi-lingual contextual information on both the LID and ASR portions of the transcription problem. A tunable feature extractor and transformer model front-end which has been pre-trained on multilingual speech will help to provide usable language and contextual information for the down-stream modules of the model as a whole.

II. Use-Case Requirements

An implementation capable of addressing an instant messaging use-case must meet requirements for size, speed, and accuracy first and foremost. The trained model without the feature extractor/encoder head must be no larger than 20MB prior to any compression attempts. This corresponds roughly with the size of the Apple's latest mobile chip's L2 system cache (A12). UPL must be no more than 3 seconds, similar to an iPhone's current response time for the first word returned from an ASR transcription. We expect the system's accuracy (measured with WER) to be directly related to the input audio's SNR so we formulate this as a bucketed requirement. For any audio with a SNR greater than 20dB (30dB is considered clean speech), the WER should be no greater than 25% while audio with a SNR down to 5dB should be no greater than 30%. A reasonably large vocabulary size of

5000 words will accommodate most day-to-day conversations we can expect in instant messaging. Though we do not expect most users to be submitting messages longer than 10 or 20 seconds, we will require that the system be capable of handling single messages which correspond with up to a minute of audio. Since the average typing speed on a phone is about 50 words per minute [2], the system should provide a comparable transcription speed. This results in an end-to-end throughput requirement of 1.2 seconds/word.

The user interface to the system will need to abstract away any depence the implementation has on file formats, converting raw audio into a standard file format for model consumption. The system must also not suffer any performance loss due to concurrent users of the system. This requires that the cloud-based application allow for theoretically arbitrary scalability with independent transcription requests that are submitted simultaneously.

III. Architecture and/or Principle of Operation

Our architecture consists of a web-based user-interface (UI), cloud-hosted application backend and language model. Fig. 1 page 11 shows this organization. This architecture reflects the fact that we anticipate the need for considerable computational resources for translation. Though we target a mobile-centric instant messaging use-case, we judged that the development effort needed to implement an mobile application as well as compress our language model for partial deployment on a mobile device would exceed the time available for this project.

The system's architecture is broken into six software functional modules. Two modules run on a user's device locally while the rest are hosted in the cloud.

Module 5 is the webapp UI frontend, which includes an audio recorder supporting start/stop recording through button clicks and a text output displayer supporting real-time text propagation. A user should click to start audio recording, start speaking and expect text transcript to real-time propopate in the text output displayer. The text displayer accumulates the text outputs within an entire start-to-stop recording session. As new text transcription comes back from the server it will be appended to the already displayed text. When the user stops recording and restarts recording, the transcription from the previous recording session will be kept on the web page, and the transcription for the new session will be displayed in a new text area mimicking a new message.

Module 3 is the audio stream sender that runs within the user's browser, which chunks an incoming recorded audio stream into 2-second audios (which should contain more than one full word) to send to the server through http requests. Besides chunking the audios, this module also adds metadata on each request, which includes audio properties (sampling rate, number of channels) and a sequence ID that notifies the server of the order of these audio chunks. The sequence IDs can then be referenced to manage the order of text transcription when multiple server responses come back, possibly in different order from how the audio chunks were originally ordered.

Module 5 is the backend APIs of our webapp. The main APIs of our app include wav file generator and the loader of our recognition model. The wav generator receives http requests that contain audio data, which is encoded in webm format, and generates .wav files (.wav files will be inputs for our CTC model and LID model). The model runner module creates a singleton instance of model runner class that loads the same model (Module 6) on initialization. The model runner class keeps track of a queue of .wav file paths and incrementally processes them.

Module 6 encapsulates an instance of the system's transcription module. For scalability and workload balancing, independent instances of this module may be launched independently. Inside, this module contains an input buffer for transcription requests received from the web applications Backend API (Module 4). Each request is treated as logically independent of others. As requests are fulfilled, their transcriptions populate an output buffer which returns transcription results to the requesting clients. Transcriptions are differentiated by unique request IDs which identify the clients which made the original request. The same request IDs are associated with each request which populates the input buffer. Between these buffers lies the audio-to-text transcription pipeline. Raw audio files are fed into the feature extractor/decoder head which returns frame-level contextual representations of the audio. Representations then flow into both the LID module as well as the ASR module.

Module 1 is responsible for identifying the language of audio frames while Module 2 is responsible for transforming context representations into the words of the system's supported vocabulary. The outputs of these modules are logits repreeenting the probability of a given language or given word for modules 1 and 2 respectively. For final prediction, these logits are combined before passing into a CTC component. This final component will be responsible for completing a "soft-alignment" of audio frames into predicted words, thus there is no need for the system to attempt to segment the audio input into words segments during an early step.

During training, modules 1 and 2 will be evaluated using separate loss criterion which will be interpolated into a single loss value for combined end-to-end training. Gradients will be allowed to flow all the way from this interpolated loss through the pre-trained extractor/decoder head which allows for the head's fine-tuning to our specific use-case and its languages. This procedure is central to the formulation of our solution as it allows the language model to adapt specifically towards the case of code-switching as it should result in a model that is more smoothly optimized for recognizing language switches between frames as well as contextual information from

multiple frames. Joint training will cause the feedback from mispredictions to assign blame to the submodules in a way that is proportional to each submodule's contribution to the misprediction. It will also allow both submodules to provide to the feature extractor/decoder about what information is relevant and what is not.

