HomieBot - The Homie-fication of AI

Devinn Chi, Quang Aethermai, Arnika Abeysekera

Macalester College

Comp 480

Table of Contents

I. Ideation

Our Initial Idea and Introduction

When we think about the current state of AI and its recent rise in popularity and usage, we imagine chat bots like ChatGPT. It is easy to understand how useful a tool like this is, but how accessible is it? Initially, our group, Arnika Abeysekera, Quang Aethermai, and Devinn Chi, set out with the goal of creating an interface where a computer could effectively communicate with a LEGO® MINDSTORMS® EV3 robot[1] through morse code, and use a Large Language Model[2] to generate a conversation between person and machine. Our idea was for a separate computer to be able to act as a morse code transmitter, and the EV3 to have the ability to convert the morse code into plaintext, and generate a conversational response, then speak this response back to the user in either morse code or vocally. However, as we considered the practicality of this ambition, we decided to alter our path towards creating an interface that would allow for vocal conversation with AI, through the likeness of an EV3 robot.

Purpose

Our idea of setting up an EV3 robot as an interface of communication between person and machine has many real-world applications, as one might imagine. While the purpose of our project was to create an interface in which a user could effectively communicate with an EV3 robot in a conversational manner, one could easily tweak our product and create an assistive AI

---

[1] (Kartunov, 2016) Kartunov, S., P, R., & M, S.. Application of robots LEGO Mindstorms Education EV3 in the learning process of specialty Mechatronics. 25–27. 2016 https://doi.org/10.15224/978-1-63248-115-3-71

[2] (Minaee & Mikolov & Nikzad, 2024) Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., & Gao, J. Large Language Models: A Survey. 2024

robot for many different applications. This technology could allow us to be better able to complete our own tasks with the added benefit of machine learning.

Interaction with generative AI without the need for direct interaction with a computer screen can also be beneficial for people with visual and physical impairments as well, opening up our project's applicability to cater towards numerous other potential users.

II. Background

Problem

Limited natural language interaction capabilities hinder the potential of EV3 robots to deliver engaging and intuitive user experiences. Traditional programming methods, often complex and demanding, restrict accessibility, particularly for younger audiences or those new to robotics. This project addresses this challenge by exploring the integration of Large Language Models (LLMs) with EV3 robots. By enabling AI-powered voice chat conversations, we aim to bridge the gap between human-robot interaction. Imagine conversing with your robotic companion, issuing instructions, or even engaging in simple dialogue. This innovative approach unlocks exciting possibilities for the educational and creative realms. First, natural language interaction fosters a more intuitive and engaging user experience when interacting with robots. Second, interactive robot communication allows for a more accessible exploration of AI concepts and applications. Finally, voice chat empowers the development of captivating narratives where robots actively participate, fostering creative expression.

Solution

This project investigates the integration of OpenAI Whisper for real-time speech recognition, Langchain's ConversationalChain with the Ollama backend and Llama-2 LLM for conversation management, and Python socket programming to facilitate communication with EV3 robots. This combination empowers the EV3 to understand spoken language, engage in interactive conversations, and respond through synthesized speech. The project's success hinges on its ability to achieve the following criteria: 1) accurate speech-to-text conversion using

Whisper, 2) effective conversation flow management and generation of relevant responses via Ollama and Llama-2, and 3) seamless translation of textual responses into synthesized speech while controlling the EV3 robot through the ev3dev library and Python socket programming. Achieving these functionalities will demonstrate the project's efficacy in enabling AI-powered voice chat conversations on the EV3 platform.

