FRED: Friendly Real-Time Embedded Dino

Nick Brandis
Electrical and Computer Engineering
University of California, Los Angeles
Los Angeles, USA
niklbrandis@gmail.com

Hayden Friedman
Electrical and Computer Engineering
University of California, Los Angeles
Los Angeles, USA
haydenfriedman@ucla.edu

William Huang
Electrical and Computer Engineering
University of California, Los Angeles
Los Angeles, USA
whuang37@ucla.edu

Spencer Stice
Electrical and Computer Engineering
University of California, Los Angeles
Los Angeles, USA
smstice@ucla.edu

***Abstract*—The modern entertainment industry fails to offer products that bridge the gap between traditional toys and digital interfaces. Specifically, there exists no standstill toy that provides both the familiar comfort of playing with a stuffed animal and the rich stimulation of a video game. With the growing sophistication of artificial intelligence and large language models, the children's toy market has failed to adopt new technological developments and lags behind the progress of digital media. We addressed these shortcomings by creating an embedded system using only a Raspberry Pi 4, BerryIMU v3, small audio-jack speaker, and usb microphone that combines sensing technology and ChatGPT under a conventional aesthetic. Dovetailing speech processing, kinematic sensing, and network protocols, the achieved game quality of our friendly real-time embedded dinosaur (FRED) toy demonstrates the possibility of integrating a video-game experience into a traditional handheld toy and leverages ChatGPT to create a device with limitless replayability.**

***Index Terms*—generative AI, toys, LLM, games**

I. Introduction

Existing toys are too impersonal and static. Consumers are forced to make a choice when they purchase a children's toy; either they select something that has tactile pleasure and is aesthetically recognizable, such as a stuffed animal or action figure, or they select a device that provides a digital interface, such as a tablet. Both choices require the consumer to forfeit an important aspect of entertainment. The stuffed animal lacks the dynamic capability and replayability provided by the tablet, while the tablet does not resemble a prototypical handheld toy and lacks the personality of such a product. Though the two categories receive continued interest and developments, there has yet to emerge a robust device that offers a middle ground and merges the key features of both categories.

The primary objective of our capstone project was to establish this middle road by augmenting the traditional playing experience with a medium that seamlessly integrates sensing technologies and generative AI. We designed our project to appeal to today's children through interactivity. Playing sessions with our device revolve around the fabrication of a choose-your-own-adventure-style game. The choose-your-own-adventure game rests on the interplay between prompts and decisions. Essentially, the game will provide the user with cues and options that affect the trajectory of a given story based on the user's choice. A story will start with a general theme and will become more fleshed out and complex as interactions between the user and the game progress. The generation of this style of game can be successfully fulfilled by calls to ChatGPT, as the service can produce an effectively unlimited number of scenarios and create a coherent story out of any user input. Using this LLM approach combined with our additional game logic, the system can be repeatedly played and the child will experience a novel story each time. With ChatGPT serving as the foundation of our backend, we created a smart-speaker system with the primary constituents of a Raspberry Pi 4, an IMU, a microphone, and a speaker to facilitate this unique playing experience. All of the hardware is embedded inside the toy and connects wirelessly to a game server, which handles most of the computational load. After shipping this product, children would connect their toy to our central servers instead of requiring a local computer.

Each group member assumed a specific role and supervised particular aspects of the project throughout its development. William specialized in the networking components of FRED, creating a TCP server protocol and boot process to enable complete wirelessness. Hayden investigated the hardware of the project and worked on the overall assembly of the device. Spencer and Nick organized the game logic of FRED, with Spencer focusing on game logic, LLM and user response augmentation, and prompting techniques and Nick implementing the IMU and kinematic sensing.

II. State of the Art

A. Existing Market Competitors

Currently, there exist few to no products already released that can compete with our project. Our primary competition are "Fuzzible Friends", stuffed animals that use the Amazon Alexa to enable speech recognition and allow children to play voice-based games. While seemingly comparable, "Fuzzible

Thanks to the ECE 180D staff.

Friends" have a key difference in that they are tethered to the use of an Amazon Alexa which is both costly and tethered strictly to the home. FRED enables children to be able to freely move around to different areas and parts of the home as long as there is an internet connection. This difference alone enables a far higher degree of creative freedom and recreational creativity as the very setting of the toy can drastically affect the play experience.