IV. Design Requirements

To achieve the system's use-case requirements they must be further partitioned into specifications for modules and submodules of our architecture. Certain requirements naturally envelope multiple modules. Combining these parent requirements with information about inter-module dependency informs the specifications for the sub-modules themselves.

Model size is dependent on the sizes of both Module 1 (LID) as well as Module 2 (ASR). The overall budget of 20MB will be roughly allocated such that Module 1 receives 9.5MB while Module 2 receives 11.5MB to account for the larger BLSTM component of Module 2.

The processing time between the moment after a 2-second audio chunk is recorded and the transcription for this chunk to appear on the webapp frontend needs to be within 3 seconds to achieve the real-time requirement of our app. This requires high performance design in Module 3 (audio stream sender), Module 4 (backend APIs) and Module 6 (language model) so the critical path from Module 5 → Module 3 → Module 4 → Module 6 → Module 5 are within 3 seconds.

A. Module 1&2: LID & ASR Model

The requirements for these modules are almost identical since they are considered to be running in parallel. As prescribed earlier, Module 1 must be no more than 9.5MB in size while Module 2 must be no more than 11.5MB. Both modules must be able to achieve the system's overall throughput requirement of 1.2 seconds per word. Since the maximum amount of data allowed for a single transcription request is more a function of other modules, it is not a requirement which applies to Module 1 or 2. Both modules should achieve the required noise tolerance however which will be tightly linked with their individual accuracy requirements. Module 1 is measured by a frame-by-frame classification error rate which is the percentage of frames it misses compared with the ground truth. This should be no more than 10% at greater than 20dB (clean audio). Module 2 is measured by WER which is a function of the number of changes needed to change a predicted transcription to the ground truth. This should be no more than 16%. These are roughly derived from the use-case requirement of a WER of 25%. We'd expect about a 75% percent chance of a successful transcription per word which requires both models to be correct thus $0.9 \times .84 = .756$. A similar calculator can be performed for the modules' performance in noisy conditions. A classification error rate of no more than 12% for Module 1 will require that Module 2 have a WER of no more than 20.3% which is similarly derived from $.88 \times .797 = .701$.

Latency requirements per module are difficult to estimate but since they are derived from the budget allocated to Module 6 (1.5s) they can be roughly derived based on the relative model size of the components within Module 6. Allowing for throughput budget to be allocated both the extractor/transformer head as well as the logit scaling and CTC algorithm, both Modules 1 and 2 are allocated .4 seconds of throughput time.

B. Module 3: Audio Streaming

Module 3 chunks the ongoing recording audio stream into 2-second chunks to the server. The latency between a user finishing speaking for a 2-second interval and the corresponding http request being received on server needs to be within 0.2 seconds so that there is enough room for backend processing and server response to come back to the frontend. We cannot control the network latency, so a user under a flaky network may experience longer processing time; however, each request only contains 2-second audio data, each request is not a heavy load, so the bandwidth will not impose significant limitations on our app performance. The text transcription should be rendered on the web page in the same order as the audio recording chunks.

C. Module 4: Backend API

Module 4, the backend API group should include a .wav generator API and a model runner API. The .wav generator should be able to write audio data received at the server end into .wav files which will be the inputs to our ASR model. The .wav generation time should be under 100ms for a 2-second audio so that we leave enough time for the model to run on the audio. The model runner API should be capable of processing audios in parallel to optimize our deployment server's utilization and our app performance.

D. Module 5: Web Frontend

Module 5 is the frontend interface of our web app. It should be accessible through a http:// url on a laptop Chrome browser that is relatively recent (Chrome version after 1/22/2022). The interface has an audio recorder for our users to click to start/stop recording using their laptop microphones. As they are recording, the text transcription of their speech should be generated in a text area on the web page as they are recording. The transcription will be a mix of English and Chinese texts that match the code-switch pattern of the users' speech. The transcription should display words or phrases in the same order of the users' speech. The delay between a word being spoken and its transcription being displayed should be under 3 seconds.

E. Module 6:Language Model

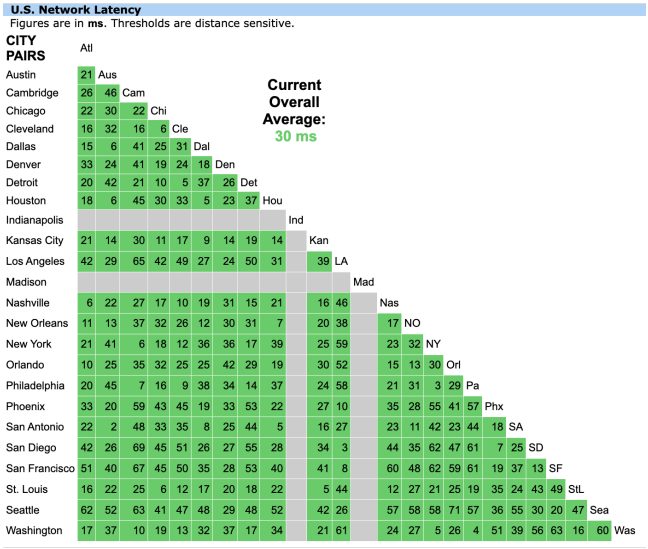
The throughput and accuracy requirements for this module are identical to the design-level requirements for the system

since the language model fully encompasses the modules which will affect the accuracy metric and since the throughput requirement remains constant throughout the system. This applies to the modifications in those requirements concerning noise tolerance. Its accuracy metric will be solely WER since its output is the combination of the two submodules 1 and 2. As described earlier, its total size must be no greater than 20MB. The latency requirement for this module is budgeted at 1.5s, exactly half of the design-level requirement since the calculations performed within will likely be the greatest single contributor of UPL in the entire system. Modules 1 and 2 are allocated .4s of this time so that the remaining time (1.1s) may be allocated between the a forward pass of the XLSR-53 [13] model as well as the logit scaling calculation and CTC algorithm.

