UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

DIPLOMA PROJECT

A Robot with the Brains in the Cloud
Giving a LLM a quadruped robot body

Bîrjovanu Ioan-Cristian

Thesis advisor:

Conf. Iuliu Vasilescu

BUCHAREST
2025

CONTENTS

ABSTRACT

This paper showcases the design and implementation of a quadruped robotic platform controlled by a Large Language Model (LLM) hosted in the cloud. The system aims to achieve a natural interaction between human and robot by offloading the high-level reasoning and decision-making tasks to a LLM, while the robot's onboard components handles low-level movement control. The project's architecture integrates a Raspberry Pi as the central piece, interfacing with an Arduino-based motion controller and sensors, including a camera, distance sensors, and a microphone.

A Python script collects the sensor data, including visual input and voice commands, and forwards it to the GPT Assistants API. The assistant processes this data and returns a response that includes movement instructions and a reply, enabling the robot to respond both verbally and physically in real time. The quadruped is based on a 3D-printed structure inspired by the Boston Dynamics Spot Robot, using 12 servomotors controlled via a PCA9685 module and an Arduino Pro Micro.

The paper highlights the two key components: a cloud-integrated AI control pipeline, and the robotic system's design. Results show that the robot is able to understand spoken commands, respond accordingly, and demonstrate interactive behavior. This prototype provides a foundation for exploring LLM-powered physical agents and opens up possibilities for more advanced human-robot interactions in the future.

1 INTRODUCTION

Recent advances in Artificial Intelligence and hardware computing power are allowing robots to achieve higher reasoning and more advanced behaviors by using powerful models. For example, Boston Dynamics' Spot robot [1] demonstrated the potential of autonomous quadruped robots in real-world scenarios.

Sadly, replicating those capabilities on a hobby-grade project is difficult due to the limited on-board computing capacity and complex software requirements. So, a popular workaround for lower computing capabilities and complexity is to offload the processing to the cloud, by using large language models (LLM's) to handle high-level perception, reasoning, and decision-making.

This project aims to enhance a relatively low-cost 3D-Printed quadruped robot with a cloud-based intelligence, effectively giving the robot a "brain in the cloud" that can understand spoken commands, speak with users and plan actions. By integrating a modern LLM (OpenAI's GPT 4.1 [2]) as the robot's reasoning system and using a simple onboard microcontroller for motor control, complex interactive behavior can achieved on lower-end hardware. The increased popularity of cloud AI combined physical robotics has the potential to greatly expand the accessibility to intelligent robots in research and education.

1.1 Motivation

This project started from an idea I got while watching a video on the OpenAI API [3] and how it could be used to control and navigate a robot around its environment [4]. The idea of creating a robot that would have the awareness of the average AI assistant was very intriguing at the time. After deciding upon using the OpenAI Assistants API for the so-called "brains" part of this project, I started to look for ideas for the physical frame, the actual robot body. I was not sure what type of robot I wanted for this project, but I always thought about building a quadruped robot, similar to the Boston Dynamics' Spot [1], and after searching for a "3D-Printed robot dog", I stumbled upon the SpotMicro project [5].

1.2 Aim and Objectives

The primary aim of this project is to design and implement a quadruped robotic system that combines a physical robot body with a cloud AI based on a Large Language Model(LLM). The robot should be capable of understanding natural language voice commands, responding verbally and with movement actions. To achieve this goal, I had to complete the following objectives:

- Developing a **cloud-integrated software stack** that can collect multimodal data (audio, video, distance), perform speech-to-text transcriptions, communicate with a GPT-based assistant in the cloud, and interpret the assistant's replies, including movement instructions.
- Designing a **prompt** for the GPT assistant so that it produces useful and relevant responses in a structured JSON format, that can be interpreted by the robot control system.
- Building a **quadruped robotic platform** inspired by Boston Dynamics' Spot, using 3D Printing, servo motors for movement (3 per leg) and a microcontroller for the servo control.
- Implementing **motion control firmware** on the microcontroller that can parse the high-level movement commands from the assistant and, move the leg servos accordingly (walking, turning, gestures).
- Integrating the **sensors** (camera, ultrasonic distance sensors, microphone) on the robot and adding their data in the assistant's message, enabling basic environmental awareness.
- Evaluating the system in **example human-robot interaction scenarios**, focusing on the robot's ability to execute voice commands, engage in simple dialogue, and move accordingly.

1.3 Contribution

This project brings several contributions related to the combined field of AI and robotics. Firstly, it demonstrates a practical architecture for a *cloud-enhanced robot*, where a local system uses a remote LLM for advanced natural language processing and decision-making. Unlike traditional robots, with all their processing taking place on-board, my project exemplifies the concept of **cloud robotics** [6], offloading computationally intensive AI tasks to remote servers.

Secondly, it contributes with a **software pipeline** written in Python[7] that combines Whisper speech-to-text transcription and GPT-4-based dialogue with control logic, illustrating how to integrate these advanced AI services into embedded systems.

Thirdly, this project showcases a custom **hardware design** for a 12-DOF quadruped robot (mechanically and electronically) and an Arduino **control firmware** that serves as a helpful robot agent that outputs natural language combined with action commands.

Finally, this work provides an analysis of the challenges (network delay, real-time control difficulties, sensor noise, and servo calibration) that I encountered while working on this project, along with documented solutions and recommended future improvements.

1.4 Results

The outcome of this project is a functional prototype of an interactive quadruped robot assistant. The robot can understand spoken user commands thanks to Whisper's advanced speech recognition[8]. It can engage in a simple conversation with the user using GPT-4, which provides reliable responses to a wide range of questions. The GPT assistant is prompted to output a structured JSON in its replies, that include movement movement commands. Those commands are parsed and sent to the microcontroller, which in turn executes the corresponding servo movements. The robot can perform a set of movements, such as: walking forward, backward, turning left or right, standing, sitting, laying flat, waving and holding paw. Basic obstacle avoidance is demonstrated: if an object is detected by the distance sensors or observed in the camera view, the assistant will choose to not walk right into it, but to either avoid it using a turn command, or ask for a different instruction.

A sample interaction shows the robot engaging in a short dialogue, following a series of movement instructions given by the user, and responding accordingly when an invalid instruction was given (trying to avoid an obstacle that got too close). These results, detailed in the 5th chapter, validate the usefulness of integrating a cloud-based AI controller with a physical robot body. The system's limitations were also observed - notably the approximately **5 second delay** between speaking a command and the robot's reply and action, due to network speed and API processing times.

1.5 Structure

The rest of this paper is organized as follows:

- Chapter 2 gives background information on cloud robotics, large language models, the Whisper speech recognition model, and existing quadruped robot frameworks that inspired my design.
- Chapter 3 describes the implemented software architecture, including the sensor interfaces, the voice transcription process, the integration with OpenAI's GPT assistant (prompt design and JSON output parsing), and the logic for sending commands to the microcontroller.
- Chapter 4 details the robot's hardware design and control: the mechanical construction of the quadruped (CAD and 3D printing), the servo control scheme using a PCA9685 driver, the electronics and wiring, and the Arduino program that interprets commands and generates coordinated leg movements. The physical challenges posed by the servo calibration and the gait stabilization are also mentioned.
- Chapter 5 presents experimental results and a discussion on the system's performance. We walk through an example user-robot conversation and analyze the response times, accuracy of the transcription and GPT responses, and the reliability of movement ex-

ecution. It highlights the current limitations and the trade-offs made in this design (for example, the dependence on a reliable internet connection). In the last section, the future work directions are presented, such as improving autonomy with on-board processing, and expanding the robot's list of skills.

- Chapter 6 brings and end to the paper, summarizing the achievements made with this project.

The paper also includes Appendices with additional details: the full prompt given to the GPT assistant defining the robot's control scheme and command format (Appendix A), and a sample conversation illustrating the robot's behavior (Appendix B).

2 BACKGROUND AND RELATED WORK

2.1 Cloud Robotics and AI Assistants

The concept of cloud robotics refers to a robotics architecture where a big part of the computation and data storage are offloaded to cloud infrastructure, enabling robots to benefit from powerful remote resources via network connection. This contrasts with traditional robots that rely only on on-board processing. By using the cloud, even smaller robots can have access to advanced processing – for example, advanced AI models or shared knowledge bases – that would be impossible to run locally on limited hardware. Early versions of cloud robotics included networked robots sharing maps and skills (e.g. RoboEarth project[9]) and offloading heavy tasks like image processing to data centers.

In recent years, the expansion of IoT[10] and faster wireless communication has made cloud-connected robots more practical, and the model is seen in commercial products like Amazon's Alexa devices[11] and Google's Home assistants[12] (essentially stationary robots with voice interfaces that rely on cloud AI). This project builds on this idea: the quadruped robot uses the cloud for its AI assistant and voice transcription, which is a form of AI-as-a-Service[13] for robotics. One main advantage for this approach is that improvements or upgrades to the AI (e.g. a newer language model) can be used via an API call without changing the robot's hardware. The trade-off is that the robot becomes dependent on network connection and suffers from higher latency in control loops, which must be addressed in the system design.

In particular, AI assistants[14] powered by Large Language Models have become accessible via APIs (e.g. OpenAI's GPT-based services, including the new Assistants API[15]) that allow integration of conversational AI into applications. These assistants can follow complex instructions, retrieve information, and perform reasoning. By defining a suitable system prompt (role and instructions), we can customize the assistant's reply in specific scenarios. In this project, I created a robotic cloud assistant – a custom GPT-4-based agent designed to interpret sensor data and user requests, then output both a conversational response and a robotic action. This approach is inspired by emerging frameworks like the OpenAI Function Calling capabilities[16], where the model can return structured data (like function arguments or a JSON) in addition to text. Rather than hard-coding robot behavior, we "prompt" the AI with the robot's capabilities and let it decide the appropriate action, which is an example of using an LLM for high-level robot control. Prior works have started exploring LLMs in robotics: for example, Google's SayCan[17] and related research combine LLMs with robot perception to plan actions from instructions, and community projects like MachinaScript[18] have demonstrated controlling Arduino-based robots via GPT-generated JSON instructions.

MachinaScript, in particular, outlines a pipeline where a central “brain” (a computer or a Pi) listens for input (voice or visual), uses an LLM to generate a sequence of actions in a structured format, parses these instructions, and then sends them to microcontroller “body” which executes them. My system uses a very similar pipeline, confirming that cloud-based brains for robots are a popular research direction.

2.2 Large language Models and GPT-4

Large Language Models like GPT-3.5 and GPT-4 are neural networks trained on massive text databases, capable of generating human-like text and performing a range of language tasks. GPT-4, introduced by OpenAI in 2023, is particularly notable for its ability to handle nuanced instructions and even accept image inputs. It exhibits what OpenAI describes as *“human-level performance on various professional and academic benchmarks”*, and is more reliable and creative compared to its predecessor. For the purpose of controlling a robot via natural language, GPT-4’s abilities fit well: it can understand context, reason about user requests, and produce intelligent multi-part answers. More importantly, GPT-4 can be guided via *prompt engineering*[19] to output structured data. We utilize this by instructing GPT-4 to format its responses in JSON with specific fields (like "speech" and "action"), effectively turning the free-form text generation into a fixed command interface for the robot.

One challenge with using GPT-4 (or any LLM for that matter) in robotics is ensuring correctness and safety of its outputs. LLMs are stochastic and can produce irrelevant or incorrect responses if not guided properly. They also have no built-in knowledge of a robot’s physical constraints. We fix this by carefully configuring the system prompt given to the model with explicit instructions and examples, explaining exactly which movement commands are available and what they mean. This aligns with how the author of the OpenCat framework[20] integrated ChatGPT with the Petoi Bittle robot dog[21]: by *“training ChatGPT to understand what commands Bittle supports”*, he made sure that the AI would respond within the robot’s abilities. The result, as reported, was a robot that could communicate and follow instructions with personality, dancing and performing tricks on command.

Similarly, this project’s GPT-4 assistant is made aware of the set of actions that the quadruped can do, and it’s instructed to avoid any output that doesn’t match the JSON format or tries something unsafe (like walking forward into an obstacle). GPT-4’s wide training data also means it can handle general conversations – if the user asks a question unrelated to movement (e.g. a trivia question), the model can answer factually while still remaining “in character” as a robotic dog.

2.3 Voice Recognition with Whisper

For the robot to receive commands in natural spoken language, a reliable Speech-To-Text (STT) engine is required. I selected OpenAI's Whisper model for this task, thanks to its high accuracy and tolerance to noise and accents. Whisper is an open-source neural net trained on 680,000 hours of multilingual data, and it approaches human-level speech recognition performance. Unlike traditional keyword-based voice control, Whisper can transcribe free-form user speech, which is essential since the GPT assistant expects full sentence inputs (e.g. "Can you walk forward a few steps?" rather than a single pre-set command phrase like "Ok Robot, move forward."). There are two ways of using Whisper in this project: running it locally on the Raspberry Pi 5, or using a cloud API. Running the small Whisper models on a Pi 5 is doable, but it is quite slow *(15-20 seconds for a 10-second long audio recording in my case)*, and uses a significant portion of the CPU, as noted in recent benchmarks of Whisper software on Pi hardware[22].