B. Research Backing

Given how LLMs have only recently made it to the consumer market, we must also note that there exists very few systems, software or hardware, that reflect our existing work. Instead, we base our work on novel research conducted into the use of LLMs to generate games and play. Our primary backing is based upon the "Ambient Adventure Pipeline" [1], an experimental pipeline that proposes a novel way to structure LLM prompts to generate unique and engaging storylines that can integrate user action and response at once. While originally created as an interface for common video games, we believe that the concepts from this work are directly applicable to physical toys and have adapted much of the pipeline to fit physical user actions and prompting. Under this framework, we also acknowledge that the nature of an LLM does not provide a goal or cohesive play experience without external modifications. We address this problem by integrating traditional game design concepts through probabilistic round selection with point-based rewards to create a win condition and give purpose to the gameplay experience. Using our combined framework, we create a more guided storyline and gameplay experience while allowing for other forms of interaction that extend beyond voice (motion via IMU). Our framework also enables future developers to easily expand upon the existing gameplay storyboard without the need for cumbersome and domain-specific knowledge on LLM prompting.

C. Project Overview

We implement FRED using a novel client-server system integrating a base Raspberry Pi 4 with custom server side architecture to allow for future scaling and reduced latency in the entire game. Initially, we selected a Raspberry Pi Zero in tandem with a WM8960 extension header as our core hardware, though we encountered issues with streaming implementation and incompatibility with the soundcard and IMU. We ultimately relied on the Raspberry Pi 4 as our base due to its powerful Quad-core 64-bit ARM cortex and 4 GB of onboard RAM; the processing power of the Raspberry Pi 4 ensured that the tasks offloaded to the Raspberry Pi would not be limited by the device's computational power. Furthermore, the Raspberry Pi 4 included a 3.5 mm headphone jack and multiple USB A ports, making the addition of microphones and speakers a simple process. A 5000 mAh Pisugar2 Plus portable battery was harnessed to the Pi to enable fully wireless functionality for multiple hours on a full charge. The microphone we selected was a simple mini USB microphone, designed to fulfill rudimentary communication needs and minimize background noise and echo. The speaker we selected was similarly a standard mini 3.5mm speaker with 3W of power. Our choice of microphone and speaker centered around portability, and we were willing to forfeit power and volume to have a compact product and hardware that could easily fit in a stuffed animal casing. The IMU we integrated with our speaker system was the BerryIMU v3, and we mainly relied on its gyroscope features to actualize kinematic sensing and controls. The final piece of hardware in our design was the stuffed animal exterior; we used a PREXTEX plush dinosaur with a zipper to house the core speaker system. Our systems use a custom TCP server-client architecture to allow for low-latency streamed file and signal transfer between the server and toy. The toy acts as an audio recorder, playback device, and speech recognizer proxy which outputs all LLM text outputs in the form of a .wav speech file. Input audio is recorded and converted to text through the Google Speech-to-Text API and sent to the server through our TCP connection. The server then takes recognized text and converts it into an LLM prompt which is fed into ChatGPT 3.5 Turbo. The LLM response is then streamed back to the client in the form of Text-to-Speech converted audio files which are asynchronously played through the entire prompt-response process. A general diagram of our architecture is displayed in figure 1. More information on individual steps and their impact are provided in III.

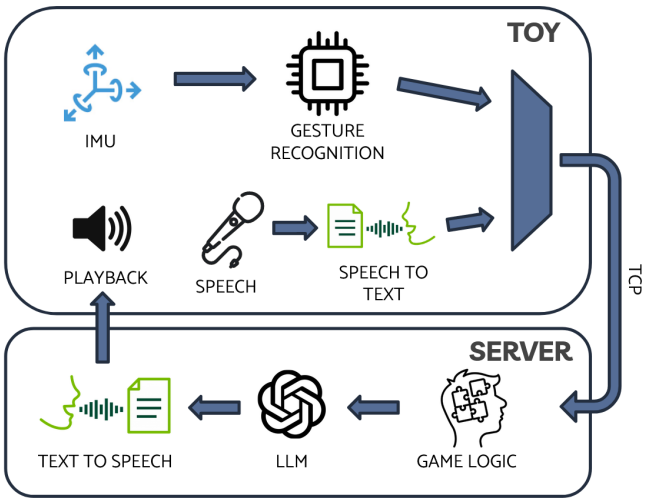

Fig. 1. Overview of the technical implementation of FRED.

D. Extendibility

While our primary goal is to implement and demo a fully functional end product in the form of a toy, we also wanted to develop a general framework to allow for future developments and extensions to our existing product. In order to enable this, we have written a full software suite in python that is easily extendible for developers. All LLM prompting is also abstracted into a simple text file interface to allow curious kids

or interested developers to experiment with prompt engineering without the need to tap into software interfaces. In this line of thought, we also have developed a full developer interface to enable textual based testing of the gameplay and storyline without the need for the actual toy.

