Imperial College London

Department of Electrical and Electronic Engineering

EEE2/EIE2 Electronics Design Project

Balance Robot

Group - WhatdoUmean

Group members:
Aobing Yin
Dawud Khan
Wei Xu
Will Zhang
Yikai Su

Submission date: 16/06/2025
Submitted to: Ed Stott

1 – Introduction

1.1 - Motivation

This project originated as a core academic challenge within the EEE/EIE curriculum, with the defined aim of building a robot that can balance on two wheels. The fundamental technical problem was rooted in classic control theory: to stabilize the inherently unstable "inverted pendulum" dynamics of the provided robot platform using a control algorithm.

However, the project brief deliberately left the robot's high-level function undefined. A key requirement was for the team to choose and develop a specific application that would "demonstrate interaction with human users or the environment". This open-ended nature was the primary spark for the project's direction, shifting the focus from a pure engineering exercise to a design challenge for us in human-robot interaction.

Faced with this choice, our team conceptualised a high-impact application: a prototype for a robotic guide companion for the visually impaired. This decision framed the project's central challenge. It was no longer sufficient to just achieve balance; we had to augment this core stability with the added complexity of a multi-layered system involving advanced sensing, multi-modal remote control, and AI-driven interaction to create a truly assistive and interactive platform.

1.2 - Aims and Objectives

To transform this vision into a tangible outcome, a set of specific, measurable objectives was established, directly expanding upon the requirements detailed in the project brief. These objectives were structured around four key pillars of development.

Pillar I: Core Robotic Platform & Dynamic Stability

- Primary Objective: To implement a control algorithm that enables the robot to robustly balance on two wheels, with its centre of gravity maintained above the wheel's axis of rotation.
- Secondary Objective: To develop a remote-control interface allowing a user to manually move the robot in two dimensions across a flat surface.

Pillar II: Intelligent Human-Robot Interaction

- Primary Objective: To fulfil the requirement for human interaction by developing an advanced, voice-driven command interface using natural language processing. This system would allow a user to issue intuitive, high-level commands.

- Secondary Objective: To incorporate a speaker system for direct auditory feedback, allowing the robot to confirm commands and report its status, thus reinforcing the 'companion' paradigm.

Pillar III: Environmental Awareness & Autonomous Navigation

- Primary Objective: To "demonstrate a form of autonomous behaviour" by implementing a "walk-with-me" functionality pertinent to the guide-dog application, using sensors to interact with the user's position.

- Secondary Objective: To add sensing components to the robot that allow it to detect obstacles, ensuring safe navigation through its environment.

Pillar IV: System Monitoring & Physical Integration

- Primary Objective: To create a remote user interface that displays useful information about the power status of the robot, specifically "power consumption and remaining battery energy".

- Secondary Objective: To augment the provided chassis with a custom-built "head unit" designed to suit the chosen demonstrator application by housing sensors and interactive components.

The ultimate measure of success for this project would be the effective integration of these modular subsystems, demonstrating a cohesive system that is not only technically complex but also intuitive for a user to interact with.

1.3 - Scope and Report Structure

The project's scope was extensive, requiring a multi-disciplinary approach that encompassed: Control Systems Theory for the core balancing algorithms; Embedded Systems Programming for real-time hardware control; Electronic Circuit Design for interfacing sensors and managing power; Network Engineering for the local and cloud communication stack; and Artificial Intelligence for the natural language command interface.

This report is structured to mirror the project's development lifecycle, starting with foundational systems and progressively building towards the integration of advanced features. The document follows a specific order:

1. **Control System Design:** Detailing the foundational PID architecture for stability and movement.

2. **Communication and User Interface:** Explaining the stack that enables remote operation and data telemetry.

3. **Higher Functionality:** Showcasing the advanced autonomous and AI-driven features built upon the core platform.

4. **Battery Monitoring:** Explaining the methods we used to implement battery monitoring into our robot

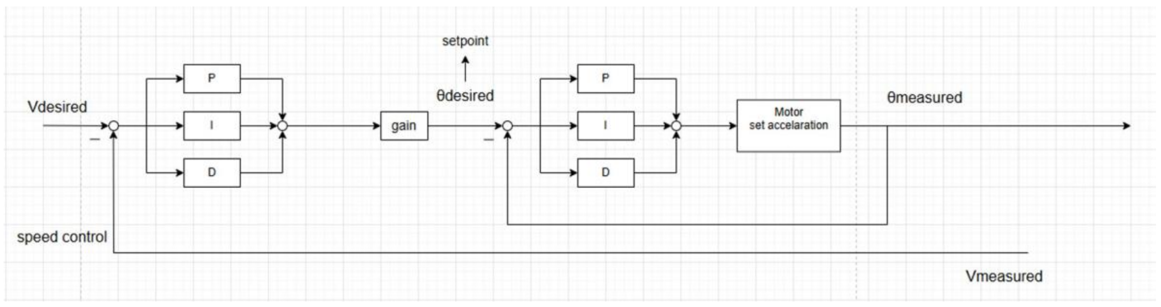
This structure reflects the integration process itself, where different layers were developed in parallel and integrated at last. The report concludes with details on the supporting hardware, a summary of the project's achievements, and an appendix.

2 - Control System Design

The robot’s stability and movement are governed by a sophisticated control system, the design of which was a primary focus of the project. The architecture is built on a foundation of PID control theory, tailored specifically for the dynamic challenge of a two-wheeled, self-balancing inverted pendulum.

2.1 - Dual-Loop PID Architecture: Balance and Speed

The core of the control system employs a nested, dual-loop PID architecture, a design choice that decouples the critical task of balancing from the secondary task of movement.



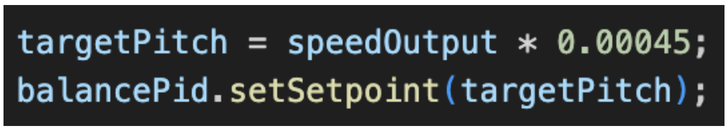
2.1A - The Inner Loop (Balance Control):

This is the most critical loop, responsible for maintaining the robot's balance second by second. It functions as a classic negative feedback system, constantly working to minimise the error between the desired pitch angle (a vertical setpoint) and the actual pitch angle measured by the onboard sensors. The output of this PID controller directly dictates the required motor acceleration, named as balanceOutput as shown in the figure below. When the robot tilts, this loop calculates the precise torque needed - delivered via wheel acceleration - to counteract the fall and restore the robot to its equilibrium position.

```
step1.setAccelerationRad(balanceOutput - turnVal - yawCorrection);
step2.setAccelerationRad(balanceOutput + turnVal + yawCorrection);
```

2.1B - The Outer Loop (Speed Control):

This loop gives the robot purposeful movement. It translates a desired speed command into a temporary change in the balance setpoint, as shown in the figure below. To move forward, the outer loop slightly shifts the desired pitch angle forward, causing the inner loop to accelerate the wheels to "catch up" and maintain the new tilted equilibrium. To slow down, it shifts the setpoint backward. A carefully tuned gain of 0.00045 is applied to the output of this loop. This is a critical detail, as it scales the larger output of the speed controller down to the very small and delicate adjustments of the pitch angle required for smooth, stable motion.



```
targetPitch = speedOutput * 0.00045;
balancePid.setSetpoint(targetPitch);
```

Experimentation showed that a targetPitch value in 0.00-0.01 led to slow acceleration, while 0.07-0.09 produced faster but uneven acceleration. A value of 0.10 caused the robot to accelerate too quickly and lose balance. Ultimately, a targetPitch in 0.02-0.06 was found to provide fast acceleration and good balance, corresponding to a gain of 0.00045.

This hierarchical structure, where the speed controller commands a desired tilt and the balance controller executes it, is fundamental to the robot's stability.

2.2 - Loop Dynamics and Tuning

System stability hinges on the significant difference in the response speeds of the two loops. The inner balance loop was designed to be substantially faster than the outer speed loop, allowing it to make rapid micro-corrections to maintain balance before the outer loop's slower adjustments to overall speed can destabilise it. The bandwidth, or reaction rate, of each loop is primarily determined by its Proportional (P) gain. The final tuned system exhibited an inner loop P value of approximately 900 and an outer loop P value of 4.5, comfortably satisfying the common engineering rule-of-thumb that the inner loop's bandwidth should be at least five times greater than the outer loops.

The PID tuning process was conducted empirically, following a structured methodology to ensure a stable yet responsive system:

1. **Integral (I) Term:** The I-term, which typically corrects for long-term error, was intentionally set to zero. Any minor steady-state error in the robot's physical resting point was instead corrected by calibrating a fixed "bias" value. This simplified the tuning process by eliminating the risk of integral wind-up, a phenomenon that can cause runaway oscillations.

2. **Proportional (P) Term:** The P-term was incrementally increased to provide a fast, proportional reaction to tilt. This was a careful balancing act: too low a P-value resulted in a sluggish, "lazy" response, while too high a value caused aggressive over-correction and rapid oscillations. In addition, the acceleration value was limited to a range of -70 to +70 to prevent jerky, unstable reactions.

3. **Derivative (D) Term:** With a responsive P-term in place, the D-term was introduced to act as a damper. Its role is to anticipate the future trend of the error. It was observed that both insufficient and excessive D values were detrimental: too small a value failed to prevent overshoot, while too large a value introduced high-frequency chatter. A suitable middle value was chosen to provide critical damping, resulting in a robot that could return to its setpoint smoothly and quickly with very small oscillations. To further smooth out the high-frequency jitters when controlling speed, an exponentially weighted moving average (EWMA) was used. EWMA places a greater weight on the most

recent speed measurements and hence reacts more significantly to recent speed changes than a simple moving average, which was desirable.

2.2A Testing Results and Discussion

Testing Methodology

The balancing robot was evaluated under various controlled conditions to assess its stability and responsiveness. To minimise human-induced variability, a self-induced impulse method was employed to ensure consistent testing conditions. The key evaluation metrics included stability, responsiveness, and oscillations.

Metrics for Evaluation

- **Stability**: Assessed by measuring the duration for which the robot can maintain balance without toppling.
- **Responsiveness**: Determined by the time taken by the robot to correct its position following a disturbance.
- **Oscillations**: Frequency and amplitude of oscillations around the balanced state.

Test Results

- **Stability**: The robot successfully maintained an upright position for 6 minutes during continuous testing. This result indicates a high level of stability.
- **Responsiveness**: The robot was able to correct its tilt within 2 seconds after experiencing a disturbance.
- **Oscillations**: when balanced over a period of 50 seconds, the amplitude of oscillations was within $[-0.17, 0.17]$ radians with an average tilt of 0 radian.