V. Design Trade Studies

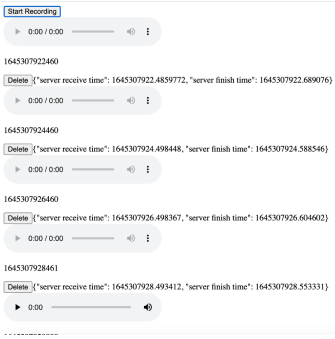
A. *Module 3: Audio Streaming*

To achieve a 2-second delay from when a user finishes speaking a word to the word's transcription appearing on the web page, the amount of time taken for audio stream transmission from the frontend to the server needs to be as low as possible to reserve more time for model runtime. The transmission delay is highly dependent on the network latency between user device location and our deployment server. According to our research, the round trip latency between northern Virginia (our deployment server location) and Pittsburgh, PA is 14ms, but the latency across states in the US can be as long as 60ms. So ideally, the users within the US will experience under 60ms (3% of the 2 second total time) round trip network latency for each request. Each 2-second audio data in webm format is 10KB whereas audio in wav format is 50KB for the same length. Therefore, we choose to send audios from the frontend to the server using webm format, so given a 1MBps network (we choose to use somewhat pessimistic network assumption), the small audio data chunk will still be trivial compared to the bandwidth.



B. *Module 4: Backend API*

We did not find specific data about the optimal conversion rate from webm to wav and disk write rate into a wav file, so we performed multiple runs of tests to use ffmpeg to convert webm data to wav and write into a .wav file and measured the time taken. The time was measured between the server receiving a request embodying audio data and the server finishing generating a .wav file. The average time was 100ms.



We designed our model to load at server start, so we do not expect the model loading to take any portion of the 2-second per audio processing time limit. We will use the python multiprocessing package to start multiple processes to process audio data coming from different clients concurrently. So our app load tolerance can be scalable by adjusting the GPU core number of our deployment server.

C. *Module 5: Web Frontend*

To make the audio recorder, for implementation efficiency, we chose to use the standard JavaScript navigator class to trigger user device microphone permission on the browser and display the default audio recording box interface at the webpage.

To make the text transcription displayer, we plan to use HTML textareas, of which their innertext can be easily modified. This will support our initial plan for text propagation is to append text transcription to the already

displaying transcription to create the real-time text propagation experience.

JavaScript that is runnable on most browsers today will enable us to implement the sequence ID management logic.

D. Module 6: Language Model

Model architecture was the first choice to be made in the language model. Since we are using an XLSR-53 head (which is already a very large model of 300M parameters), we considered there to be a significant amount of frame-level contextual information already at our disposal. A choice between using a Recurrent Neural Network (RNN) structure and a simple fully-connected layer for the LID and ASR modules had to be made. Fully connected layers would be much smaller and introduce far fewer complications in terms of inference since they only require a single time step to be passed through in order to make predictions. This could have been sufficient for the LID task since it is much simpler but likely would not have been adequate for the ASR task. This is because ASR generally relies on contextual information (that is information more than just the current frame) to make an accurate prediction of the word to be transcribed. Audio frames are unlikely to correspond to an entire word, but more likely a part of one. A memory based layer like BLSTMs allows for contextual information to be considered both from the future of the sequence as well as the past which allows for not just better combinations of multiple frames to create a word, but also to decide precisely what word that should be given the previous and future words predicted. Since accuracy must ultimately be the primary requirement, the choice of using a BLSTM architecture seemed logical though we may consider making comparisons between a single fully-connected layer and BLSTM architecture just for LID task if time permits.

Achieving a reasonable model size, throughput, and latency all rely partially on the components used in the language model. Currently, the largest component by far is the XLSR-53 model which contains 300M parameters. Attempting to distill or compress this model to fewer parameters will likely result in greater throughput and latency times as well as increase the feasibility of deploying the overall architecture in a mobile application. Research suggests that shrinking this model by a factor of 3.6x [1] is possible with only a .1% degradation in WER. This would bring the model below 300M parameters, significantly increasing space efficiency and inference time. This is achieved through quantization of the model. Though we do not consider addition to the design a requirement for our minimum viable product, it is an improvement we are strongly considering for additional iterations of our system to drastically increase our chances of meeting our system requirements.

Vocabulary size will affect our model's size requirements, inference time, and practical useability. We currently plan on a combined vocabulary size of 5000 tokens between Mandarin and English. We can roughly approximate the effects of this size on our model's usability by beginning with Heap's law which can be used in reverse to approximate the number of tokens we could encounter before our vocabulary would need to grow bigger. Given a vocabulary of size 5000 the law would predict $\frac{5000}{50}^{2.5} = 100000\ tokens$ using standard constants of the law. This is of course, a very rough estimate of the number of tokens we would need to encounter before we encountered one that is not within our vocabulary. The probability that this would prove to be false is, of course, still significant but it allows us to begin to characterize the effect that our vocabulary size would have on the usability of our system. Growing this vocabulary would only be a boon to the system in terms of usefulness but we feel that this is a reasonable size for the scale of this project. It should be noted that these laws describe fundamental aspects of language distribution and are not specific to any one language.