III. Methods and Results

As described in II-C, our product consists of a highly involved cyclical pipeline involving a variety of different pieces of software and hardware. All of these steps were developed from multiple iterations of testing described below.

A. Hardware

1) Insufficient Compute: During the initial design of FRED, we began with the use of a Raspberry Pi Zero WH. While our software pipeline was fully functional on the Zero WH with short LLM responses, we found that longer speech audio files would maximize CPU utilization during the reception process in TCP. Furthermore, audio drivers for the selected sound card required the use of a fully locked audio file during playback. This both meant that our audio file had to be fully loaded into RAM and could not be simultaneously played back as TCP packets for the same file were received. Given these constraints, we found that any LLM response that was above 4 sentences (50-60 words) was simply far too large to be loaded onto RAM and would either crash or hang the client process until a manual restart. While these problems were partially corrected through future improvements to LLM output streaming, they could only be fully resolved with the use of the more powerful Raspberry Pi 4.

2) Incompatible Audio Hardware: Since our initial prototype used a Raspberry Pi Zero WH, we were forced to find external hardware that could enable audio playback and recording. We decided upon the WM8960 Audio Hat RPI Sound Card due to its relatively low cost and strong documentation. During the initial installation process, we found that the Zero WH was preconfigured to take its built-in HDMI output as the primary audio output device. Since we could not convert HDMI to a direct audio signal, we had to disable the HDMI output source to successfully install the WM8960 soundcard. While this initial implementation worked with the Zero HW, we found that the sound card had multiple driver and compatibility issues with the I2C pins of the Raspberry Pi 4 which were necessary for the IMU. Through rigorous testing, we believe that this issue is caused by the IMU sending small signals to the I2C headers of the main Raspberry Pi 4 which would tell the system that the I2C pins were taken and unable to receive any data signals from the connected WM8960. This hypothesis was then confirmed as we found we could enable either audio playback or recording if we started up the recording/playback software and connected the IMU as it was running. To address this issue, we used the full USB-A and built-in 3.5mm audio output jack of the Raspberry Pi 4 to use a small USB microphone and 3.5mm speaker for audio playback and recording. While our original intention in upgrading the Pi was to enable better computing, it also became a core component in enabling IMU integration and, in retrospect, a necessary upgrade even without the compute improvements.

3) Toy Selection and Assembly: The aesthetics of our design was a key aspect of the project; we needed to package the core speaker system in an attractive manner that invited prolonged interaction. The original idea for our project was to employ an exterior casing that resembled a traditional stuffed animal with tactile appeal. The main obstacle in achieving this goal was identifying an easy way to access and remove the electronic system from the exterior, and ensuring that the electronics would be safe from damage and violent movement. We decided that the simplest way to satisfy this requirement was to order a stuffed animal with a zipper pocket and fasten the interior with adhesives to secure the speaker system. Initially, we tested a soft, dog-shaped pencil pouch but found it was too small to accommodate the electronics and was not comfortable to hold. We followed up by ordering a 15-inch tall PREXTEXT dinosaur plush with a large zippered pouch. The toy's pouch was large enough to house the electronics without any noticeable lumps, and we secured the Raspberry Pi 4, battery pack, and IMU using velcro tape. The microphone and speaker were located at the top of the pouch near the head of the dinosaur and were held in place by being cinched in the zipper. The toy's zipper and the velcro tape provided an elegant way to minimize movement of the electronics and user access for charging and calibration. No other modifications were necessary to make the plush toy serviceable; the interior lining of the dinosaur was a soft mesh, allowing ventilation and preventing the Raspberry Pi 4 from overheating, and the shell was not so thick that it severely muffled output from the speaker or obscured user audio to the interior speaker.

B. Software

1) Incompatible Audio Drivers: During our hardware installation process, we found that the existing Raspberry Pi ecosystem did not come prebuilt with audio drivers. While common Linux Distros now all handle audio setup in the initial operating system setup process, we found that the difference across Raspberry Pi models made a pre-installed audio driver system too general for most end-users' purposes. Thus, we tested a series of different drivers and software implementations. We first began by using AlsaAudio, the general Linux audio driver package, to test our hardware installation. Upon confirming both the speaker and microphone were working properly, we then tested the following set of packages to programmatically use audio: AlsaAudio-python was the logical first choice due to its relation to our underlying AlsaAudio drivers but was found to be incompatible with modern versions of Python and selected Python packages. Furthermore, AlsaAudio could only be run on the Raspberry Pi and would need to be swapped out on other devices if the developer wanted to test audio playback locally. PyGame worked on both the Raspberry Pi and server but had a long delay on startup that created a poor user experience. PyAudio worked on both the Raspberry Pi and server and

