Università Degli Studi
di Milano-Bicocca

Marvin: Developing an Intelligent Question-Answering AI consumer technology product

Andrea Yachaya
913721

Mirko Morello
920601

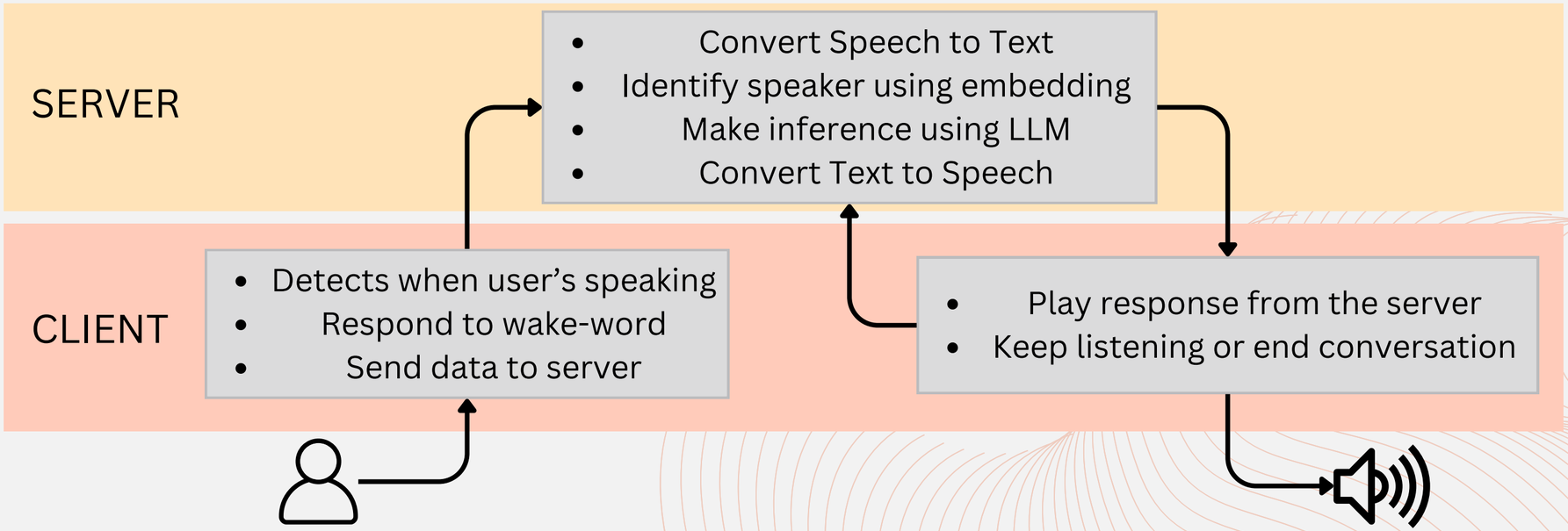
Artificial Intelligence For Science and Technology - A.Y. 2024/2025

Content

General
Overview

The project aimed to...

- Design a working AI-based virtual assistant
- The client has to listen to audio, communicate to server and play audio back
- The server is able to:
  - Separate speech in segments (diarization), and identify speakers (with embedding calculation)
  - Perform speech to text, communicate with LLM, and text to speech
  - Having real time multiple clients connected, with minimal latency

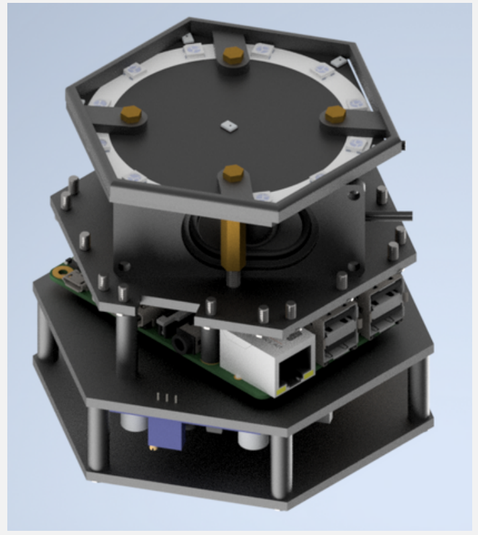


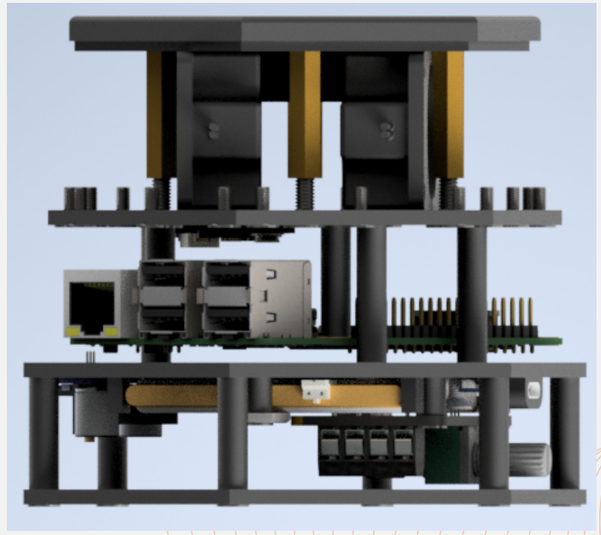
Client
Overview

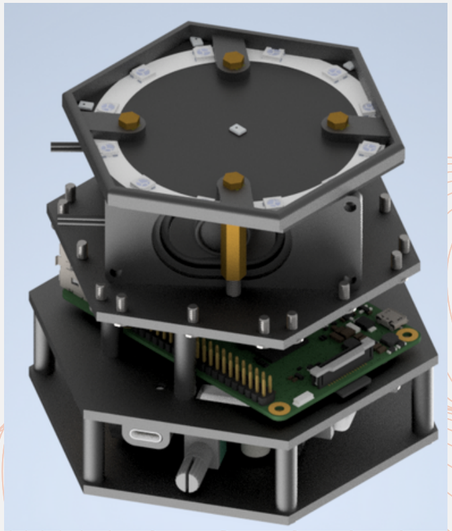
Hardware Details

The client is composed of a Raspberry Pi Model B connected with the necessary hardware for powering it up, acquire the audio from the user, and play the response aloud. All this components are enclosed in a specifically designed 3D printed casing.