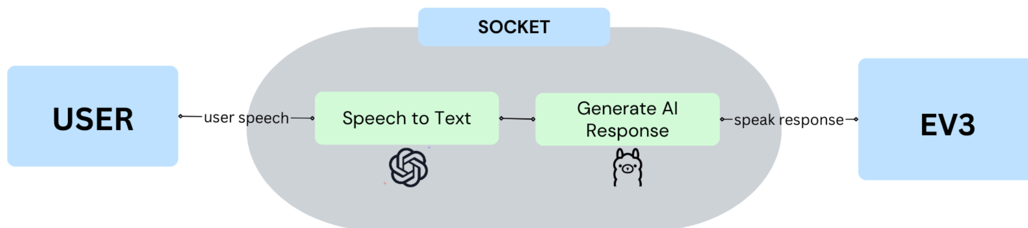

Figure 1: High-level Structure of Project

Related Work

This project leverages expertise from several areas of Artificial Intelligence (AI) and Robotics. Large Language Models (LLMs) trained on vast amounts of text data are employed to generate human-quality text, translate languages, and create informative responses to queries. Conversational AI, a subfield of AI, informs the development of systems that engage in natural language conversations with humans. Additionally, Text-to-Speech (TTS) technology allows for the conversion of written text into synthesized speech, enabling audible communication from machines. Finally, the field of Human-Robot Interaction (HRI) guides the design and development of robots that interact with humans in a natural and intuitive way.

Several projects have explored voice interaction with robots. Hanschmann et al.[3] present Saleshat, an LLM-based social robot capable of human-like sales conversations. While Saleshat

[3] (Hanschmann & Gnewuch & Maedche, 2024) Hanschmann, L., Gnewuch, U., & Maedche, A.. Saleshat: A LLM-Based Social Robot for Human-Like Sales Conversations (pp. 61–76). 2024 https://doi.org/10.1007/978-3-031-54975-5_4

focuses on a more advanced platform, it demonstrates the potential of LLMs for facilitating engaging robot conversations in specific domains, such as sales. Additionally, P. Vashistha et al.[4] built a voice assistant using a Raspberry Pi and open-source tools. Although not specific to robots, their project showcases the integration of speech recognition and text-to-speech for voice interaction. These efforts highlight the growing interest in enabling natural language communication with AI systems and robots. Our project builds upon this foundation by specifically targeting the EV3 platform. This focus expands accessibility of AI-powered voice chat conversations to a wider audience, particularly within educational and creative contexts.

[4](Vashistha & Singh & Jain & Kumar, 2019) P. Vashistha, J. P. Singh, P. Jain, & J. Kumar.. Raspberry Pi based voice-operated personal assistant (Neobot). 2019 3rd International Conference on Electronics, Communication and Aerospace Technology (ICECA), 974–978. 2019 https://doi.org/10.1109/ICECA.2019.8821892

III. Implementation

Libraries Overview

<u>OLlama</u>

The usage of libraries, packages, and other online resources made the implementation of our project much more simple and refined. These resources can be used in similar ways in the creation of alternative and uniquely helpful products using AI.

One of the most crucial parts of our project was the usage of an LLM, which initially was planned to be from OpenAI, however due to the limiting use of tokens which was not freely available on OpenAI's API access platform, we decided to use **OLlama**[5]. OLlama was much more appealing to us due to the fact that it was free to use and that we could implement it locally on our own devices, without internet access. By integrating OLlama into our project, along with other supporting conversational components such as **langchain**[6] (langchain.memory, langchain.chains, langchain.prompts, langchain_community.llms, ConversationBufferMemory, ConvesationChain, PromptTemplate) we were able to simulate conversation. Imports from langchain primarily handled memory for our OLlama text generation, so that when using our product, it would be able to respond to a user while remembering any names or preferences that may have been previously mentioned.

PromptTemplate[7] was particularly helpful for our project in that it allowed us to somewhat manipulate the "personality" of our generative AI chatbot. By changing and altering

[5] Ollama. (n.d.). Retrieved May 3, 2024, from https://ollama.com
[6] Introduction | LangChain. (n.d.). Retrieved May 3, 2024, from https://python.langchain.com/docs/get_started/introduction/
[7] langchain_core.prompts.prompt.PromptTemplate—LangChain 0.1.17. (n.d.). Retrieved May 3, 2024, from https://api.python.langchain.com/en/latest/prompts/langchain_core.prompts.prompt.PromptTemplate.html

our template, we were able to transform our bot into a more conversational entity, specifically opting towards the general personality of a “homie” (friend).