had no delay on startup but was found to occasionally hang for over 10 seconds if the audio file being played had just been received from the TCP connection. We finally settled on calling AlsaAudio commands through the subprocess which minimized delay and had no compatibility issues with any of the preexisting software used in the client code. One drawback to this approach is that it is incompatible with any device without AlsaAudio (Linux computers without AlsaAudio and Non-Linux computers). Thus, we also designed a PyAudio alternative that can be switched out on runtime for developers to locally test audio playback.

2) Speech-to-Text Methods: We used Google's speech recognition API for converting the child's voice recorded input into a textual representation of the words expressed. Before settling on this design choice, however, we experimented with some other methods. Initially, we tried a smaller recognition model called Vosk that runs completely locally, hoping it would have lower latency. However, due to memory constraints on the Raspberry Pi, we were unable to download Vosk's largest and most accurate model, and the smaller model we incorporated had much lower accuracy. Resorting to using an API, we tried using Google's version of the API that takes an audio file as input, rather than facilitating the actual recording of voice input. This worked remarkably well, but constricted us to using our own means of recording audio on the Raspberry Pi. We had been using arecord of AlsaAudio, which forced us to define a fixed length of recording before generating a wav file. In an effort to allow the child to speak for as long as they wanted, we experimented with the listen() and adjust_for_ambient_noise() functionality of the speech_recognition library. While this necessitated a migration of speech-to-text code to the client, it surprisingly had a minimal effect on latency. In fact, since the data being sent to the server was now the already-processed text instead of a large audio file, the average overall latency decreased by around 0.01 seconds.

3) Engaging Voice Output: We leverage gTTS as our Text-to-Speech interface to convert LLM responses into audio files. During initial user testing, we found that the generated voice was very robotic, leading to a more sterile and impersonal gameplay experience. We addressed this problem by designing a set of audio signal processing steps that pitched up and sped up the response to create a more toy-like sound. Our initial implementation was done directly in Python and found to be extremely inefficient, taking $\approx 1.5s/sentence$. We addressed this problem by moving all audio processing to FFMPEG and calling it through its related Python API wrapper. Our final processing steps are as follows: Multiply the generated audio file's tempo by 1.3. Rubberband Filter of pitch 8/12 (transpose up 4 semitones). Convert to pcm_s16le ultrafast. These preprocessing steps also stood to lower the generated file size and decrease the amount of time spent in TCP transfer.

4) IMU Implementation: A key objective of our project was to offer kinematic input. This was accomplished through the incorporation of an IMU into our core design. The BerryIMU was configured to the I2C pins of our Raspberry 4, which did not interfere with the functionality of the microphone or speaker. We used the accelerometer and gyroscope data of the X and Y axes to create a naive motion controller. We extracted information from the axis associated with horizontal movement and set a threshold to identify if the IMU was turning to the right or the left. Additionally, we set a threshold on the sum of the magnitude of differences of X and Y values over time to determine if the IMU was being rapidly shaken. Logging on this motion is shown in figure 2.

```
IMU shake round.
Loop time:  0.01        # Sum Distances  0.00 #
Loop time:  0.04        # Sum Distances 20.55 #
Loop time:  0.04        # Sum Distances 23.51 #
Loop time:  0.04        # Sum Distances 25.15 #
Loop time:  0.04        # Sum Distances 29.09 #
Loop time:  0.04        # Sum Distances 33.78 #
Loop time:  0.04        # Sum Distances 41.06 #
Loop time:  0.04        # Sum Distances 165.33 #
Loop time:  0.04        # Sum Distances 210.66 #
Loop time:  0.04        # Sum Distances 304.59 #
Loop time:  0.04        # Sum Distances 340.72 #
Loop time:  0.04        # Sum Distances 350.60 #
Loop time:  0.04        # Sum Distances 516.30 #
Loop time:  0.04        # Sum Distances 549.28 #
Loop time:  0.04        # Sum Distances 693.69 #
Loop time:  0.04        # Sum Distances 745.27 #
Loop time:  0.04        # Sum Distances 756.45 #
Loop time:  0.04        # Sum Distances 922.98 #
Loop time:  0.04        # Sum Distances 954.71 #
Loop time:  0.04        # Sum Distances 1066.91 #
First X angle: -17.41
First Y angle: -30.90
Sum distances: 1066.91
Shake detected.

Shake detected

Signal IMU_SHAKE sent successfully.
```

Fig. 2. Example output of the IMU shake function, where the IMU collects twenty rounds of accelerometer and gyroscope values and sums the distances to eventually compare to a threshold "shake" value.