VI. System Implementation

A. Module 1: LID Model

The LID model will convert context representations from the XLSR-53 feature extractor/transformer head into logits for three categories: English, Mandarin, and blank. Blank will indicate that the model believes there is no output and is an essential part of using the CTC algorithm for final transcription. The architecture itself is simple with a fully-connected layer used to summarize the outputs of the head. This sequence of summary vectors is then fed into a single layer BLSTM module whose hidden outputs (size 1024) are then passed back through a fully connected layer which performs the final classification into English, Mandarin, and Blank logits. When trained on its own (without back-propagating through to the head module or jointly training with the ASR module), this model will use a Cross-entropy loss criterion. To perform this loss calculation, Montreal Forced Aligner [4] will be used to segment between words in the audio samples for calculating the loss of a forward pass. It will be implemented using the TensorFlow library.

B. Module 2: ASR Model

The ASR module uses a very similar architecture to Module 1. It is deeper since we use two-layer BLSTM components. It uses a similar fully-connected layer for summarizing the output of the XLSR-53 contexts as well as a fully-connected layer for classifying the current frame into one of the tokens of the vocabulary. As a consequence, the size of the output fully-connected layer is the size of the vocabulary (including a blank token) as opposed to 3 for Module 1. It will also be implemented using the TensorFlow library. Since Module 2 and Module 1 make logically different predictions, they can be initially trained separately along with the XLSR-53 head. Once training or implementation of the opposing module is

complete, they can be combined to proceed with joint training along with the head component.

C. Module 3: Audio Streaming

The Django framework supports JavaScript logic that sends http requests and waits to receive responses to the backend. Each http request embodies a unique sequence ID before being sent. The JavaScript also maintains an ordered array of these sequence IDs so that in Module 5, this array can be referenced as a ground truth of the order in which the transcriptions should be displayed. The sequence ID of a request marks the request-corresponding chunk's order compared to other chunks. The server receiving a request keeps the sequence ID for this request and includes the same ID along with model output in the response to the request. In this way, the text displayer in Module 5 can refer to the sequence ID of responses to manage the displaying order of response texts.

D. Module 4: Backend API

Under the Django framework, the .wav generator logic will be contained in views.py. The request url routing will be set so that http request bodies passed from the frontend can be received by the views.py .wav generator function. When receiving a http request, the .wav generator logic reads the FILES field of the incoming request to get the webm audio data and writes them into a .webm file. Then the API will use ffmpeg toolkit to convert the .webm file into a .wav file.

To implement a model runner, a python script will be triggered to run by a bash script at server boot time. The script creates a singleton python instance of model runner class. The model runner instance will maintain a queue as its attribute that contains the list of unprocessed .wav file paths (along with their corresponding sequence IDs. Using multiprocessing.pool supported in python, the instance can run the model on multiple .wav files concurrently in separate processes. For each processed .wav file, the model output (the text transcription) will be packed into a http response body along with the .wav file's sequence ID. The response will be sent back to the frontend. The processed .wav files will be then deleted from the server to optimize the server disk memory efficiency.

E. Module 5: Web Frontend

The JavaScript logic that is responsible for Module 3 audio chunking and making requests will also be responsible for waiting to receive the responses from the server. When a request's response comes back, the JavaScript will read the transcription text and sequence ID from the response body. The sequence ID will be searched in the array of IDs maintained in the script. If the text transcription of all IDs before this ID have all been displayed, then the transcription of this sequence ID will be also appended to the already displayed text and thereby achieving the effect of real-time text transcription. If the server response of some ID before the current response's sequence ID is still not received, we will hold from the current sequence ID's transcription and keep the transcription of the ID until it is ready to be displayed.

F. Module 6: Language Model

The language model as a whole encompasses both Modules 1 and 2 as well as other pre-built and custom components. This module represents a single instance of the language model which may be replicated as needed. Transcription requests will be distributed to instances of the language model by Module 4. Should multiple requests be distributed to a single instance, an input buffer will store and process them in a FIFO input buffer as well as return their corresponding transcriptions using a FIFO output buffer. Upon system startup, Module 4 (which will be the initial trigger for the entire system startup since we will consider it to be the primary server of our web application) will be responsible for prompting language model instances to first pre-load models from SSD and then transfer them into GPU memory. This will include loading the extractor/transformer, the LID module, and the ASR module.

The feature extractor/transformer head will be the pre-trained multilingual model XLSR-53 from the fairseq [14] library which contains 300M parameters. It has been trained using a self-supervised learning technique to extract and encode features from 53 different languages. For fine-tuned training, it is capable of being switched from a SSL loss to whatever loss criterion is used by the modules it is attached to downstream. Once the LID and ASR modules are complete in their implementation and preliminary training, they will be logically grouped into a single object. This object will then be further logically combined into a Wav2Vec2Processor object which can then directly combine with the CTC tokenizer component described later in this section.