Audio Handling and Transcription

In order to handle the recording of user vocals, we used methods from API’s such as **numpy**[8], **sounddevice**[9] (as sd), **queue**[10] (from Queue), and **whisper**[11]. The numpy API is primarily used to handle mathematical/numeric computing for arrays and matrices. In our script, numpy is used primarily to handle array manipulation, which directly correlates to audio handling. The queue API is used in a similar way, as it allows us to store audio data for further processing in a queue (first in first out) data structure. The sounddevice API provides functions to be used for audio recording and playback in Python. In our script, we use this API to record audio data from the user for later processing/handling. The most important API that we use in our file for transcription is the whisper API, which is a locally hostable speech recognition model from OpenAI. We use functions from whisper to handle user audio data and convert it into plaintext, which is then able to be processed by OLlama and its supporting libraries. My team was very happily surprised with the accuracy of whisper’s speech transcription, it is even capable of recognizing some slang terms, which came very much in handy for our implementation.

For our final product, we integrated an EV3 robot using the **ev3dev2**[12] library, which allowed us to access ev3dev2.sound. This library includes functions for the EV3 robot that allow

[8] NumPy documentation—NumPy v2.1.dev0 Manual. (n.d.). Retrieved May 3, 2024, from https://numpy.org/devdocs/
[9] Play and Record Sound with Python—Python-sounddevice, version 0.4.6. (n.d.). Retrieved May 3, 2024, from https://python-sounddevice.readthedocs.io/en/0.4.6/index.html
[10] queue—A synchronized queue class. (n.d.). Python Documentation. Retrieved May 3, 2024, from https://docs.python.org/3/library/queue.html
[11] Speech to text—OpenAI API. (n.d.). Retrieved May 3, 2024, from https://platform.openai.com/docs/guides/speech-to-text
[12] API reference—Python-ev3dev 2.1.0.post1 documentation. (n.d.). Retrieved May 3, 2024, from https://ev3dev-lang.readthedocs.io/projects/python-ev3dev/en/stable/spec.html

for text to speech capabilities, and specifically the speak() function is what we used to output the text response that was generated by OLlama based on user audio data.

Helper Libraries

Additionally, we used a few more crucial APIs to hold our product together. The **threading**[13] API was used in our project to help speed up audio capturing by implementing a form of multithreaded processing within our audio recording. This allowed us to increase the speed of our overall response generation. The **socket**[14] API was very crucial for us to be able to connect with the EV3 robots for our final product; it contains functions which allow for the capability of both hosting a local server or connecting to a remote server by wired or wireless connection. We used functions from the socket API to create a wired connection to the EV3 robot to reduce time that would have been required to make repeated calls to the EV3. Finally, we used the **Console**[15] API from rich.console to add coloring to text in the console for the output of our project.

The Starter Code

The main structure of our project's AI generation is drawn from the starter code and instructional guide provided by Duy Huynh published on HackerNoon titled "*How to Build Your Own Voice Assistant and Run It Locally Using Whisper + Ollama + Bark*[16]". Through this guide

[13] threading—Thread-based parallelism. (n.d.). Python Documentation. Retrieved May 3, 2024, from https://docs.python.org/3/library/threading.html

[14] socket—Low-level networking interface. (n.d.). Python Documentation. Retrieved May 3, 2024, from https://docs.python.org/3/library/socket.html

[15] Console API — Rich 13.6.0 documentation. (n.d.). Retrieved May 3, 2024, from https://rich.readthedocs.io/en/stable/console.html

[16] (Huynh, 2024) Huynh, D. How to Build Your Own Voice Assistant and Run It Locally Using Whisper + Ollama + Bark. HackerNoon. 2024

we were able to acquire a starter code template that we based our program's structure on, as well as a helpful step-by-step guide on what APIs we would need. Upon installing the dependencies and understanding the starter code, we were able to record audio data, have it transcribed by whisper, and have it generate and speak a response with a combination of OLlama and **bark**[17] (a local text to speech model that uses AI).

The starter code, pre-modification, is pretty simple. The initial product uses two classes, one called converstaionService.py and another called textToSpeech.py. The conversationService file handles audio recording, AI generation, and audio output; the textToSpeech file allows us to synthesize AI vocal output of the chatbot using bark's pretrained model, "bark-small" (this model is meant to be faster because of its lesser size). First and foremost, in the conversationService file, all of the necessary libraries (as mentioned above) are loaded into the working environment via import statements.

```
console = Console()
stt = whisper.load_model("base.en")
tts = TextToSpeechService()
```