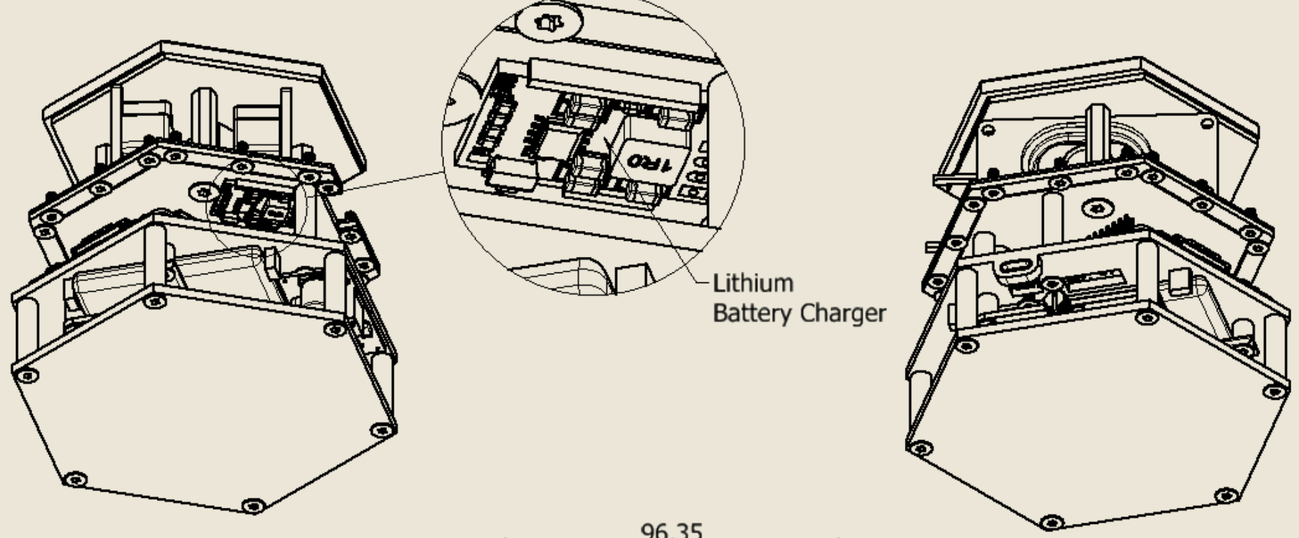
We used a lithium battery for powering up all the client components, a battery manager module which includes a voltage regulator to step-up the voltage compatible with raspberry, 2 speaker to have stereo sound with associated amplifier and a acquisition module which provides 7 microphones and 12 addressable LEDs.