As data streams from IMU are notorious for unpredictable and noisy output, we experimented with several mathematical filters for extracting the true, important features of the data. Specifically, we tested a Kalman filter and a complementary filter, and compared the effects of these to just using raw accelerometer and gryoscope data. We found that both the Kalman and complementary filters produced a cleaner output than raw data, but the Kalman filter was more computationally expensive due to its recursive nature. Based on this experimentation, we decided to commit to the complementary filter as our filter of choice. We also experimented with dynamic IMU calibration, where the calibration API was called before collecting any IMU data. The motivation was to obtain calibrated maximum values for accelerometer and gyroscope values on all axes to dynamically adjust thresholding values of left/right and shake logic. This worked well, but added close to 1 second of latency for the first IMU round of every game, which we later decided was not worth it despite the

minor measurement improvements. With the two identifiers, right vs left and shaken, being more reliable, we were able to incorporate this functionality into our game logic. On the server side, there is a chance that each round is an IMU round, in which case we randomly select between a left/right or shake round and send a corresponding prompt to ChatGPT. To inform the client, a unique signal is sent over TCP, and the client updates its game loop to call our IMU APIs instead of the traditional recording through the microphone. The results of the IMU functions are calculated and a corresponding signal (e.g. IMU_TURN_LEFT) is sent back to the server, where ChatGPT is given a custom prompt depending on the received signal. For example, if the received signal is IMU_NO_SHAKE, our prompt tells ChatGPT that the child did not shake the toy quickly enough and the game is over. ChatGPT is then instructed to come up with a sensible explanation for why the child lost the game given the context, and tell the child that the game is over in a kid-friendly manner.

5) LLM Streaming: In our initial prototype, we developed a single-threaded implementation of our project pipeline which would wait for the full LLM response before beginning the Speech-to-Text and TCP transfer process. This design had one key limitation: The delay of the LLM API call was dynamic and could extend depending on the response length of the LLM. We saw this fallback directly affect our product's reception during the Quarter 1 demo where the LLM would take upwards of 10 seconds to generate a response, creating long periods of silence that were distracting and off-putting to the audience. This was a major wake-up call to improve the general user experience before expanding upon the features and creating an enticing core product before stretching ourselves too thin. Our final solution to the LLM problem leveraged the streaming feature of the API to asynchronously stream LLM outputs as it was generating. Instead of a single-threaded solution, we moved the LLM generation to a separate thread which would read the output and combine them into full sentences before placing them into a buffer in a producer-consumer architecture. The main thread would preprocess sentences and transfer them through TCP individually as the LLM was generating the rest of the response. On the client, each received audio file was placed in a buffer which was then sent to a separate audio playback thread where audio files were played in order. We could ensure the order of files through the use of TCP as our selected network transfer algorithm (ensures ordered packet reception). This approach made the LLM generation delay constant time (the amount of time to generate one sentence) as the rest of the output could be generated during audio playback.

6) Latency Testing: As a part of our work to create a more enticing core product, we also tested a set of different software suites to see if we could further optimize processing delay. A series of ablation tests were conducted to select the best combination of softwares to minimize delay. Each trial was run 10 times to ensure reproducibility. The full data and final selections are shown in I.

As discussed earlier, the speech recognition latency was slightly improved by moving to the client. We hypothesize that this is due to lower latency of text transfer to the server as opposed to transferring a large audio file, despite the reduced CPU speed on the Raspberry Pi. For Text-to-Speech, the task was relatively computationally inexpensive, and was therefore only marginally faster when running on the server. The TTS processing was where we experienced some drastic differences. Running our initial Python TTS algorithms on the client was almost three times as slow as on the server. Furthermore, the transition to FFMPEG for TTS processing decreased latency by over one order of magnitude. Though expected, the use of the streaming option of ChatGPT was also responsible for a huge cut in delay. Finally, the sound card initialization code was several orders of magnitude faster for 2 channels at 44.1kHz than for a single channel at a 16kHz sampling rate.