Next, as shown above, the whisper model, "base.en" (pre-trained to recognize the English language), alongside a TextToSpeechService object (to access functions in the textToSpeech class), "tts", for bark integration. A console object, "console", is also created to handle creating more user-friendly console stylings. Following these initializations, a template is created:

```
template = """
You are a helpful and friendly AI assistant. You are polite, respectful, and aim to provide concise responses of less
than 20 words.
The conversation transcript is as follows:
{history}
And here is the user's follow-up: {input}
Your response:
"""
```

Figure 2: Original Chatbot Template

[17] Suno-ai/bark. (2024). [Jupyter Notebook]. Suno. https://github.com/suno-ai/bark (Original work published 2023)

This template aims to personify the OLlama-run AI chatbot as an AI assistant. A PromptTemplate object is created, “PROMPT”, with this template as a parameter, then a ConversationChain object is created, “chain”, with “PROMPT” as a parameter. The “chain” variable is used to access the OLlama LLM and its helper functions.

Furthermore, there are a number of main functions within the conversationService class. Here, they will be described alongside an overview of the workflow in this program. First, after the user is prompted to press “enter” to start recording, record_audio() uses a nested function, callback(), alongside functions from the sounddevice library to handle user input audio recording and gather recording as audio data (the user presses “enter” again to stop recording).

```
    def callback(indata, frames, time, status):
        if status:
            console.print(status)
        data_queue.put(bytes(indata))
    with sd.RawInputStream(
        samplerate=16000, dtype="int16", channels=1, callback=callback):
        while not stop_event.is_set():
            time.sleep(0.1)
```

Then, transcribe() takes in the data gathered from record_audio() as input, then applies it to the previously loaded whisper model. transcribe() then takes the string returned by the whisper model and prints it onto the console (with some styled coloring from the rich library) after some cleaning is done using strip(). This string is also saved as the variable, “text”.

```
result = stt.transcribe(audio_np, fp16=True)
text = result["text"].strip()
return text
```

Once this is done, the program calls get_llm_response() using the user’s sentence, “text”, as input. get_llm_response() uses the previously initialized (with the template)

ConversationChain variable, “chain”, to access the OLlama LLM, then generate and return a response based on the user’s sentence.

```
response = chain.predict(input=text)
    if response.startswith("Assistant:"):
        response = response[len("Assistant:") :].strip()
    return response
```

Once this response is available, long_form_symthesize() is accessed by using the TextToSpeechService object, “tts”. This function uses the previously trained bark model to generate AI vocal audio output of the string returned by get_llm_response(). This audio output is then played by the computer’s speakers using play_audio(), which uses the sounddevice library’s play() function as method of output.

```
sd.play(audio_array, sample_rate)
sd.wait()
```

This entire process is held in the main script under a while loop which can be exited if the user presses “ctrl+c”. This is so that the user can ask repeated questions to the AI chatbot.

After examining and experimenting with the starter code, we decided that there were some optimizations and expansions that we had the ambition to attempt the implementation of.

Our Implementation and Modifications

Optimizations

Upon further testing and examination, we noticed that bark, the guide’s method of speech to text, was extremely slow on our laptops, and was even very slow when running the program on Devinn’s at home very high end gaming PC. The majority of runs of the program took upwards of 2-5 minutes to complete. Because of this, just for testing purposes on our devices without considering usage of the EV3 robot, we opted to use functions from the **py3ttx** API, which is a much simpler text to speech model with audio output that sounds less "human" than bark, but was unimportant for the purpose of our project (Bhat, n.d.). This allowed us to run the program and make adjustments at a much faster pace.

Initially, the guide that we used provided us with code that initializes the chatbot's generative output to take on the personality of an assistant. Because we wanted to modify this personality to act more conversationally, we opted to change the given template that is interpreted by functions in the PromptTemplate library. After extensive experimentation, mainly trial and error, we were able to adjust the personality of our chatbot accordingly with this template:

```
template = """
You are a friendly homie that seeks to converse with a fellow homie (user).
You speak as a close friend of the user would.
You aim to provide responses as fast as possible, to maintain the flow of conversation.
Your main objective is to be a conversation partner,
engaging in what one may call small talk. Speak in the manner that the user speaks to you, including
slang and such. You aim to provide responses in less than 30 words.
The conversation transcript is as follows:
{history}
And here is the user's follow-up: {input}
Your response:
"""
PROMPT = PromptTemplate(input_variables=["history", "input"], template=template)
```

Figure 3: Modified "HomieBot" Template

Once we successfully adjusted our template, we sought to add EV3 functionality.