I instead opted for the OpenAI's Whisper API for faster transcription, given that the robot is already internet-connected for the GPT assistant. The recorded audio from the robot's microphone is sent to the Whisper API, and a text transcription is returned, usually in under 2-3 seconds for a short command. This cloud STT approach uses the cloud robotics philosophy and offloads the heavy lifting from the Pi. However, an offline fallback could be implemented in future versions for limited-case scenarios, making sure that the robot can still respond to basic commands in the absence of an internet connection (at the cost of slower and less accurate transcriptions). It's worth noting that voice interface technology has a rich background in robotics and consumer devices. Systems like Amazon's Alexa and Google's Home Assistant introduced the scenario in which one can speak to a device and get intelligent responses. These systems also perform audio capture locally and send it to cloud servers for interpretation, so quite similar to what's done in this project. The difference is that my robot does not just respond in words, it also acts with physical actions. This integration of voice commands, dialogue, and a physical body is a step towards more natural human-to-robot interaction. It was important to make sure that the transcription is accurate, especially for command phrases like *"stop", "forward", "turn left"*, since any misinterpretation there could lead to a wrong movement. Whisper's accuracy on those simple words was great in testing, even with a bit of background noise and weird accents, thanks to its advanced speech recognition.

2.4 Quadruped Robot Platforms and Spot Micro

The mechanical design of my robot takes inspiration from existing open-source quadruped projects, particularly the Spot Micro family of hobby robots[23]. Spot Micro is a scaled-down, 3D-printed version of Boston Dynamics' Spot, typically using 12 servos (3 per leg) forming the main joints: hip (horizontal translation), shoulder (vertical rotation), and knee (vertical translation). This provides a total of 12 degrees of freedom, which allows the robot to perform a wide range of leg motions and poses. Many makers have contributed with variants for Spot Micro. For example, the Nova Spot Micro project by Chris Locke[24] provides STL files and assembly guides for this robotic framework.

My robot uses a modified version of these community designs, adjusted to fit my specific components (The Raspberry Pi 5, a camera module, the speaker etc.). Low-cost quadrupeds like Spot Micro use hobby-grade servos for actuation. Common choices are metal-geared servos such as the TowerPro MG996R or DS3218, which provide decent torque (10-15 kg-cm) at 6V and are affordable. However, as noted by hobbyists, a fully assembled Spot Micro can weigh a few kilograms, and using cheap or plastic-gear servos isn't enough, as they will not be able to support the robot's weight and can fail. In the beginning, I tried using TowerPro MG996R servos in my robot, but they proved to be difficult to work with (more details in Chapter 5). The servo calibration is an important process as well: each servo's zero position must be aligned correctly when assembling the legs, and some trim adjustments have to be coded in, so that the robot stands evenly. The Nova Spot Micro project prioritizes calibrating the motors during leg assembly for proper movement. In my build, I used a *servo tester* to get the correct angles while mounting the servo horns, making sure all the joints are symmetric.

Several electronic control architectures exist for DIY quadrupeds. One approach (used by Petoi's commercial Bittle and Nybble robots) is to use a microcontroller (like an *Arduino* or a *Teensy*) for reading sensors and driving servos, and an optional *Raspberry Pi* or *ESP32* for higher-level tasks. Another approach is to use a microcontroller purely for motion control and a companion board (Pi or another MCU) for higher level logic and sensor data. My design uses the second approach: the *Arduino Pro Micro* is used for servo control ("the body"), and the Raspberry Pi acts as the high-level processing unit for the sensor data and movement commands ("the brain"). This separation is useful because the servo pulses handled by the Arduino require precise timing, while the Pi, which runs a full Linux OS and resource-intensive software, can occasionally lag or spike in latency. This split architecture is also seen in *OpenCat*, an open-source quadruped framework, which uses an Arduino-compatible board for locomotion and an optional Raspberry Pi for running AI algorithms.

Finally, we'll look at the *gait* and control software in related work. Fully dynamic autonomous quadrupeds (like *Boston Dynamics' Spot*) use advanced control algorithms, sensor arrays (*IMUs*, vision, *lidar* etc.), and even machine learning to achieve advanced navigation. In the hobby space, simpler gaits (like crawling gaits where at least 3 legs touch the ground) or predefined walking sequences are typically implemented. Some projects use inverse kinematics (IK)[25] to compute leg joint angles for the desired foot positions, providing more flexibility in movements (e.g. the Alpha ESP32 Spot Micro[26] uses IK for motions like sitting and walking.

IK requires solving the servo angles for given desired XYZ positions of the feet relative to the body. I opted for not using IK in the leg positioning because of the issues encountered with the *MG996R servos*. Instead, my robot's walking gait is encoded as a series of key frames (servo angle sets) that move the legs in a cycle – an *open-loop* approach. This method is enough for demonstrating walking on flat ground, though it lacks the adaptability of a full IK or feedback-controlled gait[27]. The implications of this choice (stability vs flexibility) are discussed in Chapter 4 and Chapter 5.

In summary, this project takes inspiration from prior work in cloud-connected AI, voice interfaces, and quadruped robotics. By combining these elements – a cloud AI assistant, a voice-controlled interface, and a quadruped platform, this project makes a solid entry in the interactive personal robotics space.

3 CLOUD-INTEGRATED CONTROL SOFTWARE

3.1 System Architecture Overview

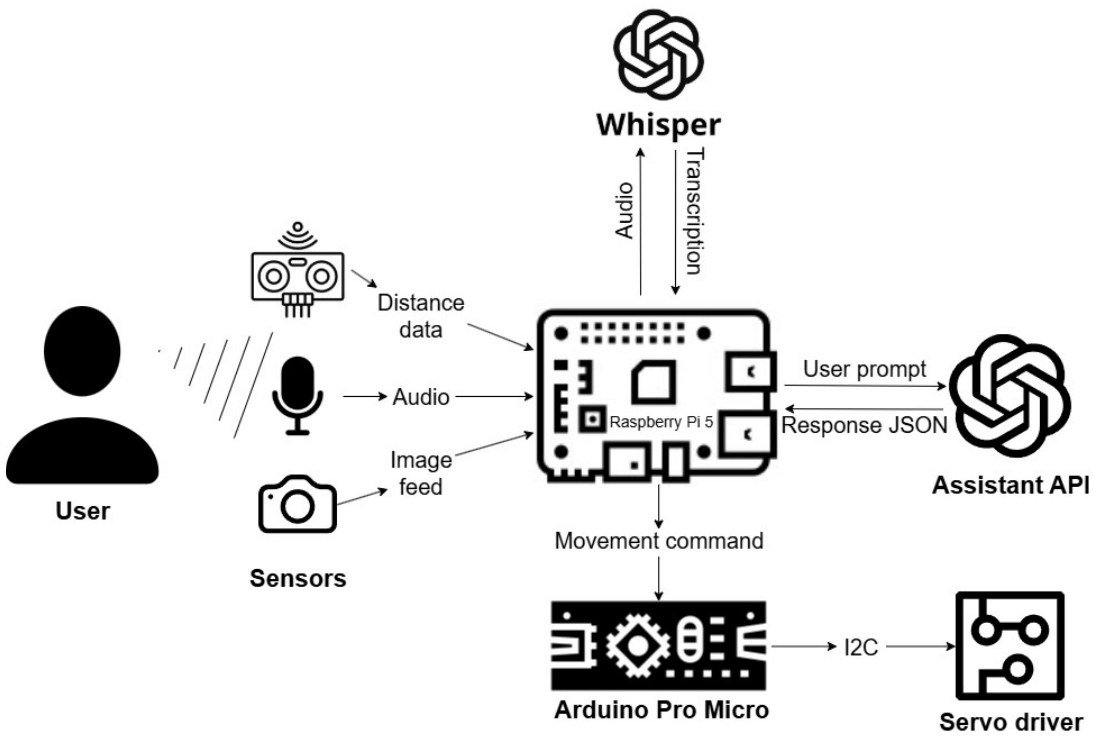

Figure 1: High-level system architecture, showing the flow from sensor input to robot motion.

The core of my system[28] is a pipeline that connects human input to robot action output via cloud API's. Figure 3.1 illustrates the architecture. A user's voice command is captured by the robot's microphone, along with a picture from the *Pi camera* and the data from the distance sensors, and then it is processed by the Raspberry Pi 5 (the onboard computer). The Pi runs a Python-based program that executes the following steps:

1. **Input Reception:** The raw audio is recorded at the beginning of the control loop for 10 seconds, then sent to the cloud-based Whisper API for transcription. The script also captures a picture from the front-facing camera and the distance readout from the 2 distance sensors, also mounted up front.
2. **LLM Query Formation:** The transcribed text from the user's recording, along with the sensor readings (distance measurements and the camera feed), is formatted into a prompt, and sent to the GPT-4 assistant API. The assistant has been configured with

a system prompt that gives it context about being a robot control system, the input data, its moveset, and the required response format.

3. **Assistant Processing in the Cloud:** The GPT-4.1 model, running on OpenAI's servers - the *"brains in the cloud"* - receives the query and generates a response. Thanks to the given prompt, the response is a JSON object containing two main parts: a natural language reply to the user (the "speech" field), and a set of movement commands (the "action" field, which may contain a keyword like "forward", "turn-left", etc., or a direct control structure with parameters, and the "steps" field - how many steps of the movement should the robot make).
4. **Response Handling:** The Raspberry Pi parses the JSON response. The "speech" reply is extracted and is output via the speaker (running the *Espeak text-to-speech* software), and displayed (for debugging purposes, I logged it to the console). The action instructions are translated into a series of movement commands, which are sent over a serial connection to the Arduino board.
5. **Microcontroller Servo Control:** The Arduino receives the serial commands and executes the corresponding servo movements (e.g. starting a walking sequence or turning). This is the final step of the input-reasoning-action loop. After the Arduino reports back via Serial that the movement is finalized, the Python script will repeat all of the steps from above in the next loop.

With this architecture, the Raspberry Pi works as an intermediary between the cloud AI and the robot's body. It handles all network communication and local sensor data. The Pi runs a Python script: it firstly configures all the components at the startup (the camera, Arduino Serial and the distance sensors) and announces via the TTS that the robot has successfully initialized, then it runs in a loop format: it asks for a command from the user, starts recording 10 seconds of audio, sends that recording to the Whisper API for transcribing, then, after the transcription is done, it packages it, alongside the camera picture and the distance sensors readout into a prompt, and sends it to the GPT assistant API for processing. I chose Python thanks to its vast library collection (for example, using the *OpenAI Python library* to access the GPT assistant, and *serial library for Arduino* communication) and ease of use. In terms of performance, Python on the Pi 5 runs smoothly overall. Also, because the time constraints aren't critical since the servo control is offloaded to the Arduino, I haven't encountered any problems regarding the functionality of the robot. There are some delays introduced by the network, the AI processing times and the audio recording, but they don't affect the functionality of the robot, and since the STT software is configured to run on a separate thread, the real-life experience doesn't include significant delays.

The data flow and message formats are very important in this architecture. The recorded audio is handled in chunks of 10 seconds – the program loop starts a 10-second audio recording after a TTS message (in my case it's "Please give an instruction"). The audio is saved as a *WAV file* and sent to the Whisper API. The returned transcription is inserted into the assistant prompt in the "user_instruction" field. Sensor data (the two ultrasonic distance readings in cm) is encoded as a small JSON snippet. For example, we have the "environment_data" field indicating 45 cm to an obstacle in the left and 50 cm to the right. This is due to the sensor angles being 50 degrees towards the center:

```
environment_data = (
        "Left sensor: 45 cm (50° right-inclined), "
        "Right sensor: 50 cm (50° left-inclined)."
)
```

These values are included in the user prompt to the assistant, alongside an image capture from the front camera. The GPT JSON reply will look like this:

```
{
    "movement": "forward",
    "steps": "4",
    "speech": "Okay, I will walk forward."
}
```

Then, the Pi would process the output JSON. It will speak the "speech" text using the TTS program and send the movement commands to the Arduino for the servo movement. In other interactions, the assistant might return something like this:

```
{
    "movement": "left,forward,right",
    "steps": "3,5,3",
    "speech": "Obstacle detected. Walking around it."
}
```

Here, it observed from sensor data and camera feed that the forward path is blocked by an obstacle and decided to go around its left side. More details about the JSON structure in section 3.4.2.

One important addition is that I implemented basic error handling in the Pi's software. If the Whisper API fails to return a valid transcription (network issue) or returns an empty one, it will fallback to a default case (*"There are no instructions, act on your own"*). If the GPT API call fails or returns invalid JSON, the Pi will log an error and retry in the next loop. To keep the processes in sync, only one GPT query is active at a time. Once the Arduino finishes executing a movement, it will send a "Finished Instruction" signal over serial, which the Pi listens for. This process helps to synchronize command execution (the Pi won't send a new movement command until the old one is finished).

Overall, the system architecture represents a modular design: the cloud AI module, the Pi control module, and the microcontroller module work independently and connect through clear interfaces (API calls and serial commands). This modularity allows for future upgrades, such as using a fallback local LLM, adding additional sensors, or upgrading the hardware, all without rethinking the entire system.