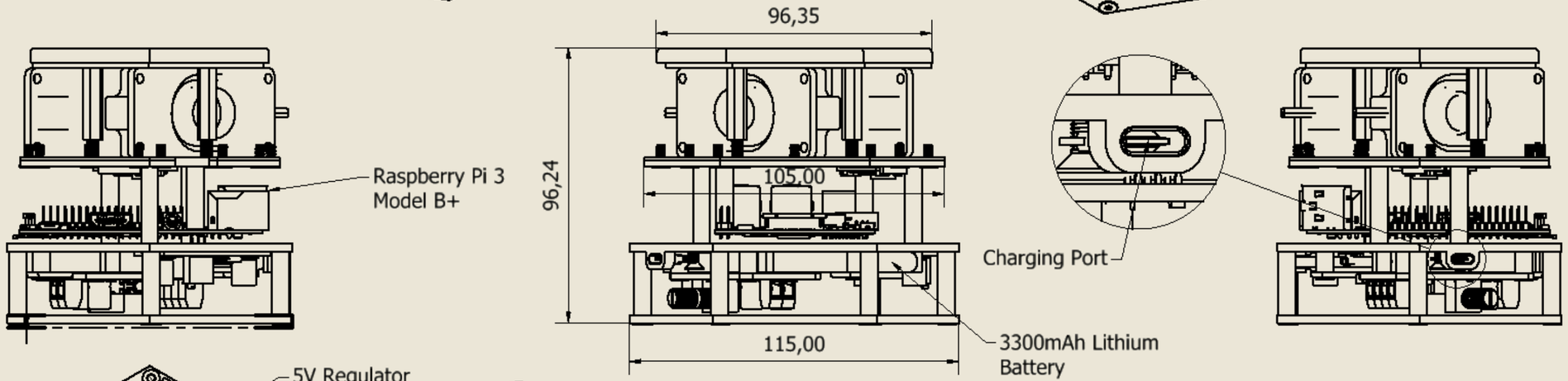


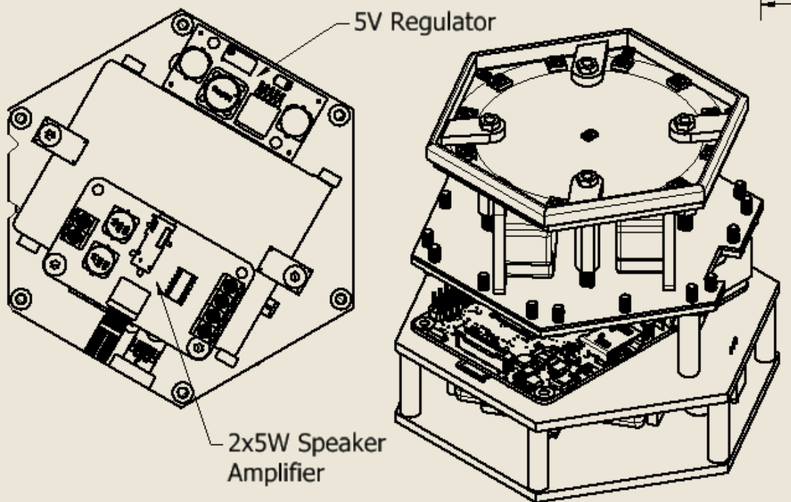


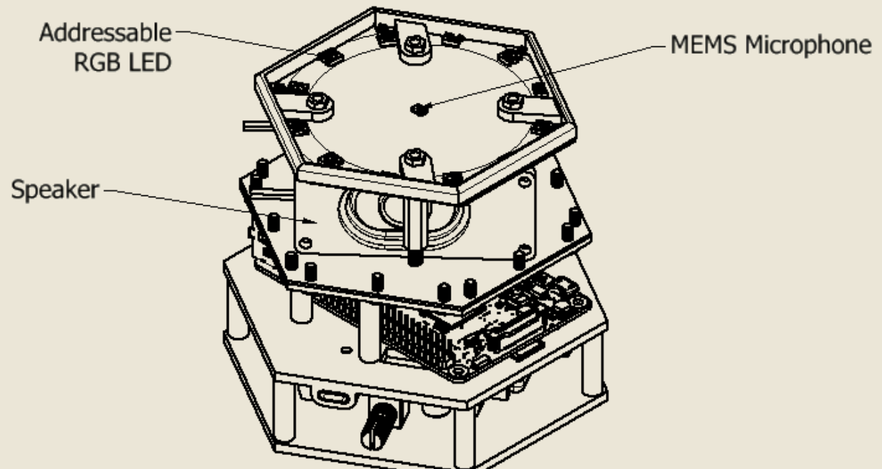


Client
Overview







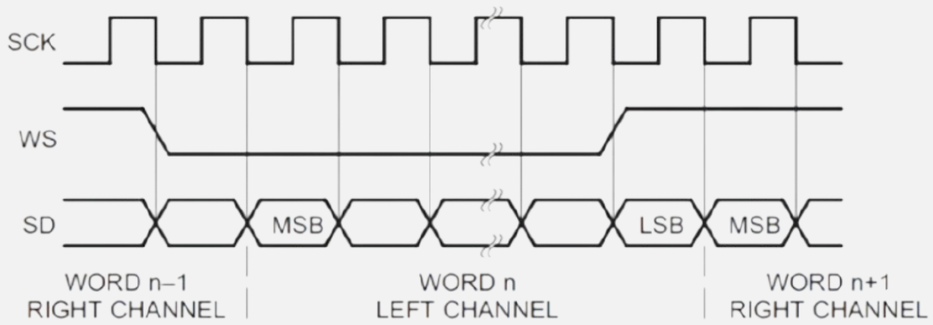


Client
Overview

- General Overview
- Client Overview
- Server Overview
- Future Work
- Conclusion

Electrical Details

While the LEDs are driven using the i2c comunication protocol, the data from the microphones are readed using the i2s comunication protocol.

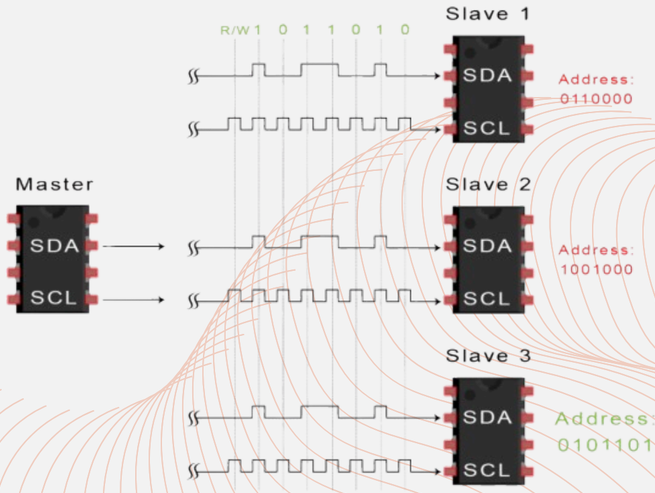


Timing diagram for i2s protocol

The clock (SCK) times the data transfer from the microphones, while the word select (WS) allow for the selection of the channel from which a data word is transfered from.

i2c is used to transfer data from low-speed peripherals to a microcontroller or processor. In our case is used for controlling the colors of each RGB LED. It follow a master-slave architecture , where one or more master devices can communicate with multiple slaves. SDA transmits data between devices while SCL synchronizes the data transmission

Example of multiple devices comunicating using i2c protocol



Client Overview

Wake Word Detection

Wake word detection is performed on the client for two main reasons:

- Privacy concern
- Avoid overloading the server of requests

The initial model we evaluated for this task the MIT/ast-finetuned-speech-commands-v2, which is an Audio Spectrogram Transformer (AST) model fine-tuned on the Google speech commands v2 dataset.

This model, composed of around 85M of parameters, was too big for a system which lacks any computing acceleration onboard like the raspberry Pi 3. For this reason we designed a different architecture highly inspired by MatchboxNet.[1]

This allowed to create a fast neural network with a total of 77k parameters.
The network was trained on the same dataset, the Google Speech Command V2 dataset, which contains 35 classes and 105,829 audio files on the training set.
The final network weights 7.27 MB.

[1] MatchboxNet: 1D Time-Channel Separable Convolutional Neural Network Architecture for Speech Commands Recognition https://arxiv.org/abs/2004.08531

Client
Overview

Wake Word Detection

The model is designed for audio classification and operates on raw audio inputs by first transforming them into frequency-domain features (MFCCs) and then processing them through a series of convolutional blocks.

- Preprocessing (Time-to-Frequency Domain):
  - AudioToMFCCPreprocessor: This module converts raw audio waveforms (time-domain signals) into MFCC (Mel Frequency Cepstral Coefficients) features.This step provides a compact, perceptually motivated representation of the audio that emphasizes important frequency content for speech tasks.
- Encoding (Feature Extraction with Convolutional Blocks):
  - ConvASREncoder: After obtaining the MFCC features, the model passes them to an encoder composed of multiple JasperBlocks. Each JasperBlock consists of:
    - MaskedConv1d Layers: These are specialized 1D convolution layers that take into account the valid lengths of the input (handling padding/masking appropriately) to prevent computations on padded regions.
    - Depthwise and Pointwise Convolutions.
    - Batch Normalization and Activation and Residual Connections
- The encoder gradually transforms the MFCC feature maps, extracting higher-level representations while potentially reducing or altering the temporal resolution based on convolution parameters.
- Decoding (Classification Head):
  - ConvASRDecoderClassification: Once the features are extracted by the encoder, the decoder aggregates the temporal information using an AdaptiveAvgPool1d layer, which pools the features along the time dimension to a fixed size. The pooled representation is then passed through a linear layer to produce logits corresponding to the different classes.

The model employs a CrossEntropyLoss for training, which is standard for multi-class classification.