Consistent Communication with Robot

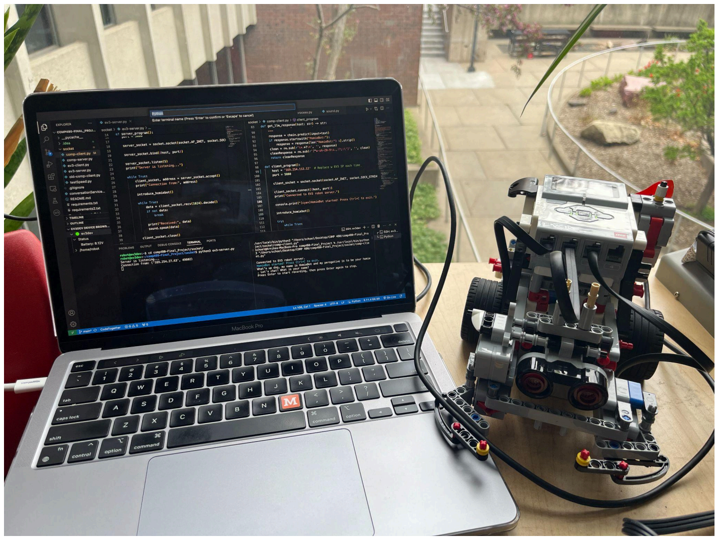

Figure 4: Wired Connection between Laptop and EV3

In order to integrate the base implementation with the EV3, we simply removed the text-to-response method utilizing py3ttx, and used the EV3's Sound.speak() method instead. It was surprisingly simple, involving only a single import of the Sound module and a single call to the speak method, taking in the generated response as an input.

And once we implemented one cycle of:

speech-to-text → text-to-response → response-to-speech

with the EV3, we had to implement a way to repeat this process such that the user could have a sustained conversation. The method we chose for this was using socket programming and creating our own locally hosted server for communication between laptop and EV3.

To do so, the computer and EV3 were binded together in a socket connection, with the EV3 as the client and laptop as the server. The EV3 instantiated a socket connection at the address 0.0.0.0, thereby allowing it to accept more connections. It then listened for incoming connections. In this case, the laptop was connecting to the EV3, using the wired connection IP address of the EV3 (Figure 3) and the same local port as initialized by the EV3.

Once the socket connection was established, it allowed us to send information back and forth between the two in a communication loop. The computer was able to send data (in the form of the text response) to the EV3:

```
response = get_llm_response(text)
client_socket.send(response.encode())
```

Meanwhile, the EV3 listened for incoming data from the computer (to be taken in and "spoken" by the robot):

```
data = client_socket.recv(1024).decode()
sound.speak(data)
```

IV. Testing and Performance

To achieve the final implementation above, we incrementally tested each main function of our project and made improvements when appropriate. The voice recognition, transcription, response generation, and response to speech were the primary functions that were tested.

Voice Recognition

Starting with the first function the user encounters, we tested how well voice recognition works. Fortunately, the sounddevice API was able to pick up the primary speaker with surprising accuracy. Regardless of distance from the laptop, number of other speakers around you, and the volume at which the user is speaking, the audio was recorded clearly enough for the next step (transcription). We tested in classrooms, where numerous other conversations were occurring around the laptop's microphone. We (separately) tested with the user around four feet from the laptop, and speaking below the average speaking level. In each case, the recorded audio and speaker recognition was accurate. However, it is important to note that we only tested on the microphones of Macbook devices. Therefore, performance may worsen on different devices and/or external microphones. For the purpose of our project, however, where the laptop was necessary for the socket connection, the laptop microphone was sufficient.

Transcription

The next step - accurate transcription - was crucial, as the text transcription would be sent to the generative AI model and the accuracy of its response would be dependent on the initial prompt given. Our chosen API, whisper, was able to accurately transcribe the given audio file.