3.2 Sensor Suite and Data Collection

The robot is equipped with a general set of sensors used for gathering the environmental context and the user instruction:

- **Camera:** A *Raspberry Pi Camera Module* is mounted in the front of the robot and used for capturing images of what is in front of the robot. In the current implementation, I didn't add any on-board image recognition to run on the Pi, since I wanted to "test" how much context would the AI be able to gather from the picture alone. Future versions could implement an object tracking feature and face recognition, so the robot could recognize users when looking at them and reply based on the person it's talking to. The camera takes one picture for every loop, so every prompt sent to the assistant API includes visual data for context.
- **Distance Sensors:** Two *HC-SR04 ultrasonic distance sensors* are located in the front of the robot, on one side and the other of the camera, each angled inward by 50 degrees. They are positioned like this purely because of the way the *Spotmicro model* is designed. The Raspberry Pi interfaces with these sensors through its GPIO pins (the trigger and echo pins for each sensor). The *time* Python library was used to handle the timing (emitting a 10 µs trigger pulse and measuring the echo pulse width for distance). These values are then added to the prompt every time a query is made. The assistant can use this information to decide on movements. For example, if the right ultrasonic records a small value (less than 15 cm), the assistant will suggest avoiding the obstacle from the left (due to the inwards angle, these sensors help the robot determine the lateral obstacles, that aren't covered by the camera FOV). Basically, the ultrasonic distance sensors give the robot a basic idea for obstacles, like the simple collision avoidance in a *vacuum cleaner robot*. While GPT is not performing advanced path planning, it can output sentences like "I sense an obstacle in the right, I will turn left", which proves a good environment awareness.
- **Microphone** A *USB Wireless Microphone* is connected to the Pi for capturing audio from the user. Since Whisper is optimized for noisy and unclear audio, the quality of the mic doesn't have to be great. Although, these wireless mics are quite decent, and the rezulting transcriptions rarely differ from the real spoken phrase. The Python script uses the *Arecord* Linux library for recording audio. It opens a subprocess with the arecord command, selecting a 16kHz rate and mono audio channel, which is the recommended Whisper audio format. The audio is the only source of input from the user, and is automatically recorded in 10-second batches at the beginning of the script loop. A future improvement could include a separate program thread that continuously records audio, and when it detects voice activity (some noise after a period of silence), it sends that snippet to the Whisper API for transcription, avoiding the 10-second pause in the robot control loop for when the audio is recorded. Some specific voice commands could also be added, like the addition of a "force-stop", where the robot would suddenly stop when hearing the word "stop" or "pause" from the user.

3.3 Voice Command Transcription

The voice command segment converts the user's speech into text that can be sent to the GPT assistant. As presented earlier, I chose the *OpenAI Whisper transcription model* for this task. The whole transcription process works like this:

- At the beginning of the loop, after a spoken announcement via the TTS (*"Please give an instruction"*), the script starts recording a 10-second segment of audio, using the arecord Linux library. The recording segment of code is:

```
command = [
    "arecord",
    "-D", AUDIO_DEVICE,
    "-f", AUDIO_FORMAT,
    "-c", AUDIO_CHANNELS,
    "-r", AUDIO_RATE,
    "-d", str(RECORD_SECONDS),
    AUDIO_FILE
]

try:
    subprocess.run(command, check=True)
    print(f"[Audio] Recording complete. File saved as: {AUDIO_FILE}")
except subprocess.CalledProcessError as e:
    print(f"[Error] Recording failed: {e}")
```

- The recorded audio is then sent to either the Whisper API for transcription, or transcribed locally using *whisper.cpp*. The transcription method is decided by the boolean configuration variable *USE_API_WHISPER* (it has the value "True" if we want to use the API, and "False" for the local api). The audio transcription is sent to the Whisper API via this code:

```
transcript = ""
# Use the API to transcribe the audio
print("[Whisper] Transcribing audio with API...")
try:
    # Open the audio file
    audio_file = open(wav_file, "rb")
except FileNotFoundError:
    print(f"[Whisper] WAV file not found: {wav_file}")
try:
    # Call the API to transcribe the audio
```

```
            transcript = client.audio.transcriptions.create(
                model="whisper-1",
                file=audio_file
            )
        except Exception as e:
            print(f"[Whisper] Error during transcription: {e}")
```

- Next, we get the resulting JSON from the call, and extract the 'text' field containing the recognized speech. Whisper tends to include punctuation quite well, so I opted to send the direct transcription to the assistant API, to keep a small bit of context that would've been lost from text post-processing. One interesting observation though - Whisper will transcribe everything it hears, so if the user says something and then there's a prolonged silence or an "*umm...*", it will also include that. In the case where no speech was detected and Whisper returned an empty string, I added a feature so that the command will be changed to "*There are no new instructions, act on your own*".

The latency introduced by the transcription isn't noticeable: recording 10 seconds of audio adds 10 seconds. Uploading that to the API and waiting for the transcription takes 2-3 seconds, which isn't a long time. This is similar to a conversation with other popular voice assistants(Gemini, Siri, Alexa, where similar delays exist). Whisper can operate in different modes (translating to English, or just transcribing). I set it to translate to English since there are instances where the robot would be spoken to in my native language (Romanian).

Using Whisper yielded a high accuracy - in testing, even pretty complex phrases were correctly transcribed. Whisper's tolerance to noise eliminates the need for a noise cancellation step. In one test, background music at an average volume did not affect the understanding of the command "turn right" – it was transcribed correctly. This performance proves that using a modern AI-based automatic speech recognition (ASR) is a valid method for human input.

3.4 GPT Assistant Integration

This section describes the interface with the GPT-4.1 model via OpenAI's API and how I configured the assistant's output to match the robot's architecture. The integration has two main components:

1. The **Prompt Design**, which is responsible with the construction of messages (system and user prompts) sent to GPT-4 to receive the desired output.
2. The **JSON Response Format** and parsing, which describes how the Raspberry Pi extracts the assistant's reply and the movement commands to be sent to the microcontroller.

3.4.1 Prompt Design

Designing the system prompt (also known as the initial instruction for the AI) is a very important step. My goal was to make the GPT-4 model act as a robot control system for a quadruped robot dog. The assistant has to include movement commands for every reply. I kept adding details for the system prompt as I developed each feature of the project and through experimentation. The final prompt (full text provided in Appendix A) can be summarized in these steps:

- **Establishing identity:**

  ```
   You are a robotics control system for a quadruped robot. You receive:
  1) The data from the robot's distance sensors and an image of what
  he sees through its camera. The camera and the 2 distance sensors
  are frontally mounted. The camera is oriented straight ahead, and
  the sensors are 5cm apart(on one side and the other of the camera),
  each rotated inwards by 50 degrees.
  2) A user instruction - a voice command transcribed into text.

  Give the robot a playful and curious personality, eager to discover
  the environment and people nearby.
  ```

- **Establishing desired output format and clarifying the movement set:**

  ```
  You must output a JSON object with the following format:

  {
    "movement": "STRING",     // The actions that the robot will execute
  - must be one of: forward, backward, left, right, stand, flat, sit, paw,
  greet . The actions are sepparated by commas
    "steps": list of numbers,  // Number of steps done for each movement,
  only the forward,backward,left and right commands require a number of
  steps, for the others the number can be 0
    "speech": "STRING"      // The text the robot should speak via TTS
  }

  Clarification and brief explanation of the poses:
  - forward: the robot steps forward once, for around 4cm
  - backward: the robot steps backward once, for around 4cm
  - left: the robot spins left for one step, for around 10 degrees of rotation
  - right: the robot spins right for one step, for around 10 degrees of rotation
  - stand: the default position of the robot, standing straight
  - flat: the robot lies down, touching the ground with its belly, similar to
  ```

```
its powered-down state
- sit: the robot sits like a dog, good for an idle position
- paw: the robot stretches one of its front feet in the front
- greet: the robot waves with an arm
```

This tells the model how to reply, making sure that the output can be parsed by the Pi.

- **Valid output JSON examples:**

  ```
  Example of a valid json:
  {
      "movement": "forward,left,sit",
      "steps": "2,5,0",
      "speech": "Hello, I am moving forward, turning left, then sitting down"
  }

  You can also only send one movement, like this:
  {
      "movement": "forward",
      "steps": "5",
      "speech": "Hello,I want to explore, moving forward"
  }
  ```

  I listed some examples of valid output for the model.

- **Guidelines**:

  ```
  Guidelines:
  - Always return a valid JSON.
  - If the space permits it but the instruction is unclear, try to turn
  the robot to the user and ask for clarification.
  - If the users request is unclear, and the space is constrained so that
  you can’t turn, set the movement to a stationary position, steps = 0,
  and in speech, clarify what you need.
  - As long as the space permits it, try and explore the environment
  you’re in, unless given a clear command from the user.
  - Steps values should be integers.
  ```

  These guidelines were added to increase the answer reliability and set the desired action responses.

I also added an experimental direct control schema to the prompt: instead of a default movement like "*forward*", or "*left*", the agent could send the angle position for every servo index. This was mostly implemented to test the reasoning capability of the GPT model for the direct positions. The prompt is:

```
There is also the option of a direct position control:
You could have a json like this:
{
"movement": "1:30 2:80 5:100,2:60",
"steps": "2,0.5",
"speech": "Hello, I am moving the servos directly now"
}

In this way, you can directly set the angles for each servo on the robot.
The movement consists of a list of servo indexes and their values.
Different positions are sepparated by commas. The steps value now
represents the amount of time each direct position is held.
The positions don't reset after the timeout, the legs remain in those
positions, but you can use the delays for orchestrating new movements.
You can set a position, put a step value that determines the time it
holds that, then add a new movement with a new time period to create
your own complex movements.

The indexes of the servos are:
Each leg: [hip, shoulder, elbow]
{0, 1, 2},   // Front left
{4, 5, 6},   // Front Right
{8, 9, 10},   // Rear Left
{12, 13, 14}  // Rear Right

The indexes go from 0-15, but we're not using the 3, 7, 11 and 15 indexes.

The positions for each servo that the robot uses in the standing pose are:
{40, 50, 65},  // FL
{100, 90, 80},  // FR
{45, 100, 80},  // RL
{90, 60, 70}   // RR

And the positions for the laying flat pose are:
{0, 15, 65},  // FL
{140, 130, 80},  // FR
{0, 140, 80},  // RL
{140, 10, 70}   // RR
```

```
The angles go from 0-180, but I recommend only going up to ~140-150,
and be careful with the elbow 2, 6, 10, 14 servos - they have less
movement space because of the robot's joints, and can try to bend
into the frame, so not reccomended to flex them a lot.
```

More info about the direct control schema can be found in Section 3.5.

All these instructions were combined into a single system message. This resulted in a long prompt (several hundred tokens), but GPT-4 does not have any problem with that context length. I configured it as the system prompt in the *OpenAI's Assistants Playground*. then the user message (which contains the transcribed command and sensor data as described).

While testing the assistant, GPT-4 performed great with this prompt. I didn't encounter any scenarios where the LLM output didn't match the desired JSON format, and the environmental awareness was almost surprinsing. Again, more details about the results can be seen in the fifth Chapter.

3.4.2 JSON Response Format

The formatting of the AI's response is very important, since the script running on the Raspberry Pi needs to read the movement instructions automatically. I chose the JSON format because it's easy to parse in Python (using the *Python json library*). The content of the JSON is basic: a "movement" string, a "steps" list and a "speech" string.

To ensure that the LLM always returns a JSON, I set the "*Response format*" field as a "*json_object*" in the assistants Model configuration. Also, the design of the JSON output means the system can easily be extended or integrated with other modules. For example, if we wanted a GUI to display what the robot wants to do, it can parse the same JSON. Or, if the Arduino got replaced with another microcontroller, the integration remains the same if it receives the same action strings.

3.5 Movement Commands

After the Raspberry Pi extracts a movement command ("movement" string) from the assistant's response, it will send it to the Arduino, which controls the servos. The communication between the Pi and Arduino is achieved via a USB serial connection (the Arduino Pro Micro appears as a /dev/ttyACM0 serial device for the Pi). I configured the serial link at 115200 baud. The command protocol used designed for serial communication is simple: Each command is sent as a line of text, terminated by a newline character ("`\n`"). The Arduino program reads lines from serial and parses them. There are 2 types of commands:

1. **Movement Commands:** Sending a direct moveset command, like "forward" or "sit".

The Arduino moves the servos in predefined positions based on the selected movement. In case of walking commands (e.g. "forward", "backward", "left", "right"), the "steps" field represents the number of times that moveset is executed (e.g. for "left,2", the robot takes 2 steps to the left). For the rest of the moves that represent just a pose (e.g. "stand", "sit", "greet" etc.) the prompt instructs the AI to send a "steps"="0" value.

2. **Position Commands:** Sending a string with direct servo target angles by order index, like "0:15 1:30 2:90". This type of command was mostly used for debugging, but with enough details in the prompt, the LLM was able to control the servos directly as well. But, since the aim was to let the AI handle high-level planning, and have the low-level control run on the Arduino, the first option is used by default, leaving the second one for isolated cases.

