Alexa 2.0 Report

Microprocessor Based Design - Final Project

Brett Sullivan, Iris Wang, Johnny Cao, Trevor McDonald

Abstract

Voice assistants have become a centerpiece among modern consumer electronics. Most voice assistants function by recording audio locally and streaming it to the cloud for processing. This limits such systems by requiring an Internet connection to function. Cloud processing has additionally worried users about potentially exposing private information to voice assistant vendors. This report describes a voice assistant system which is constructed with open-source hardware and software which is controlled by the user. The final version of the system successfully operates offline and performs processing locally, which addresses the limitations and concerns that arise with cloud-driven voice assistant solutions. All example automations were successfully triggered by voice commands, and the system was found to function well even with background noise and with low latency.

Table of Contents

Abstract 1
Table of Contents 2
1 - Introduction 2
2 - Related 3
3 - Design 4
3.1 - Matrix Voice: 5
3.2 - Rhasspy: 7
3.3 - Home Assistant: 10
3.4 - Spotify: 11
4 - Experimental Results 13
4.1 - Hot Word Results: 13
4.2 - Matrix Stream Latency: 14
5 - Conclusion 17
6 - Future Work 19
References 21

1 - Introduction

Voice assistant products have experienced a meteoric rise in the consumer electronics space over the last decade. Indeed, all of the major players in big tech have thrown their names into the hat: between Google's Home solutions, Amazon's Alexa, Apple's Siri and many others, there is no shortage of offerings for those looking to purchase a voice assistant. Unfortunately, corporate interests in voice-driven automation are often at odds with user privacy. Sensitive data such as personally identifiable information and HIPAA-protected information can be exposed through voice assistants when it is sent to cloud servers for processing. Amazon directly states that they utilize voice samples from Alexa requests to train their language-processing models [1]. In spite of controls available on many voice assistants that claim to limit intrusion into user privacy, the impact of such controls remain a mystery as they are locked behind a black box cloud server. There is no way to verify that sensitive data is not being utilized for purposes not authorized by the user. Thus, the most effective way to ensure user privacy is to ensure that none of the data leaves the user’s network. The system outlined in this report aims to demonstrate a voice assistant which can be used for home automation while respecting the privacy of the end-user.

2 - Related

The major tech companies retail voice assistants such as Google’s Home devices, Amazon Alexa, Apple’s Siri, Microsoft’s Cortana, Samsung’s Bixby, among others. In the case of Amazon’s Alexa, the device will monitor for hotwords which triggers the voice recording. The recording is then streamed to Amazon’s cloud for processing. In any other case, all voice data is discarded [2]. The other voice assistants function in a similar manner. This seems like a logical method of saving on memory and network bandwidth in that only the relevant commands uttered by the user after the hotword will be sent as opposed to continually recording and streaming all audio picked up by the microphone. By performing the audio processing with a model stored in the cloud, commercial voice assistants can better recognize commands, offer more features and automations and improve ease-of-use by hiding away all of the automation machinery from the end-user. However, this means that commercial voice assistants are beholden to corporations for them to work. They require an Internet connection to access the cloud, and not all end-users enjoy having their automations hidden behind a black box over which they have no control. The goal of the system outlined in this report is to build an alternative voice assistant using an open-source software stack which allows the user full control over the entire ecosystem, works without an Internet connection and allows end-users to self-host to preclude sending potentially sensitive data to a black box cloud service.

3 - Design

When selecting hardware and software components for this system, the team looked to existing open-source solutions in order to avoid reimplementing solutions to already-solved problems. The prompt for this system was to perform home automation, and so a clear starting point for this aspect was utilizing Home Assistant for automation. Home Assistant offers a web interface for creating automations and offers integrations with a large variety of devices, some of which were available to the team for testing. Home Assistant specifically points out the Rhasspy project [3], which has extensive documentation on integrating with Home Assistant and creating automations using voice intents in addition to supporting multiple languages. To add an embedded component to the project, an ESP32-based satellite which is able to send audio data to Rhasspy was found [4]. This, in turn, supports several boards; out of which the Matrix Voice was selected. The Matrix Voice supports programming over a Raspberry Pi, which precludes the need to purchase a dedicated debugger probe and was a fleshed-out product as opposed to a development or a breakout board like the other supported offerings on the ESP32 Satellite page. Finally, a Wi-Fi LED strip controller running WLED was selected as it was available and could be reconfigured to work with a Wi-Fi router without reflashing. With the subcomponent selection complete, the system was integrated as found in Figure 3.0.1.