Layer (type:depth-idx)	Output Shape	Param #
EncDecClassificationModel	[1, 35]	–
AudioToMFCCPreprocessor: 1-1	[1, 64, 101]	–
MFCC: 2-1	[1, 64, 101]	–
MelSpectrogram: 3-1	[1, 64, 101]	–
AmplitudeToDB: 3-2	[1, 64, 101]	–
ConvASREncoder: 1-2	[1, 128, 101]	–
Sequential: 2-2	–	–
JasperBlock: 3-3	[1, 128, 101]	9,152
JasperBlock: 3-4	[1, 64, 101]	18,304
JasperBlock: 3-5	[1, 64, 101]	9,408
JasperBlock: 3-6	[1, 64, 101]	9,536
JasperBlock: 3-7	[1, 128, 101]	10,304
JasperBlock: 3-8	[1, 128, 101]	16,768
ConvASRDecoderClassification: 1-3	[1, 35]	–
AdaptiveAvgPool1d: 2-3	[1, 128, 1]	–
Sequential: 2-4	[1, 35]	–
Linear: 3-9	[1, 35]	4,515

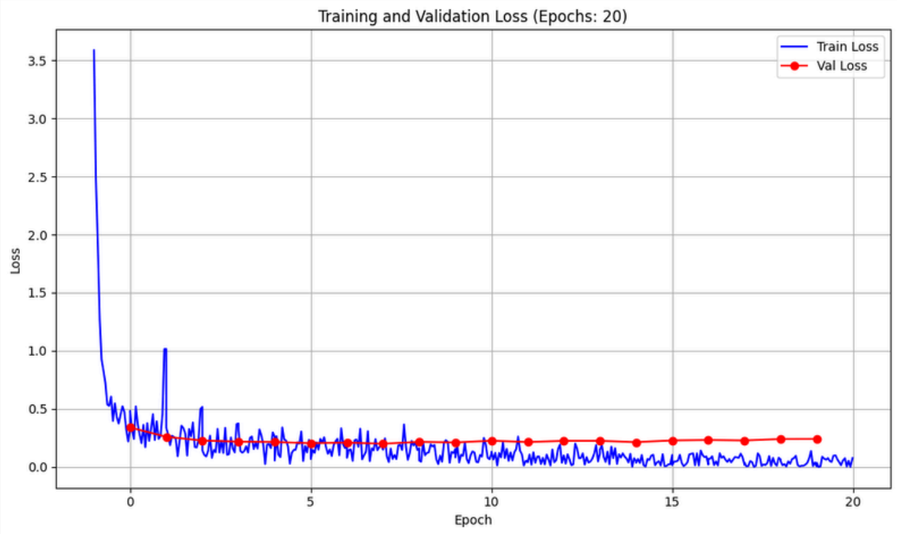
Table 1: Detailed Model Architecture

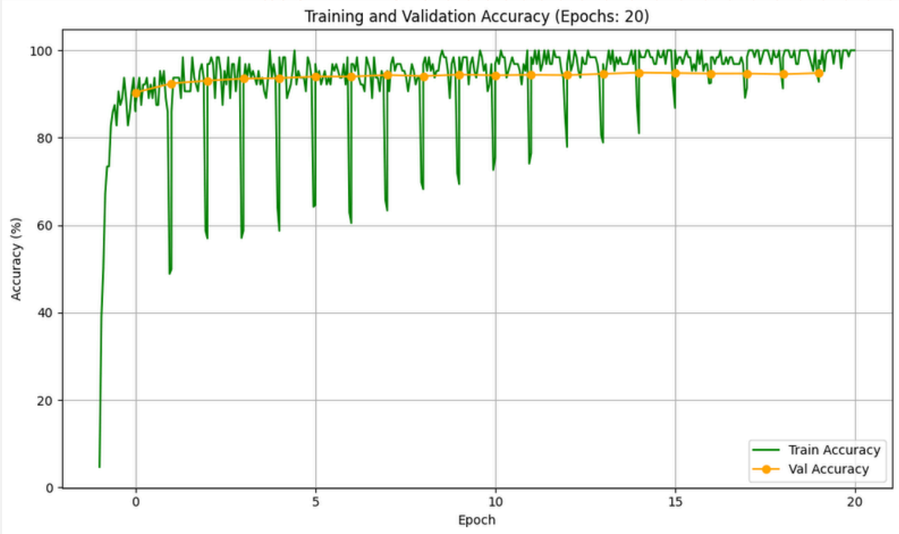
Parameter	Value
Total params	77,987
Trainable params	77,987
Non-trainable params	0
Total mult-adds (Units.MEGABYTES)	7.27
Input size (MB)	0.06
Forward/backward pass size (MB)	1.66
Params size (MB)	0.31
Estimated Total Size (MB)	2.03

Table 2: Model Parameter Statistics

Client Overview

Wake Word Detection





The network was trained for 20 epochs using Adam as optimizer, CrossEntropy as loss function and ReduceLROnPlateau as scheduler for the learning rate. We also considered a batch size of 64 elements and an initial learning rate of 0.001.

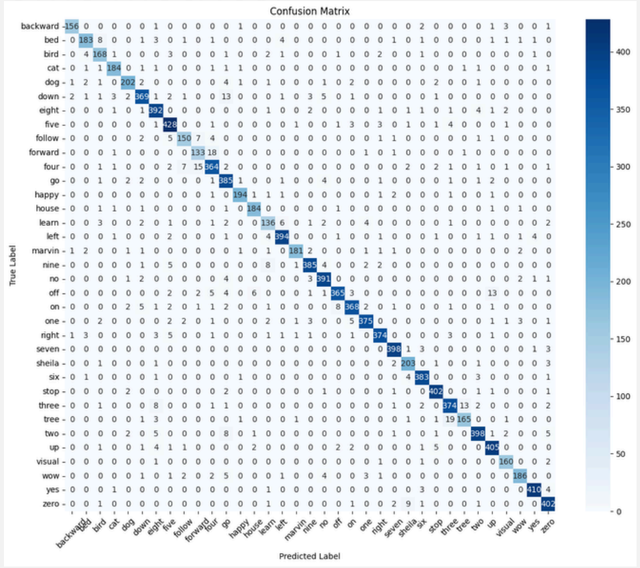
The final evaluation on the testing set, lead to an average accuracy of 94.02%