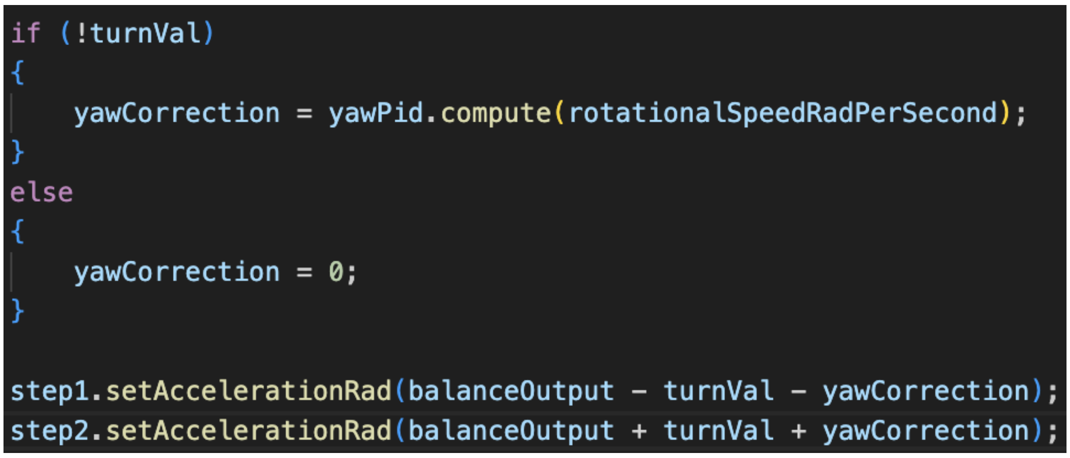
Discussion

The tuning process revealed a trade-off between responsiveness and stability. Increasing the proportional gain Kp enhanced the robot's responsiveness but also introduced a higher risk of oscillations. Incremental parameter adjustments and rigorous testing were essential to achieving optimal performance. The integration of filters for speed measurement played a significant role in stabilising the inner and outer control loops.

Potential future enhancements may include the use of advanced techniques such as gain scheduling or model-based tuning to accommodate varying operational conditions. Despite these possible improvements, the current configuration effectively demonstrates the capability of the cascaded loop control architecture in maintaining the robot's balance.

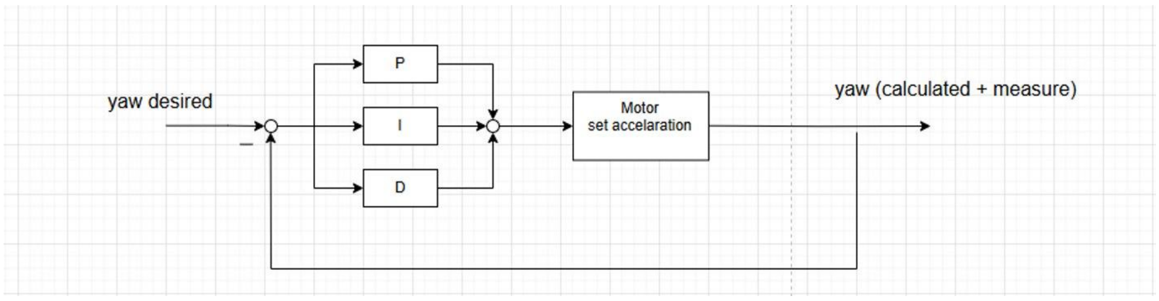
2.3 - Yaw and Orientation Control

Inspired by drone stabilisation technology, a third independent PID loop was implemented to control yaw, or the robot's rotation about its vertical axis. This allows the robot to maintain a straight orientation and execute precise turns. Yaw rate is calculated in real-time from the difference in the measured speeds of the two wheels, divided by the distance between them. This calculated yaw rate is fed into its own PID controller, whose output makes corrective adjustments by creating a slight difference in the acceleration of the two wheels, thereby steering the robot to maintain its heading.



```
if (!turnVal)
{
    yawCorrection = yawPid.compute(rotationalSpeedRadPerSecond);
}
else
{
    yawCorrection = 0;
}

step1.setAccelerationRad(balanceOutput - turnVal - yawCorrection);
step2.setAccelerationRad(balanceOutput + turnVal + yawCorrection);
```

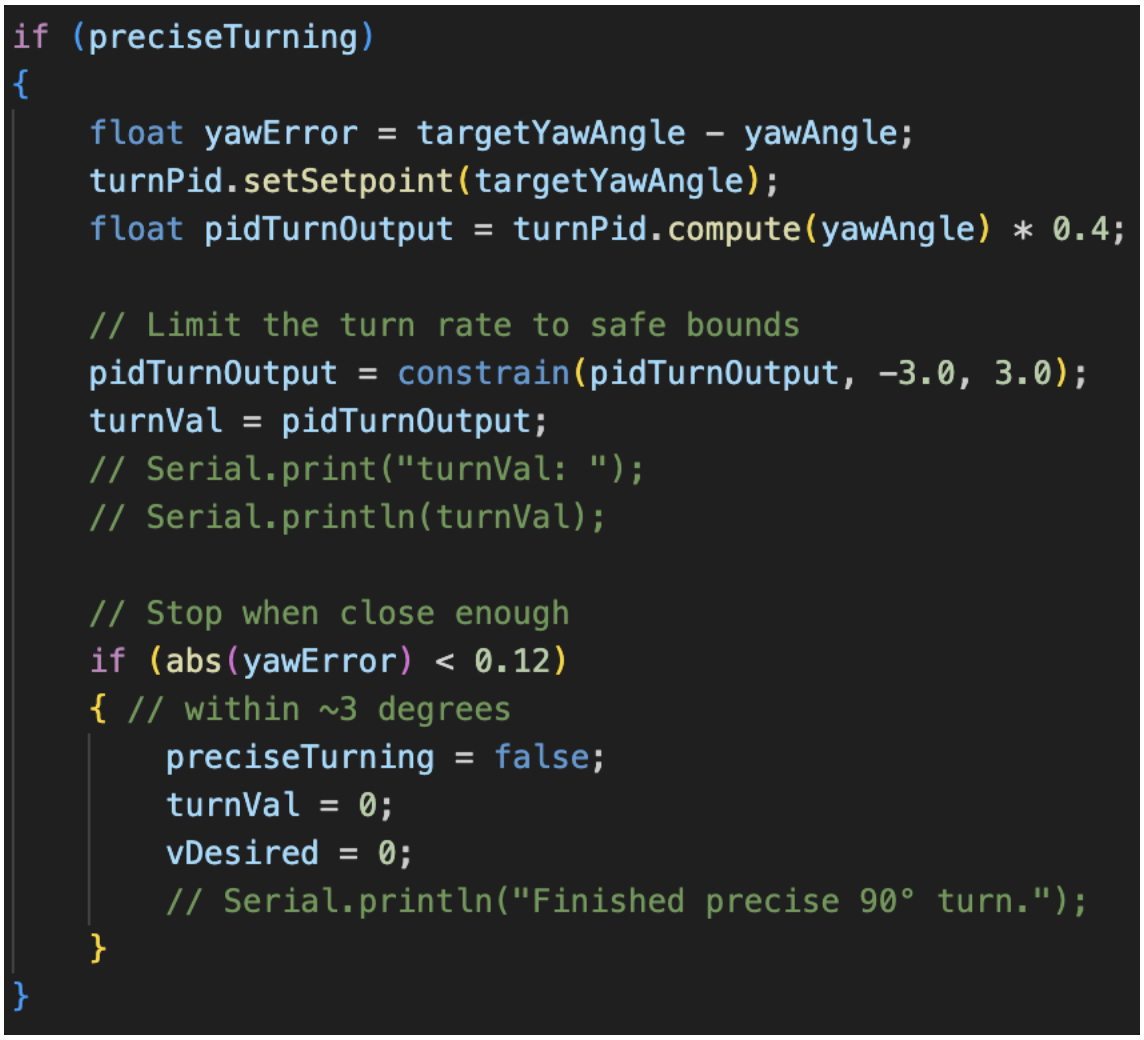
As shown in the figure above, PID control is used to compute yawCorrection when the robot is not turning. During turning, a differential acceleration - equal to twice the value of turnVal -is applied between the wheels to adjust the robot’s orientation.



2.4 - Precise Movement Control

This yaw control system was leveraged to enable fine-grained movement commands:

- For executing precise turns (e.g., 90 degrees), the desired angle is fed as a setpoint to the yaw controller. A gain of 0.4 is applied to the turning value (turnVal) to scale the command to small acceleration values compared to the balancing acceleration values so that turning won’t affect balancing.



```
if (preciseTurning)
{
    float yawError = targetYawAngle - yawAngle;
    turnPid.setSetpoint(targetYawAngle);
    float pidTurnOutput = turnPid.compute(yawAngle) * 0.4;

    // Limit the turn rate to safe bounds
    pidTurnOutput = constrain(pidTurnOutput, -3.0, 3.0);
    turnVal = pidTurnOutput;
    // Serial.print("turnVal: ");
    // Serial.println(turnVal);

    // Stop when close enough
    if (abs(yawError) < 0.12)
    { // within ~3 degrees
        preciseTurning = false;
        turnVal = 0;
        vDesired = 0;
        // Serial.println("Finished precise 90° turn.");
    }
}
```

As shown in the figure above, turning acceleration bounds and stopping condition are also added to ensure smooth turning.

- The same principles can be applied to move a precise forward distance by integrating speed over time discretely in the outer loop.

2.5 - Sensor Fusion with a Complementary Filter

A single sensor is insufficient for obtaining an accurate pitch angle. An accelerometer provides a stable sense of gravity's direction over time but is easily corrupted by the robot's own linear acceleration. A gyroscope provides an excellent instantaneous measure of rotation but is prone to accumulating drift error over time.

To get the best of both worlds, a complementary filter was implemented. This sensor fusion algorithm combines the high-pass filtered output of the gyroscope (trusting its short-term accuracy) with the low-pass filtered output of the accelerometer (trusting its long-term stability). This effectively uses accelerometer data to continuously correct for gyroscope's drift. The filter was implemented with the following equation:

$$\Theta_n = (1 - C)\Theta_a + C(\frac{d\Theta_g}{dt}\Delta t + \Theta_{n-1})$$

Θ_n = *the calculated* ***tilt*** *at iteration* ***n***

Θ_{n-1} = *the* ***tilt*** *from the* ***previous*** *iteration*

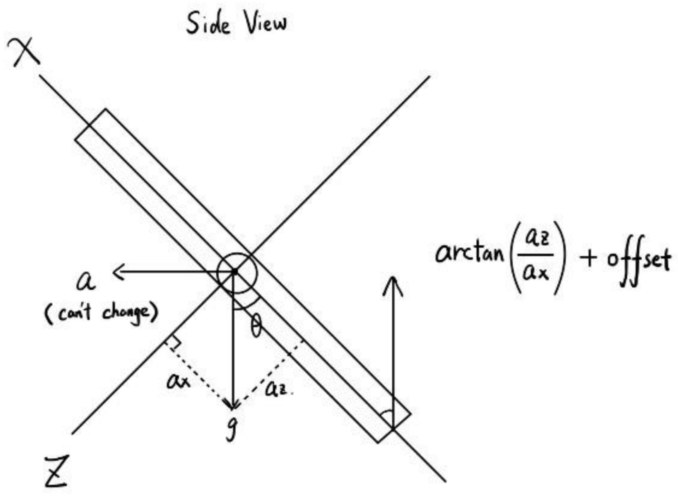
Θ_a = *the tilt* ***angle*** *from the accelerometer*

$d\Theta_g/dt$ = *is the* ***rate of change of the tilt angle*** *from the gyroscope*

Δt = *the* ***time interval*** *between iterations*

C = *a factor between 0 and 1, typically close to 1*

The tuning of the filter constant C was critical. A value of 0.98 was chosen, heavily weighting the clean, responsive gyroscope data while still allowing the accelerometer to slowly and gently pull the estimate back to the true vertical, resulting in a highly accurate and stable pitch angle measurement.