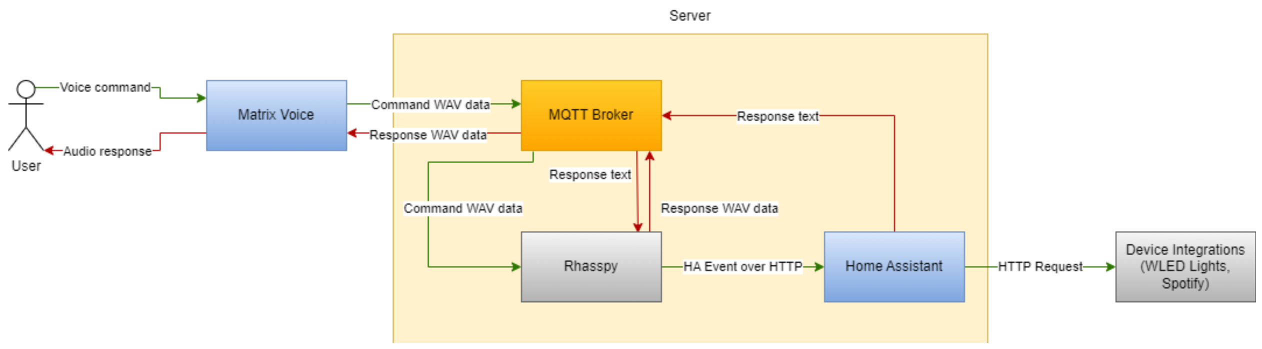


Figure 3.0.1: System block diagram

Figure 3.0.1 summarizes the interactions between each subcomponent, which shall be elaborated in the following sections. Voice commands from the user are recorded by the Matrix Voice, which forwards all WAV data to the MQTT broker. This broker passes the WAV data to Rhasspy, which dispatches the intent as a Home Assistant event using an HTTP request. Home Assistant then performs any actions with the WLED controller by sending another HTTP request or sends a vocal response as text to the MQTT broker. Rhasspy detects the response text and turns it into WAV data to send back to the Matrix Voice. The Matrix Voice will then play any audio Rhasspy sends over MQTT.

3.1 - Matrix Voice:

The Matrix Voice is a hardware package which contains the interface for recording and playing audio. It also has a set of RGB LEDs, array of MEMS Microphones, an FPGA running beamforming, and an ESP32 allowing it to be customizable for various Rhasspy integration methods. The simplest is to have the Matrix Voice connected to a Raspberry Pi running Rhasspy, and directly use the Matrix Voice as a microphone.Then the Raspberry Pi runs hot word detection and processes the audio to home assistant commands.

The second method runs the ESP32 Rhasspy Satellite software [4], which publishes the audio data captured from the microphone as WAV packets to the server over an MQTT broker. The Rhasspy server instance listening to the MQTT topic then captures the WAV packets and performs the audio processing. When a response should be played by the Matrix Voice, the WAV packets are sent by Rhasspy to the MQTT broker. The satellite listens at its assigned MQTT topic and outputs the WAV packets as an audio signal. The Matrix Voice's ESP32 connects to a Wi-Fi network on which the server software package runs in order to connect to an MQTT broker. All messages to and from the broker follow the Hermes protocol [5], which defines the set of topics and messages that Rhasspy recognizes to power the voice recognition functions.

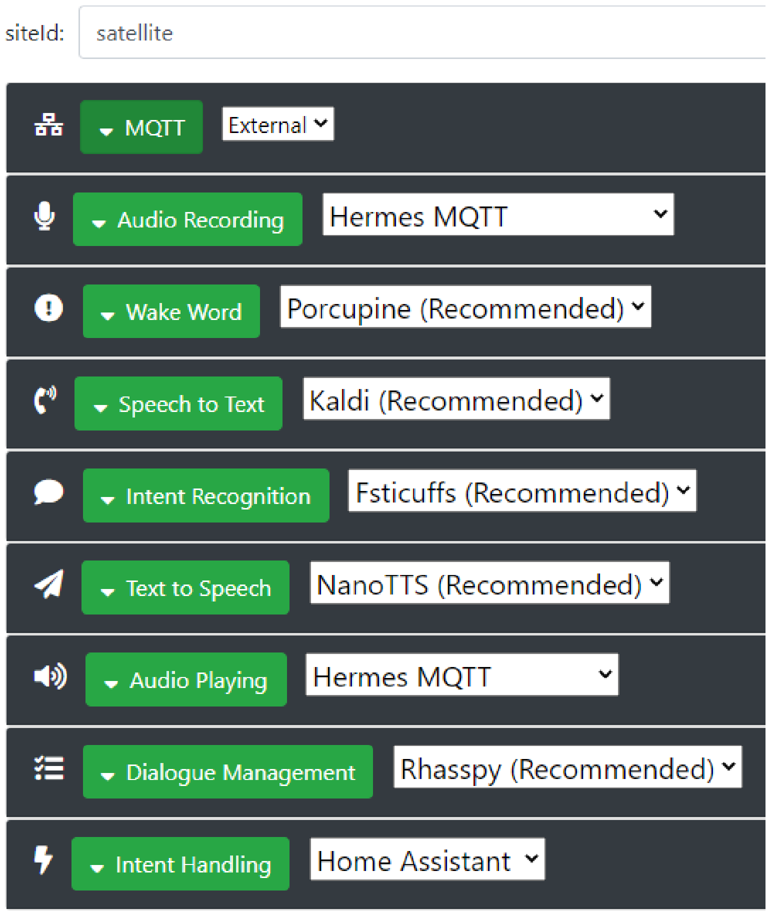
The third method is similar to the previous version using ESP32 Rhasspy Satellite release V6.0.0 [6] to locally run Wakenet 3 [7], a low-power neural network based hot word detector. After detecting a hot word the Matrix Voice publishes the audio data captured from the microphone as WAV packets to the server over an MQTT broker. Just like the previous method, the server processes the audio. We were able to get the Matrix Voice flashed with this version of the code and setup, however the hotword detection did not work locally and we did not have enough time to finish troubleshooting it. There may be incompatibilities as this is an old version of the Rhasspy Satellite software and Wakenet 3 is an old version of hotword detection.