To combine the output logits of Module 1 and 2, one of whose outputs is size 3 while the other is the size of the vocabulary, respectively, the logits must be combined (referred to as scaling in the diagram). This is done by iterating over the values returned for each token by Module 2 and adding to that value each the value returned by Module 1 for the given token's language (e.g. the token "the" will have the logit value for English added to it). Then a softmax is performed. When training, these logits are passed into the CTC loss component. Logits from Module 1 are also passed into a Categorical Cross-entropy loss. These two losses are linearly interpolated with a hyper-parameter λ to create a single final loss which is used for back-propagation.

The CTC module implementation known as Wav2Vec2CTCTokenizer is also provided by the same fairseq library from Facebook AI which provides the XLSR-53 head. This provides a soft-alignment scheme which is capable of translating a sequence of distributions over the vocabulary into a likely sequence of output tokens without the need for hard alignment between audio frames and ground-truth labels. It is

also capable of calculating a loss value which is related to the probability of calculating the ground truth sequence given the current estimates of token probabilities.

All of Module 6 as well as its submodules will run on a GPU-enabled AWS EC2 instance. Any non-tensor operations will be performed on the instance's standard CPU. The model itself will sit on GPU memory with input audio sequences and output logits having to be transferred into and out of GPU memory which we anticipate will be a source of additional latency.

VII. Test, Verification and Validation

The testing and verification process for our system will be divided into three stages: unit, integration, and system testing and verification. Unit testing will focus on measuring whether individual modules meet their assigned requirements as described later in this section. These tests will need to be completed before integration testing begins. As each model's testing is completed, modules will be integrated with each other one by one with further testing being performed for those requirements which may be affected by the integration (e.g. an increase in latency once language modules 1, 2, and 6 are fully integrated. System-level testing will focus on measurements for meeting high-level design requirements. Unit tests will be derived from the procedures of system tests unless otherwise specified. Each test is identified with an identifier (e.g. T0).

A. System Tests

An UPL test (T0) will measure the time required for the system to produce the first transcription response when it first begins receiving audio. A software timer will be used to make the measurement, starting when the system first begins accepting audio and terminating when the first word of the transcription has been displayed to the UI. Three clips, one English, one Mandarin, and one mixed will be utilized. Throughput (T1) will be similarly measured using a software timer and three one-minute audio clips. The timer shall begin at the moment we begin accepting audio and terminate once the last transcription has been returned to the UI. One clip will contain only English, one clip Mandarin, and one clip a mixture. Five trials will be conducted for each clip with the resulting throughput measurement calculated as the average of all trials. The same procedure shall be used for the UPL test as well.

Model accuracy (T2) will be tested using the WER metric from the JiWER open-source library [7]. This metric shall be calculated by passing audio samples which are already withheld in the SEAME [8] testing directory through the model. There are separate test sets biased towards English and Mandarin. Though our final WER value shall be an average of average WER returned from running against these two sets, we'd expect both the average WER from both sets individually to meet our design WER requirement. Samples used in this test shall be considered "clean" audio (which we'll consider to be >20dB SNR). A separate test (T3) to measure the robustness of our system to audio noise will be conducted using a subset of both the English and Mandarin biased tests sets. Three audio files from each shall be selected and augmented with gaussian noise to produce files with 5dB, 10dB, 15dB SNR. The standard WER test will be repeated at each SNR with the final noisy WER returned as the average across trials.

System portability (T4) will be tested by running a single trial of T0 and T2 on the Chrome browser on Linux (Ubuntu 18.04), Windows 10, and MacOS (Monterey) systems. The system shall pass this test if UPL and WER are no more than 5% different than the values found by running T0 and T2.

All of these tests directly measure the metrics prescribed in the design requirements. Meeting these tests will directly indicate meeting the design requirements which are themselves derived directly from our use-case requirements.

B. Module 1 Unit Testing

Testing Module 1 will involve very similar tests to the system tests since its requirements are either identical or directly derived from them. Module 1 will need modified versions of T0, T1, T2, and T3 performed. The only changes necessary would be to update the T2 and T3 metrics to use classification error rate as opposed to WER. The passing requirement for T0 will also be different since Module 1's throughput budget is different from that of the design-level requirement.

C. Module 2 Unit Testing

Like Module 1, Module 2 will require tests T0, T1, T2, and T3 to be performed. It will have the same passing criteria for T0 as Module 1 while all other aspects of the procedure and metrics will be identical to those of the system level tests.

D. Module 3 Unit Testing

Module 3 Audio Streaming will be tested on two aspects: latency and sequence IDs. The latency between a user finishing speaking a 2-second interval and the corresponding http request is received on the server will be tested and compared with our network network approximation (60ms). 100 requests will be made to verify if all of the requests are received on the server and correctly processed into .wav files. We will also verify that unique sequence IDs are indeed generated for each request. The procedure to measure one-way latency will be as follows: keep recording audio for 1 minute, log the physical timestamp right before a request is sent out on the Javascript side and the physical timestamp when the server receives that request; the difference between the 2 timestamps is the one-way latency if assuming the server and the client share global physical time.

E. Module 4 Unit Testing

Module 4 Backend APIs will be tested on audio quality of output .wav files from the wav generator API, the latency to

convert webm data and generate .wav files and the model preloading. We will compare the audio quality of the original audio recording and .wav file generated on the server.