2.6 - Testing Results & Discussion

2.6A - Testing Methodology

The movement control subsystem was evaluated under a range of controlled conditions to ensure accurate and responsive performance. The primary evaluation metrics included movement precision, responsiveness to commands, and stability during motion.

2.6B - Metrics for Evaluation

Precision: Defined as the accuracy with which the robot executes movements in relation to the commanded target positions.

Responsiveness: Determined by the time required for the robot to initiate and halt movement in response to control commands.

Stability: Evaluated based on the robot's ability to maintain balance both during and after executing movement commands.

2.6C - Test Results

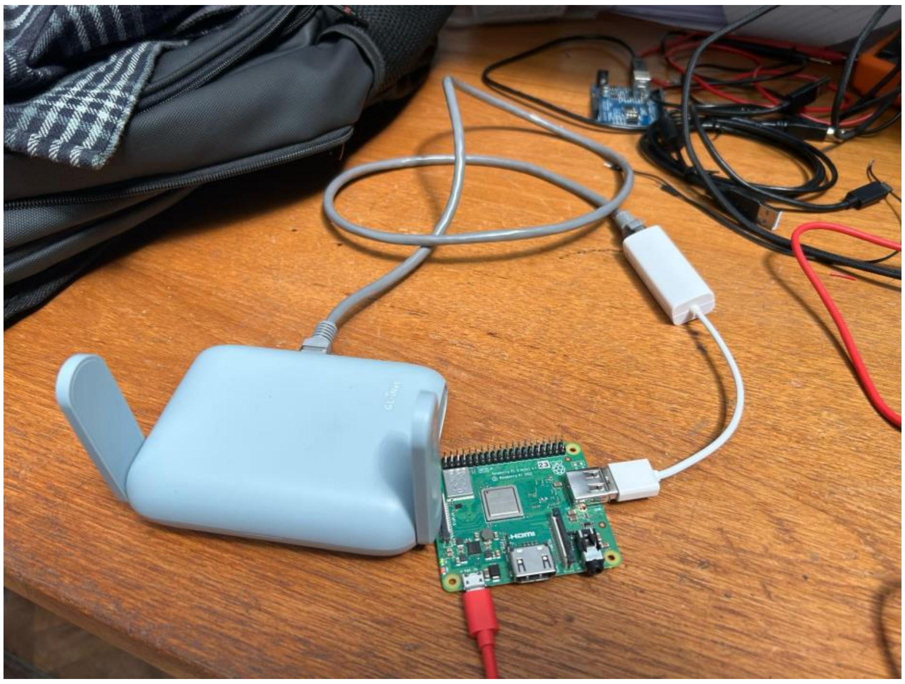
- **Precision**: The robot exhibited a high degree of precision in following movement commands, maintaining a straight trajectory and performing accurate on-the-spot turns.
- **Responsiveness**: The robot consistently initiated and stopped motion within 0.4 seconds following a control input. For context, the average human reaction time is approximately 0.25 seconds, indicating that the robot's response time is perceived as nearly instantaneous from a user's standpoint.
- **Stability**: The robot maintained balance effectively throughout movement. Oscillations remained within acceptable limits, and no instances of toppling were observed.

3 - Communication System Design and Implementation

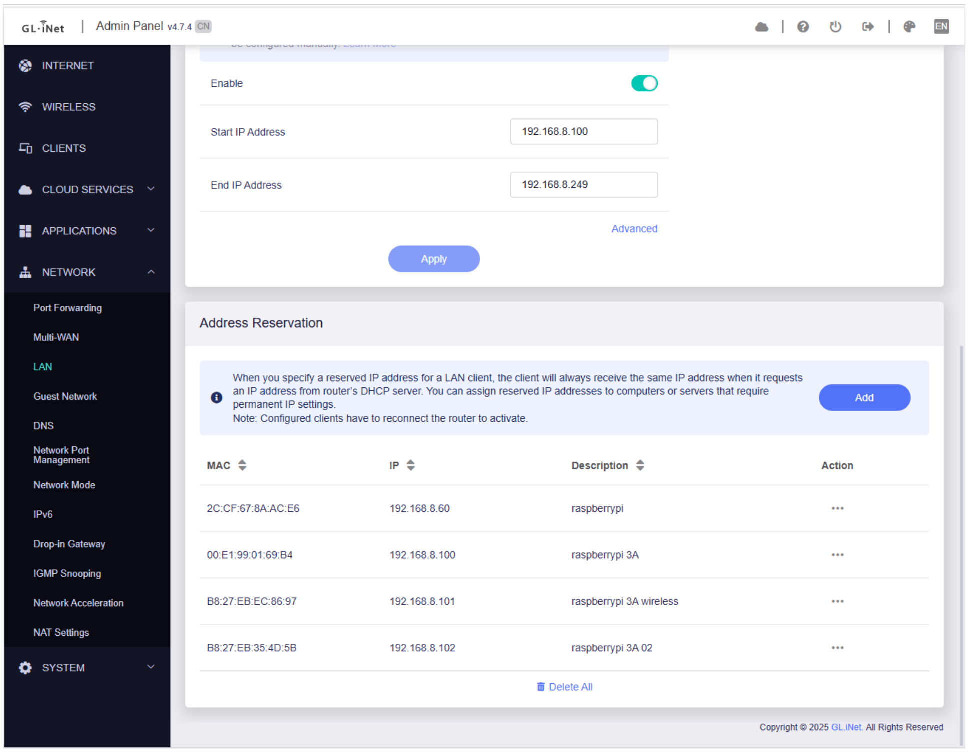
The communication infrastructure serves as the critical backbone enabling coordination between various components of the robotic system. This section focuses on the design and implementation of a robust, multi-layered communication system that ensures reliable data exchange across heterogeneous platforms. From low-level serial communication with embedded devices to high-level network interactions between the Raspberry Pi and remote servers, the system was engineered to handle real-time control demands while maintaining modularity and fault tolerance.

3.1 - Raspberry Pi Network Configuration and Remote Access

To facilitate both development and runtime remote access to the Raspberry Pi, a stable LAN configuration was adopted. During the initial setup phase, a physical Ethernet connection was established using an RJ45-to-USB-A adapter and a cable, enabling the Raspberry Pi to be temporarily connected to the router. This allowed configuration of the reserved IP address and activation of SSH access via standard username-password authentication.



Subsequently, the Raspberry Pi was configured - remotely via SSH - to automatically connect to the router's Wi-Fi network on boot. Its wireless interface was also assigned a static IP address. This setup significantly simplified future connections, eliminating the need for repeated physical access during deployment.



Enable

Start IP Address: 192.168.8.100

End IP Address: 192.168.8.249

Address Reservation

When you specify a reserved IP address for a LAN client, the client will always receive the same IP address when it requests an IP address from router's DHCP server. You can assign reserved IP addresses to computers or servers that require permanent IP settings.
Note: Configured clients have to reconnect the router to activate.

MAC	IP	Description	Action
2C:CF:67:8A:AC:E6	192.168.8.60	raspberrypi	•••
00:E1:99:01:69:B4	192.168.8.100	raspberrypi 3A	•••
B8:27:EB:EC:86:97	192.168.8.101	raspberrypi 3A wireless	•••
B8:27:EB:35:4D:5B	192.168.8.102	raspberrypi 3A 02	•••

Copyright © 2025 GL.iNet. All Rights Reserved

3.2 - TCP Communication Between PC and Raspberry Pi

To implement control signal transmission and data return between the PC and the Raspberry Pi, a TCP-based communication module was developed using Python's built-in socket library. In this setup, the Raspberry Pi acts as the server, persistently listening on port 65432, while the PC functions as the client initiating the connection and sending csommands.

This TCP communication framework serves as the foundation for all LAN-based interactions between the PC and the control system. It enables structured message exchange and provides a reliable transport layer for both real-time control and feedback.

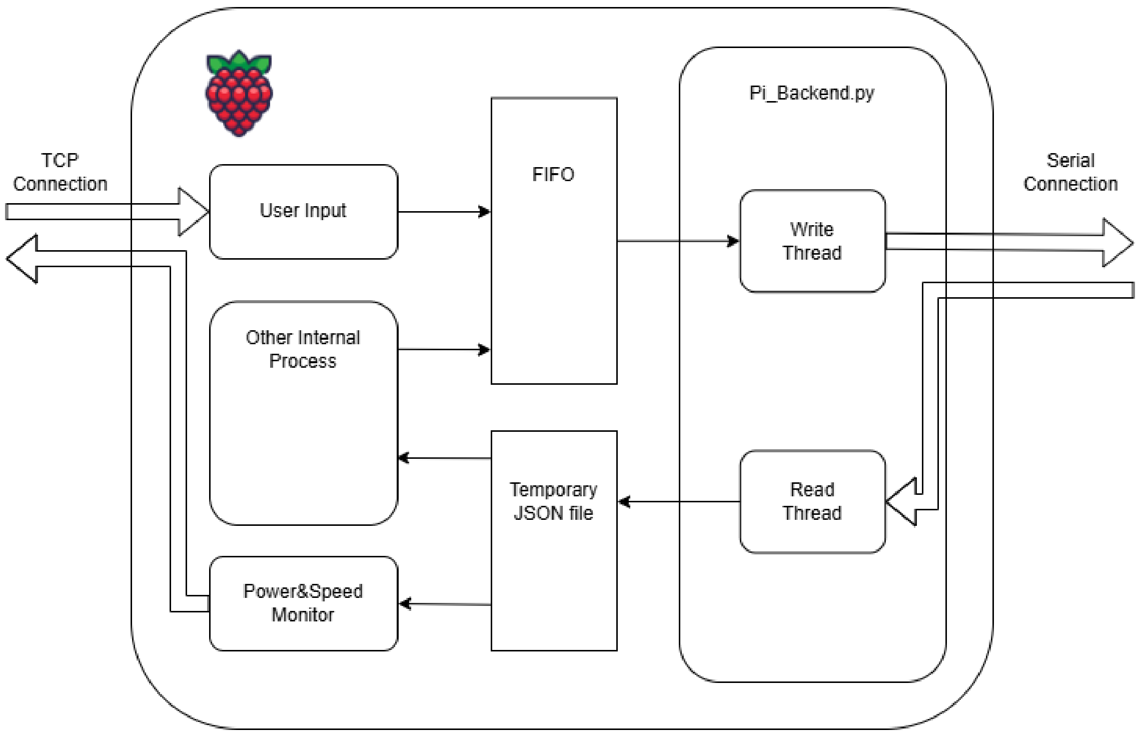
End-to-end communication was successfully tested under typical network conditions, achieving millisecond-level latency. This result confirmed that the system is well-suited for responsive remote operations in a local network environment.