The Matrix Voice is flashed using Platform IO [8]. The most reliable method is by connecting the matrix voice to the Raspberry Pi and flash using the GPIO with Platform IO from another host computer as explained in the GitHub Repository. There is also over-the-air (OTA) flash support for the ESP32 but this has proven to be less reliable and the Matrix IO must already be flashed and connected to the same network as the flashing host, with a known IP address.

The final system utilized the second method of streaming all audio from the Matrix Voice to the server, which eliminated the need for an additional Raspberry Pi for each satellite. However, during the development process, we encountered a challenge with the Matrix Voice. After a certain period, we were unable to reflash the ESP32 on our original board, either through direct connection with a Raspberry Pi or OTA. Despite reinstalling all software and reflashing the FPGA on the Matrix Voice, our attempts were unsuccessful in resolving the issue. We hypothesize that the flash module on the board was damaged in some way. Although we purchased a second Matrix Voice, it arrived late in our development process, and we had limited time to debug issues with the Wakenet method.

3.2 - Rhasspy:

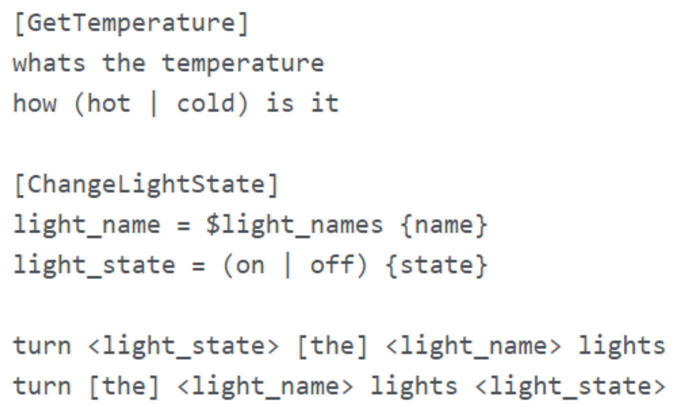
Rhasspy is a software package which performs the following tasks: hot word detection, voice data ingress over MQTT, speech-to-text conversion, intent recognition and finally intent handling, dispatching the Home Assistant event using an HTTP request to trigger an automation to occur. When an automation returns a response to Rhasspy, it then performs a text-to-speech conversion and sends the audio data over MQTT to the Matrix Voice to be played.


siteId: satellite

MQTT: External
Audio Recording: Hermes MQTT
Wake Word: Porcupine (Recommended)
Speech to Text: Kaldi (Recommended)
Intent Recognition: Fsticuffs (Recommended)
Text to Speech: NanoTTS (Recommended)
Audio Playing: Hermes MQTT
Dialogue Management: Rhasspy (Recommended)
Intent Handling: Home Assistant

Figure 3.2.1: Rhasspy web server

For each of the above steps, the Rhasspy image comes with several pre-built implementing libraries in addition to external commands, and where possible, remote HTTP servers. Therefore the voice processing is completely configurable for users, and Rhasspy itself is mostly a tool for users to configure the services that then handle the actual execution of each task and forward data from each step to the next. Settings can be changed either via a web interface started on port 12101, or directly via its profile.json file. In Figure 3.2.1, the actual settings used for the final version of the project are displayed. Each white dropdown menu shows all the options available for each task, while selecting the green button will allow users to set additional parameters based on which option is selected.



```
[GetTemperature]
whats the temperature
how (hot | cold) is it

[ChangeLightState]
light_name = $light_names {name}
light_state = (on | off) {state}

turn <light_state> [the] <light_name> lights
turn [the] <light_name> lights <light_state>
```

Figure 3.2.2: Part of sentences.ini

The other component of Rhasspy is the configuration of intents. Intents are defined in a file called sentences.ini, where different types of intents are defined as well as the sentence structures and values within the intent. For example, in Figure 3.2.2, a simple GetTemperature intent will be dispatched by one of three set commands, as the (hot | cold) syntax means that either hot or cold will be recognized and result in the same intent. In contrast, the ChangeLightState intent will collect information about the name of the light, as well as the desired state. As intents are passed to Home Assistant in JSON format, the intent will have fields for light_name and light_state, which can then be parsed and handled by Home Assistant. In order to add new commands, users need to add the intent name and sentence structure to sentences.ini, then handle that intent by adding an automation in Home Assistant. Finally, the dollar sign syntax shown above in $light_names simply means that light_names can be one of several names which are defined in a separate file. This prevents the sentences.ini file from becoming cluttered with long lists of options.

The Rhasspy instance is deployed along with the MQTT server which brokers communication using the Hermes protocol [5] in the server software package. The work done for this system was twofold: to integrate all the software packages and to configure the intents to demonstrate the system functionality. The team set up Rhasspy to communicate with this MQTT server and to link it with the Home Assistant instance also running on the server. Then, the sentences.ini file was modified to add intents which will be supported by Home Assistant.