The testing procedure for .wav audio quality is as follows: keep recording audio for 1 minute, which should generate ~30 .wav files each with 2-second audio on the server. Use matlab to invert the audios in the .wav files and add each inverted audio to the original audio. The total difference should be < 1% of the original audio volume.

The testing procedure for .wav conversion and generation will be to log the physical timestamp right after receiving a request and the timestamp right after finishing .wav file generation and get their difference.

For the model loading test, we will print a message for every model loading action. We will restart the server to see if the model loading is triggered. Then we will send a 1-minute audio from our web app to the server to verify if the server does not have to load the model again and is ready to directly use the preloaded model for processing.

F. Module 5 Unit Testing

Module 5 Web Frontend will be tested on the "real-timeness" and order correctness of text transcription display. We will measure the time from speaking a word into our recorder to the word transcript appearing on frontend by continuously recording an audio for 1 minute using our app; server returns a log for every incoming request; we logs the http responses to the browser console and measure the average number of http responses logged within 1 minute. We will check the correctness of text outputs by comparing the displayed transcription to our speech content and measure word error rate of the transcription. The word error rate also accounts for the ordering difference of words, so incorrect transcription word ordering will be detected with a high word error rate.

G. Module 6 Unit Testing

Unit tests for Module 6 can be thought of as integration tests for Modules 1 and 2 since they are both encompassed within this module. Module 6 will also require tests T0, T1, T2, and T3 to be performed on it with no variation from the system tests except for the passing criterion for T0 since its latency budget is not the same as the entire system. Otherwise, we expect the results of these tests to be identical to those of the system-level tests. If they are not, we can be sure there is some confounding factor elsewhere in the system leading to worse performance.

H. System Validation

Our use-case involves direct user interaction, thus our validation process will rely on random users to interact with our system and provide feedback. We'll aim to have a group of at least 5 participants interact with the system through the UI. We'll ask them to try giving the system different phrases and evaluate the performance they observe. We will ask that the group in all try at least 5 English phrases, 5 Mandarin phrases, and 5 CS phrases. They will then provide short written or verbal feedback as well as a rating between 1 and 5 in each of the following categories (speed, accuracy, ease of use, and usefulness). We could consider average scores of 4 and above to be a success in each of these categories.

VIII. Project Management

A. Schedule

The project development schedule is shown in Fig. 2 on page 13. Work on the web application and language model will occur concurrently. A total of thirty days is allocated for system implementation with another two weeks allocated for integration. Model testing is included with development time while the week allocated for system testing is partially included with integration.

B. Team Member Responsibilities

Nick and Marco are responsible for the implementation and training of the deep ASR model. Shared tasks will include fine-tuning of the off-the-shelf transformer/decoder, data augmentation for generalizability, and final system end-to-end training. Nick will focus on training and designing the LID model while Marco will focus on training and designing the English/Mandarin ASR module. They will work together to monitor and adjust the joint training of the two modules as well develop the peripheral components surrounding the model such as the logit combination, joint training loss, and input and output buffers.

Tom is responsible for the implementation of all front-end and back-end web app modules. This will include designing a front-end UI for accepting raw speech audio, chunking this audio as needed and developing an API backend which will accept audio chunks and requests for transcription. Finished transcriptions will be returned to the frontend where a live text display will present transcriptions message by message as they arrive from the cloud. He will also be responsible for providing scalability to the web application to allow for the servicing of multiple independent users concurrently.

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 14, Table I.

D. Risk Mitigation Plans

There are several areas of the project which pose potential 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. Though we are feeling somewhat more confident now that we have gained 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].

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. Should training proceed more quickly with limited performance a second but larger instance could be launched to compete against the former with the best being selected for deployment.

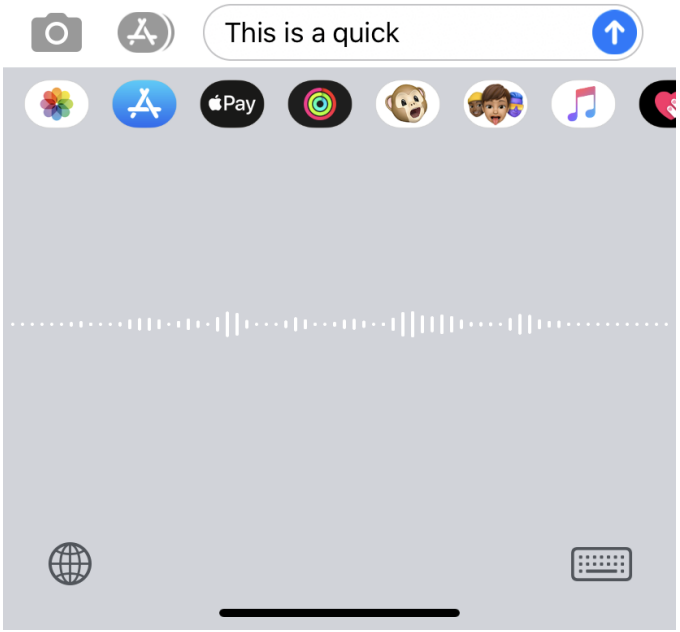
Finally, achieving our throughput targets remains an ambitious and central goal in order for our system to be viable for real deployment in our messaging use-case. Major sources of latency will be the network jump to and from the web application backend (which are difficult to predict day to day) as well as inference time through the model. Shrinking the latter is our best option if we find that the end-to-end UPL does not meet our target during integration. We can achieve this by shrinking or compressing both the model as well as the feature extractor/encoder head. We will have to carefully balance this against accuracy. Another option would be to try a larger EC2 compute instance (one with more virtual CPUs to allow more of the model to run in parallel at a time) or to fully parallelize the LID and ASR models by running them on separate machines altogether.