3.3 - Serial Communication Between Raspberry Pi and ESP32

The Raspberry Pi and ESP32 are connected via USB for serial communication. During setup, the user identifies the correct serial port on the Pi using the ls /dev command (e.g., /dev/ttyUSB0), and manually configures it within the code. Communication on the Raspberry Pi side is implemented using Python's Pyserial library, while the ESP32 leverages the Arduino Serial library for interaction.

In the early stage of development, we implemented a **basic version of the serial communication module** to enable data exchange between the Raspberry Pi and the ESP32. This version provided simple read/write interfaces for serial operations, which could be called directly from different parts of the system. However, as the project evolved, this design proved insufficient for supporting more advanced and concurrent features:

- Serial ports are **exclusive resources** in the operating system and **cannot be accessed by multiple processes simultaneously**.
- Advanced system functionalities - such as **manual control**, **battery monitoring**, and **future extensions** - require concurrent access to the serial interface, leading to conflicts and runtime errors.
- The function-call-based implementation **lacked a proper inter-process communication (IPC) mechanism** and was vulnerable to race conditions and misread values under concurrent operations.



To overcome these limitations, we redesigned the serial communication module as a **robust, scalable, and concurrent system**, incorporating the following improvements:

1. **Dedicated Serial Communication Process:**
 Instead of exposing serial communication as callable functions, we encapsulated it into a **standalone background process** responsible solely for maintaining communication with the ESP32. This ensures that **only one process owns the serial port**, effectively eliminating resource contention.
2. **Inter-Process Communication (IPC) with FIFO and JSON:**
 Higher-level control modules no longer interact with the serial port directly. Instead, they communicate with the serial process through **named pipes (FIFO)** for command input and **intermediate JSON files** for output retrieval. This architecture allows **multiple**

independent processes to issue serial commands concurrently, in a structured and non-blocking manner.

3. **Multithreaded Serial I/O:**
   Inside the serial process, **read and write operations are handled by separate threads**, avoiding blocking delays and allowing simultaneous handling of incoming and outgoing data. This design ensures that commands from different sources are processed without interference and that responses are returned accurately and promptly.

This modular and thread-safe architecture greatly improves the **reliability, concurrency, and extensibility** of the serial communication system. It enables seamless coordination among multiple Pi-side processes such as user input handling, power monitoring, and task automation. It also lays a solid foundation for integrating more sophisticated features and control logic in future development.

Thanks to the Serial.available() method on the ESP32, the microcontroller only reads data when new input is detected, avoiding unnecessary delays and ensuring efficient communication responsiveness.

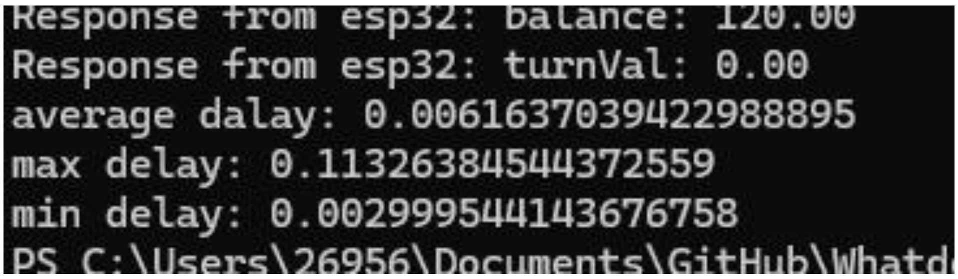
3.4 - Remote Control Testing Based on Communication Module

To validate the full communication pipeline, a terminal-based remote-control prototype was developed. This system enables real-time control of the ESP32-based vehicle using either keyboard or joystick input from a PC. The implementation leverages Python's pynput and pygame libraries to asynchronously monitor user input and translate it into control commands.

On the PC side, directional inputs (w, a, s, d) are interpreted from keyboard or joystick sources. These inputs are processed in real-time and mapped to movement commands. To reduce network and processing overhead, each distinct command is transmitted only once - new commands are sent only when the user input changes. When no active input is detected, a placeholder command (p) is used to indicate an idle state. If both input types are present, joystick input is given higher priority.

The commands are transmitted over a TCP connection to the Raspberry Pi, where a server process listens on port 65432. Upon receiving a command, the Pi relays it to the ESP32 via serial communication.

On the ESP32 side, the system employs Arduino's Serial.available() method to non-blockingly detect and process incoming data, minimising delay and avoiding unnecessary CPU usage.



```
Response from esp32: balance: 120.00
Response from esp32: turnVal: 0.00
average dalay: 0.006163703942298895
max delay: 0.11326384544372559
min delay: 0.0029995441436767758
PS C:\Users\26956\Documents\GitHub\Whatd
```

Latency testing across the complete control loop - from PC input to ESP32 execution and response back to PC - showed an average round-trip delay of 6 milliseconds under a Wi-Fi 6 LAN environment.
This result is based on a 10000-command stress test, during which the system maintained a 0% packet loss rate, indicating exceptional stability under high-frequency bidirectional communication.
These results confirm the system's capability for real-time, low-latency robotic control, with reliable command delivery and timely feedback across the entire communication chain.

3.5 - Cloud Server Integration and Remote Web UI Access

To support internet-based remote control and monitoring, a cloud server hosted on AWS was integrated into the system architecture. The server runs a lightweight Python-based web service using the Flask framework and is accessible through the **registered domain,** http://whatdoumeanrobot.com.

The remote access architecture is composed of the following components:

PC → Server (HTTP/HTTPS Web UI):
Users interact with the system via a browser, connecting to the AWS server over HTTP (port 80) or HTTPS (port 443). Flask handles the user's requests and translates them into control commands for the Raspberry Pi.

Server → Raspberry Pi (Reverse Proxy via frp)

Due to the **NAT firewall environment of the Raspberry Pi**, which **lacks a public IP address**, the server cannot establish direct incoming TCP connections to the Pi. This is a common challenge in scenarios involving devices deployed in home or campus LANs where port forwarding or public address assignment is not available.

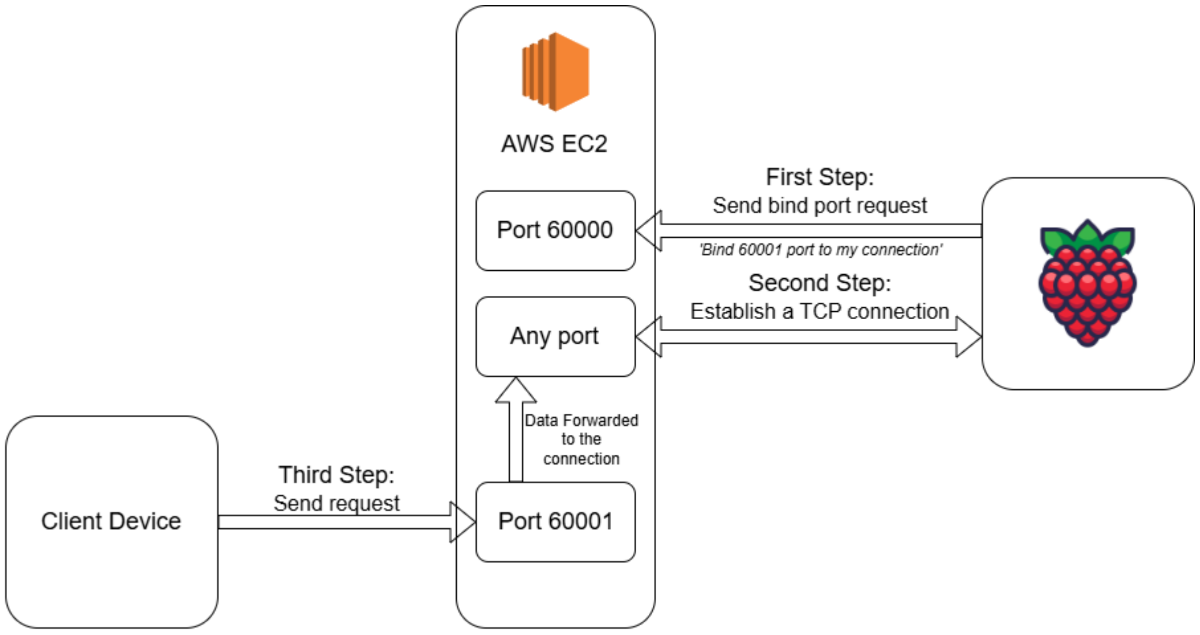
To overcome this limitation, the system adopts **frp (Fast Reverse Proxy)** - an open-source, lightweight reverse proxy solution that supports **NAT traversal**. In this architecture:

- The **frp server runs on the public AWS instance**, which has a static public IP.
- The **frp client runs on the Raspberry Pi** and actively establishes a persistent connection to the cloud server.
- **External traffic to specific ports on the AWS server is forwarded through this secure channel** to corresponding services hosted on the Raspberry Pi, such as SSH or custom TCP control ports.

This approach was chosen based on several key considerations:

- **Simplicity of deployment**: frp requires minimal configuration and can be set up within minutes using systemd for persistent background operation.

- **No changes to LAN or router settings**: Users do not need to configure port forwarding or request public IPs from their local ISP or campus IT.
- **Efficient and reliable tunneling**: frp maintains a stable connection with low overhead, suitable for low-latency, high-reliability control communication.
- **Compared to VPN solutions**:
  - **Self-hosted VPNs** (e.g., OpenVPN, WireGuard) require a public IP address or port-forwarded entry point on the Raspberry Pi, which is not feasible in this setup.
  - **Mesh VPNs** (e.g., ZeroTier, Tailscale) introduce additional **connection steps, client installation, and potential latency overhead**, and **some solutions are subscription-based or commercial** for multi-device teams.