The team encountered and addressed several challenges during the setup phase. First, because Rhasspy was set up on a different device from the MQTT broker, the Rhasspy instance would fail to start when turned on again. This was resolved by pulling Rhasspy into the docker-compose.yml file. Another issue was the address of the Home Assistant server must include an http:// prefix or automations would not be dispatched by Rhasspy. Once these issues were resolved, the work for Home Assistant was to add automations and then pull them into Rhasspy by adding intents which trigger those automations.

3.3 - Home Assistant:

Home Assistant is a software package that acts as the automation engine for the system. Home Assistant takes events triggered by Rhasspy and turns them into actions. Those actions include controlling a WLED light strip, getting the temperature, or controlling Spotify by sending HTTP requests to some external API and returning a response to Rhasspy by sending a textual response to the MQTT server for Rhasspy to process.

As was the case with Rhasspy, the work done for this system was twofold; the setup to integrate Home Assistant with the rest of the server ecosystem as well as creating automations to demonstrate functionality. This was a matter of connecting the Home Assistant instance to the MQTT broker in order to send voice response text to Rhasspy and connecting Home Assistant with the WLED LED controller, AccuWeather, and Spotify. Once the initial set up was complete, automations were added in order to test the system working together. The automations supported include turning the LED strip on and off, individual LEDs on the strip on and off, turning the strip on with a delay or for a certain duration, retrieving a temperature, pausing/playing music on Spotify, switching playback device, and searching Spotify [9].

As with the set up procedure from Rhasspy, one of the problems with integrating the WLED was the IP address changing as the Wi-Fi router was turned on and off. Since the address of the WLED controller was hard-coded into Home Assistant, it would not pick up the new IP of the WLED controller when the router was rebooted. However, this was easily fixed by assigning a static IP to the WLED controller. Realistically, the WLED module is not intended to be unplugged and end-users typically have enough control over their network to also reserve a static IP if this also became an issue for them as well.

The initial automations created for testing the system primarily centered around turning the lights on and off. This is a common application for voice assistants; it is easy to determine whether the automations function or not and the team had convenient access to an LED controller which ran the WLED software. Some of the common functions such as turning lights on and off were obvious candidates for voice automation. Since much of the project was integrating existing software together, more automations were added to increase the scope of the project. This included the capability to turn on and off individual LEDs on the strip and setting a timer for either scheduling the lights to turn on later or to turn on for a duration of time. To ensure that a response could be retrieved from Home Assistant, there are additional automations for retrieving the current time and for retrieving the temperature from AccuWeather.

3.4 - Spotify:

In addition to such devices like the WLED strip, Home Assistant can automate cloud services like Spotify. This integration does require an Internet connection in order to access the Spotify API and control music playback, but this does not affect Rhasspy or the Matrix Voice, so sensitive sound data remains local. Home Assistant comes with options to pause and resume playback, switch devices, and execute other simple commands. In order to integrate this with Rhasspy, voice commands for playing and pausing and switching devices were added.

Rhasspy's intent handling system is intended for a predetermined, finite set of commands, as the speech recognition library by default only trains on words provided by the user in the aforementioned sentences.ini file. Therefore it can support controls such as “play music” or “pause music”, but it does not handle commands like “search Spotify for ____”. In order to add this functionality to the Home Assistant Spotify integration, the team enabled open ended speech recognition with the speech recognition library, causing it to train on a much larger language model and therefore recognize a much wider vocabulary. This decision caused other commands to become more likely to fail as the wider vocabulary made it more likely for the speech recognition to misidentify words. Rhasspy comes with an option to specify minimum confidence in a command to execute or ignore it, so the team adjusted this parameter until a satisfactory threshold was reached where fixed and open-ended commands were both working semi-regularly. Importantly, this illustrates the major tradeoff of a local voice processing system like the one created in this project: cloud-based speech recognition is simply more powerful than local software like the Kaldi library used.

After enabling open ended speech to text, two problems remained: Home Assistant does not support playing specific songs by title, and Rhasspy's intent recognition service does not handle arbitrary strings in intents and strips unknown words in the intent itself, making it difficult to pass the actual search query to Home Assistant. Therefore the team implemented a custom python script to make search and play music calls to the Spotify API using the Spotipy library, which the built-in Home Assistant integrations use as well. The event trigger from Rhasspy, in addition to the intent, also contains the raw text from the speech-to-text, so the script takes that raw text as an argument and parses the search query from the text string.

A separate issue that occurred while setting up the Spotify integration is that of authentication: Home Assistant's Spotify integration handles authentication, but having written an external script that also makes API calls, this script then needs to authorize Spotify access separately. In order to accomplish this, the Spotify Authorization code OAuth flow was used as it is intended for long running applications that authenticate once at the beginning and never again. This also involved figuring out how to add environment variables to the docker container, and caching the access token so that the Spotify integration works across different development machines. The end result is that using the docker container, different team members could access the Spotify integration without any additional authentication.