Around 99% of the words were written correctly with accurate punctuation in between. Ellipses were added to the transcription when there was a pause in the speech. Tone was even interpreted correctly. For example, question marks were added to the end of the spoken questions. Proper nouns were also detected with similar accuracy, and spelled and capitalized correctly. For example, the following speech to text was observed (Figure 4), and matches the speech perfectly:

```
You: My name is James Jordan Bingham Jr. and I hail from the great state of Washington
 DC in the United States of America. How art thou today?
```

Figure 5: Example of Accurate Transcription (Special Focus on Punctuation and Proper Nouns)

Both the names and the punctuation are also accurate. One mistake we observed, however, when testing was that Macalester College was transcribed as McAllister College (but with the correct capitalization on the A).

To test the speed at which transcription occurs, we utilized Python's time() module, encasing code we wanted to test with the following:

```
startTime = time.time()

# CODE TO TEST HERE

endTime = time.time()

executionTime = endTime - startTime
```

In this case, the code we tested were the lines that took in an audio file and transcribed it. For a single sentence of average length (around 7 words), when averaging 15 tests, it took approximately 1.387 seconds to transcribe. For an audio file containing 5 sentences of average length, transcription took, on average, 2.775 seconds. Therefore, transcription occurred fast enough not to slow down conversation with the AI.

Response Generation

Most of the issues we encountered in the project were related to the following sections. Immediately after basic implementation, we noticed lags in the response generation time. For a single sentence of shorter length (around 4 words), when averaging 15 tests, it took approximately 19.919 seconds to generate a response. For an audio file containing 3 sentences of average length, transcription took, on average, 29.531 seconds. This significantly slowed down the rate of conversation with the robot.

Therefore, to improve the response time we ran the *stable-code* model of Llama rather than *llama2.* Stable-code is slightly optimized, containing only the foundational modeling necessary for generating correct responses, and is 1/4th of the size of *llama2.* Therefore, it can run faster than the previous model of Llama we were running. For this model, it took 14.213 seconds to respond to a short sentence, and 19.665 seconds for a longer sentence. Therefore, response time was slightly improved.

The responses themselves, however, consistently and accurately reflected the prompt, and matched the template. For example:

```
You: What is the history of Minnesota?
HomieBot: Oh, man, Minnesota? That's like way up north, ya know? It's got a rich history, though. Native American tribes, like the Ojibwe and Dakota, used to roam there. Then there was the fur trade, and later on, it became a hub for industry and immigration. You feel me?
```

Not only was the bot speaking in the informal, friendly, slang detailed in the template, but it referenced correct information about Native American tribes in Minnesota in answer to the question.

Speaking the Response

Finally, we tested the text-to-speech (TTS) function. Most of the issues that we encountered, and improvements we made were in this section of the project. In the first implementation of our project, we had the laptop speak the response (rather than the EV3). To do so, we first utilized the Bark TTS library. However, the response time was incredibly slow. It takes >1 minute every time to speak a response. Assuming approximately 30 seconds of that time was the actual response generation, simply converting text-to-speech took around 30 seconds to start. Therefore, we had to switch models in order to speed up conversation time.

The library we opted for instead was pyttsx3. When using this library, there was virtually no delay in text-to-speech startup. After a response was generated by the AI, it took, on average, 0.823 seconds for speech to start. This reduced conversation time.

Finally, when we integrated this implementation with the EV3, we eliminated the need for a TTS library altogether and used the EV3's built-in Sound library and speak function. Again, the speed of speech startup was sufficient. On average, it took around 2.844 seconds for the EV3 to start speaking the given text. No modifications relating to speed were made.

However, we had to modify the response itself based on the EV3's capabilities. For example, the responses generated by Llama include Emojis. The EV3, however, could not recognize the character or translate them to speech. Furthermore, the responses included action phrases closed within asterisks. (For example, `HomieBot: That's so funny! *chuckles* Man, that's a good joke.`). Therefore, we had to clean the response text to exclude Emojis and action phrases, for more natural speech.

Python Profiling

To further assess the performance of our project, we utilized the cProfile module in Python (The Python Profilers, n.d.). This built-in tool acts as a code profiler, pinpointing sections that consume the most execution time. By analyzing the profiling data, we can identify potential bottlenecks within the system. This information is crucial for optimization, allowing us to focus on areas that can significantly improve the overall responsiveness and efficiency of the AI-powered voice chat experience on the EV3 robots.