Overall, frp offers a **robust yet lightweight solution** tailored for this project's use case: enabling transparent, secure, and persistent **reverse proxying from a public cloud server to a NATed embedded device**, without requiring manual configuration from the user.

frp Overview:

frp is a high-performance reverse proxy application that operates in a client-server model. The server is deployed on a machine with a public IP address, while the client runs within a private network. Through this setup, internal services can be securely exposed over the internet.
frp supports multiple protocols including TCP, UDP, HTTP, and HTTPS, and offers additional features such as encryption, compression, authentication, rate limiting, load balancing, and even peer-to-peer communication through xtcp.

Domain Configuration and DNS:

A custom domain (whatdoumeanrobot.com) was registered and configured with an A-type DNS record pointing to the AWS server's IP address. This allows users to easily access the system without needing to remember IP addresses.

Automation and Service Reliability:
To ensure long-term reliability, both ends of the frp architecture are configured for automated startup and persistent availability:

On the **server side**, screen sessions are used to run the frp server daemon in a detached, persistent terminal, allowing the service to continue running even after SSH disconnection or system interruption.

On the **Raspberry Pi**, a custom systemd unit file was created for the frp client, enabling it to start automatically during system boot and recover from failures.

This cloud-integrated architecture provides a seamless bridge between private IoT hardware and public web access, allowing users to control the robot platform via a unified Web UI interface in both local and remote scenarios. The reverse proxy approach ensures security, compatibility, and flexibility in diverse networking environments. A demonstration of the robot being controlled via the Web UI is included in **Appendix 9.3**.

3.6 - Summary

The communication system successfully established robust links between the PC, Raspberry Pi, ESP32, and cloud server using TCP, serial, and HTTP protocols. It overcame challenges such as concurrent process access and NAT traversal using threading, FIFO/JSON intermediates, and reverse proxy solutions. This foundation supports future development of advanced features and user interfaces.

4 - Robot Web UI: Voice and Manual Control

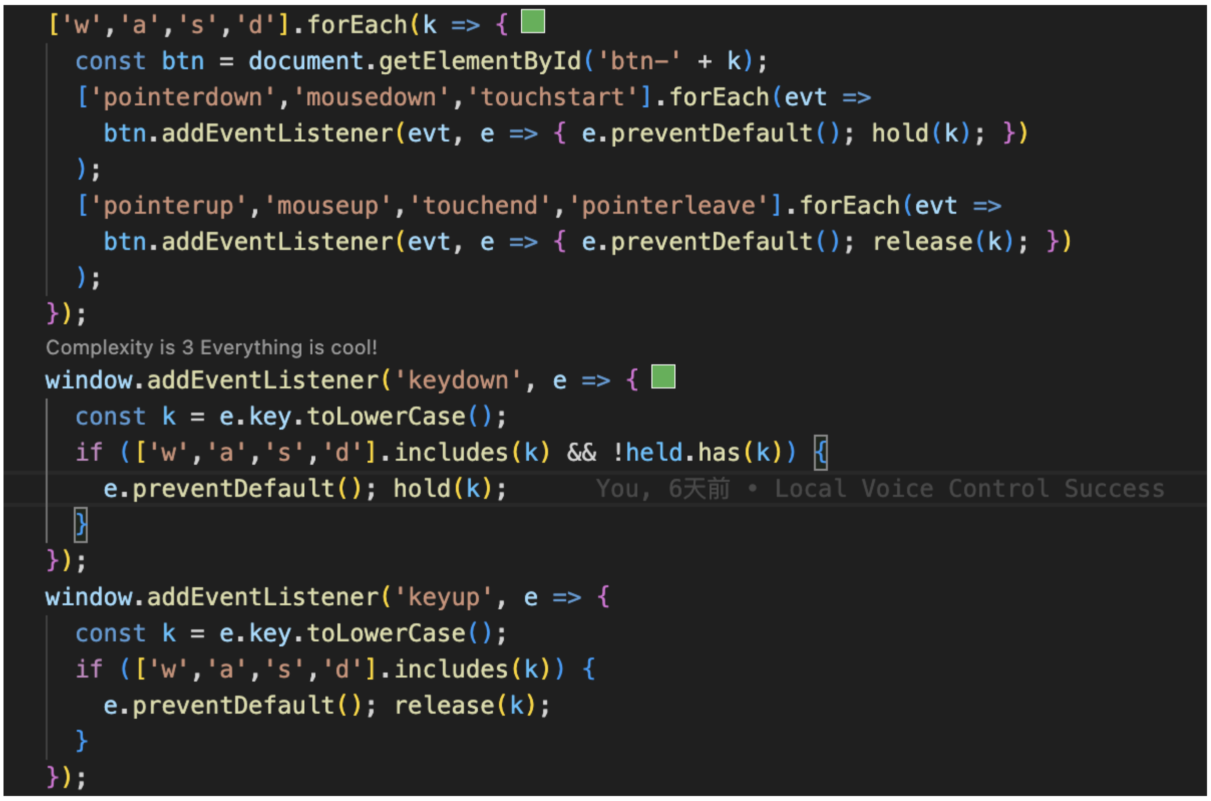
4.1 - Overview

The Web interface is a single-page application constructed entirely with Tailwind CSS. Beyond providing a visually coherent dark-theme dashboard, the UI fulfils **four key functions**: it delivers real-time telemetry, offers manual D-Pad driving, exposes quick toggles for autonomous modes, and provides a full voice-command pipeline that converts speech into timed movement directives. Adopting Tailwind's utility-first framework let the team prototype and adjust those features in rapid cycles while keeping every component - buttons, progress bars and status messages - high-contrast and legible. Gradients and soft shadows supply depth without custom style sheets, and Tailwind's responsive classes ensure the same code base renders correctly on a four-inch phone and a computer monitor alike.

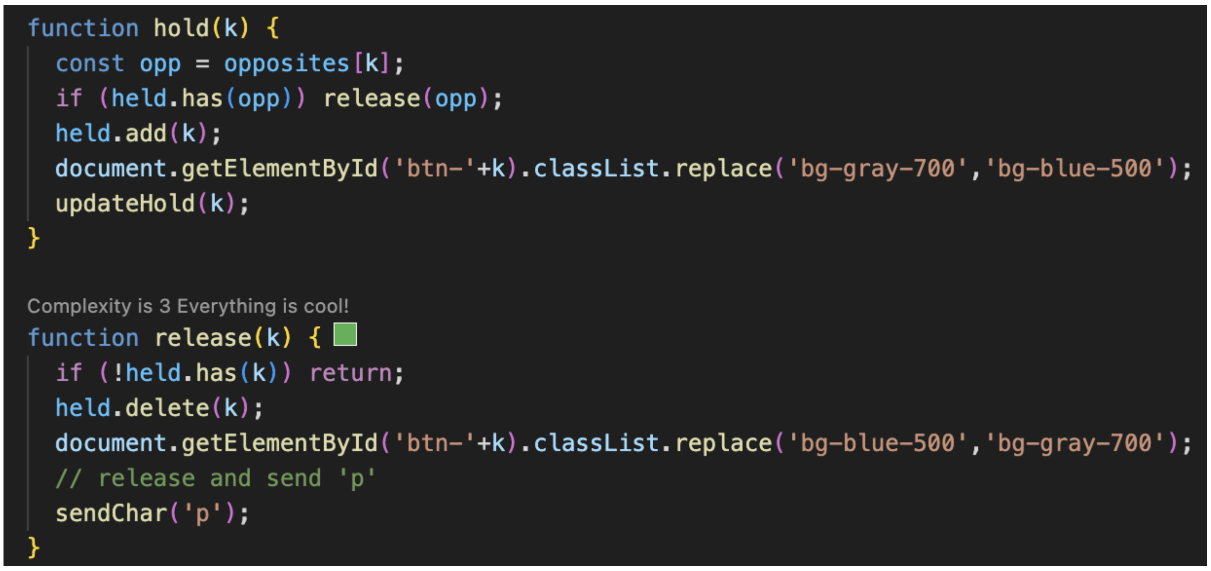
4.2 - Client-Side Frontend- Interactive Web UI & Command Dispatcher

4.2A - Directional (D-Pad) Control System

At the heart of manual driving lies the on-screen D-Pad, which is rendered with an explicit 3 x 2 CSS Grid. Each cell contains one arrow button, and both touch events (***pointerdown*** / ***pointerup***) and standard keyboard events (***keydown / keyup***) are bound to the same logic. When a user presses a direction--whether by tapping the screen or holding a keyboard key--the UI issues a single-character command (***w, a, s*** or ***d***) to the backend endpoint ***/api/command***. Releasing the pointer or key always triggers a second call that sends the stop character ***p***, guaranteeing that the robot never continues to move once the user has let go.



```
['w','a','s','d'].forEach(k => {
  const btn = document.getElementById('btn-' + k);
  ['pointerdown','mousedown','touchstart'].forEach(evt =>
    btn.addEventListener(evt, e => { e.preventDefault(); hold(k); })
  );
  ['pointerup','mouseup','touchend','pointerleave'].forEach(evt =>
    btn.addEventListener(evt, e => { e.preventDefault(); release(k); })
  );
});
Complexity is 3 Everything is cool!
window.addEventListener('keydown', e => {
  const k = e.key.toLowerCase();
  if (['w','a','s','d'].includes(k) && !held.has(k)) {
    e.preventDefault(); hold(k);        You, 6天前 • Local Voice Control Success
  }
});
window.addEventListener('keyup', e => {
  const k = e.key.toLowerCase();
  if (['w','a','s','d'].includes(k)) {
    e.preventDefault(); release(k);
  }
});
```