4 - Experimental Results

We followed the agile development strategy by testing each individual component (Home Assistant, Rhasspy, Microphone Options, Matrix Voice) for basic functionality, and then rapidly integrating all components together and triggering the supported voice commands to determine whether they had the desired effect. Then we would iterate to improve or add more functionality until we achieved our final setup for this project.

For our testing we were unable to record sentences and play it back through a speaker for consistency because a playback of the recorded hot word would never activate the system. Our theory is that converting the audio from analog to digital through a microphone and back to analog through a speaker, changes the audio so that the AI model no longer recognizes the hot word. Specifically, hot word detection is completed by converting the audio to Mel Frequency Cepstral Coefficients (MFCCs) that are stacked into time frames and inputted into the ai model [10]. This is just a method of converting audio data into discrete values over time that is a better format for ai models to take in and classify. Thus, for some reason playing back audio changes something about the resulting MFCCs that causes the AI to no longer recognize the hotword.

4.1 - Hot Word Results:

We compared the hot word latency between using the Matrix Voice and the Laptop as microphones. Contrary to our initial expectations, the remote MQTT audio streamer version of our project had a lower hotword latency than the fully local version running on docker (see Fig.

4.1.1). However, this was likely due to the background noise when each test was run. Since we did not test the hotword latency for each version in the same environment, the background noise was likely a significant contributor to the disparity between the two versions. In reality, the howard latency between the remote and local versions are likely comparable.

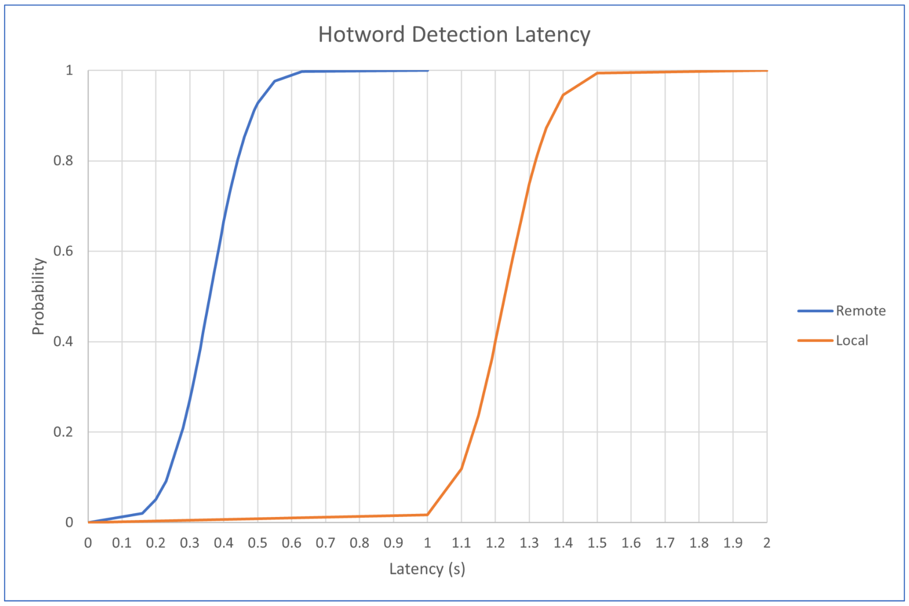


Figure 4.1.1: Hotword Latency (Local vs Remote)

For the Matrix Stream version we also tested the hotword detection with various levels of background noise. This was accomplished by playing white noise in a room at various volume levels, measuring the decibel level near the microphone, and then speaking the hot word as consistently as possible at a fixed distance from the microphone. Through this we were able to calculate a percentage of how often the hot word was detected at various levels of background noise as seen in Figure 4.1.2. As it can be seen the hot word was consistently activated until ~60 dB of background noise. Then the success rate dropped off until it stopped working completely around 75 dB. One interesting note is that with no background noise the hot word being spoken raised the decibel level to ~70 dB with our test setup, so the fact that it stopped working at 75 dB makes sense.

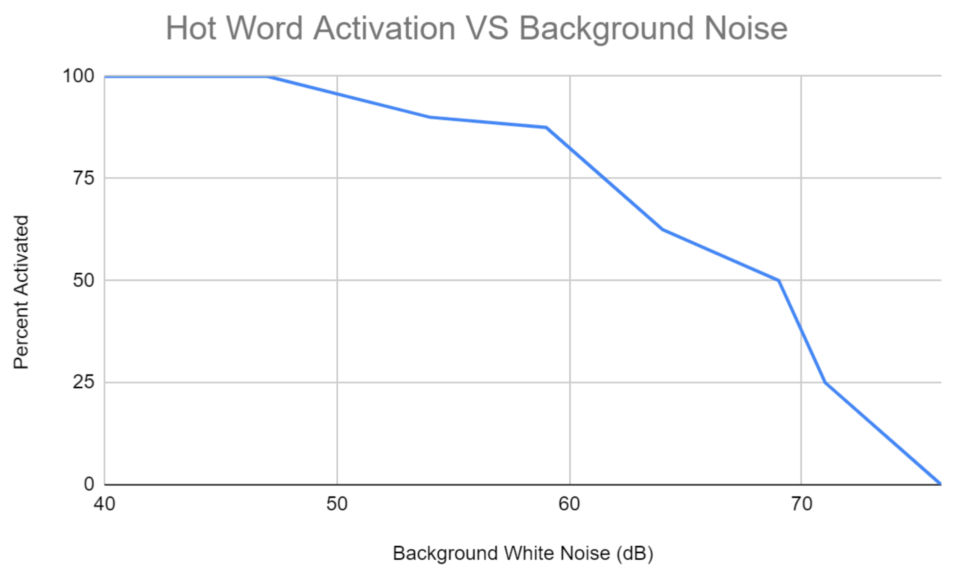


Figure 4.1.2: Percent of successful hot word activations against background noise.