7) Connecting to Network: In designing our user experience, we also had to contend with the initial setup process. Most testing was done by spoofing a TCP connection through a wire to remotely access the Raspberry Pi from an external source. Yet in deployment, our toy had to be able to connect and run to any device without the need for developer knowledge. We also wanted to avoid the use of external applications or screens to reduce product cost and development time. Thus we developed a novel boot procedure that leverages a single flash drive to connect and run the toy. The user is expected to take a flash drive and enter their network information into a predefined file on the drive. Then the user plugs in the flash drive into the Raspberry Pi and powers on the toy. During the boot process of the Raspberry Pi, custom startup code will mount the drive using fstab and then read in the network configuration information. This network configuration is then directly written into the built-in wpa_supplicant.conf file and reloaded to reconnect to the new network. Finally, the boot script starts up the client and awaits a server connection to begin playing. We found that this procedure drastically reduced the amount of setup time and received large amounts of praise for its simplicity of use and cost-effectiveness.

8) LLM Safety: Given the nature of LLMs, we must acknowledge that these systems are unpredictable and potentially dangerous at times. This is especially apparent given the phenomenon of "hallucinations", where LLMs will generate a response that is completely unrelated to the given response. While we were not able to directly modify model architecture, we still must contend with these issues to create a safe product. We did so by implementing an extensible word censoring system, which identifies key phrases in the LLM's output. Upon identification, users can choose to either replace the word or ask the LLM to regenerate the output before sending it to the toy. We believe our approach offers a logical middle-ground to strict parental control rules, enabling the user to modify and add phrases that may be inappropriate on a case-to-case basis.

9) Save States: One key advantage of leveraging ChatGPT is its highly developed set of data structures and API endpoints. Since our entire backend hinges on API calls to

TABLE I
A BREAKDOWN OF LATENCY TESTING OUTCOMES.

Step	Parameter	Run Location	Mean Latency (s)
Speech Recognition	Google	Raspberry Pi	**1.01**
Speech Recognition	Google	Server	1.02
TTS	Google	Raspberry Pi	0.33
TTS	Google	Server	**0.32**
TTS Processing	Python	Raspberry Pi	11.91
TTS Processing	Python	Server	4.23
TTS Processing	Ffmpeg	Server	**0.37**
File Transfer	Transfer time	RBP->Server	**0.45**
LLM	GPT 3.5 Turbo Unstreamed	Server	3.00+
LLM	GPT 3.5 Turbo Streamed	Server	**1.58**
Sound Card Init	1 Channel 16kHz	Raspberry Pi	4.46
Sound Card Init	2 Channel 44.1kHz	Raspberry Pi	**0.01**

ChatGPT, we can directly pull the chat history as a JSON file to effectively save the state of our game and rerun at any time. Thus, our server architecture enables users to not only load and save previous chat histories from older runs with the toy, but also generate new save states by interacting directly with ChatGPT to create new gameplay experiences that we have not predefined. These JSON files can also work as a direct input into image diffusion models which can generate interesting visuals of each game in future works.

10) Game Logic: The LLM is largely responsible for generating the content for the game, therefore only a rudimentary skeleton was coded for the game's progression where we handle the raw sensor inputs from the child and convert them to suitable messages for the language model. Then, after receiving the next story segment from the model, we modify it as needed to enhance the experience for the child. Thus, our focus beyond integrating the model itself was dedicated to prompt engineering, implementing different types of interactions for the child, and other game features.

We hardcoded two initial questions for the system to output: the first question will ask the user for their name and the second question will ask the user for an overall theme for the game's story. Since the LLM can generate a story based on nearly any imaginable theme, the child could explore space, travel back in time, or role-play a career they may be interested in. We noted here that if the child selects an inappropriate theme, the LLM is generally capable of identifying such a problem and telling the child to select another option. Past these two questions, all of the following cues are created by the LLM. The LLM will present hypothetical scenarios and weave together an adventure story regarding the chosen theme, offering the user control over how the story progresses. Noting that the model is probabilistic in nature and that children will typically respond to the same scenario differently, our system offers effectively infinite story possibilities that are all generated in real time as the child progresses. The pairing of an LLM's story content output and the user's response can be classified as a round, and we implemented 5 different types of rounds. The inclusion of a variety of types of rounds creates an additional layer of dynamic interaction in the game on top of the LLM itself, thus even if the language model happens to generate similar scenarios in repeated trials, the way the game progresses is likely to be different. The first type of round is a "standard round" and consists of the LLM providing the user with multiple options for the story's advancement. This is the most simple of the round types and is the first type we implemented to test the game. The second type of round is a "star round", where if the user makes a good choice they are awarded a "star"; if the user collects a predefined number of stars, they win the game; and the length of the game is largely determined by this predefined number. To determine if a child's choice is considered a good one, worthy of obtaining a star, we implemented a separate mechanism. First, we realized that it is necessary to determine which choice the child selected (as a number). For example, we wanted to know if the child selected option 1 or option 2. This was implemented through an additional request to the language model that is kept separate from the rest of the core story. In other words, we crafted a prompt, excluding the rest of the game history, asking the model to output a single number corresponding to the option number selected by the child.