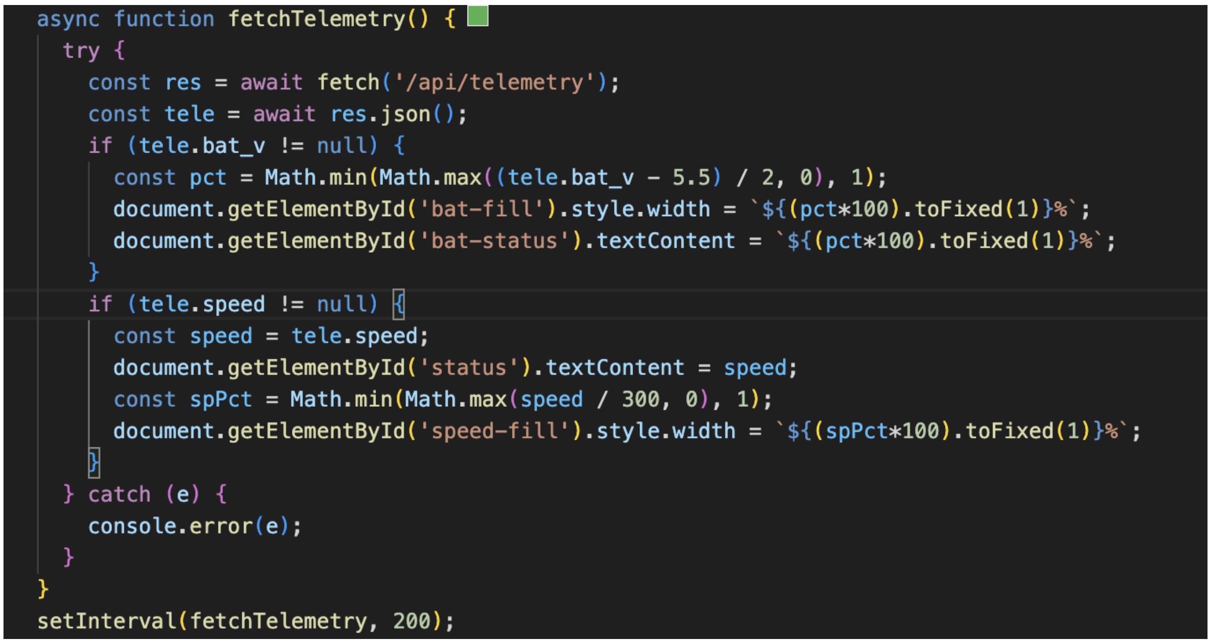


```
function hold(k) {
  const opp = opposites[k];
  if (held.has(opp)) release(opp);
  held.add(k);
  document.getElementById('btn-'+k).classList.replace('bg-gray-700','bg-blue-500');
  updateHold(k);
}

Complexity is 3 Everything is cool!
function release(k) {
  if (!held.has(k)) return;
  held.delete(k);
  document.getElementById('btn-'+k).classList.replace('bg-blue-500','bg-gray-700');
  // release and send 'p'
  sendChar('p');
}
```

To avoid contradictory instructions, the script first checks whether the opposite direction is already held; if so, it releases that direction before engaging the new one, eliminating the risk of deadlock or cancelling momentum mid-course.

4.2B - Real-Time Telemetry Display

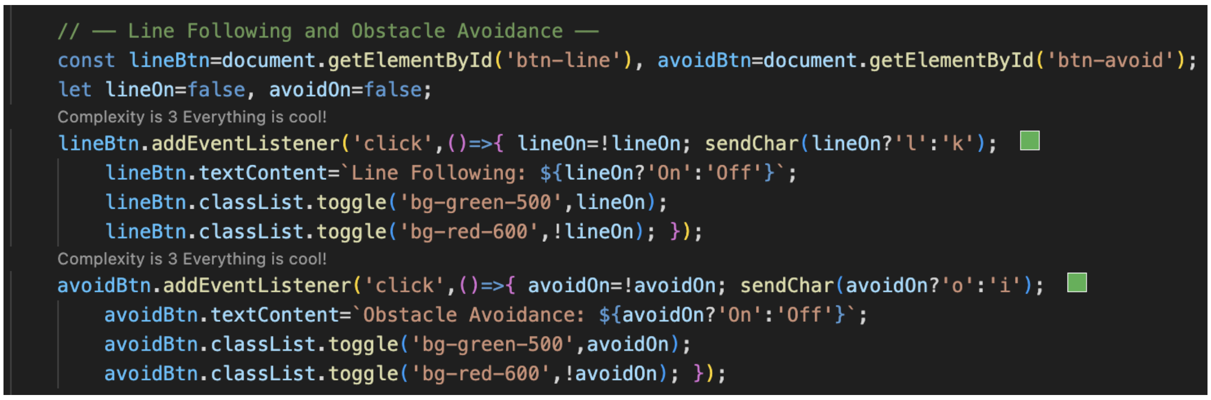
Two live widgets - battery and speed - occupy the centre of the dashboard. A polling function runs every two-tenths of a second and requests the latest telemetry JSON from ***/api/telemetry***. Battery voltage, which the robot reports as a floating-point value, is linearly mapped to a percentage between zero and one hundred and drives the width of a green progress bar. Likewise, an integer speed value, bounded on the robot at three hundred units, is normalised and poured into a blue bar of matching scale. All asynchronous calls are wrapped in a ***try-catch***; should the network drop momentarily, the last reading remains on screen and the console quietly records the exception, ensuring the visual layout never flickers or collapses.



```
async function fetchTelemetry() {
  try {
    const res = await fetch('/api/telemetry');
    const tele = await res.json();
    if (tele.bat_v != null) {
      const pct = Math.min(Math.max((tele.bat_v - 5.5) / 2, 0), 1);
      document.getElementById('bat-fill').style.width = `${(pct*100).toFixed(1)}%`;
      document.getElementById('bat-status').textContent = `${(pct*100).toFixed(1)}%`;
    }
    if (tele.speed != null) {
      const speed = tele.speed;
      document.getElementById('status').textContent = speed;
      const spPct = Math.min(Math.max(speed / 300, 0), 1);
      document.getElementById('speed-fill').style.width = `${(spPct*100).toFixed(1)}%`;
    }
  } catch (e) {
    console.error(e);
  }
}
setInterval(fetchTelemetry, 200);
```

4.2C - Feature Toggles (Line-Following & Obstacle Avoidance)

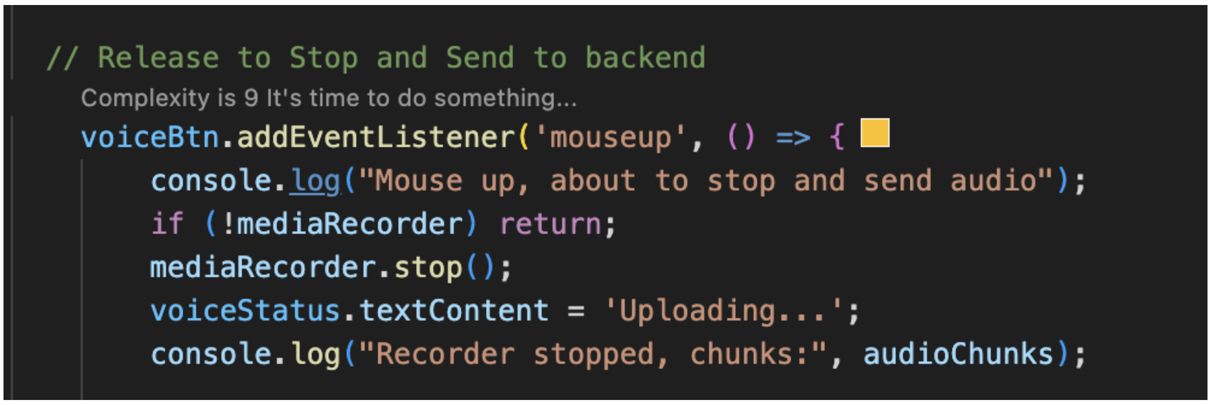
Beneath the telemetry panel two square buttons activate or deactivate extended driving features. The first governs line following. When it is pressed, a Boolean flag flips and the code dispatches ***l*** to switch the algorithm on; pressing again sends ***k*** to turn it off. The second button performs the same role for obstacle avoidance, sending ***o*** to enable and ***i*** to disable. Each click updates the button label from "Off" to "On" and replaces a red background with green, giving the operator unequivocal confirmation that the requested mode is now active.



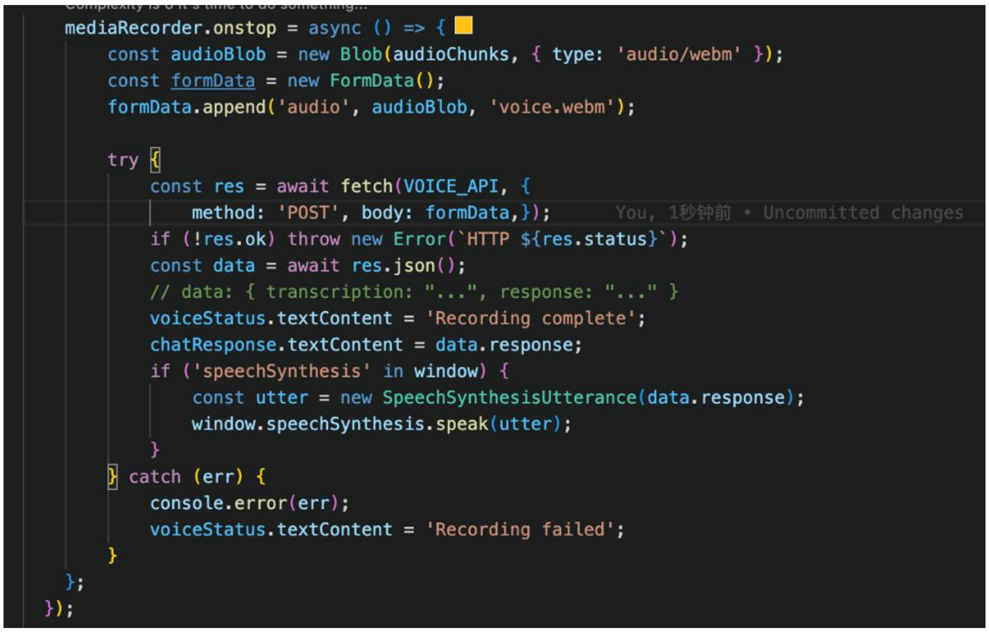
```
// —— Line Following and Obstacle Avoidance ——
const lineBtn=document.getElementById('btn-line'), avoidBtn=document.getElementById('btn-avoid');
let lineOn=false, avoidOn=false;
Complexity is 3 Everything is cool!
lineBtn.addEventListener('click',()=>{ lineOn=!lineOn; sendChar(lineOn?'l':'k');
    lineBtn.textContent=`Line Following: ${lineOn?'On':'Off'}`;
    lineBtn.classList.toggle('bg-green-500',lineOn);
    lineBtn.classList.toggle('bg-red-600',!lineOn); });
Complexity is 3 Everything is cool!
avoidBtn.addEventListener('click',()=>{ avoidOn=!avoidOn; sendChar(avoidOn?'o':'i');
    avoidBtn.textContent=`Obstacle Avoidance: ${avoidOn?'On':'Off'}`;
    avoidBtn.classList.toggle('bg-green-500',avoidOn);
    avoidBtn.classList.toggle('bg-red-600',!avoidOn); });
```

4.2D - AI-Driven Voice Interaction

The voice subsystem is responsible for translating free-form speech into precise movement directives. The first time the operator presses and holds the "Press to Record" button, the browser requests microphone permission through the MediaDevices API. Once approved, an in-memory ***MediaRecorder*** starts capturing audio at every subsequent press. Releasing the button finalises a WebM blob, which the interface immediately uploads to ***/api/voice***.