4.2 - Matrix Stream Latency:

In order to obtain quantifiable latency results, we developed a method that utilized GPIO outputs of the ESP32 on the Matrix Voice to track notable events during each Rhasspy interaction. To do this we had to develop this system both on the Matrix Voice side to toggle the output GPIO and the Raspberry Pi side that read the GPIO toggles and timed them. For all of the following data, open-ended speech transcription was disabled. Enabling it consistently causes higher latency as more processing is required to match words on a larger dictionary.

Firstly, we developed a new version of the StateMachine.hpp file (flow control for ESP32 satellite program) to toggle one of the ESP32's GPIO pins during notable events. These notable events included playing a tone to indicate a hot word detected, a tone that no more audio is being collected, and the response audio based on the command. This approach was better than timing on the matrix and printing as it was less expensive and had less of an impact on program latency.

Secondly, we developed code on the Raspberry Pi using pigpio to record these GPIO events accurately to a few microseconds, utilizing a callback function that runs whenever the state of the specified GPIO pin changes. By connecting the GPIO from the matrix to the Raspberry Pi, we were able to record the latency of these events, and this method could easily be adapted to record other events and multiple GPIO pins for different time differential measurements if required.

The entire Matrix Stream transaction is visualized in Figure 4.2.1 with the color coded latency measurements that correlate with the CDF plot in Figure 4.2.2. This is useful in understanding where the latency lies within the system.