After a few iterations of prompt types, we were able to successfully generate the correct selection number upwards of 95% of the time. To handle cases where the detection is unsuccessful, we determine that option 1 was selected by default. We note that this is a relatively simple solution to handle this problem and would like to improve on it moving forward. The third type of round is a potential "game-ender" round, where if the player makes a bad choice, determined the same way we determined if a choice was good, the result is ending the game in defeat. The fourth and fifth types of rounds rely on kinematic input through the IMU rather than the user's voice; the fourth type prompts the user to make a decision based on directionality and turning the toy left or right, while the fifth type prompts the user to shake the toy. The player cannot earn stars during the turn rounds but can during shake rounds if they shake the toy fast enough. Additionally, the player will lose the game if they fail to shake the toy quickly enough. The combined effect of these types of rounds is to provide: an objective for the user, through the star currency, replayability, through the element of probability/chance, and physical engagement, through kinematic sensing. To realize

these various round types, we augment the child's response to a given scenario with information to the LLM regarding the next round type. For example, if the next round is determined to be an IMU shake round, after the server receives the child's response from the previous round, we add a system message to the LLM instructing it that during the next round of play, the LLM should create a scenario in which shaking the toy makes sense. In this way, we have created a set number of prompts to augment the LLM message with, yet the prompt does not include any hard coded information about specifically how to intertwine the input instructions with the current story, thus opening the creative generation ability of the model for the player to enjoy. We use a similar prompting technique to achieve various effects during the gameplay, all of which have predetermined prompts to the model while allowing it to freely generate the next scenario as it sees fit. Given the relative success in realizing the vision of our product, this augmentation technique seems to have at least some efficacy. Finally, to add an additional layer of dynamics to the game, the round type is selected probabilistically. Near the beginning of the game, it is most likely that the rounds will be either standard rounds or IMU turn rounds, since in these rounds the player can neither get closer to winning or losing and would like to be introduced to the general mechanics of the game. Then, after a set number of "safe rounds", the other round types have a chance to be selected. At first, there is a fairly low chance that any of these rounds, which include potentially earning stars or losing the game, will be chosen. However, as the child plays through more rounds, the odds of the rounds occurring slowly increases, thus making it more likely that the child will both win and lose the game. Clearly, one could adjust the probabilities and exact details of the incorporation of such game logic to adjust the difficulty and length of the game, both of which are areas that could be further explored in the future.

To test the quality and perform iterations on the prompting techniques, we implemented a local test/debug mode where the text outputs and useful debugging information is displayed without the need for the entire system as seen in figure 3. In fact, the mode is run on the server itself without the need for the client. Using this mode, iterating on game logic ideas and testing new features became dramatically faster and easier. This was important considering that at times, the language model simply would not oblige with the given prompt or with the order of the prompts. Use of this debugging mode allowed us to quickly modify the order/prompt types and minimize the chances that the model would not produce a desirable output. This process was somewhat frustrating and time consuming, but eventually landing on useful, working set ups was rewarding. Using a combination of this feature and full system testing at integration time, we were able to create a functional prototype of our game logic system.

C. User Testing and Feedback

The final Q1 demonstration made it apparent that latency was the most glaring issue afflicting our design. We remedied this by implementing the LLM streaming feature and relocating speech processing from the server side to the client side of our system. The second flaw exposed by the demonstration was the limited time allotted to user input; our prototype only allowed the user a strict 3-second window to speak. This was resolved through dynamic speech capturing; we employed thresholding to enable a constant stream of input with termination contingent upon a determined level of quietness. User testing revealed that the setup process for our project was involved and confusing. In particular, waiting times and locations for certain processes were ambiguous, retrieving API keys was unclear, and no information was provided for establishing the client. These concerns were addressed by modifying the user manual and README for the project, including ample information to make the initialization procedure as transparent as possible.

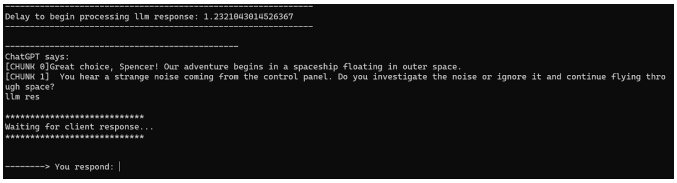

Fig. 3. Local debug mode used to quickly test game logic and prompting techniques without the need for the entire system to be connected.

