UCSB
CS190B - F23

Final Project

Team Members:

First and last names	Perm Number
Joseph Yue	5750633
Carson Kopper	5471271
Madhav Rai	6505317

Report

Report should be 5 pages minimum with the below sections

1. Problem Overview

I. Background

ChatGPT and its contemporaries have revolutionized many aspects of the modern world. However, this technology is still not accessible to everyone. Those who have had limited exposure to modern technology or are hampered by certain disabilities cannot fully utilize the capabilities provided to us by AI. We hope that our VoiceGPT project will bridge this gap and bring the power of AI to more people.

While existing smart speakers such as Amazon's Alexa can provide this kind of functionality to their users, they tend to focus too much on the ability to interact with other devices and as a result provide a subpar conversational experience to their users. Our VoiceGPT device on the other hand places a heavy emphasis on the vocal interactions between user and device by leveraging the power provided by LLMs.

Additionally, our prototype is also capable of interfacing with external devices, much like other smart speakers. Currently, as a proof of concept, we only enable the ability to manipulate basic lighting systems. However, the general code/hardware used can be extended to encompass many other I/O devices. Due to this design decision, we can deliver this advanced conversational proficiency without sacrificing any of the benefits provided by existing smart speakers.

II. Task Divisions

We tried to divide tasks based on what we felt we specialized in. Joseph was responsible for most of the hardware setup, both physically and in code. Madhav handled the API calls due to his familiarity with using OpenAI tools. Carson dealt with database-related tasks, such as setting it up, writing all the code to interact with it, and the voice command processing middleware. We would like to mention that although we chose to split the tasks in this manner, every team member is familiar with the entire system and is well aware of how the different components work together.

2. Methods and Solution

I. Laying the Basic Hardware Foundations

The most basic functions of a smart speaker are dependent on being able to record and play audio. It would logically follow that our first step towards creating our final product was setting up a proper speaker/microphone interface. After some deliberation, we

landed on using a USB microphone and speaker. These devices were chosen because we found that they were the quickest to set up. This was especially advantageous for our team because we all had relatively limited hardware knowledge. To actually make use of these devices in our code we used the pyaudio library, which allows us to record and play .wav audio files. Surprisingly, there were virtually no difficulties with this step. Once we finished writing the code, it was as simple as plugging in the two devices and running the script. In the test directory of our GitHub repository, we have recorded 2 tests that show us successfully recording and playing audio. Although this step was relatively easy, our inexperience with the hardware made this accomplishment very exciting.

II. Audio/Text Conversion

The next step in this process was getting the speech-to-text and text-to-speech to work. This way, we could take the audio input and feed it into ChatGPT then take ChatGPT's text output and play it out of the speakers. Due to the low performance of our PiZero, we knew beforehand that performing this kind of resource-intensive processing locally would be very difficult. Luckily, the same libraries that allow us to interface with ChatGPT also provide the audio and text conversion that we were looking for. However, this solution presented us with a new challenge. The text-to-speech conversion only worked with .mp3 files, whereas our code for audio I/O worked with .wav files. We initially thought that it would be simple enough to just convert between the 2 file types as we needed, but it soon became clear that this would drastically increase the response time of our device. Despite this, at this point in the process, we were simply happy that our device could now convert between audio and text so we decided to come back to this step at a later time and come up with a solution (which we detail in the challenges section of this report).

III. Integrating the AI and First Words

Now we were ready to finally implement the initial goal of the project, which was the ability to converse with ChatGPT through an audio interface. Out of all the steps in this project, the initial integration of ChatGPT into our system was by far the simplest. We already had the user's input as text, which we then fed into ChatGPT using an API call. Once we received a response from this API, we sent this response through our text-to-speech converter and played the resulting audio file through the speakers. We made rapid progress in this phase of the project and before long we were asking our device its first question, which was "How tall is Mount Everest?", to which it responded, "Mount Everest is approximately 8,848.68 meters tall or 29,031.7 feet tall".