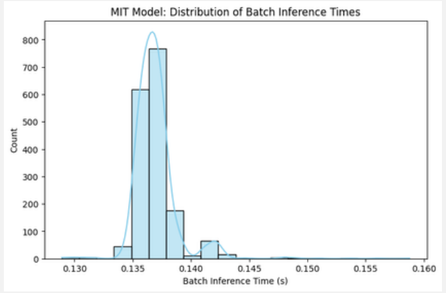
Client Overview

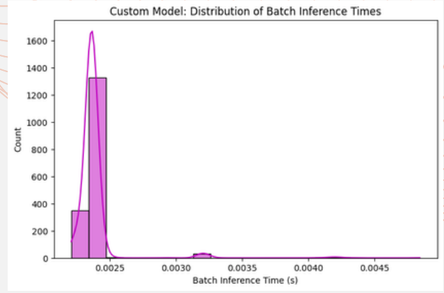
- General Overview
- Client Overview
- Server Overview
- Future Work
- Conclusion

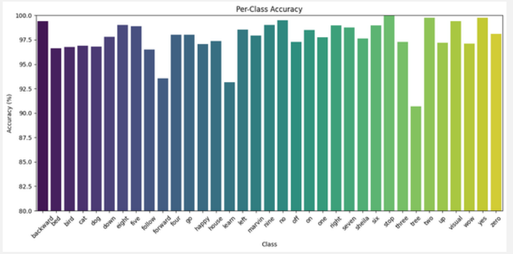
Wake Word Detection

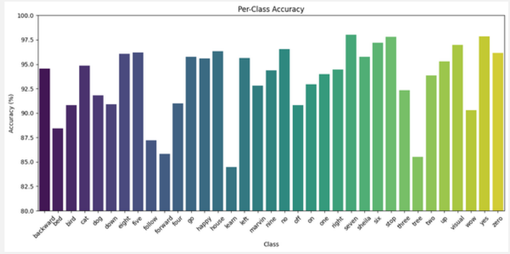
Some metrics:

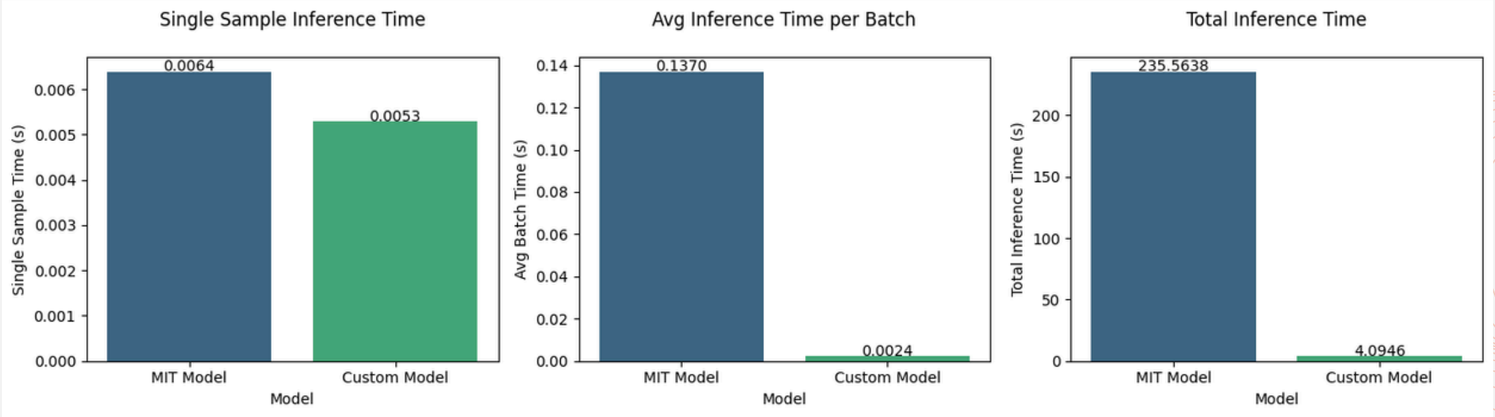










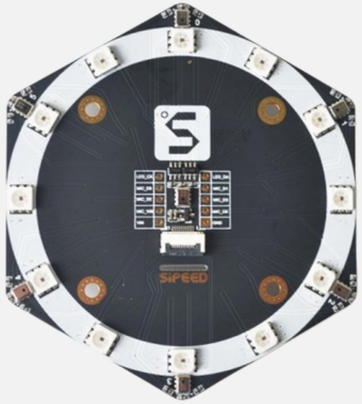


Client
Overview

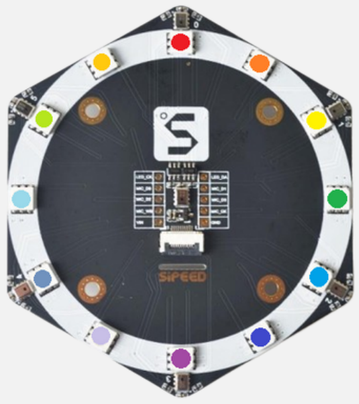
Client States

After the wake-word has been detected, the client enter in the listening state, and in that state remains until the smoothed-energy decreases below a fixed threshold. The energy is the intensity of the audio signal. A louder or more energetic sound (like speech) will have a higher energy value, while silence or background noise will have a lower energy. The energy for each time interval is recorded and then smoothed using an exponential smoothing in order to provide a more stable indication of speech activity.

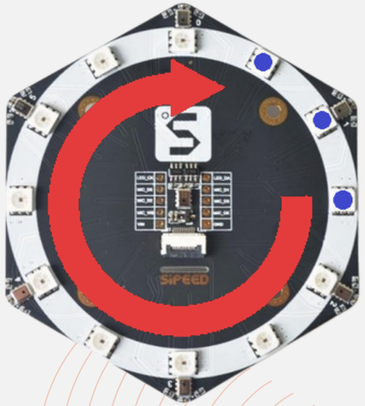
The current state in which the client is in is showed using a user frendly pattern of lights using the LEDs attached to the microphones module. The color pattern associated to each state is showed below