To send the commands, the serial port has to be initialised first, so I open the serial with 115200 baud rate at the beginning of the script, and keep it open, to avoid resetting the Arduino mid-run. I also set timeout=1 so that reading from serial doesn't block the script for too long. The Arduino will send back completion messages, like "Finished instruction" or debugging info like "Standing still". Those are printed by the Python script for logging. On the Arduino side, the serial parsing is done in a simple loop: it reads words from the lines until the newspace, then compares the keyword from the received command against known commands. The code looks like this:

```
void loop() {
  // Accept commands via Serial Monitor for now
  if (Serial.available()) {
    String line = Serial.readStringUntil('\n');
    line.trim();
    if (line.length() == 0) return;

    if (line.startsWith("move,")) {
      int first = line.indexOf(',');
      int second = line.indexOf(',', first + 1);
      String dir  = line.substring(first + 1, second);
      int steps   = line.substring(second + 1).toInt();

      if (dir == "forward") moveForward(steps);
      else if (dir == "backward") moveBackward(steps);
      else if (dir == "left") moveLeft(steps);
      else if (dir == "right") moveRight(steps);
      else if (dir == "stand") standPosition();
      else if (dir == "flat") layFlatPosition();
      else if (dir == "sit") sitPosition();
      else if (dir == "paw") pawPosition();
      else if (dir == "greet") salutePosition();
      else standPosition();

      Serial.println("Finished instruction.");
    }
```

An important part of sending the movement commands was avoiding their overlap. I had to make sure that the Pi doesn't send commands faster than the Arduino can execute them. So, if the robot receives multiple commands from one LLM query, it will send them in order, and wait for the Arduino to reply via serial with the "*Finished instruction.*" string, so that it knows to hold on to the new command until the current completes. To achieve this, I implemented a *busy-waiting* sequence of code that continuously reads the serial output until it detects the finished instruction string. While testing, because interactions are sequential and the received commands have relatively low numbers (1-3), this configuration worked well. The program loop waits until the robot finishes moving (which is only a second or two per position command, but around 5 seconds for a step forward/backward or a directional left/right). After the movement finishes, it restarts the loop with the recording of instructions and sensor data collection.

In summary, the movement command mechanism is a lightweight protocol that sends the "movement action" from the Pi to the Arduino. In the case of movement commands (which are mostly used unless special cases), it is doing a level of abstraction for the details of the servo movements and gaits, so the high-level AI doesn't have to calculate the kinematics for the movement. This kind of separation is recommended in robotics: the high-level planner (AI) gives abstract actions and the low-level controller takes care of the precise movement. This approach is also seen in *MachinaScript* for example, where they use serial commands like "A:45,10;B:0,10;" to specify servo targets, so very similar to my direct position approach. Though, my movement command is even higher-level (just choosing a pre-set command). Both methods have their pros and cons.

3.6 Movement Gaits

After I finished assembling the legs, I discovered a step that I overlooked: I didn't know how to make the robot walk. I could have started from a community Spot Micro project and use its locomotion code, but I chose to implement it myself, which in retrospect wasn't a great idea. So, I first started out by configuring the servos to have the robot stand up straight, using these values:

```
// The offsets for each leg servo
uint8_t legOffsets[4][3] = {
  {40, 50, 65},  // FL
  {100, 90, 80},  // FR
  {45, 100, 80},  // RL
  {90, 60, 70}   // RR
};
```

Then, I implemented a function that changed the leg angles relative to the standing position, so it would be easier to code movements for it:

```
// Set servo angles relative to per-leg offsets
void setLegRelativeAngles(uint8_t legIndex, int8_t hipDelta,
int8_t shDelta, int8_t elDelta) {
  setLegAngles(
    legIndex,
    constrain(legOffsets[legIndex][0] + hipDelta, 0, 180),
    constrain(legOffsets[legIndex][1] + shDelta, 0, 180),
    constrain(legOffsets[legIndex][2] + elDelta, 0, 180)
  );
}
```

And after experimenting with this relative angle change function, I ended up with this forward gait:

```
void stepForwardCycle() {
  // Phase 1: Lift FL + RR
  setLegRelativeAngles(0, -10, -10, -10);
  setLegRelativeAngles(3, 10, -10, -10);
  delay(100);

  // Phase 2: Swing FL + RR forward
  setLegRelativeAngles(0, -10, 20, 0);
  setLegRelativeAngles(3, 10, 20, 0);
  delay(100);

  // Phase 3: Place FL + RR down
  setLegRelativeAngles(0, 10, 20, 0);
  setLegRelativeAngles(3, -10, 20, 0);
  delay(100);

  // Phase 4: Lift FR + RL
  setLegRelativeAngles(1, 10, 10, 10);
  setLegRelativeAngles(2, -10, 10, 10);
  delay(100);
  // Phase 5: Drag FL + RR backward
  setLegRelativeAngles(0, 0, 0, 0);
  setLegRelativeAngles(3, 0, 0, 0);
  delay(100);

  // Phase 6: Swing FR + RL forward
  setLegRelativeAngles(1, 10, -20, 0);
  setLegRelativeAngles(2, -10, -20, 0);
  delay(100);

  // Phase 7: Place FR + RL down
  setLegRelativeAngles(1, -10, -20, 0);
  setLegRelativeAngles(2, 10, -20, 0);
  delay(100);

  // Phase 8: Lift FL + RR
  setLegRelativeAngles(0, -10, -10, -10);
  setLegRelativeAngles(3, 10, -10, -10);
  delay(100);

  // Phase 9: Drag FR + RL backward
  setLegRelativeAngles(1, 0, 0, 0);
  setLegRelativeAngles(2, 0, 0, 0);
  delay(100);

  standPosition();
}
```

I implemented the simplest mode of stepping forward on a quadruped, by lifting 2 legs at once, in the opposite sides of the robot (the front left leg with the back right leg, ant the front right with the back left), moving them forward, placing them down and dragging them back.

For the backwards stepping, I implemented those same steps, but mirrored the angles so the legs move backwards.

The turning gaits proved more challenging to program, but I ended up using the same strategy, lifting 2 legs at once and moving the other 2 to achieve the desired motion:

```
void turnLeftCycle() {
  // Phase 1: Lift FL + RR
  setLegRelativeAngles(0, -10, -10, -10); // FL
  setLegRelativeAngles(3, 10, -10, -10);  // RR
  delay(100);

  // Phase 2: Swing FL back + RR forward
  setLegRelativeAngles(0, -10, 20, 0);    // FL backward
  setLegRelativeAngles(3, 10, -20, 0);    // RR forward
  delay(100);

  // Phase 3: Place FL + RR down
  setLegRelativeAngles(0, 10, 20, 0);     // FL
  setLegRelativeAngles(3, -10, -20, 0);   // RR
  delay(100);

  // Phase 4: Lift FR + RL
  setLegRelativeAngles(1, 10, 10, 10);    // FR
  setLegRelativeAngles(2, -10, 10, 10);   // RL
  delay(100);

  // Phase 5: Reset FL + RR to neutral (drag forward/backward)
  setLegRelativeAngles(0, 0, 0, 0); // FL
  setLegRelativeAngles(3, 0, 0, 0); // RR
  delay(100);

  // Phase 6: Swing FR back + RL forward
  setLegRelativeAngles(1, 10, 20, 0);     // FR backward
  setLegRelativeAngles(2, -10, -20, 0);   // RL forward
  delay(100);

  // Phase 7: Place FR + RL down
  setLegRelativeAngles(1, -10, 20, 0);    // FR
  setLegRelativeAngles(2, 10, -20, 0);    // RL
  delay(100);

  // Phase 8: Lift FL + RR again
  setLegRelativeAngles(0, -10, -10, -10); // FL
  setLegRelativeAngles(3, 10, -10, -10);  // RR
  delay(100);

  // Phase 9: Reset FR + RL to neutral
  setLegRelativeAngles(1, 0, 0, 0); // FR
  setLegRelativeAngles(2, 0, 0, 0); // RL
  delay(100);
```

```
  standPosition();
}
```

Although complete, the movement code could greatly benefit from further development and testing to optimize and improve it. There is also the option of implementing *inverse kinematics* to improve movement.

3.7 Software Challenges and Trade-offs

Designing the software pipeline raised a couple of challenges, and I had to make some trade-offs. They are listed below:

- **Latency and Real-Time Interaction:** As mentioned, the total time from speaking a command to an action execution took a few seconds. This is mostly due to the cloud API calls (Whisper and GPT-4) and the set 10-second recording time, instead of an automatic speech detection. While GPT-4 is very advanced, its replies take a bit of time: typical API response time for a short prompt and short answer was 3-4 seconds in my tests (which is quite good for a modern LLM). Adding the 2-3 seconds for Whisper and some overhead, I often got 5-6 seconds of latency in total. This latency, while noticeable, since it's not exactly instantaneous, it's decent in a conversation context. The user experience is also helped by the filler phrases I added via the TTS (e.g. the phrase "*Processing instruction..*", which runs on a separate thread, so it doesn't have to wait for the api calls). There is also the option of using a faster model, like the GPT-o4-mini, which could respond faster (1-2 seconds), but in practice the bigger model also performed well.
- **API Reliability:** Relying on external APIs adds extra points of failure. There were instances of network issues where the API calls would return error messages (due to failing to connect to the servers). But since the program runs in a loop, it uses a simple retry logic: if Whisper API call fails, the current loop will have no new instruction, and the GPT will act on his own. And if the GPT call fails, there will be no new actions, and the loop will start again. Thankfully, in my use case, the frequency of commands is quite low, so I never got close to the official rate limits of OpenAI (which allow much more requests per minute for Whisper and chat). Another important trade-off is the **token cost**: each Whisper and GPT-4 API call costs money (although a very small amount per call). Over heavy use it can add up, which is not ideal for a freely roaming robot. For this prototype project though, it was fine (I barely ended up using 60 cents, and the minimum payment of credit for api calls is 5 dollars - so it's not a real issue for a hobby project). But in the case of a commercial product or widespread deployment, the api pricing becomes meaningful - so a fully offline solution would be preferable in that case. The local processing is also something to consider for future work.

- **Continuous Listening vs Set Interval of Recording:** I initially wanted the robot to be continuously listening for a wake word, like “Hey Robot!”. Implementing a wake word detector (like the *Porcupine* or *pyttsx3* libraries) on Pi is doable, but integrating that with Whisper proved more difficult (so that I didn't end up constantly sending audio to Whisper, which would consume more resources). In the end, I went with a 10-second listening period announced via TTS at the beginning of the loop, so that the structure would remain more back-and-forward between the user and the robot (the robot listens for a command, processes it, executes it and starts again). The trade-off is between technical complexity and user experience. But for a more advanced process, there would be a need to integrate a local wake-word model to trigger the rest of the pipeline only when the user addresses the robot, and let it act independently in the meantime.
- **Sensor Data in the GPT Prompt:** Sending sensor data to GPT helped with obstacle avoidance. But GPT-4 is not a dedicated path planning algorithm, so it might not always know what to do with the sensor values. For example, if there is an obstacle in the front of the robot, and the user says "go forward", ideally GPT should output a refusal or walkaround action. The prompt tells it to do so. In tests, it did respond as expected most of the times, but sometimes it might say: "I'll try to move forward carefully." which is not ideal because it's ignoring the obstacle in a way. This highlights that using a general AI for critical logic requires very explicit instructions. A fix to this problem could be to add a rule in the code (e.g. the Pi checks the distance and changes any forward command with a stationary one, adding a mention to the command change in the next prompt to the assistant). I didn't end up implementing a safeguard, since the GPT prompt itself was enough, but it is a good implementation idea for future work.
- **Memory and Context:** I used the assistants API for the GPT calls, so it automatically keeps all the conversation history per session, with no need of sending the last message as a reference. A "session" is basically an assistant thread created at the beginning of the Pi control script, so the robot would recall everything that was told to him in that conversation. Each query to GPT only included the user prompt, since the system prompt is defined in the assistants workspace. Although, the robot can't remember a previous conversation for when it was last used, since for every boot cycle, the assistant thread changes. This means that the robot doesn't recall older conversations, just what was told to him in the current session. I could technically create an assistant thread manually and change the code so that only that one thread would be used for API calls, but that would mean that the token usage would increase, and the query time would take longer. Also, there is the extra cost for the calls, but that is not a big issue, since it would still be difficult to go over budget. The trade-off is the loss of conversation history (the robot can't remember your name if you told it the last time you spoke with it), but I didn't really find it a big problem, since the current conversation scope was more relevant. For future improvement, maintaining the context through all the conversations could make interactions smoother.

In conclusion, the software side is what makes the robot interactive. By using advanced AI models and focusing on the prompt engineering and protocol design, the resulting system successfully combines a cloud AI with a physical robot, despite some latency and simplicity of motion. The challenges encountered while designing the system made me aware of the importance of fail-safes when dealing with AI outputs, the need for programming the AI's outputs for the robot, and the user experience details of interacting with a semi-autonomous robot. These conclusions will help improve the next versions of the system, as discussed in Chapter 5 and 6.