IV. Making it Remember

Even though we now had a device that could answer individual questions, it still did not possess the capability to properly converse with a user because previous interactions were not stored in any form. Our initial solution to this was to create a JSON file that would be stored locally. We would then store past user/assistant interactions in this file and whenever we called the ChatGPT API, the contents of this file would be sent alongside the current prompt made by the user. This implementation is convenient because the API inherently supports messages in the JSON format. This means that we did not even need to provide additional context in the prompt we sent to ChatGPT as it already knew that the input format meant it was looking at a conversation history. The input was formatted with 2 keys for each message; the “role” key, which in this case specifies whether the message is spoken by the user or the assistant, and the “content” key, which specifies what the actual message is. We later found a better way to store this data but for now, we settled with a locally stored JSON file.

V. LIGHTS!

With the completion of the memory feature of our device, we could now move on to the next phase. We now had a system that could have proper conversations with the user, allowing them to utilize ChatGPT completely hands-free. However, the task of enabling the ability to interface with a lighting system was still left to be completed. From the moment we decided this was a feature we wanted to implement, we were well aware of the potential difficulties involved. While 190B did prepare us for properly using breadboards and GPIO pins, all our work had been in an online simulator, and our lack of hands-on experience made this task especially daunting. Our primary focus at this time was to just get something to turn on, how it looked was not an issue. As a result, we chose to use simple LEDs and avoid any fancy devices. The library we chose, RPi.GPIO, was remarkably easy to use and coding with it was quite similar to the work with Arduinos we did early in the quarter. We found that the main challenge was the hardware, which required researching the pin configuration of the PiZero and ensuring that we had correctly wired everything on our breadboard. Our initial assessment proved to be correct and this turned out to be a rather difficult task. However, we were ultimately able to get the lights to function outside of our system.

VI. State your Intent

Now that we understood how to interface with the lighting systems in our code, we worked to integrate this into our device. If this was our only feature being implemented, then the process would be pretty much the same as in step III, where we send the user

prompt to ChatGPT and process a response. However, because we wanted to also support conversations, we needed a new approach. Luckily, the OpenAI API once again provided us with the appropriate features. As mentioned earlier, the API takes a number of keys that specify certain messages to the AI, including the “role” key. We have already used the “user” and “assistant” roles but there is also a “system” role. This role indicates to the AI how the prompt should be processed. For each voice command, we added a system prompt describing a specific JSON output for certain types of user phrasing. For example, we could ask it if the prompt being processed is requesting a modification to an output device, and if it is, we make it return a command name, the name of the light or output device, and a true or false value (on or off). When we receive JSON from ChatGPT, we can handle any necessary database actions and manipulate pins on the Pi inside the command processor. Then we produce a new user-friendly text output. Otherwise, the output is passed through directly to the text-to-speech processor.

By utilizing this “system” role, we were able to rapidly integrate the lights into our device. However, this strategy also presented an additional problem. Recall that we were already feeding the previous exchanges into ChatGPT as well. When combined with these new “system” prompts, the AI could oftentimes get confused. We would ask it a normal question and it would try to change the lights or vice versa. This took quite a while to debug but we eventually came up with a solution. The details of this can be found in the challenges section but the basic outline is that we ended up making 2 calls to the ChatGPT API. One call would determine the user’s intent, whether it was a normal conversation or light control. The other call would take the full history of the conversation and return a standard response. This response would then be used if the first call identified the prompt as a general question. Otherwise, it would be discarded and the lights would be modified according to the user’s intent. With this solution in place, pretty soon we had a device that could both converse with the user and change the lighting system.