IX. Related Work

Solving ASR in the context of code-switching for practical use is an ongoing area of research.

A research paper published in October, 2021 is closely related to our objective. The paper is titled Mandarin-English Code-switching Speech Recognition with Self-supervised Speech Representation Models. Similarly to our objective, the paper aims to define a model architecture that can determine the language mode (English/Chinese/blank) for each frame of input audios. The paper's architecture basically contains one feature extraction layer, two parallel models that both take in extracted audio features and output probability distributions that will be combined to compute a loss for iterative training using logit scaling. The feature extraction layer used by the paper is wav2vec 2.0 that takes in .wav files and extracts audio features into vector form. The vectorized features are fed into a CTC model and a LID model that run in parallel. The CTC model outputs a probability distribution of word tokens for the transcription of the input audio; the LID model outputs a probability distribution of language for each frame of the input audio. The pair of probability distributions are joined to compute a loss, which completes a training pipeline. The paper then tests different model choices for language detection and speech-to-text transcription hoping to find the optimal model combination that gives the lowest translation error rate. The result of the paper shows that using the multilingual XLSR model to learn features from mandarin-English-mixed audios and then applying the joint CTC/LID architectures gives the lowest translation error rate of 18%.

We also see a similar objective in the dictation feature in iOS devices. The dictation feature supports users to speak into their device and their voices will be transcribed into texts in a real time manner. Below is the screenshot of the dictation feature.



The transcription is accurate when the input language is single language mode; however, a test with speaking a mix of Mandarin and English into the dictation app shows low transcription accuracy, which inspires our project.

X. Summary

Our design uses a joint LID and ASR model training scheme to separate the burden of performing the LID and

speech transcription tasks onto different models. The theory is that this will allow for better handling of the CS use case in instant messaging. The system will be deployed in the cloud to allow for a powerful compute backend (GPU) to perform model inference attached to a frontend UI which will be responsible for accepting raw audio input and returning live transcriptions. The use of a pre-trained multilingual feature extractor and decoder will help provide useful language contexts before end-to-end fine-tuned training even begins.

We anticipate most of our implementation challenges to lie in the training and fine-tuning of the model given that this will require significant software development effort to orchestrate as well as many hours of continuous training. The anticipated performance of the system is difficult to predict until significant training has been performed which presents a risk to achieving a useful final product. This must be completed successfully in order to meet our accuracy requirements. Given that achieving accuracy relies on using large models and special dedicated compute resources, meeting our latency requirement will likely be an equally challenging task that will require carefully optimizing every step of the system's pipeline.

Glossary of Acronyms

ASR – Automatic Speech Recognition
BLSTM - Bidirectional long short-term memory
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
WER - Word Error Rate

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] Panayotov, 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.