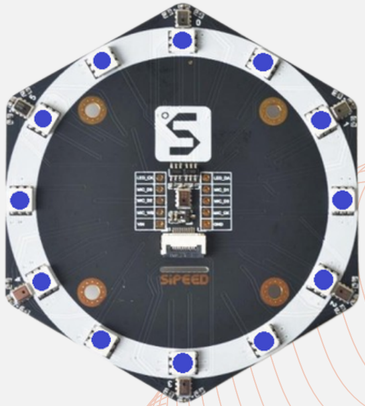
Waiting for wake-word



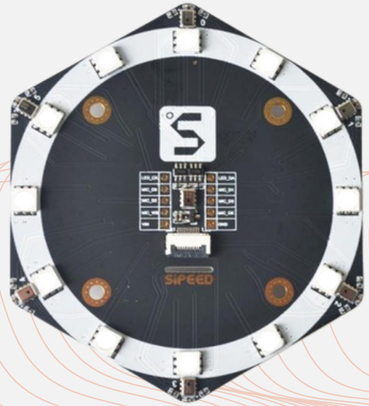
Listening to conversation



Grace period before sending the data



Data sent to the server



Response is being played

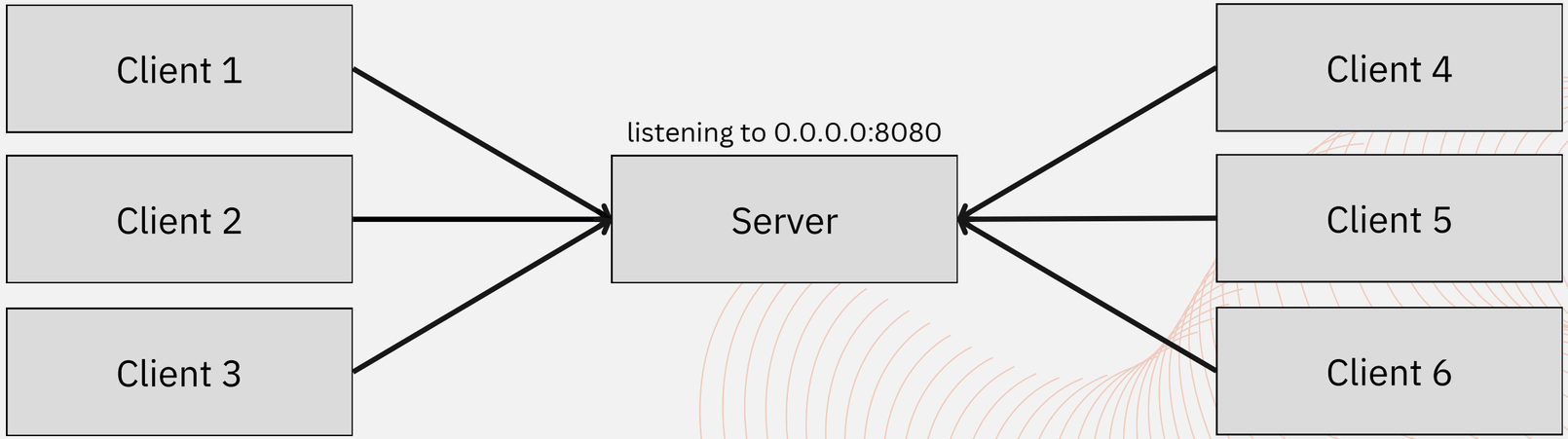
Client
Overview

- General Overview
- Client Overview
- Server Overview
- Future Work
- Conclusion

Client-Server Architecture

The client and the server comunicates using a client-server paradigm over raw TCP. Instead of using an already established L7 comunication protocol like a websocket, we decided to opt for a simpler binary protocol defined by our own for the specific context of our application.

It is build on top of TCP, which is a L4 comunication protocol, and uses custom binary protocol at level 7 for handling data exchange between client and server. This way, the data overhead is reduced to the possible minimum, and we are able to send next state indicator from the server to the client alongside audio.
The server is listening to port 8080 for incoming connection coming from one or more clients.



Server
Overview

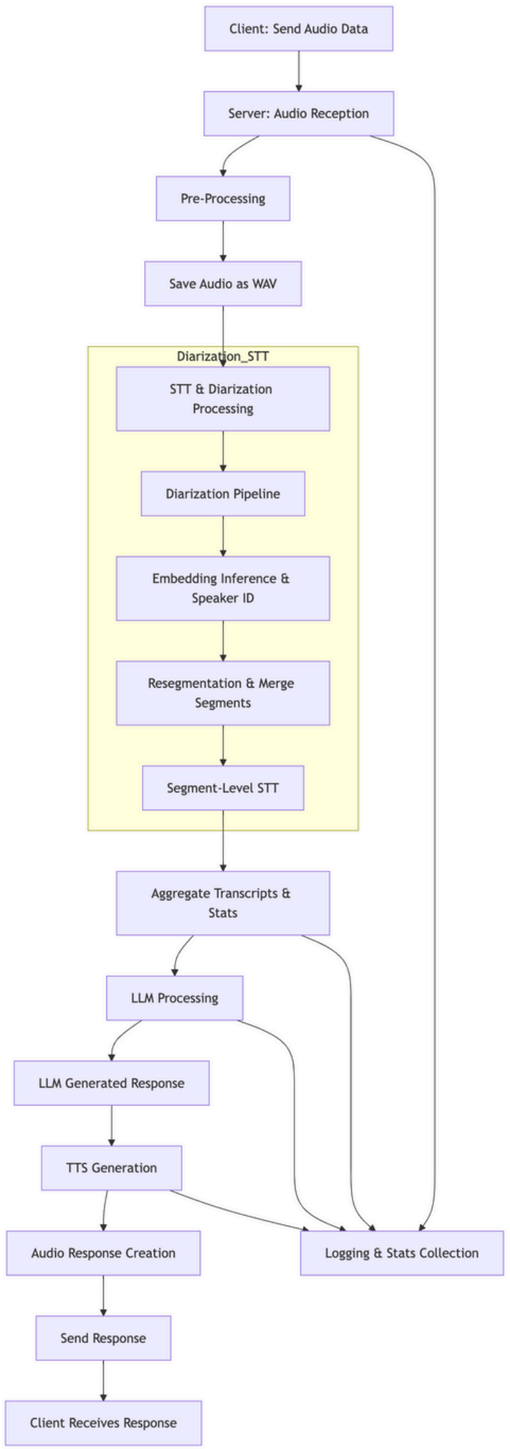
- General Overview
- Client Overview
- Server Overview
- Future Work
- Conclusion

Server Pipeline

This End-to-End Conversational Pipeline from the perspective of the server:

- Audio Reception: Client sends audio over a socket and the server receives raw audio data.
- Pre-Processing: Before working on the audio, the data is converted to a suitable format (WAV) for further analysis (this isn't privacy friendly).
- Diarization & Speech Recognition:
  - Speaker Diarization: We segment and identify the audio by speakers using pyannote models (Diarization 3.1 and Embedding)
  - Segment-Level STT: We then transcribe each segment with Whisper-based ASR (Large v3 Turbo).
- Aggregation & LLM Processing:
  - Aggregated transcripts and diarization stats form the conversation context that is happening in the room.
  - LLM (LLama 8B) processes context to generate a concise assistant response, with a custom made prompt (it knows it's on a speaker).
- TTS Generation: Converts the LLM response into audio using Kokoro TTS.
- Response Transmission: Encoded audio response is sent back to the client.