VII. Persistent Data Storage

At this point, we were almost done with the second phase. Although our device could interact with the LEDs attached to it, the settings for these lights were not stored anywhere, so when the system shut down, it would forget the configuration the lights were in originally. Our solution to this was to incorporate a PostgreSQL database into the system. We chose this approach because not only did it enable us to persist the user’s settings but we realized it could also be used to store the conversation history. The primary table to control lighting systems and other outputs contains a pin ID, corresponding to the pin number on the Pi. Associated with this pin is an alias and a boolean stating whether the output is on or off. The alias is used to provide a

user-friendly name for when the user asks to change the status of a lighting system. With this setup, multiple pins can be assigned to an alias, but only one alias can be assigned to each pin. Instead of using local memory to store conversation data, we later added a log table that stored the previous user prompts and the responses provided by ChatGPT. These logs stored both the message type (“user” or “assistant”) and the content of the message and were assigned a serial ID to enable in-order retrieval. The application can then be configured to inject a certain number of previous prompts and responses into the API request for the current prompt so that ChatGPT remembers this conversation.

VIII. Finishing Touches

In this last section, we will explain the details we added in the final week of the project to enhance the user experience with the device. The main feature we had yet to implement at this point was a voice activation command. On paper, this functionality should be very simple to implement because we already had speech-to-text capabilities so we could just check the user input for the activation word. However, it turned out that OpenAI’s speech-to-text API was not well suited for this purpose. While this tool is highly accurate, it is also unfortunately too slow in this context. Constantly checking user input for the activation word requires many calls to the API, and this means that even if OpenAI’s API is only marginally slower than DeepGram, the service we settled on, this small difference would quickly add up. In testing, we found that DeepGram was not nearly as accurate as OpenAI, but the faster response times meant that it was a trade-off we were willing to take. In this use case, we are only checking for a single word so the reduced accuracy is less detrimental.

When it comes to response times from the APIs, we found that there were situations where it took too long to respond to a request. As a result, we also decided to implement a caching feature that would store the last exchange between the user and the device. If the current prompt by the user matches the previous one, then we simply return the previous response without needing to call the ChatGPT API. Initially, we felt that we could just query the conversation for the user’s prompt but soon came to the conclusion that this could cause some errors. For example, the user could ask the device “Could you repeat that?” in multiple instances with different contexts each time. If we simply queried the response history, we would get completely incorrect answers. That is why we chose this smaller cache instead, which would provide fewer chances to use it in exchange for better accuracy.

Lastly, we added an indicator light and more generic audio outputs to give the user a better idea of what the device was doing. This time the light was easy to set up due to

how much experience we had gained up to this point with the GPIO pins. Adding simple statements like “powering on” and “switching to passive mode” was also quite simple because we could generate the necessary audio files using a call to the text-to-speech API we were using.

3. Results and Findings

I. On the Effectiveness of AI Tools

We used two speech-to-text APIs for this project, one from OpenAI and one from Deep Gram. The one from OpenAI matched what we said in over 95% of cases but it was slightly slower than the DeepGram API. While the OpenAI API was more accurate, we used DeepGram to constantly listen for the activate command. By the same token, OpenAI’s text-to-speech generated speech that felt more human than alternatives like Google Cloud but it created a few extra seconds of delay. We also found that when we inputted “14”, into the text-to-speech API, the generated speech would not say 14 so a conversion of numbers like “14” to “fourteen” fully spelled out was necessary.

The system prompts were largely successful in interpreting user intent and outputting a response our command process could handle, but with generative AI tools, there’s never a guarantee that the correct intent will be determined. While system prompts could be re-engineered to be more specific and handle edge cases correctly, this change could greatly increase the number of tokens sent per call. Moreover, with added tokens comes an increase in processing time by ChatGPT, so we wanted to keep request payloads as small as possible. Thus, we became familiar with the tradeoff between processing reliability and cost.