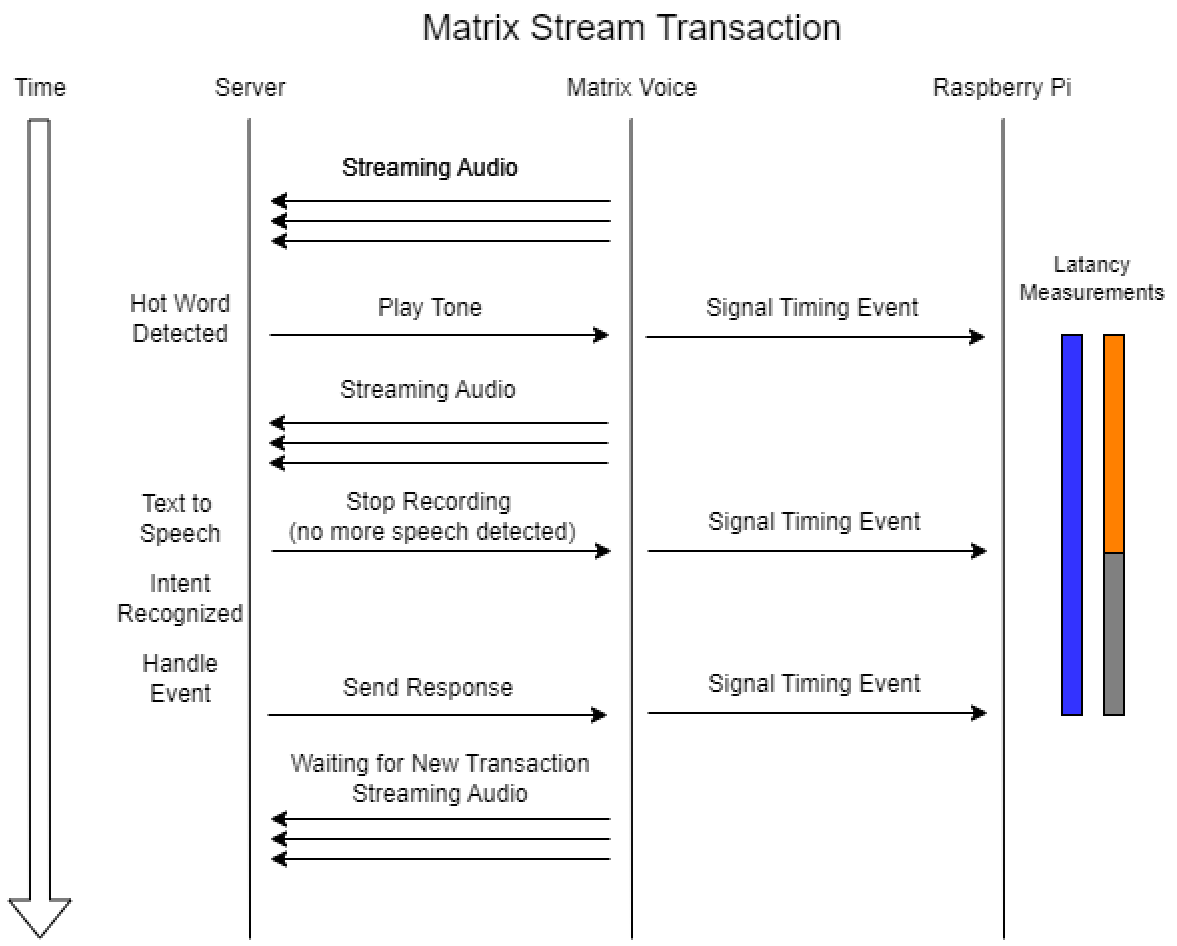


Figure 4.2.1: Matrix Stream Transaction with Latency Measurements

Running this system gave us the CDF plot in Figure 4.2.2 for the latency of various aspects of the Matrix Voice command transaction. This shows that on average an entire transaction from after the time a hot word is detected to the time the response starts to be played (blue) is about 6.5 seconds, and ranges from 3.5-8 seconds. Most of this time is taken up by the amount of time the system is listening for the command (orange) at around 5 seconds on average. The actual processing time is rather short around 1.25 seconds on average and is much more consistent as shown by the steeper slope.

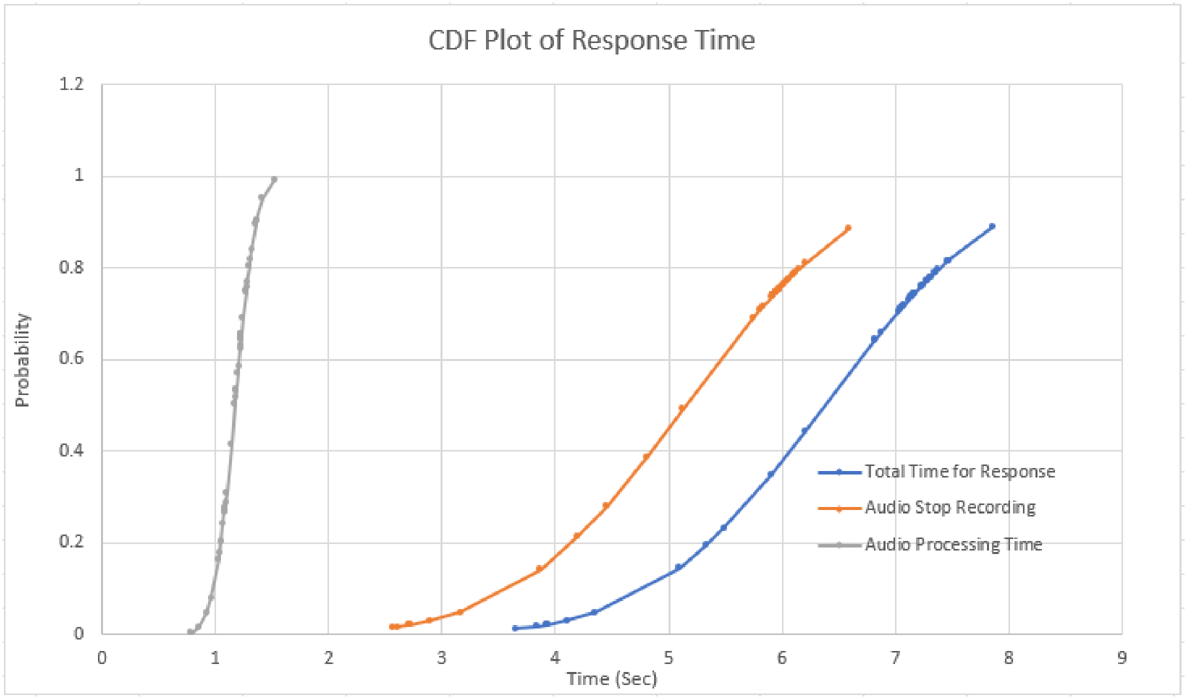


Figure 4.2.2: CDF Plot of latency results for a Matrix Stream transaction

5 - Conclusion

Commercial voice assistants offered by big tech corporations offer a streamlined voice automation solution for end-users by streaming data to their cloud services. The system outlined in this report offers a voice assistant that does not rely on a cloud to function. This offers users interested in self-hosting their own software stack a solution which functions without an Internet connection, can be inspected and modified to suit the user's liking and does not leak potentially sensitive information to black box cloud service.

The solution was implemented by integrating existing hardware and software elements to create the voice-operated home automation system. A Matrix Voice device was used to send voice data to a Rhasspy instance, which then triggers an event in Home Assistant to perform automation actions. Overall, the system was a success in that it was able to control various home automations without an Internet connection by performing local voice capture and intent recognition. Additionally, the software stack used resulted in a highly customizable system, where users can add only the automations relevant to themselves and design voice commands as they like. All of the team's example integrations were successfully triggered by voice commands, including WLED control and timing helpers, Spotify playback control and search, and weather and time getters. Further, the data collected showed that the system is robust in both background noise tolerance and command latency. Hotword activation succeeded at a rate of over 50% up to a threshold of about 70 dB of background noise, so it was very successful in a standard household environment, which is the intended use case. The hotword processing time was consistently just over 1 second, which the team qualitatively found to be a tolerable wait time.