4 QUADRUPED ROBOT DESIGN AND CONTROL (HARDWARE)

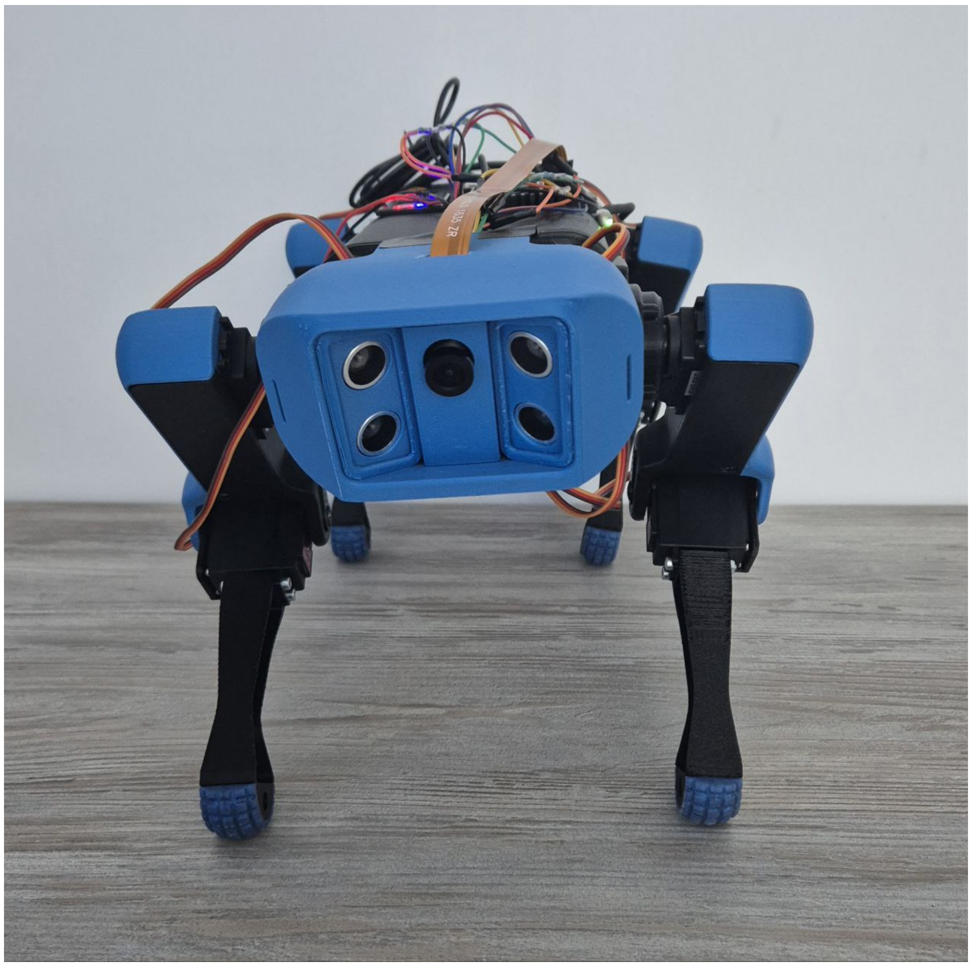

Figure 2: The finished robot, standing while waiting for a command

4.1 Mechanical Design and CAD Model:

The robot's mechanical design is a quadruped structure, based on *Boston Dynamics' Spot* and the open-source *Spot Micro designs*. Each leg has three segments (similar to the hip, thigh, and shin of an animal), that are moved by three servos, giving the robot 12 degrees of freedom in total. The design process involved two main tasks:

1. Obtaining the 3D models for the chassis and legs and modifying them for my specific usecase
2. Assembling the robot and calibrating the servos to provide the desired range of motion and stability

I started with existing CAD models from the internet. The frame of the robot is adapted from the project that started the SpotMicro community: *KDY0523's Spotmicro - robot dog*, which provides the *STL files* for a miniaturized Spot clone that is designed to use 12 *hobby-grade servos*. I imported these models into *Autodesk Fusion 360* to measure and modify them. The main modifications I made are:

1. **Speaker Mount:**

   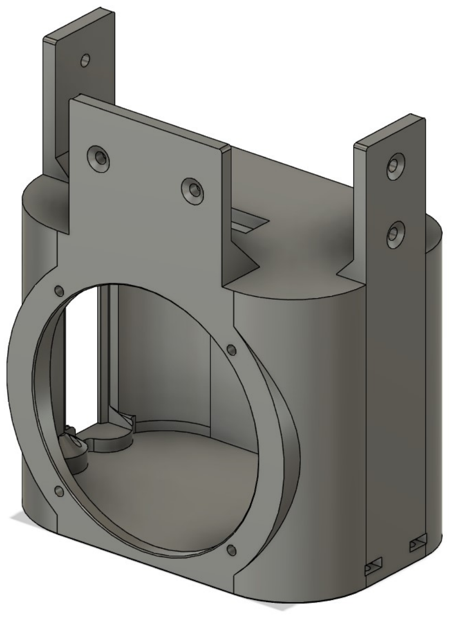

   Figure 3: The designed 3D model for the speaker mount

   For the robot to be able to respond to the user, it needed a speaker. I started from a *Omega 5W portable speaker*, then disassembled it and saved the circuits. I sketched a rectangular box that would hold those parts and made it compatible with the robot's front leg assemblies, so it could be mounted to them and act as the front half of the robot's torso. To power the speaker, I used a *MP2307DN buck converter*, and connected it to the Raspberry Pi via USB using an *external audio card*, since the Raspberry Pi 5 doesn't have a 3.5mm audio jack port.
   This improvised speaker turned out to be effective, having the audio come straight from the robot. It did achieve quite a high volume, so there wouldn't be any issues with users not hearing the robot.
   To generate the robot's response, as detailed in Chapter 3, I used the *Espeak TTS library*. I liked the retro and robotic sound of the voice, fitting for a robotic assistant.
2. **Electronics Mount:** This was a more difficult part to design. I needed something to connect to the speaker mount and the back legs, thus completing the robot body. I also needed to house the rest of the components that would make the robot work: the

battery mount, the voltage regulator, the ampmeter, the Raspberry Pi, the Arduino Pro micro, and the PCA9685 module.
I placed the battery tray on the underside of the part, the voltage regulator on the side, leaving space for its heatsync and the Raspberry pi up top. The Arduino and the servo control module were seated in the remaining interior space, between the Pi and the battery.
After printing the mount, thankfully all parts fit in nicely and I was able to assemble the robot. The servo cables routed through the hole at the back and connected to the PCA9685 module, which was powered by the voltage regulator.

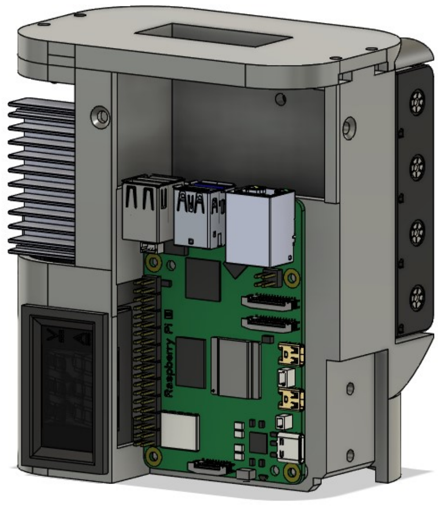

Figure 4: The designed 3D model for the electronics mount

3. **Camera mount** I wanted the camera to capture the front view of the robot. The front cover, which holds the ultrasonic distance sensors has an unused 25 mm wide space between the distance sensor mounts, the perfect space for the camera. I added 2 mounting holes to the underside of the cover, which I used to screw in the camera mount.
The model itself isn't very complex, just a slanted profile to match the front cover angles, and a hole to pass the camera objective through.

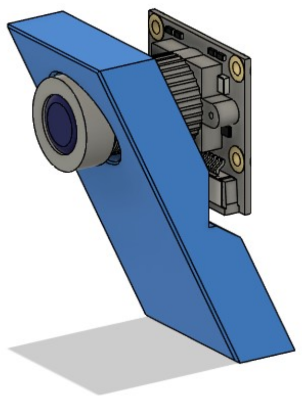

Figure 5: The designed 3D model for the camera mount

The parts were 3D-Printed[29] on my 2 machines: a *Creality CR-10S Pro V2* and a *Bambu Lab A1 Combo* using *PLA* and *PETG* filament at 15-20% infill with 0.4mm nozzles and 0.2 mm layer height. The structural parts (the black pieces of the legs and the torso) were printed from a combination of generic *PLA* and *PETG*, since I ran out of my PETG roll halfway through the printing process. The blue parts were printed in *bambulab's Matte blue PLA*. The total print time was around 40 hours for all the parts (torso parts, legs, brackets, mounts), but since I own 2 printers, I could print them in parallel, basically cutting the printing time by half. I encountered some typical 3D-printing issues, like holes not fitting screws perfectly and slight tolerance differences, but those were quickly solved with a drill and some sandpaper.

After printing the parts, I started assembling the components:

- **Legs:** Every leg assembly contains 3 segments: the tibia, which is the segment that touches the ground and holds the knee servo, the hip, which is the middle segment, holding the hip flexing servo, and finally, the joint support for the last servo, the hip rotation actuator. Between the knee and hip servo there is an extra part, that supports the knee servo with a bearing.
- **Front and back leg assemblies:** Each leg assembly contains a left and right leg. Those are held together by 2 supporting parts that act as a *pelvis*.
- **Torso:** The robot's torso is formed from the speaker and electronics mounts (detailed in Section 4.1). Those are the parts I designed to fit the custom electronics for this project. They use the existent mounting holes from the leg assemblies and interlock together using 6 M3 screws. When I was assembling the robot, I first had to install the electronics in their respective mounts (the speaker and electronics in the first part, and

pretty much all of the electronics in the other), then slowly routed the leg servo cables from the leg assemblies through the parts, and into the servo control module.

- **Front cover and camera mount:** After completing the torso assembly, I moved to a simpler one - the front cover assembly. Here, I first mounted the camera into its small support, then fastened it into the front cover, and added the distance sensors. I wired up the sensors and connected the camera ribbon cable, mounted the assembly in the front of the robot, and connected everything to the Raspberry.

One mechanical challenge was making sure that the robot could lift a leg without falling over, since the SpotMicro design is top-heavy, so the gait needs at least three legs on the ground at all times for stability. I did not implement dynamic balancing (e.g. by using a gyroscope), so the mechanical design has to ensure stability: I set the legs a bit wider by mounting the hip servos slightly to the exterior, and tuned their orientation further in the Arduino code. This small change helped with stability without affecting the walking process.

In conclusion, the mechanical design phase was a very important step in the making of this project. It required modifications and designing new parts from the ground up to accommodate my choice of hardware and ensure reliable operation. The result is a quadruped robot body that is capable of performing the movements ordered by the software while fitting all the electronics that give it full functionality.

4.2 Actuation: Servos and Kinematics

The movement of the robot is achieved with 12 servo motors arranged as described (3 per leg). These servos are standard RC (Radio Control) style servos that take a *PWM* signal to set an angle. I started with using the *MG996R* (a popular low-cost metal gear servo), but they quickly proved to be sufficient for supporting the robot's full weight. So, I ended up replacing most of the servos with the *TD-8125MG* model. Those really improved the motion of the robot. All servos run on 6V supplied from the battery via a 20A step-down regulator.

Servo characteristics: The MG996R and TD-8125MG servos both have approximately a 180 degree range, corresponding to a pulse width of 500-2500 μs, with 1500 μs as center (90 degrees). In practice, each servo's actual range and center can vary slightly, and I calibrated them in the code by defining the servo offsets array (more details in Section 3.6). Their speed is about 0.2 seconds per 60 degrees at no load, but they're slower under load. At first, when I started testing with the MG996R model, because of the robot's weight, the knee servos in particular got overloaded when the robot was standing – they can hold the position but it draws significant current and they heat up if held for too long. Also, they can't reach the standing position unassisted. Thankfully, using the TD-8125MG model mostly solved those problems.

PCA9685 servo driver: To control 12 servos, I ended up using the *Adafruit 16-channel PWM driver board*, based on the PCA9685 chip, which is connected to the Arduino via

I2C. This board generates 12-bit resolution PWM signals for all the servos without using the microcontroller's timers, since the Arduino Pro Micro (*Atmega32u4*) has limited timers and could only directly control a few servos reliably. The PCA9685 acts as a hardware PWM controller: the Arduino sends it commands over I2C specifying channel and pulse width, and the chip handles the rest. As a result, the Arduino can send updates to servo positions and then continue with other logic while the PCA9685 continues outputting the required pulses. Another advantage is that it only uses two pins (SDA and SCL) to control all 12 servos, simplifying the wiring. I set the PCA9685 PWM frequency to 50 Hz, a typical value for servos (20 ms period). With 12-bit resolution, this gives pulse width increments of about 4 µs, which is finer than the average hobby servo precision. I did not end up needing that fine of a resolution, but it is nice to have for future versions.

Kinematic configuration: Each leg's kinematics can be described by two segments: thigh (upper segment) and shin (lower segment) and a base roll axis from the hip (to laterally move the leg). In the leg geometry, the thigh length is approximately 10.7 cm and the shin length is 13.4 cm (including the servo horn offsets). The default standing pose has both hip and knee at roughly 45 degree angles (0 degrees meaning the leg is fully extended straight down). In that pose, the foot is about 15-16 cm below the hip horizontally, giving the robot a standing height of around 16 cm, and a total height of approximately 23 cm including the body.

Gait Implementation: I decided to use a parallel gait for walking, where one leg moves at a time while the other three remain on the ground to support the robot. This gait is pretty stable if the center of mass stays in the triangle of support. The sequence is: lift front-left leg, move it forward, put it down, then rear-right leg, move it forward, put it down, and the same for the front-right and rear-left. After one cycle, the robot has moved forward. The turning gaits also only use one leg per movement to remain stable, but instead of advancing all the legs in the same direction, they reverse the directions for the diagonal legs to make the robot rotate (more details about the gaits and code in Section 3.6).

During testing, the TD-8125MG servos handled the load, but did get warm, especially the hip and knee servos after walking. This was expected, and thankfully they never overheated. The 3D models included supporting bearings on the knee joints and the lateral rotation of the hip, but none for the forward hip servo, where the whole leg was held only by the servo horn. In my case this didn't affect the movement, but in future versions additional support for the hip would be recommended.