II. The Benefits of Threading

Threading is a very useful tool that allows us to execute multiple functions at once. From a purely functional perspective, this is useful for how we approach sending prompts to ChatGPT. As we explained in the “Methods and Solutions” section, we make 2 calls to the ChatGPT API, one to classify user intent and one to produce a response based on past exchanges. Threads allow us to perform both calls at once, without having to wait for one to finish before executing another. Given that the main bottleneck in our design is the wait time for the API calls we make, this optimization turned out to be quite beneficial for our system.

The second use for threading is to improve the user experience. There are times when we want to signify to the user that certain operations are being performed. Doing this without threads would require us to play the audio file we want, wait for that to finish,

then begin the task. By using threads, we can instead play the audio while making the API calls at the same time, once again optimizing the response time of our device.

III. Potential Points of Improvement

In this section, we will detail additional ideas we had for our device that we did not end up implementing, mostly due to time constraints. Currently, the only form of interaction with our database we support is through talking with our device. However, it is possible to put together a web app that also makes queries to the database. This would have 2 main benefits. Firstly, the web app could list past messages for the user, so if they wanted to remember a previous message, they could just look it up in the app. Secondly, this would allow the user to configure the GPIO pins without needing to speak to the device, which would greatly simplify the process of connecting devices to the pi.

Additionally, with only one Raspberry Pi, we were restricted to manipulating lighting systems connected directly to that Pi. With more resources, we could extend the system to multiple Pis, potentially each with its own microphone and speaker, and have voice commands assign lighting system aliases to specific pins on any Pi connected to the network. Lastly, while our application can handle many different types of outputs, we did not include support for inputs. Therefore, we could potentially extend this system to read temperature sensors or other input devices in the future. Also, due to the strength of ChatGPT's ability to consistently recognize user intent, it could respond to other commands from the users that we could implement such as playing music, surfing the web, or retrieving information from the user's data like their messages, search history and email to customize the assistant to the user.

4. Challenges

I. File Conversion Latency

The first challenge we encountered during this project was the amount of time it took to convert between MP3 and WAV file formats. To put this into perspective, we were constantly seeing that the file conversion by itself was taking up to 50-70 percent of the wait time. We theorized several potential solutions. We first considered changing our method of conversion. However, both APIs and local processing took up considerable amounts of time. We also considered changing the APIs we were using so that they would all work with the same file format. The problem with this solution is that it would require extensive research and testing, which would use up what little time we had left. This felt especially unnecessary because our device was technically functioning properly with the current configuration, it was just taking too long. As a result, we

decided to simply work with the formats we had instead of trying to streamline them all. This required incorporating multiple ways to play audio into our system. The original audio library we used was pyaudio but because it only worked with WAV files, we set out to find an additional method for playing audio that would work with MP3. After some trial and error, we settled on using the VLC media player. This media player is integrated into the system and can be called directly using bash commands. As a result, we believed that it would be the fastest option out of all the other methods we attempted.

II. Combining Conversation History and Light Control

This is another issue we spent a lot of time dealing with and it was very unexpected. Once we integrated both conversation history and light control into the prompts we sent to OpenAI's API, we made sure to test both features but during this testing process, we forgot to also examine using both functions in the same session. As a result, we did not realize until much later that sending both user/assistant conversation history and a light control command in the same prompt to the API at the same time would confuse the AI. The difficulty of handling this problem lay in the fact that we had no control over how the AI processed this aspect of our prompts and therefore we were forced to work around it. After some testing, it was found that splitting light control and conversation history into 2 separate prompts would fix the confusion they caused when used together. However, this solution has a fairly obvious flaw, which is that the extra call will increase the wait time of our device. As we have emphasized throughout this report, response times were one of the core issues we grappled with during the design process so we tried everything we could to speed up the device. Fortunately, at this point, we had already implemented threading into our system for playing various audio cues. Our newfound familiarity with this tool meant we were able to quickly rewrite this portion of our code so that it utilized 2 function calls combined with a thread. Of course, this does not completely fix the latency issue as 2 calls will still take longer to process than 1, but the use of threads helped mitigate the negative consequences of this new design choice.