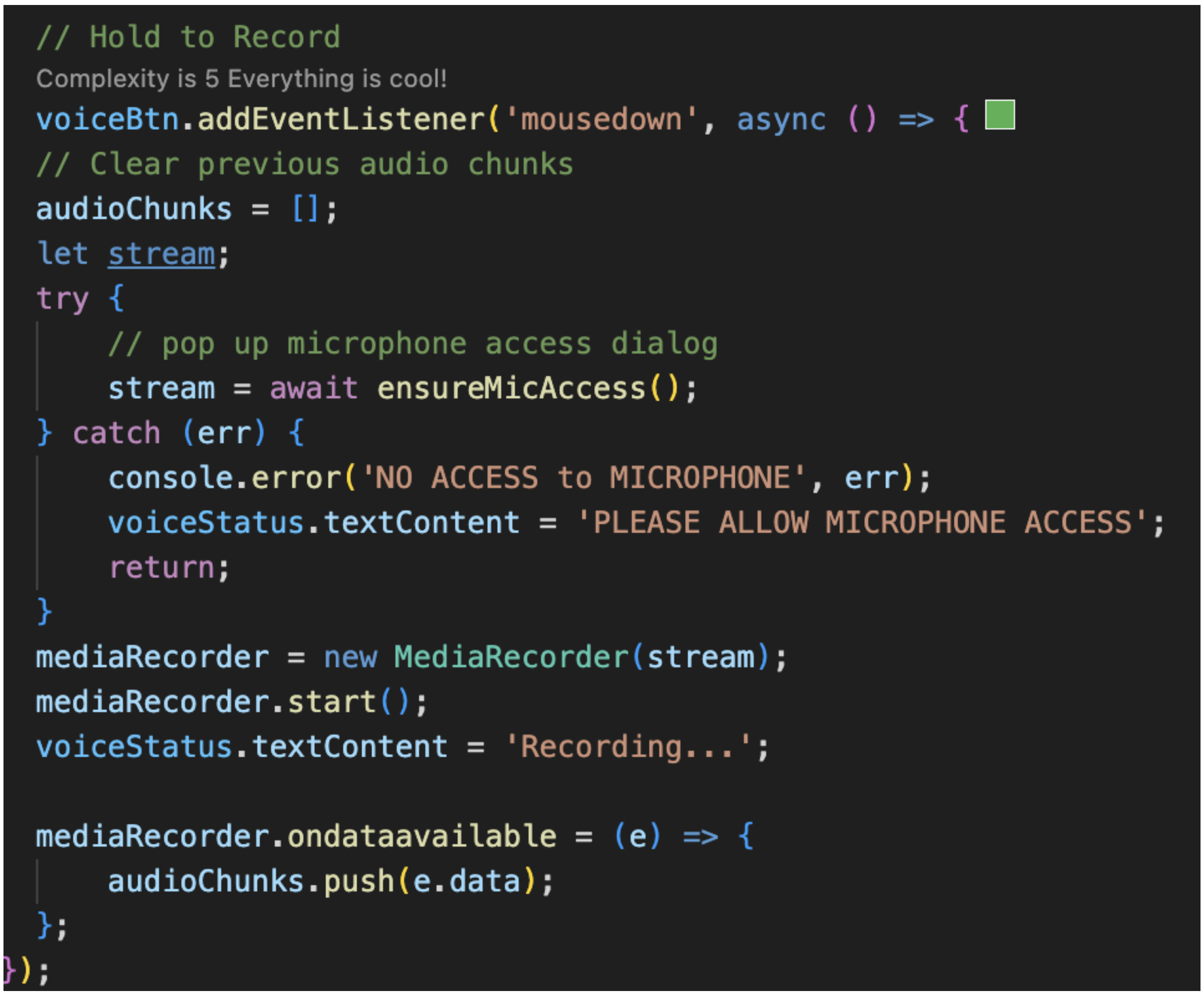


```
// Release to Stop and Send to backend
    Complexity is 9 It's time to do something...
    voiceBtn.addEventListener('mouseup', () => {
        console.log("Mouse up, about to stop and send audio");
        if (!mediaRecorder) return;
        mediaRecorder.stop();
        voiceStatus.textContent = 'Uploading...';
        console.log("Recorder stopped, chunks:", audioChunks);
```

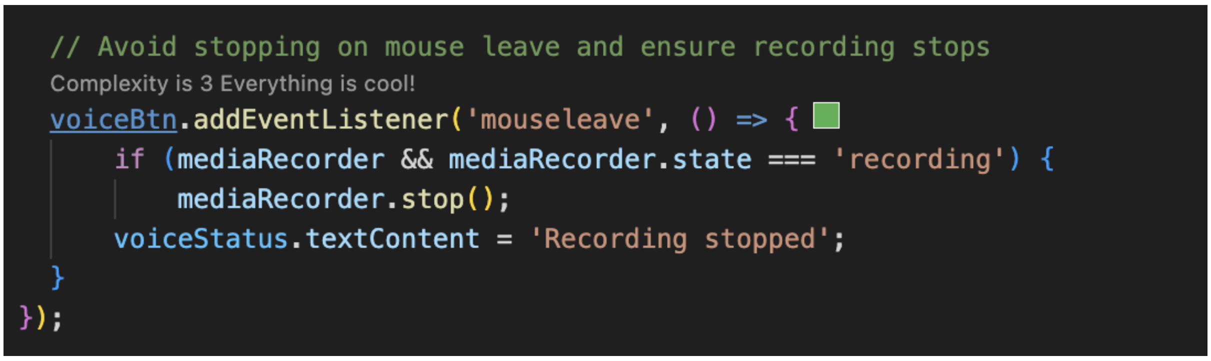


```
    mediaRecorder.onstop = async () => {
        const audioBlob = new Blob(audioChunks, { type: 'audio/webm' });
        const formData = new FormData();
        formData.append('audio', audioBlob, 'voice.webm');

        try {
            const res = await fetch(VOICE_API, {
                method: 'POST', body: formData,});      You, 1秒钟前 • Uncommitted changes
            if (!res.ok) throw new Error(`HTTP ${res.status}`);
            const data = await res.json();
            // data: { transcription: "...", response: "..." }
            voiceStatus.textContent = 'Recording complete';
            chatResponse.textContent = data.response;
            if ('speechSynthesis' in window) {
                const utter = new SpeechSynthesisUtterance(data.response);
                window.speechSynthesis.speak(utter);
            }
        } catch (err) {
            console.error(err);
            voiceStatus.textContent = 'Recording failed';
        }
    };
  });
```

On the Backend side, AI output a flat string of movement commands - e.g. ***w5 a3 p*** where each command is a single character, followed by a duration, and the sequence ends with ***p***. The client receives that string, splits it by spaces and iterates over the segments. For every pair encountered, it sends the character to ***/api/command***, pauses for the specified number of seconds by means of ***setTimeout***, and finally delivers an explicit ***p*** to halt the robot. As a courtesy to the user, the browser's speech-synthesis engine simultaneously reads the model's natural-language acknowledgement out loud. If the operator refuses microphone permission or the network upload fails, the interface replaces the status line with "PLEASE ALLOW MICROPHONE ACCESS" or "Recording failed" but leaves the rest of the dashboard fully functional.



```
// Hold to Record
Complexity is 5 Everything is cool!
voiceBtn.addEventListener('mousedown', async () => {
// Clear previous audio chunks
audioChunks = [];
let stream;
try {
    // pop up microphone access dialog
    stream = await ensureMicAccess();
} catch (err) {
    console.error('NO ACCESS to MICROPHONE', err);
    voiceStatus.textContent = 'PLEASE ALLOW MICROPHONE ACCESS';
    return;
}
mediaRecorder = new MediaRecorder(stream);
mediaRecorder.start();
voiceStatus.textContent = 'Recording...';

mediaRecorder.ondataavailable = (e) => {
    audioChunks.push(e.data);
};
});
```



```
  // Avoid stopping on mouse leave and ensure recording stops
  Complexity is 3 Everything is cool!
  voiceBtn.addEventListener('mouseleave', () => {
      if (mediaRecorder && mediaRecorder.state === 'recording') {
          mediaRecorder.stop();
      voiceStatus.textContent = 'Recording stopped';
  }
});
```

4.2E - Responsiveness and Accessibility Considerations

Because the application is likely to be opened on phone, tablet and desktop alike, every layout decision is framed in terms of flexible units. The D-Pad scales proportionally, and the gap between buttons grows when horizontal space is abundant, preventing cramped controls on larger screens. Each interactive element exceeds recommended minimum hit-target dimensions, which is essential for gloved hands in a workshop. Although our first release relies mainly on visual cues, the markup can be extended easily with ARIA labels so that screenreader software can announce “Move forward” or “Enable line following” when a button gains focus.

4.2F - Error Handling and Stability Measures

All fetch calls and media-capture operations are wrapped in dedicated ***catch*** clauses. Whether the telemetry endpoint time-outs or the speech upload is rejected, the error is logged, and the interface falls back to a safe state: command buttons remain active, and no stale animation continues unchecked. During stress-tests we repeatedly unplugged the robot’s Wi-Fi and confirmed that the UI froze the last known telemetry instead of crashing, then *seamlessly resumed* updates after reconnection.

4.3 - Server-Side Frontend

A lightweight “server Web UI” runs as a separate Flask micro-service whose sole purpose is to expose the robot dashboard to the public internet only when our tunnel or reverse-proxy is alive. The script monitors a configurable host: port pair - in our deployment, the FRP tunnel on **127.0.0.1 : 60001** - and stores the Boolean result in memory. Its static page, **index.html**, shows a status label that flips between red *Not Connected* and green *Connected*, and a disabled “Jump to Target Website” button that becomes clickable once the port probe succeeds. A tiny JavaScript loop polls /status once per second and updates the DOM accordingly, while a POST endpoint ***/set_connected*** lets any background process set the flag programmatically. All other paths are served over HTTPS with the same self-signed certificate mechanism described earlier, so remote teammates can open **website,** watch for the green indicator, then follow the button to the full Web-UI dashboard without ever exposing port 9001 directly.

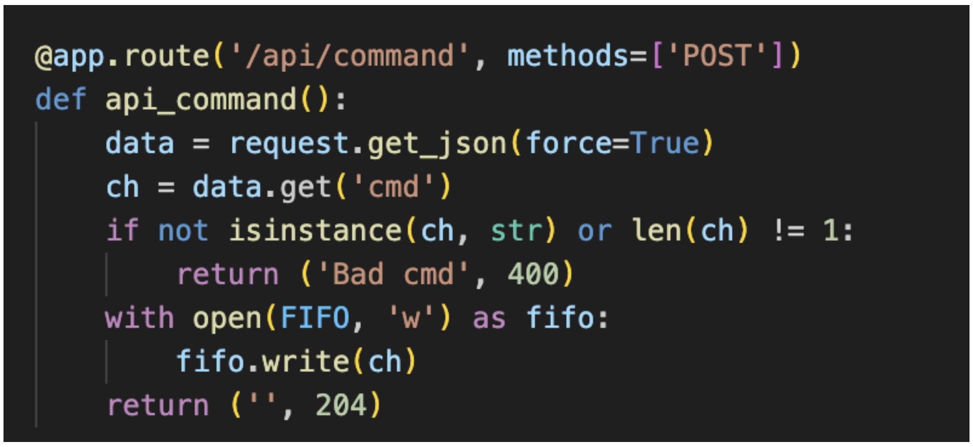
4.4 - Web Backend: Secure Flask API & Voice Interpreter

4.4A - Overview and Framework

The server side is a small Flask application that exposes three REST endpoints under the same port as the static front-end (***/api/command, /api/telemetry, /api/voice***). Cross-origin requests are permitted through Flask-CORS so that the browser can fetch data without additional headers. All traffic is carried over HTTPS; the application loads a local self-signed certificate and key at start-up, so the browser sees a secure context and is willing to grant microphone access to the front-end.