The pros of our approach is that we have a seamless integration of multiple state-of-the-art models, with real-time processing with detailed logging and statistics.
Also, our modular design allows flexible updates and component replacement.



Server
Overview

Diarization using Pyannote

First of all: Enrollment

To perform Diarization we first have a separate speaker enrollment routine:
We register new speakers by capturing multiple short utterances and aggregating their embeddings:

- We record 20 short utterances (3 seconds each) using a microphone (via PyAudio) (This isn't super solid, but more audio files can be provided).
- We then compute the embedding like the following:
  - For each WAV file we determine the audio duration and create a segment covering the entire audio. We then use the pyannote embedding model to extract a speaker embedding. If multiple frames are returned, we average them to get a single vector.
  - All enrollment embeddings are averaged to compute a centroid, which is then L2-normalized (divided by its norm) to ensure unit length.
  - The centroid is stored in a JSON database.

The JSON database is then used server side.

Server
Overview

Diarization using Pyannote

On server side, after pre-processing, the server uses the pyannote diarization pipeline to divide the audio into segments based on speaker changes. Each segment comes with start and end times.

- Embedding Extraction per Segment: For each segment, the server extracts a speaker embedding:
  - It uses the pyannote embedding inference function on the temporary WAV file (created during pre-processing) for the specific segment, just like in the enrolling phase.
  - If the function returns multiple frames, these frames are averaged to produce a single embedding vector, again, like in the enrolling phase.
  - The resulting embedding is then L2-normalized to ensure unit length.
- Similarity Comparison: The server then compares the normalized embedding of each segment to the enrolled speaker centroids (stored in the JSON database):
  - Cosine Similarity: Since all embeddings are normalized, the similarity is calculated as a dot product.
  - Then a thresholding: If the similarity score exceeds a predefined threshold, the segment is attributed to that speaker. Otherwise, it is labeled as "Unknown."

This per-segment approach allows the server to dynamically assign speaker labels during real-time processing, allowing for multiple speaker identification, which means the LLm is able to have context of multiple people on the room!

Server
Overview

Speech-to-Text (STT) Using Whisper

After diarization, each speaker-specific segment is transcribed individually using Whisper Large v3 Turbo. This approach enables us to process shorter, context-aware segments rather than one long audio file, improving both accuracy and efficiency. The resulting transcripts are then aggregated to reconstruct the full conversation while preserving speaker labels and timestamps. There are six model sizes of Whisper AI available on the market, offering speed and accuracy tradeoffs. We opted to use the Turbo version, which uses 809M parameters and requires about 6GB of VRAM, this version is optimized for handling varying audio quality and background noise.

After diarization, the audio is already divided into speaker-specific segments. Then:

- These segments, with defined start and end times, are directly fed into the Whisper Large v3 Turbo model for transcription
- By processing each segment individually, Whisper generates context-aware transcripts, which are then aggregated to reconstruct the full conversation while preserving speaker labels and timestamps.

Server
Overview

LLM Processing with LLama 8B & TTS with Kokoro

After aggregating transcripts from the Whisper model, the conversation context is processed by LLama 8B and then transformed into audio using Kokoro TTS. Here's how they work:

- LLama 8B (LLM Processing): After transcription, the aggregated conversation (complete with speaker labels and timestamps) is fed into the LLama 8B model. This model, which consists of 8 billion parameters, is loaded onto the GPU with an hybrid quantization fine tune to improve memory usage and inference speed. A custom prompt is prepended to the conversation context, instructing the model to behave as a helpful home speaker assistant. This prompt guides LLama to generate concise, context-aware responses that consider the multi-speaker environment. The processing demands of LLama 8B require a significant portion of the GPU's memory, often around 6-8GB of VRAM, depending on the stretch of the context.
- Kokoro TTS (Text-to-Speech Synthesis): Once LLama 8B produces a textual response, it is handed over to the Kokoro TTS engine for speech synthesis. Kokoro TTS converts the text into natural-sounding audio by generating phoneme sequences and synthesizing them into a waveform. The model, along with a specific voice profile (for example, voice "af" in use), is also loaded onto the GPU to facilitate low-latency, real-time audio generation. Like LLama, Kokoro leverages GPU acceleration to process the neural network computations efficiently, ensuring that the final audio output is both high quality and produced with minimal delay, but Kokoro is much smaller, comprising of 82M parameters.

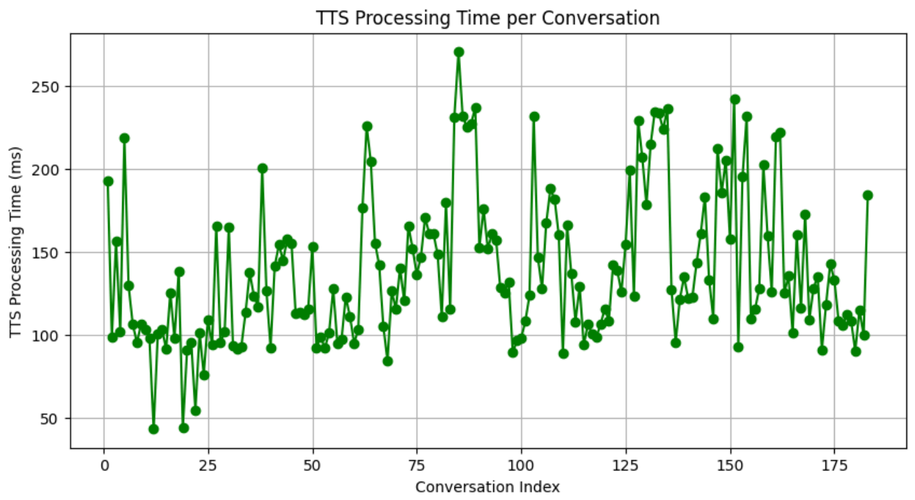
Together, LLama 8B and Kokoro TTS provide a seamless, GPU-accelerated pipeline: LLama 8B interprets and responds to the conversation context, while Kokoro TTS brings that response to life as audible speech, enabling an interactive and efficient user experience.