4.3 Electronic Architecture

This project, being this hardware intensive, needed a complex electronic architecture. The whole robot is powered by a 4-cell battery, situated in the back half of the torso. This battery then feeds the 3 voltage regulators, used for the servos, the Raspberry Pi and the speaker. The Arduino is powered directly from the Raspberry via a usb cable, and the Pi is connected to a *65W quick charge module* that is powered directly from the battery. The speaker is connected to the pi using an *external audio card*.

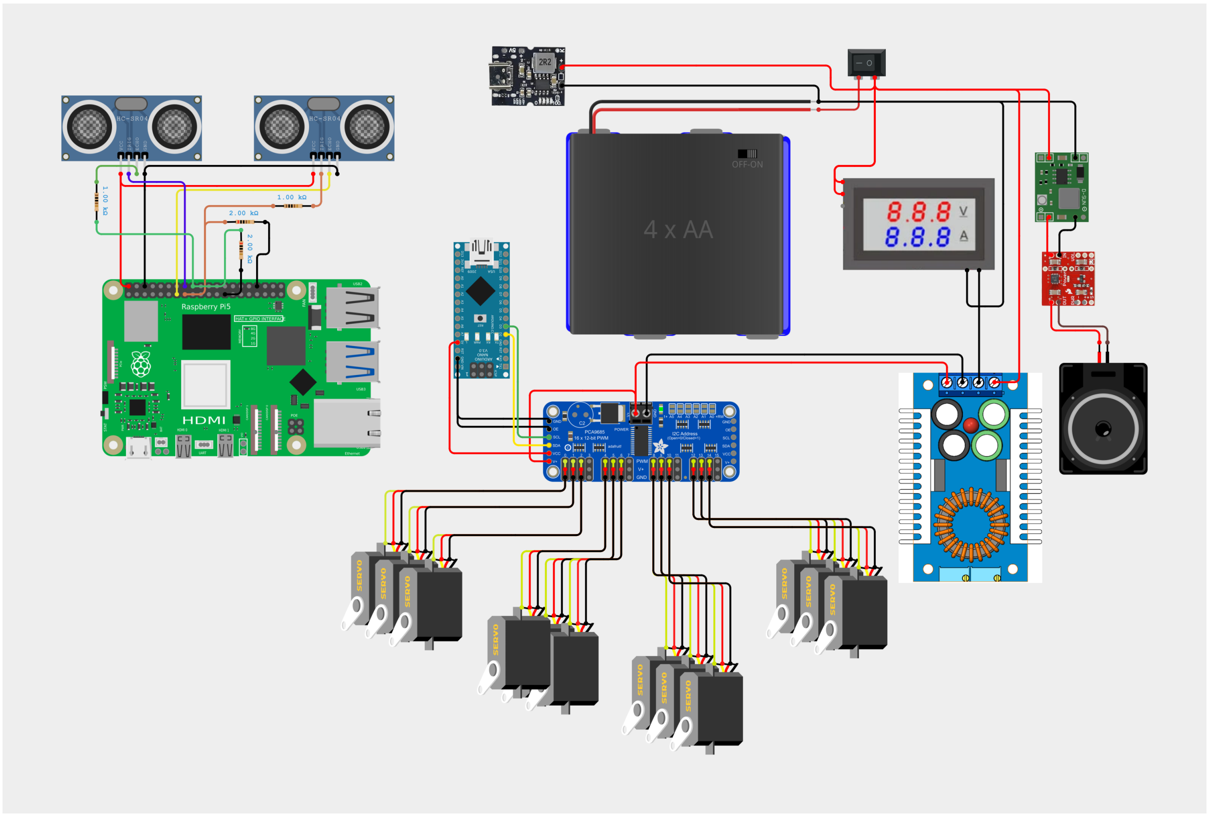

Figure 6: The wiring diagram for the project

Power Considerations: The robot is powered by a 4-cell 2200 mAh *18650* battery pack. The individual cells are mounted in a *4x case*. Running 12 servos simultaneously can draw a lot of current (a MG996R or TD-8125MG servo have stall power draws upwards of 2 A). To power them I used a *20A YD4020J* regulator set to output 6V. It is wired directly into the PCA9685 board, using the power input terminal blocks. This feeds all the servos, greatly simplifying the wiring. The speaker and raspberry used separate voltage regulators as to not consume the servomotors' power budget.

4.4 Arduino Firmware for Motion Control

The Arduino Pro Micro runs a program that interprets incoming serial commands and executes the corresponding motion sequences using the servos. It was written in Arduino C and structured to handle the different named actions. The main parts of the firmware are:

1. **Serial command parsing** (as described in Section 3.5): The Arduino is connected to the Pi via a USB cable, so it gets directly powered by the Raspberry, while using the

same cable for Serial communication. The program runs a simple line parser in the loop to read the received commands.

2. **Movement routines for each action:** After the microcontroller receives a movement command and the number of steps for it, it executes the corresponding movement function for it, thus signaling to the servos to move in the desired direction. As explained in Section 4.2, I used a *PCA9685* board to control the servos. It mostly helped on wiring, since it also supplied every servo with the connected power from the voltage regulator, so it reduced the need for power and signal cables to each servo. It was also simple to wire it up to the Arduino, since it used the *I2C protocol*, and only needed 6 pins connected.

In this project, the Arduino was only responsible for the movement of the servos, so it didn't have any sensors connected. It worked well for this scope, but in future versions, adding a *gyroscope* or an *accelerometer* to help with uneven terrain and fall detection would be a good upgrade to the system, since any slope or uneven step could trip it. All the tests were conducted on a flat floor. I didn't test outside or on rough terrain. The design can walk on carpets and wood flooring.

4.5 Sensor Integration and On-board Processing

Although the microcontroller didn't have any sensors connected, this wasn't the case for the Pi, as it had the following sensors attached:

1. **The Distance Sensors:** The robot has 2 *Ultrasonic HC-SR04* distance sensors, which are used to gather more information about the potential obstacles in its proximity. They are mounted in the front of the robot, on one side and the other of the camera (which is positioned in the middle), and at a 50 degrees interior angle. This means that, for example, the right sensor detect the obstacles in the front-left side of the robot. The Pi runs a simple distance detection command on each sensor(by sending a 10 µs pulse and measuring the time it takes to return), and sends them to the assistant API, together with the user prompt and camera picture.
2. **The Camera:** The Pi camera is mounted in the front of the robot, where his "head" is. It is held in by the camera mount part that I designed in Fusion (detailed in Section 4.1). It uses a *ribbon cable* for the connection to the Pi's camera port, and it is directly controlled by the Raspberry using the *libcamera* library.
3. **Microphone and Speaker:** To record audio reliably, I used this *wireless lavalier microphone* I got from Aliexpress for cheap. It proved to be a great choice, since the range is good (up to 10 meters) and the audio quality doesn't affect Whisper's transcription. The fact that it's wireless also meant that the audio recording would work better, since a microphone mounted on the robot would add extra noise from the servo hum and have the user follow the robot and try to speak directly at it, instead of just holding the

wireless microphone. And for the audio output (so that the robot can speak via TTS), I used a *5W portable speaker*. I removed its internals and transplanted them into the speaker mount (explained in Section 4.1).

In summary, the sensor integration was straightforward for voice and obstacle sensing. The ultrasonic sensors were used to inform the AI's decisions, alongside the camera view, demonstrating how some simple sensors can make an AI-driven robot more grounded in reality, preventing "mindless" instruction following.

4.6 Physical Control Challenges and Tuning

During building and testing of the robot, I encountered a couple issues that needed to be fixed:

1. **Servo Calibration and Alignment:** Although I set every servo to its middle position while mounting the legs using a *servo tester*, so all the legs would sit in the standing position when the robot would get powered up, some of them (due to the manufacturing or the way I assembled the joints) moved to different positions. This issue was quickly fixed by introducing a servo offset array in the Arduino code to slightly adjust each actuator's angle (code snippet in Section 3.6).
2. **Thermal Issues:** As expected, after a couple of minutes of use, the servos would start heating up, due to the load of keeping the robot standing. The temperature would slowly rise, but stabilise eventually. The servos got warm, but not burning hot, so about 50 degrees Celsius. It's a good thing they stopped there, since the materials I used to 3D-print the legs (PLA and PETG) start softewning above 60 degrees Celsius. Thankfully, the servo temperatures didn't pose a significant problem to this project, but for a future version it would be a great idea to add some heat syncs top the servos, or even some active cooling to keep them at lower temperatures.
3. **Power issues:** This was also a persistent issue for the project: The *YD4020J*, although rated for 20A, only put out around 15A max. This was fine for the most part, but in specific motions where all the legs would have to move at the same time (for example when the robot stand up straight after lying on the floor), the servos had a bigger power draw, which made the regulator voltage drop down 1-2 V . A reliable fix for this issue in future versions would be to use a more powerful regulator, or to add additional regulators to increase the power budget. I instead tuned the gaits and the motions to only ever use 2 feet at the same time, to reduce the load. This fixed the power draw issues, but it was more of a workaround than a fix.
4. **Noise and Wear:** Upon powering up the robot, the first thing that a user hears is the servo whine, caused by the actuator trying to hold its position. This is generally unavoidable, since any type of servo produces a buzzing sound when under load. More expensive models do emit less noise, but aren't silent either. This doesn't pose an issue

for this project's scope, but for a commercial version this might annoy some users. This could be fixed by upgraded actuators or a structural redesign. Another more significant issue could be the wear on the components. In my testing, and since the robot wasn't built too long ago, I didn't observe wear on any component except the old MG996R servos I used in the first version. But, after changing over to the TD-8125MG model, I didn't notice any wear or degrading performance. This is due to the higher build quality and torque spec of the servos.

Overall, the physical tuning of the robot was an iterative process: I went back and forth between adjusting the servos and joints, to adjusting the control code. I even adjusted the AI prompt a couple of times to change the robot's capabilities and range of movements.

The physical part of this project taught me a lot about servomotors, power delivery circuitry, battery configurations, and most importantly, the inner workings of walking robots.

5 EXPERIMENTAL RESULTS AND DISCUSSION

5.1 Example Interaction Scenario

To evaluate the integrated system, I conducted a series of test interactions that simulate how a user might interact with the robot. A representative scenario is the following (a full conversation can be found in Appendix B):

I start by saying, "Hello robot!" to the robot. The robot's microphone records this, Whisper transcribes it as "Hello robot!", and the text is sent to GPT-4 along with the sensor data (e.g. the ultrasonic sensors detect distances over 50 cm, so no immediate obstacle) and the camera feed. The GPT assistant processes the image, sensor data and camera picture, recognizes a human (me) in the picture, and responds in a friendly manner with the greeting motion. It returns this JSON:

```
{
    "movement": "greet",
    "steps": "0",
    "speech": "Hello, there! How can I help?"
}
```

The Raspberry Pi parses this and sends the speech to the TTS system and the commands over to the Arduino. The robot's speaker outputs the voice audio, while the servos start moving in the predefined position. This greeting move is coded to have the robot sit on his hind legs, lift its left arm and wave it a few times, returning to the sitting position afterwards.

Next, I ask the robot to walk forward - "Can you walk forward, please?" The transcription still picks this up, despite the background noise recorded by the microphone from the servos, giving us "Can you walk forward, please?". The assistant receives this, plus sensor data (there is enough space for the robot to move) and a picture of the room he is in, with plenty of space to explore, and me staying in the center. The assistant understands it as an instruction to start walking. It responds with:

```
{
    "movement": "forward",
    "steps": "3",
    "speech": "Sure, I will walk forward."
}
```

The Pi speaks "Sure, I will walk forward." through the speaker and sends the "forward" command to Arduino. The Arduino executes the walking gait sequence 3 times. The robot rises from its sitting position and moves forward a for 6 steps (a movement gait does 2 steps, one for the right legs and one for the left legs - so 30 cm forward in total). This takes about 15 seconds to complete. The robot starts to move towards me. At the end of the gait, the Arduino sends a "Finished instruction" string, which the Pi uses to know that it finished moving.

Next, I added a box in the front right of the robot, to test its pathfinding, and said "Continue going forward". Like before, the transcription was accurate and the assistant got the instruction to move forward. But because it detected the box as an obstacle via the left distance sensor (which is facing right) and into the camera view, the reply was a solution to the problem:

```
{
    "movement": "left",
    "steps": "3",
    "speech": "I detect something in front of me. I cannot move forward
safely. I am turning left to avoid the obstacle"
}
```

The robot replies and starts turning in place. This motion takes 5 seconds to rotate approximately 15 degrees and the LLM requested 3 steps, so after 15 seconds, the robot now faces a new direction, away from the box, 45 degrees to the left.

The back and forth would continue like this, and when there would be no instructions from the user (I wouldn't say anything in the microphone), the robot would either ask if I wanted it to continue going forward and stand in place, or just try to ask for a new command. He occasionally turns for 1-3 steps to look towards a person in the room when deprived of commands.

I also tried to test the Whisper transcription capabilities by speaking gibberish or too quietly into the microphone. This resulted in some random words at times, or empty transcriptions. These prompted the robot to ask for clarifications:

```
{
    "movement": "stand",
    "steps": "0",
    "speech": "I'm sorry, but I didn't understand your command.
Could you please repeat it?
}
```

I even tried speaking to the robot in Romanian, since I am running Whisper in translation mode, and it ended up transcribing correctly in those cases as well.