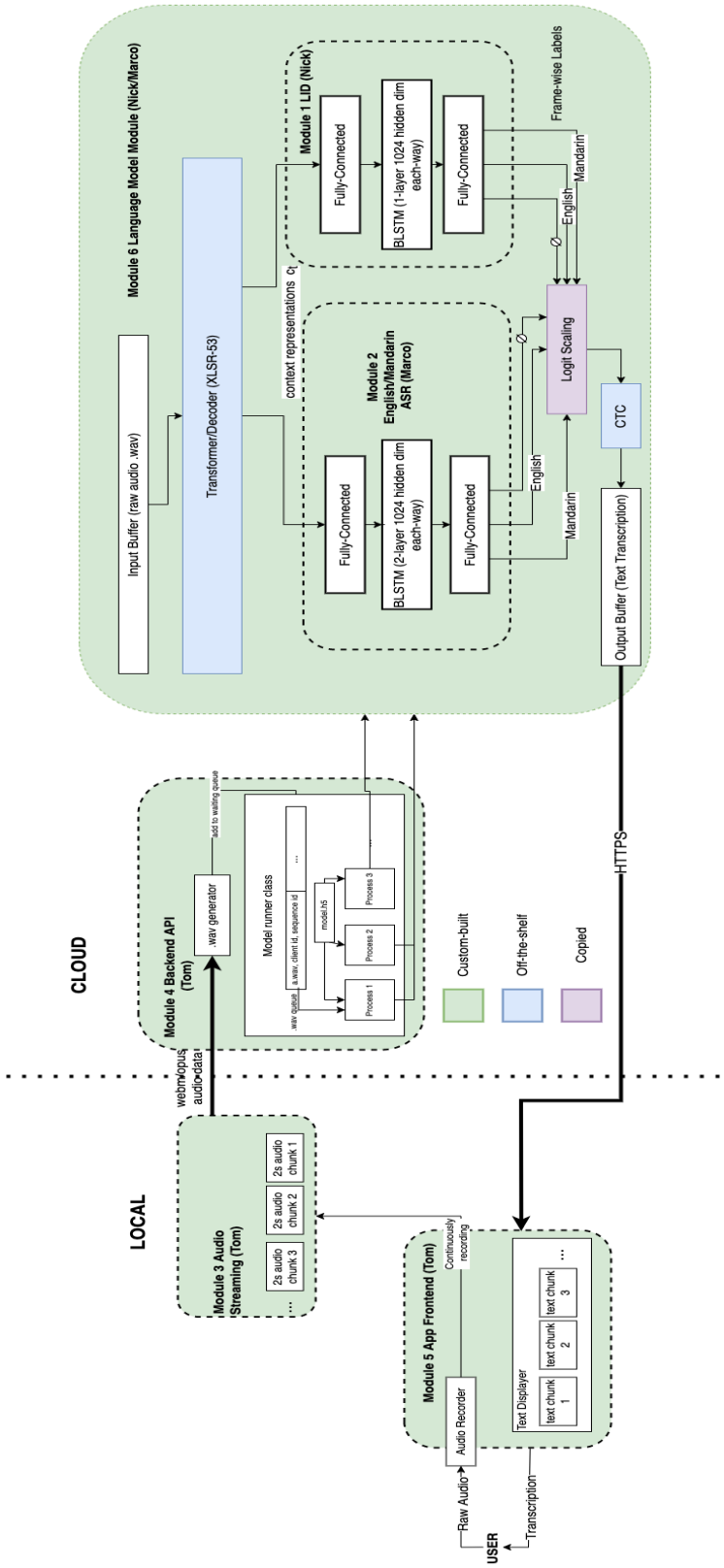


Fig. 1. System Architecture

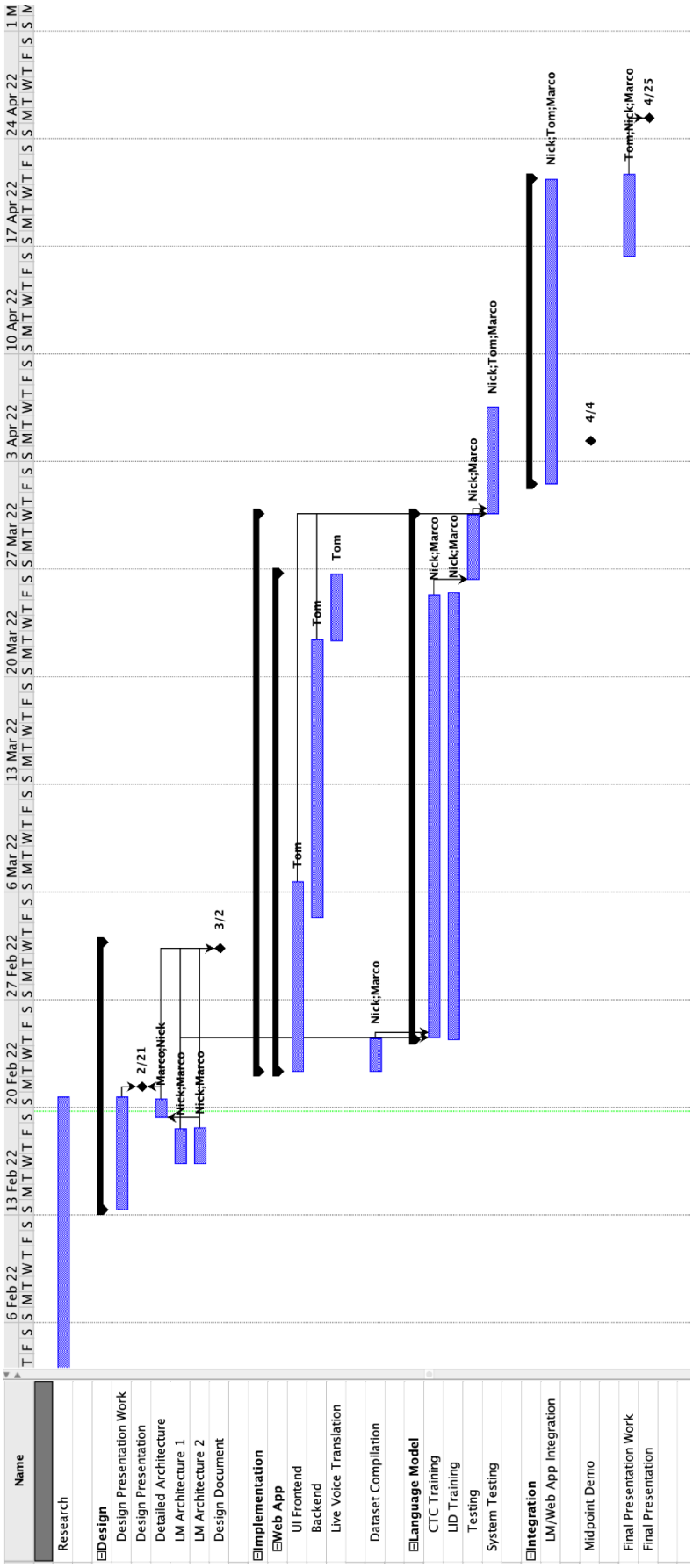


Fig. 2. Project schedule with milestones and team responsibilities

Table 1. Bill of Materials

Table I. Bill of Materials

Description	Model #	Manufacturer	Quantity	Cost @	Total
AWS GPU Instance for LID Module, CTC Module and app deployment, each instance estimated 7 full days of runtime	g4dn.xlarge	AWS	3	$.526/hr	$265.10
AWS EBS 1.5 Months SSD Storage	gp2	AWS	1	$.1/GB-Month	$19.50
				Grand Total	**$284.60**