Within the module, the `cProfile.run()` function allows us to profile code by passing the Python code as a string argument. This flexibility enables us to profile specific modules, like Whisper and Llama-2, providing a better understanding of their performance impact. cProfile reveals to us that OpenAI Whisper for speech recognition and Llama-2 for conversation management as the most time-consuming components. This is because these models are computationally expensive, especially when dealing with real-time audio processing and complex language generation. Therefore, in the future, if optimizations were to be made, they could be targeted on those models.

However, overall, with the changes made through testing, we were still able to increase conversation speed, without compromising accuracy.

V. Limitations

Hardware

After running through some tests, we noticed that there were some differences in various speeds across tasks when testing our program on different hardware. On Quang and Arnika's MacBooks, downloading dependencies took a few attempts but was successful after a short period of troubleshooting. On Devinn's windows laptop, once the MacBooks were working correctly, Devinn's speech to text using whisper stopped working, however, on his pc at home there were no problems. This meant that when observing differences between the MacOS and Windows operations systems, we were also comparing desktop hardware to laptop hardware. Because we were using completely local APIs and libraries, as well as generative AI models, almost all of the general speed depended on the internal components of whichever device we were using, so for consistency's purpose, we decided to limit testing to either Arnika or Quang's laptops.

One of the main limitations of our project is the need to create a connection between one of our laptops and the EV3 robot. The EV3 robot has an outdated, and frankly underpowered, processor which would not have been capable of running the language models that we used at a reasonable pace. In addition, the EV3 robot would not have had enough storage space or memory speed to effectively store or access the local information and dependencies required for usage of our program. Due to this limitation, our group needed to use a modern computer as the brains behind the generative AI functionality.

Network

We chose to use fully locally runnable APIs and libraries for our project, which introduced hardware limitations. If we had committed to using OpenAI as our main generative LLM, we would have needed to connect our program to OpenAI's online API, which would have needed internet accessibility in order to create requests to their API. Despite this, using an online API would have allowed us to eliminate most issues related to hardware, as all of the computational resources would have been accessible remotely. However, because of time constraints and monetary limitations, we were not able to adapt our program to use a remote based LLM.

VI. Expansion

Network Adaptations

One of the main extensions we would have loved to implement in our project would have been effectively to eliminate the need for server/client connection from the EV3 to the program. We would have wanted to use a robot that has much newer and powerful processing capabilities so that it would have been able to have been used as a singular device, which would also make it more accessible. This would have helped us create a more immersive user experience, and would eliminate the need for users to own high-end equipment of their own. An alternative method to achieve better means for accessibility could have been adapting our program to give the user the option to either use a slower, locally run model for language processing like OLlama, or to use a remote based model like OpenAI.

Movement Capabilities

Another possible extension that we would have been interested in implementing would have been introducing spoken movement commands to the EV3 interface (ex: when telling the program to “move forward”, the EV3 moves forward). This would have been likely quite inapplicable in general while using an EV3, however if our project were to have been expanded upon, this could have been adapted to be much more useful. For example, this could have allowed for the creation of an AI assistant that could be able to complete simple movement tasks for the differently abled. With organizations like Boston Scientific innovating the world of physically capable robots, this sort of technology is around the corner.

Advanced LLMs

Building upon the LLMs foundation of this project, future expansions could explore several exciting avenues regarding the extension in the ability and capacity of LLMs. One direction involves investigating the integration of more advanced LLMs capable of reasoning and factual language understanding. This would enable the EV3 to engage in more complex conversations, potentially answering questions, providing summaries of factual topics, or even participating in simple storytelling with reasoning elements. Additionally, the incorporation of sentiment analysis could allow the EV3 to tailor its responses based on the user's emotional tone, fostering a more natural and engaging interaction. Furthermore, exploring the integration with vision recognition could empower the EV3 to connect spoken language with visual cues, enriching the conversation and allowing it to react to its environment. These expansions hold immense potential to transform the EV3 platform into a powerful educational tool for fostering not only basic coding skills but also a deeper understanding of AI concepts, natural language processing, and human-robot interaction.