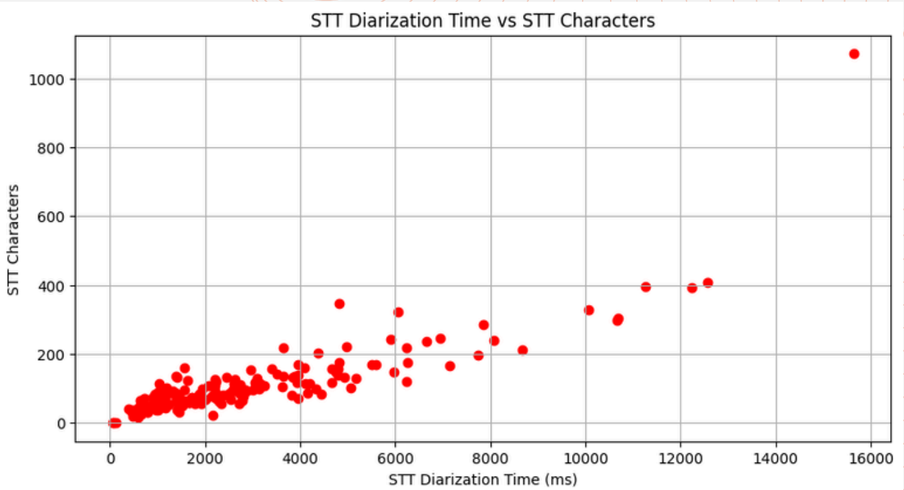
Server Overview

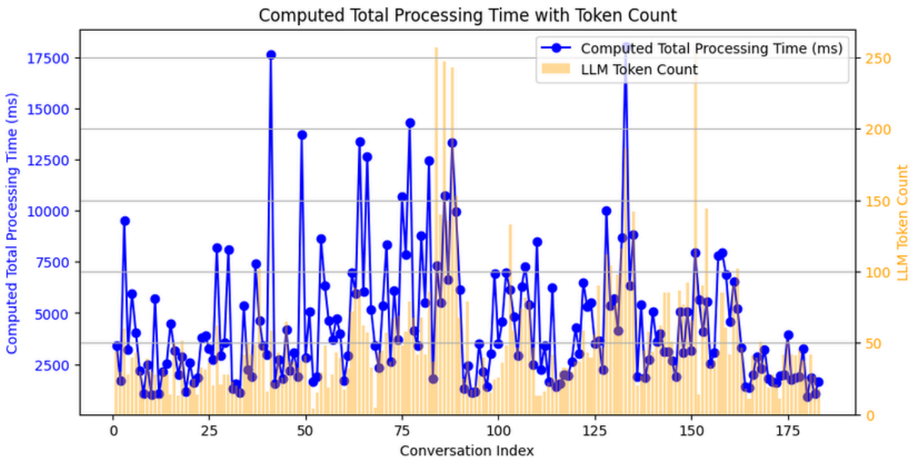
- General Overview
- Client Overview
- Server Overview
- Future Work
- Conclusion

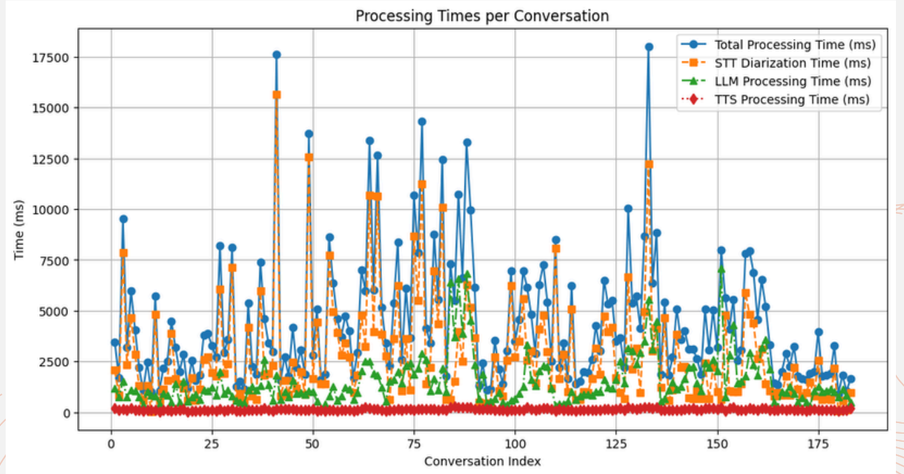
LLM Processing with LLama 8B & TTS with Kokoro

Some metrics:









Server
Overview

Future Work

Marvin was designed and deployed rapidly in order to meet the deadline. Even though it accomplish the tasks we set when we first starting working on this problem, some aspects can for sure be improved. This are the mainly ones:

- In our current implementation marvin understand that the conversation must end when a word in a set of termination words is spoken. A possible better approch would consists in the creation of a model used for infering context-aware information about the discussion in order to determine if this must end or not based on the user’s conversation. Probably a sentiment analysis' model like BERTa from Meta.
- At the current state, marvin is not able to execute any action that available commercially virtual assistant can do: like for example turning a lamp on or opening a door. Possible improvements in that direction would be to integrate an agent-based framework for controlling real-world appliances or provide more useful and personalized responses, starting from the most basic information about the the time and weather conditions, to executing API to control external appliances.

Server
Overview

Conclusion

The main challenges this project posed was the seamless interaction of multiple components both in the client and in the server pipeline. Above that, there was also a physical implementation associated to this project which required solving challenges from the hardware and electrical point of view. The use of a low power device like raspberry, also limited the use of on-device models, and pushed us to create our own lightweight wake-word detection model.
Overall, we managed to create a working intelligence computing device able to respond and solve complex tasks, which can be truly deployed as an Intelligent Consumer Technology.

Naturally, there’s room for improvements, but the device, as it is and from an empirical perspective, is able to deliver all the promises made when the project was first started.

THANK YOU