These test conversations helped me observe the following:

- **Speech Recognition Accuracy:** Whisper returned correct transcriptions consistently for my speech. Even multi-part commands like "come here, walk forward" were transcribed with all key words intact. This high accuracy is important, because any misinterpretation could make GPT misunderstand the instruction (e.g. "turn left" misinterpreted as "turn that" wouldn't make much sense).
- **GPT Understanding and Reasoning:** GPT-4 interpreted my intent well, including direct commands, questions, and complex sentences. It followed JSON format well. I didn't observe format errors after the prompt was refined. It also made reasonable decisions with sensor data – refusing to go forward when an obstacle was detected. It matched my expectations well.
- **Physical Execution:** The executed movements matched the commands. The forward movement was sluggish because of the cheaper servos I first used, but it still got there. The turns are harder since they need balancing, but the robot did indeed turn in the desired direction. A good thing is that the robot did not fall during these actions, especially after changing to the more powerful servos. There were some oscillations when a leg lifted due to the low quality of the servos, but that was again solved by replacing them with better ones. The obstacle avoidance test worked – the robot did not move further forward when the box was placed in path, and verbally acknowledged the obstacle. That is a good demonstration of integrating perception into decision-making, which many simple robots lack.
- **Latency:** I measured the delays in this scenario: the initial greeting took 5s from end of the speech recording to the robot's reply (Whisper - 1.5s, GPT - 3.5s). Movement commands took longer mainly due to movement time, not processing. For example, after "walk forward", the voice reply came in 5s, then movement took 15s. This is due to the way the movement gait is written. For the Continue going forward" command that was refused, it was 6s to reply. I didn't get bothered by those delays, but adding 5 more seconds would've made the robot seem slow, users would get bored. So keeping interactions short and the robot giving some sign it's processing (even a small LED blinking on robot) would helpful. I didn't implement anything other than the speech to signal to the user that the robot is processing the information.
- **Error Handling:** When I purposefully tried to break the transcription: e.g. mumbling or gibberish, Whisper might return "[inaudible]" or some random words. Thanks to the prompt though, GPT replied that it didn't understand and ask for a new command. When I treid speaking clearly in another language, the transcription worked well. This system could be improved to detect if Whisper has low confidence and then it would ask the user to repeat. I didn't implement that, but it is a good idea for future work.
- **Conversation memory:** Since I used the assistants API, the AI has access to the conversation history, so no context was lost. It can't remember other conversations due to the thread change though, so when the robot is restarted it starts fresh, with no conversation history, just the system prompt. This didn't represent a problem, but it

would be interesting to implement lasting memory in future versions.

The example scenario confirmed the core functionality: the pipeline for voice command to action works well, and the robot can integrate dialogue with movement. When tested with friends, they quickly started personifying the robot, so it ended with good feedback. This shows the project's goal of creating an interactive robot companion was mostly completed.

5.2 System Performance - Latency and Accuracy

For this section, I measured the performance in terms of speed, accuracy and user experience.
Latency Breakdown:

- **Audio Transcription Time:** Using the Whisper API, the average time for a 10-second sentence was 2.5 seconds to receive the text. This is very fast, so no problems with the transcription. Using a local Whisper model on the Pi would take almost 15 seconds for same input, so using the API clearly helped.
- **GPT-4 Response Time:** GPT-4 is by far the "slowest", since the api call registers the biggest latency. For some short prompts (user prompt for 100-200 tokens, and around 50 tokens for a response including the JSON overhead), the response time averaged 3.5 seconds. There were outlier cases: sometimes the first query of a session took 5-6 s, maybe due to loading. The following ones were around 3-4 seconds. Faster models, like the o4-mini would be under 2 s for similar content, but I chose GPT-4 for the better reasoning. The network also played a big part in the latency, since the loading of the image to the model added time to the query.
- **Local Processing:** The processing time for the Pi for assembling the prompt JSON, making requests via Python SDK, and parsing response is practically non-existent. I measured, at most, 100 milliseconds for these steps (using the time library in the code), so basically instant. Serial sending to Arduino is also almost instant (at most, tens of milliseconds).

In total, for a user command that makes the robot move, there is a 5-6 second delay (transcribe + gpt + parse + movement start). This was consistent. The user experience is better, because the speech fills in those gaps. The movement times are varied, since a simple pose command would be almost instantaneous, while a longer walking cycle could take up to a minute. For a user command that only triggers speech from robot, there is a similar 5-6 second delay to hear the response. These figures are quite good for a cloud-based system. It feels similar to using Gemini/Siri/Alexa in terms of response time, which a lot of people are used to.

Reliability and Accuracy:

- **Speech Recognition Accuracy:** In tests with multiple speakers (I tried two different voices - mine and a friend's), Whisper had near complete accuracy for clearly spoken

English commands or questions. It even handled other languages, if spoken clearly - like Romanian. I observed no instance of confusing one command with another (like "left" vs "right") – probably because those sound quite different. A harder one would be "forward" vs "for word" or something, but context helps. One test I also did was "flour" vs "forward": I threw a nonsense "flour", Whisper heard "forward" incorrectly, and GPT output a forward action. That was a *misactivation* due to hearing a similar sound. Thankfully, "flour" is not a common thing for a user to say, especially without context. I could add a command confirmation step to fix these scenarios, but I think that would get annoying for the user, unless it's in special cases based on Whisper's confidence level and context. Overall, I was very satisfied with Whisper's performance. The only limitation is that it doesn't do wake-word detection by default, and implementing the wake word logic for a small percentage of user satisfaction was not worth in the project's scope. It also has an upper limit for 30-second audio segments, but since the architecture only records 10-second audio batches, it didn't pose any issue.

- **GPT Response Accuracy:** By accuracy, I refer to the AI's response following the JSON schema described to him in the system prompt. So, it needs to contain the 3 fields: "movement", "steps" and "speech". In my testing since I started working on this project, I didn't encounter issues with the reply format. The potential "issue" is especially solved when using the assistants API instead of the GPT API, since it has a selectable option to output a JSON. So again, the accuracy of the GPT response is pretty much perfect. This doesn't mean that the movement commands are completely relevant to the user instruction, or the responses are always factually correct. But that is another discution entirely.
- **Sensor Integration Effectiveness:** When an obstacle was present, the AI always mentioned it, refused motion when asked to move into it, or tried turning around. This means that including sensor readings in the prompt and instructing the LLM how to use them was useful. If the sensor reading allowed for some room to move (10-20 cm), GPT could still choose a forward, but mention it. That happened in one trial: a box was placed approximately 20cm in front of the robot, the command was to advance forward, GPT returned the "speech" as "I'll move forward carefully.", with the "move": "forward" and "steps" as "2". The robot moved, but because it only moves 5 cm per step, it didn't actually hit the object (it ended up 10 cm away). Then next command it tried turning around. This is an uncertainty in how GPT might handle the sensor values. GPT can freely choose how to move. For future work, it would be recommended to add more guidelines to tem system prompt, and even add exceptions in the control code.
- **Physical Accuracy:** The robot's movement accuracy is moderate at best. This can be considered as the weakest part of the project, since the cheap hobby-grade *MG996R* servos I used weren't that accurate and also not powerful enough to keep the robot straight. They worked for the novelty of making a quadruped robot move, but they weren't up for the task. So, I ended buying a more expensive model, the *TinyTronics TD-8125MG 25Kg* servos. These pretty much solved the robot's walking issues, at the cost of my funds. And since the walking motions were programmed by me, with no

background in kinematics, they still had some small amounts of drift. For example, after around 10 steps of walking forward, the robot would end up drifting approximately 10 cm in either direction, and the turning command varied on average for 2-3 degrees from the expected 15.

- **Battery life:** As explained in Chapter 4, a full charge (around 2200 mAh) of the 4-cell 18650 battery, gave the robot 15-20 minutes of battery life (with movements form the servos and the Pi running). The biggest consumer was by far the 12 servos, which could draw up to 2.5-3 Amps when stalled. This battery life is decent for testing and quick demos, but future versions that might rely less on user input will need an upgraded battery. The good part is, since a 4-cell 18650 tray was used to support the cells, they are hot-swappable. So, a battery change takes seconds. That if there are 4 other 18650 cells to spare, of course.

We can conclude that the performance is decent for a prototype interactive robot. The main limitation is the latency for real real-time control, so we can't use GPT to adjust every footstep of the robot, since the delay is of a couple of seconds. That has to be processed locally. But for the interaction with people part it's fine. For any scenario requiring fast reflexes (like slipping or someone pushing the robot), the system can't process that via the cloud. It would need local adjustments running on the control hardware. That is a common issue for cloud brains.

The user experience was good, those who tried speaking with the robot found it responsive enough and its responses enjoyable. People are used to instant reactions in face-to-face conversation and a processing delay might weird them out, but since voice assistants have existed for a while, the delay is often subconsciously accepted. I could fill that gap with more robot expressions (some lights or idle animations to show it's computing). For example, Alexa devices spin a light while thinking. Future versions could add some lights to the robot for "thinking" effects.

Precision of Dialogue: Another point is the precision of the GPT responses and their structure. For example, in responses, it sometimes added an explanation in the "speech" field. When refusing a command, it can say "I cannot because I detect an obstacle. Do you want me to turn left or right?" That's interesting, since I didn't explicitly instruct it to explain why it can't procede or what sollutions to give, the model just did it. And it's a good thing, since it adds extra personality to the robot. So, GPT added value by giving reasons and asking for directions in its speech. The generative aspect of LLM's makes for a more natural conversation than a rule-based robot.

Finally, it's time for the **Evaluation of Objectives Fulfillment**. Looking back on the aim and objectives from the first chapter, we can say the following about the goals set on this project:

- **Voice and text understanding:** Achieved, thanks to the OpenAI models' performance.

- **Cloud-based reasoning and response actions:** Achieved, GPT returns valid actions integrated with text thanks to the system prompt.
- **Physical robot build:** Achieved, the robot works in the way it was designed.
- **Conversational ability:** Achieved, via the microphone recording and TTS output.

The integration of all these parts into this complex system was probably the biggest success: many similar projects encounter issues while trying to combine all these subsystems, but from the tests I can say they all work well together. The only part that could be vastly improved is the locomotion system - some extra time put into the walking gaits and extra movement variants could enhance the user experience and the robot's capabilities of navigating its environment.

5.3 Limitations and Observations

Despite the working prototype, this project has several limitations and areas it didn't excel:

- **Physical Limitations:** This is by far the weakest side of my project. The robot cannot navigate complex environments, since it has no vision-based mapping or path planning. It cannot climb steps or handle uneven ground. So it is only fit for flat spaces with minimal obstacles. This limits the real-world usecases, so it's more of a proof-of-concept platform. While Boston Dynamics' Spot can operate autonomously over large spaces, my version can't – it needs direct user guidance for complex moves.
  The robot moves slowly and cannot carry anything big. It can only be an "assistant" in conversation, it can't do physical work. This was outside the scope, but worth noting.
  Also, if the robot falls over or gets stuck (which can happen if it slips or if a servo fails), the system can't recover. For example, if it falls, it can't lift itself without a pre-programmed sequence - which I didn't do. So, human intervention is needed in those cases, which is common in such prototypes.
- **AI and Interaction Limitations:** The robot is dependent on internet. Without a reliable connection, it is stuck in a loop of trying to contact the OpenAI servers, being reduced to an idle state. The robot would be pretty much "braindead", except from maybe manual or pre-scripted behaviors, which weren't in the scope of this project. This raises reliability concerns – if connectivity is lost or latency spikes, the performance degrades. I designed this robot assuming the network would always be present. For field use, it will likely need an offline mode or at least a fallback to a local model and some predefined routines.
  And even when the network is present, there are still some limitations: There is the cost for the API usage, which would pose a bigger problem for a production variant, but in this case it's negligible. Also, the general GPT-4o model that I used for the assistant is not finetuned for the spatial reasoning and step planning, so it can only execute simple basic tasks. For example, it might figure out exploring the environment it's located in,

but it can't bring a ball back to the user from where he threw it.

Another popular discution in the cloud robotics space is the trade-offs between local and processing.

The **pros** for the **cloud sollution** are the access to a very capable AI model and the Whisper transcription API, which both can't run directly on the Pi, or any microprocessor at this time. They allow interaction and processing at levels that couldn't be achieved with local computing, only maybe on servers or powerful desktops, but this would again mean that the processing is done remotely. Also, when future models will be released, switching over to them will be very simple, only the configuration would need to be changed.

The **cons** for the **cloud version** are, as mentioned earlier, mostly represented by the need of a network connection and its latency. There are also some privacy concerns, since the audio and data is sent to the cloud on a private server. This doesn't pose a problem for the scope of my project, but a home assistant version of this robot might make users uncomfortable with this constant cloud transmission.

So, the **fix** is to basically just **use both**: A good idea is to use the cloud only when needed and keep some local fallback models running on the limited hardware. For example, the robot could have a wake word model for the critical commands like "stop", and only use the cloud for advanced planning routines.

5.4 Improvements and Future Work

Despite being a quite complex project, there are still many improvements and upgrades to be made. The most relevant ones are listed below:

1. **Enhanced Autonomy and Sensing:** To move past simple obstacle avoidance, integrating more sensors (like a LIDAR or a depth camera) and using them for environmental mapping could allow the robot to navigate autonomously to a target. The Raspberry could run *Simultaneous Localization and Mapping (SLAM)* algorithms to improve the robot's environmental awareness. GPT could then be used for high-level decision-making (e.g. deciding to go in another room or searching for an object), while the low-level control is handled by classic robotics algorithms. This hybrid approach is promising – LLMs for reasoning and classic control for precise movement – because the LLMs are better at reasoning and making high-level decisions.
2. **Local Model Execution:** Running smaller models on the PI can reduce the dependency on the cloud. For example, running the whisper.cpp local version and a local LLM (using *Ollama* would make the robot fully autonomous. They don't rise up to the cloud API's performance levels, but still work decently well. The better option would be to use them as fallback, so they would run locally when the OpenAI servers are unreachable. Another

option is to use a local server (on-premise) that is running more complex models, and connect to it via the local network.