Automated Speech Recognition

Currently, HomieBot relies on the user to detect when to listen for speech. The user must press the "enter" key on the keyboard to begin recording speech. However, we would like to automate this process using existing Automated Speech Recognition (ASR) technology, having Hombie, on startup, listen indefinitely for user speech. This would eliminate another level of user-computer interaction, thereby making it more accessible.

These expansions, ultimately, demonstrate the potential of human-robot interaction.

Conclusion

Regardless of future expansions, as the project currently stands, it demonstrates both the possibility of AI-powered conversations on the EV3, and the potential for more real-world applications. HomieBot demonstrates that AI assistants can be both helpful tools and friendly companions. And by eliminating the need for a visual interface and having simply vocal communication, HomieBot enhances accessibility, making AI tools more inclusive and accessible to a broader range of users. With such advancements, human interaction with both robotics and AI can be better utilized.

Bibliography

API reference—Python-ev3dev 2.1.0.post1 documentation. (n.d.). Retrieved May 3, 2024, from https://ev3dev-lang.readthedocs.io/projects/python-ev3dev/en/stable/spec.html

Bhat, N. M. (n.d.). pyttsx3: Text to Speech (TTS) library for Python 2 and 3. Works without internet connection or delay. Supports multiple TTS engines, including Sapi5, nsss, and espeak. (2.90) [Python; MacOS :: MacOS X, Microsoft :: Windows, POSIX]. Retrieved May 4, 2024, from https://github.com/nateshmbhat/pyttsx3

Console API — Rich 13.6.0 documentation. (n.d.). Retrieved May 3, 2024, from https://rich.readthedocs.io/en/stable/console.html

Hanschmann, L., Gnewuch, U., & Maedche, A. (2024). Saleshat: A LLM-Based Social Robot for Human-Like Sales Conversations (pp. 61–76). https://doi.org/10.1007/978-3-031-54975-5_4

Huynh, D. (n.d.). How to Build Your Own Voice Assistant and Run it Locally Using Whisper + Ollama + Bark | HackerNoon. Retrieved May 3, 2024, from https://blog.duy-huynh.com/build-your-own-voice-assistant-and-run-it-locally/

Introduction | LangChain. (n.d.). Retrieved May 3, 2024, from https://python.langchain.com/docs/get_started/introduction/

Kartunov, S., P, R., & M, S. (2016). Application of robots LEGO Mindstorms Education EV3 in the learning process of specialty Mechatronics. 25–27. https://doi.org/10.15224/978-1-63248-115-3-71

langchain_core.prompts.prompt.PromptTemplate—LangChain 0.1.17. (n.d.). Retrieved May 3, 2024, from https://api.python.langchain.com/en/latest/prompts/langchain_core.prompts.prompt.PromptTemplate.html

Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., & Gao, J. (2024). Large Language Models: A Survey.

NumPy documentation—NumPy v2.1.dev0 Manual. (n.d.). Retrieved May 3, 2024, from https://numpy.org/devdocs/

Ollama. (n.d.). Retrieved May 3, 2024, from https://ollama.com

P. Vashistha, J. P. Singh, P. Jain, & J. Kumar. (2019). Raspberry Pi based voice-operated personal assistant (Neobot). 2019 3rd International Conference on Electronics, Communication and Aerospace Technology (ICECA), 974–978. https://doi.org/10.1109/ICECA.2019.8821892

Play and Record Sound with Python—Python-sounddevice, version 0.4.6. (n.d.). Retrieved May 3, 2024, from https://python-sounddevice.readthedocs.io/en/0.4.6/index.html

queue—A synchronized queue class. (n.d.). Python Documentation. Retrieved May 3, 2024, from https://docs.python.org/3/library/queue.html

socket—Low-level networking interface. (n.d.). Python Documentation. Retrieved May 3, 2024, from https://docs.python.org/3/library/socket.html

Speech to text—OpenAI API. (n.d.). Retrieved May 3, 2024, from https://platform.openai.com/docs/guides/speech-to-text

Suno-ai/bark. (2024). [Jupyter Notebook]. Suno. https://github.com/suno-ai/bark (Original work published 2023)

The Python Profilers. (n.d.). Python Documentation. Retrieved May 3, 2024, from https://docs.python.org/3/library/profile.html

threading—Thread-based parallelism. (n.d.). Python Documentation. Retrieved May 3, 2024, from https://docs.python.org/3/library/threading.html