One of the major takeaways from this project is that there is a tradeoff between the security of computing on a local network and the power of cloud computing. A mainstream voice assistant is able to handle almost any search query with a high degree of accuracy and at great speeds. This system is able to handle arbitrary searches by enabling the open speech transcription option, but this option immediately degrades the performance of all other commands that do not involve searching, and the time taken to process the command also noticeably increases.

Further, the team discovered that it would have been beneficial to integrate the voice portion earlier. Rhasspy comes with the ability to enter commands as text, bypassing audio processing entirely, and the team was able to verify that all steps after this point in the pipeline were working as expected: intent recognition as well as the Home Assistant automations triggering. However, once the actual voice recognition was introduced into the mix, the aforementioned issues with reliable speech recognition and open speech transcription came to light and less time was left to deal with those problems than desired. It would also have been useful to obtain more than one Matrix Voice to begin with; part of the reason the voice recognition was left so late was due to the first Voice becoming unflashable, and having a backup would have been useful. It also would have allowed concurrent development between team members when it was not possible to work together in person. Finally, if this system were to be deployed in an actual house, it would be useful to have multiple microphones so that voice commands could be delivered anywhere in the home rather than having to go to the only microphone listening for commands.

6 - Future Work

With more time in the future, the existing system can be improved in many different aspects. During the development of the system, it was proposed that a system could be put into place which could automatically add intents to both Rhasspy and Home Assistant. This way, end-users would not need to modify two different services to add a single function. The user experience in Home Assistant can be improved significantly. Because many functions with Rhasspy involve the use of the Hermes protocol, this results in writing significant JSON snippets using the YAML editor as opposed to the visual editor in Home Assistant. If it was not necessary to use the YAML editor, this would be a more attractive option to users which are not familiar with YAML scripting.

For the Spotify integration, the most significant remaining issue is the open-ended speech recognition degrading the whole system's accuracy. It is possible just by spending more time experimenting with various confidence values and language model mix weights, the system could achieve a higher accuracy rate in identifying voice commands. If not, open-ended speech recognition could be switched to storing audio samples, and upon identifying keywords in certain intents that include arbitrary strings, send only the relevant audio samples to a more powerful cloud-based speech recognition service. Other issues include the processing of raw text: since this overlaps with the purpose of the intent recognizer, a better system design would include a custom intent handler that returns the search query in a field of the intent YAML. The end result would be the same, but the design would separate services by purpose and become more generalizable. Finally, future work that could result from the Spotify search integration is a more general set of services that can take arbitrary strings to search on any platform, such as Alexa.

There are numerous improvements that can be made on the Matrix Voice side of things. First it would be beneficial to solve running hot word detection locally on the Matrix Voice so audio is not being continuously streamed to the server and wasting network resources. Another feature would be to potentially implement audio to text conversion on the node as well so even less data needs to be transmitted over the network, reducing resources used and increasing privacy of the system. Finally the Matrix Voice contains a customizable FPGA so these processes could be accelerated in hardware using the FPGA, making the on-node processing even faster.

References

[1] “Amazon.com: True and False.” https://www.amazon.com/b/?node=23608567011 (accessed Apr. 23, 2023).
[2] “Alexa and Alexa Device FAQs - Amazon Customer Service.” https://www.amazon.com/gp/help/customer/display.html?nodeId=201602230 (accessed Apr. 23, 2023).
[3] P. Schoutsen, “2023: Home Assistant’s year of Voice,” *Home Assistant*, Dec. 20, 2022. https://www.home-assistant.io/blog/2022/12/20/year-of-voice/ (accessed Apr. 23, 2023).
[4] P. Romkes, “ESP32 Rhasspy Satellite,” Apr. 22, 2023. https://github.com/Romkabouter/ESP32-Rhasspy-Satellite (accessed Apr. 23, 2023).
[5] “Rhasspy Hermes — Rhasspy Hermes documentation.” https://rhasspy-hermes.readthedocs.io/en/latest/ (accessed Apr. 23, 2023).
[6] “Release v6.0.0 · Romkabouter/ESP32-Rhasspy-Satellite,” *GitHub*. https://github.com/Romkabouter/ESP32-Rhasspy-Satellite/releases/tag/v6.0.0 (accessed Apr. 23, 2023).
[7] “WakeNet Wake Word Model - ESP32 - — ESP-SR latest documentation.” https://docs.espressif.com/projects/esp-sr/en/latest/esp32/wake_word_engine/README.html (accessed Apr. 23, 2023).
[8] “PlatformIO IDE — PlatformIO latest documentation.” https://docs.platformio.org/en/latest/integration/ide/pioide.html (accessed Apr. 23, 2023).
[9] “sp23-prj-voice-home-automation/server at main · neu-ece-4534-sp23/sp23-prj-voice-home-automation,” *GitHub*. https://github.com/neu-ece-4534-sp23/sp23-prj-voice-home-automation (accessed Apr. 23, 2023).
[10] Sol, “Keyword spotting, What is it?,” *Medium*, Jan. 05, 2023. https://medium.com/@nandaras0103/what-is-keyword-spotting-c06c3c44945c (accessed Apr. 27, 2023).