3. **Better Real-Time Control:** The robot could benefit greatly from a subsystem for reflexes like balancing or immediate stopping for detected obstacles. These changes would have to be implemented at the microcontroller level (with IMU feedback loops, perhaps using a PID to adjust leg positions if tilt is detected). This would also mean the shift to a more advanced control schema, like using inverse kinematics instead of direct angle control, and maybe a more powerful microcontroller. The Arduino could be replaced by a *Teensy Development Board* to be able to compute IK in real-time for active balancing. This would allow the robot to walk better and handle rougher terrain.
4. **Improved Robot Hardware:** Future versions of the robot could add *servo feedback* or more advanced actuators, which would provide data to make sure that the legs are where they should be, detecting if a servo is stalled or overloaded. Also adding a small *robotic arm* to the robot would greatly increase its capabilities: from being able to play catch to opening doors and helping the user with simple tasks.
5. **Security:** Having a robot connect to the internet creates quite a few security concerns. First off, the system has to be protected from malicious actions, or from non-trusted users. For example, if a bad actor were to ask the robot sensitive details about its user, he would simply give them out. LLM's are famously known to mess up and give out sensitive details when safety precautions were not put in place. Implementing a voice detection algorythm on the Pi to differentiate from the "admin" user could help avoid those kinds of scenarios.

This project successfully met its goals of creating a cloud-connected robot and proved the advantages of using Large Language Models for robotic control. The robot hardware is much more limited, requiring careful integration so that the AI can limit its commands to the capabilities of the robot. This work is a step in the direction of cloud robotics, demonstrating both the potential and the many challenges of this approach.

6 CONCLUSIONS

This diploma project explored and implemented a novel integration of cloud AI services with a quadruped robot platform. I started out by trying to give a 3D-printed robot a "brain in the cloud" by using a Large Language Model to interpret user voice commands and make high-level decisions, while managing the low-level motion control on dedicated hardware.

The project achieved this aim and demonstrated the promising combination of modern AI and robotics. The robot, in his scope, can listen, speak, and physically act, which is a strong proof of concept for an AI-driven robotic assistant. It succesfully validated that a Large Language Model can be more than a disembodied chatbot, it can act as the control logic for a physical agent, mapping natural language instructions to real-world behavior. This involved bringing together two usually separate domains: cutting-edge Natural Language Processing and real-time embedded control. And this project shows that they can be successfully combined.

Beyond the technical accomplishments, this project offers insights into the trade-offs of cloud robotics: offloading some of the computation to the cloud can help expand a robot's cognitive abilities (as we can see with the robot handling flexible conversations and complex language), but it introduces latency and reliability problems, which have to be managed. It also underlines the importance of an integrated design: the hardware and the AI services have to be developed in parallel, so that the AI's outputs remain constant with the hardware's capabilities, and the hardware provides the AI with enough real-world sensor data to make correct decisions. My approach of configuring the AI through prompt engineering, while giving it real sensor data is a decent, albeit simplified process to achieve AI+robotics integrations.

In summary, this project contributes with a case study in giving a small robotic platform capabilities more advanced than its on-board computational power, by using cloud-based intelligence. It opens up possibilities for building more of these types of "smart" robots, that are relatively low-cost and physically simple, yet they have advanced behaviors thanks to the cloud AI. As AI models continue to advance and become more accessible, we'll likely see more robots with their "brains in the cloud," whether it's domestic service robots, interactive companions, or specialized assistants. This work serves as an early example of that idea, and the lessons I learned here could be helpful for engineers and researchers trying to combine AI and robotics in the near future. This project met its objectives and delivered an interesting result: a Spot-inspired robot dog that you can talk to and command in natural language, which represents a considerable step towards more advanced human-robot interactions.

BIBLIOGRAPHY

[1] Robotsguide, "Spot - robots: Your guide to the world of robotics." https://robotsguide.com/robots/spot.

[2] OpenAI, "Introducing gpt-4.1 in the api." https://openai.com/index/gpt-4-1/.

[3] OpenAI, "The fastest and most powerful platform for building ai products." https://openai.com/api/.

[4] N. Bartnik, "I put chatgpt on a robot and let it explore the world." https://www.youtube.com/watch?v=U3sSp1PQtVQ.

[5] KDY0523, "Spotmicro - robot dog." https://www.thingiverse.com/thing:3445283.

[6] W. P. T. N. T. U. Guoqiang Hu and Y. W. N. T. University, "Cloud robotics: Architecture, challenges and applications." https://www.researchgate.net/publication/241638237_Cloud_Robotics_Architecture_Challenges_and_Applications.

[7] Wikipedia, "Python (programming language)." https://en.wikipedia.org/wiki/Python_(programming_language).

[8] OpenAI, "Introducing whisper." https://openai.com/index/whisper/.

[9] R. d. Oliver Zweigle University of Stuttgart, M.J.G. van de Molengraft Eindhoven University of Technology and K. H. U. of Stuttgart, "Roboearth - connecting robots worldwide." https://www.researchgate.net/publication/221222976_RoboEarth_-_Connecting_robots_worldwide.

[10] Wikipedia, "Internet of things." https://en.wikipedia.org/wiki/Internet_of_things.

[11] Amazon, "Meet the new alexa." https://www.amazon.com/dp/B0DCCNHWV5?ref=aucc_web_red_xaa_evgn_tx_0002.

[12] Google, "Discover what google assistant is." https://assistant.google.com/.

[13] DigitalOcean, "What is aiaas? understanding artificial intelligence as a service." https://www.digitalocean.com/resources/articles/ai-as-a-service.

[14] TechTarget, "What is an ai assistant?." https://www.techtarget.com/searchcustomerexperience/definition/virtual-assistant-AI-assistant.

[15] OpenAI, "Assistants api overview." https://platform.openai.com/docs/assistants/overview.

[16] OpenAI, "Function calling - enable models to fetch data and take actions.." https://platform.openai.com/docs/guides/function-calling?api-mode=chat.

[17] G. Research, "Using language to better interact with helper robots.." https://sites.research.google/palm-saycan.

[18] BabyCommando, "Machinascript for robots." https://github.com/babycommando/machinascript-for-robots.

[19] V. G. A. A. . T. C. Author, "What is prompt engineering?." https://www.ibm.com/think/topics/prompt-engineering.

[20] O. Çolakoglu, "Opencat - open source quadruped robotic framework." https://hackaday.io/project/183916-opencat-open-source-quadruped-robotic-framework.

[21] P. Bittle, "Programmable robot pets for everyone to build code." https://www.petoi.com/.

[22] E. Tkachenko, "Audio transcription with openai whisper on raspberry pi 5. can text recognition be fast on a small device?." https://gektor650.medium.com/audio-transcription-with-openai-whisper-on-raspberry-pi-5-3054c5f75b95.

[23] MKme, "Spotmicro quadrupedal 3d printed robot project." https://github.com/MKme/quadrupedal-robot.

[24] C. Locke, "Welcome to the nova sm3 project." https://novaspotmicro.com/.

[25] M. G. L. X. T. U. B. C. H. Z. Y. U. Q. C. Augat, Peter Berufsgenossenschaftliche Unfallklinik Murnau and C. A. o. S. H. C. Song, Yuntao Institute of Plasma Physics, "An inverse kinematic model of a manipulator - equipped with an end effector - is a function which allows to calculate a manipulator configuration corresponding to a given end effector location (position and orientation).." https://www.sciencedirect.com/topics/engineering/inverse-kinematics.

[26] venkygaming18, "Quadruped robot - alpha! esp32 based spot micro robot.." https://www.instructables.com/Quadruped-Robot-Alpha-ESP32-Based-Spot-Micro-Robot/.

[27] L. L. S. Zhang, "Feedforward and feedback control for gait and balance." https://www.researchgate.net/publication/292936851_Feedforward_and_feedback_control_for_gait_and_balance.

[28] IonutBirjovanu, "A robot with the brains in the cloud." https://github.com/IonutBirjovanu/A-robot-with-the-brains-in-the-cloud.

[29] 3DPrinting.com, "What is 3d printing?." https://3dprinting.com/what-is-3d-printing/.

A SYSTEM PROMPT

```
You are a robotics control system for a quadruped robot. You receive:
1) The data from the robot’s distance sensors and an image of what he sees
through its camera. The camera and the 2 distance sensors are frontally
mounted. The camera is oriented straight ahead, and the sensors are 5cm
apart(on one side and the other of the camera), each rotated inwards by
50 degrees.
2) A user instruction - a voice command transcribed into text.

You must output a JSON object with the following format:

{
  "movement": "STRING",     // The actions that the robot will execute
- must be one of: forward, backward, left, right, stand, flat, sit, paw,
greet . The actions are sepparated by commas
  "steps": list of numbers,  // Number of steps done for each movement,
only the forward,backward,left and right commands require a number of
steps, for the others the number can be 0
  "speech": "STRING"      // The text the robot should speak via TTS
}

Clarification and brief explanation of the poses:
- forward: the robot steps forward once, for around 5cm
- backward: the robot steps backward once, for around 5cm
- left: the robot spins left for one step, for around 15 degrees of rotation
- right: the robot spins right for one step, for around 15 degrees of
rotation
- stand: the default position of the robot, standing straight
- flat: the robot lies down, touching the ground with its belly, similar to
its powered-down state
- sit: the robot sits like a dog, good for an idle position
- paw: the robot stretches one of its front feet in the front
- greet: the robot waves with an arm

Example of a valid json:
{
    "movement": "forward,left,sit",
    "steps": "2,5,0",
    "speech": "Hello, I am moving forward, turning left, then sitting down"
}

You can also only send one movement, like this:
{
    "movement": "forward",
    "steps": "5",
    "speech": "Hello,I want to explore, moving forward"
}

There is also the option of a direct position control:
You could have a json like this:
{
"movement": "1:30 2:80 5:100,2:60",
"steps": "2,0.5",
"speech": "Hello, I am moving the servos directly now"
}

In this way, you can directly set the angles for each servo on the robot.
The movement consists of a list of servo indexes and their values.
Different positions are sepparated by commas. The steps value now
represents the amount of time each direct position is held. The
positions don’t reset after the timeout, the legs remain in those
positions, but you can use the delays for orchestrating new movements.
You can set a position, put a step value that determines the time it
holds that, then add a new movement with a new time period to create
your own complex movements.

The indexes of the servos are:
Each leg: [hip, shoulder, elbow]
{0, 1, 2},   // Front left
{4, 5, 6},   // Front Right
{8, 9, 10},   // Rear Left
{12, 13, 14}  // Rear Right

The indexes go from 0-15, but we’re not using the 3, 7, 11 and 15
indexes.

The positions for each servo that the robot uses in the standing
pose are:
{40, 50, 65},  // FL
{100, 90, 80},  // FR
{45, 100, 80},  // RL
```

```
{90, 60, 70}   // RR

And the positions for the laying flat pose are:
{0, 15, 65},  // FL
{140, 130, 80},  // FR
{0, 140, 80},  // RL
{140, 10, 70}   // RR

The angles go from 0-180, but I recommend only going up to ~140-150,
and be careful with the elbow 2, 6, 10, 14 servos - they have less
movement space because of the robot’s joints, and can try to bend
into the frame, so not reccomended to flex them a lot.

There is also the option of combining the command types. So, if
you want to make sure that after some custom movement commands
the robot takes a known position, you can issue a default
movement: forward, backward, left, right, stand, flat, sit,
paw, greet
So like this:
{
"movement": "1:30 2:80 5:100,2:60,stand",
"steps": "2,0.5,0",
"speech": "Hello, I am doing 2 direct movements, then standing"
}

When in need of direct position commands, it’s better to first
ask the user if you should send those instead of just normal
movements, unless directly specified to use direct positioning.

Give the robot a playful and curious personality, eager to
discover the environment and people nearby.

Guidelines:
- Always return a valid JSON.
- If the space permits it but the instruction is unclear, try
to turn the robot to the user and ask for clarification.
- If the users request is unclear, and the space is constrained
so that you can’t turn, set the movement to a stationary position,
steps = 0, and in speech, clarify what you need.
- As long as the space permits it, try and explore the environment
you’re in, unless given a clear command from the user.
- Steps values should be integers.
```

B SAMPLE CONVERSATION WITH THE ROBOT

```
"Hello robot!"
```

```
{
    "movement": "greet",
    "steps": "0",
    "speech": "Hello, there! How can I help?"
}
```

```
"Can you walk forward, please?"
```

```
{
    "movement": "forward",
    "steps": "3",
    "speech": "Sure, I will walk forward."
}
```

```
"Continue going forward"
```

```
{
    "movement": "left",
    "steps": "3",
    "speech": "I detect something in front of me. I cannot move forward
safely. I am turning left to avoid the obstacle"
}
```

```
[Unintelligible]
```

```
{
    "movement": "stand",
    "steps": "0",
    "speech": "I'm sorry, but I didn't understand your command.
Could you please repeat it?     }
```