D. Project Management and Distribution

Though each member had a hand in nearly every aspect of the project, the majority of tasks were compartmentalized and supervised by a subset of the team. In the initial stages of the capstone, William and Hayden investigated the base hardware to constitute a prototype speaker-microphone system, while Spencer and Nick began organizing skeleton code for the game logic and experimented with different LLMs and speech-to-text APIs. On the hardware side of the project, William and Hayden dedicated a good portion of Q1 configuring the WM8960 extension header with the Raspberry Pi Zero, encountering and overcoming the host of obstacles previously discussed. Simultaneously, Nick and Spencer developed an Alpha of the adventure game and selected Google as the means for speech processing and ChatGPT as the LLM to generate the game's content. At the end of the quarter, the two teams evaluated and shared their progress to construct a naive first iteration of the project that could complete a simple demonstration. The team identified latency as the project's critical hindrance, and all members dedicated their efforts to resolve this issue. William coded a multi-threaded full-streaming protocol and the remaining members worked towards implementing this protocol with the hardware on hand. Nick and Spencer ordered a Raspberry Pi 4 to replace the Raspberry Pi Zero to fulfill this task, and the team split up once again to satisfy their respective roles after the streaming feature was completed and we achieved a low-latency demonstration for the midterm presentation. Collaboration and unification of the team members occurred once again after Nick concluded

that the WM8960's soundcard and IMU were incompatible, necessitating everyone's efforts to identify a resolution. We collectively decided to substitute the header with an external microphone and speakers; members continued refining their assigned portion of the project before congregating to fully integrate the systems for the final demonstration. Team interactions and overall collaboration followed an overarching pattern; members would divide themselves into subteams to work on a broad component of the project, such as software and hardware, and merge back together in the face of major obstacles or to combine these broad components. As the project progressed, these subteams were further divided into individual members working on more specific auxiliary features such as game logic and the product's initiation. The exchange of information between members was primarily executed through in-person meetings, both inside and outside of the laboratory, and a central group chat. Progress reports and updates were communicated through the chat and remote meetings on Discord, while in-person meetings between two or more members frequently occurred to address key problems and combine work.

IV. Conclusion

The development and fruition of FRED demonstrate the possibility of combining modern digital tools and sensing in a traditional package to satisfy all aspects of children's entertainment. We were ultimately successful in creating a dynamic interface that processes motion and audio inputs to generate an interactive story game. Despite a host of obstacles faced during the development process, we created a robust and consistent system that fulfilled our initial goal of bridging a traditional and futuristic toy. The most restrictive limitation that held back the enjoyability of our product was the presence of latency. Though we were able to reduce latency time from an initial time of over 8 seconds to a final time of under 4 seconds, calls to the API and the speech-to-text conversion process will occasionally present noticeable delays that impede the flow of play sessions. Another large challenge we faced was the calibration and configuration of the IMU. Rudimentary kinematic controls were successfully implemented, but the IMU will occasionally flip polarities or sensitivity, warranting calibration, and threshold modifications. Consequently, the future development of FRED should be centered around the inclusion of more robust hardware. A more sophisticated IMU along with a sensitive microphone would likely eliminate the minor sensing idiosyncrasies we faced and warrant more complex inputs. Additionally, migration to a universal server rather than one established on a personal machine would potentially reduce latency even further and significantly simplify the setup process for the user. Future goals for FRED would also include leveraging more features of a generative AI model; we are interested in the possibility of creating images from the game's story and presenting them to the user on an external mobile application or website. We also want to include more types of games with our device outside of the standard choose-your-own-adventure. The only limitation of the game logic is the amount of features we could incorporate within our time constraints, but the nature of LLMs grants the opportunity to continuously build off the existing game structure. Finally, we wish to launch FRED on a larger scale; we would like to have more options for the exterior shell (sizes and shapes other than the current dinosaur) and implement shared sessions between multiple FRED owners.

Acknowledgment

Special thanks to the ECE 180D staff and students who provided invaluable feedback and support through the entire project timeline.

References

[1] Z. Chen, E. Zhou, K. Eaton, X. Peng, and M. Riedl, "Ambient Adventures: Teaching ChatGPT on Developing Complex Stories." arXiv, Aug. 03, 2023. doi: 10.48550/arXiv.2308.01734.