4.4B - Low-Latency Command Channel (/api/command)

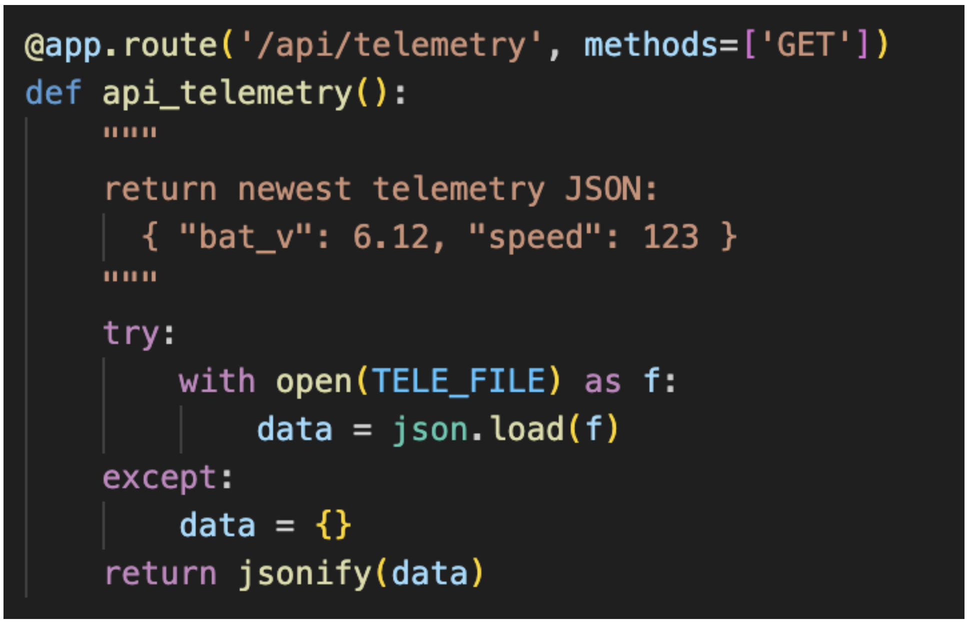
The simplest endpoint receives a JSON body that contains exactly one character. The code validates that it is a single printable byte, opens the named FIFO ***at /tmp/robot-cmd***, and writes that byte. Because both keyboard presses, on-screen buttons, and the voice pipeline converge on this call, the robot’s firmware can rely on one uniform stream of characters. The handler returns only HTTP 204, so the browser is never blocked by payload parsing.



```python
@app.route('/api/command', methods=['POST'])
def api_command():
    data = request.get_json(force=True)
    ch = data.get('cmd')
    if not isinstance(ch, str) or len(ch) != 1:
        return ('Bad cmd', 400)
    with open(FIFO, 'w') as fifo:
        fifo.write(ch)
    return ('', 204)
```

4.4C - Telemetry Relay (/api/telemetry)

The robot periodically appends a tiny JSON file to ***/tmp/telemetry.json***. The backend simply reads that file and delivers its contents unchanged. When the file is absent - such as during robot boot - the handler returns an empty object rather than an error, allowing the Web UI to keep its last valid reading instead of flashing a warning.

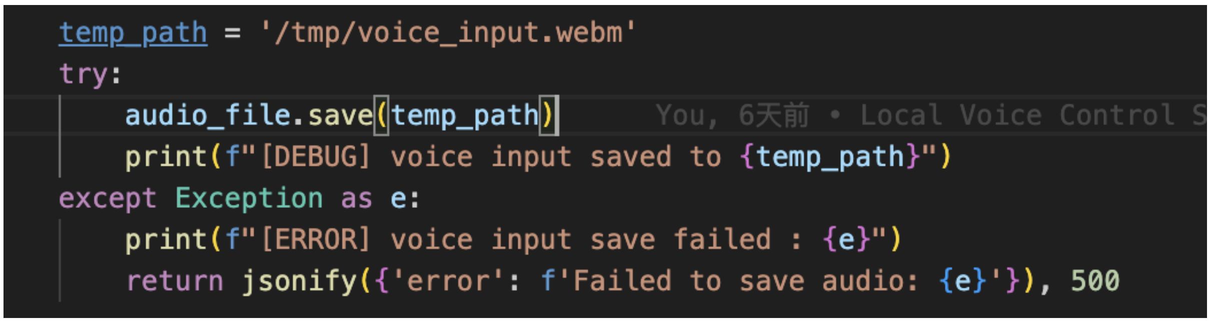


```python
@app.route('/api/telemetry', methods=['GET'])
def api_telemetry():
    """
    return newest telemetry JSON:
      { "bat_v": 6.12, "speed": 123 }
    """
    try:
        with open(TELE_FILE) as f:
            data = json.load(f)
    except:
        data = {}
    return jsonify(data)
```

4.4D - Voice Pipeline (/api/voice)

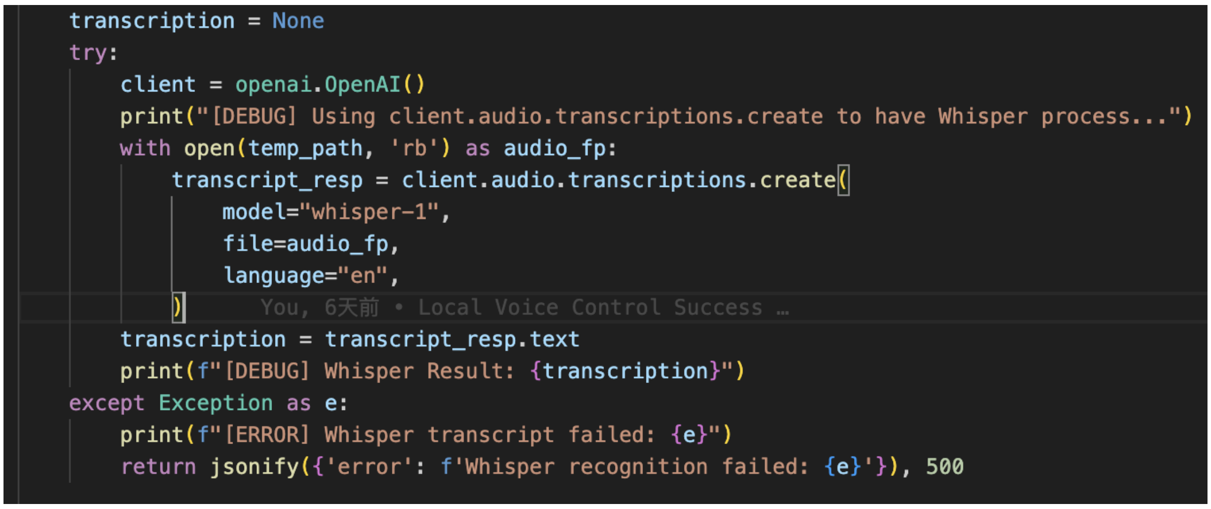
The most elaborate route implements a five-stage pipeline:

1. **Upload and persistence** – the front-end posts a ***multipart/form-data*** request; the server saves the attached WebM file to ***/tmp/voice_input.webm***.



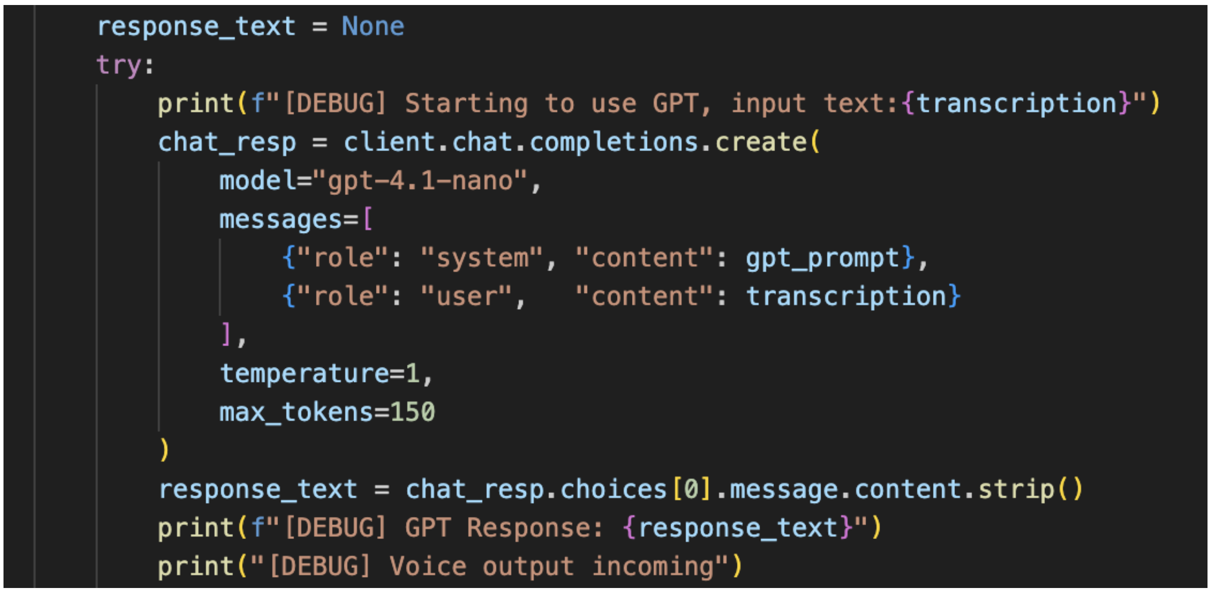
```
temp_path = '/tmp/voice_input.webm'
try:
    audio_file.save(temp_path)        You, 6天前 • Local Voice Control S
    print(f"[DEBUG] voice input saved to {temp_path}")
except Exception as e:
    print(f"[ERROR] voice input save failed : {e}")
    return jsonify({'error': f'Failed to save audio: {e}'}), 500
```

2. **Speech-to-text** – the file is forwarded to OpenAI Whisper (***client.audio.transcriptions.create***). Whisper returns plain text in the ***text*** property.



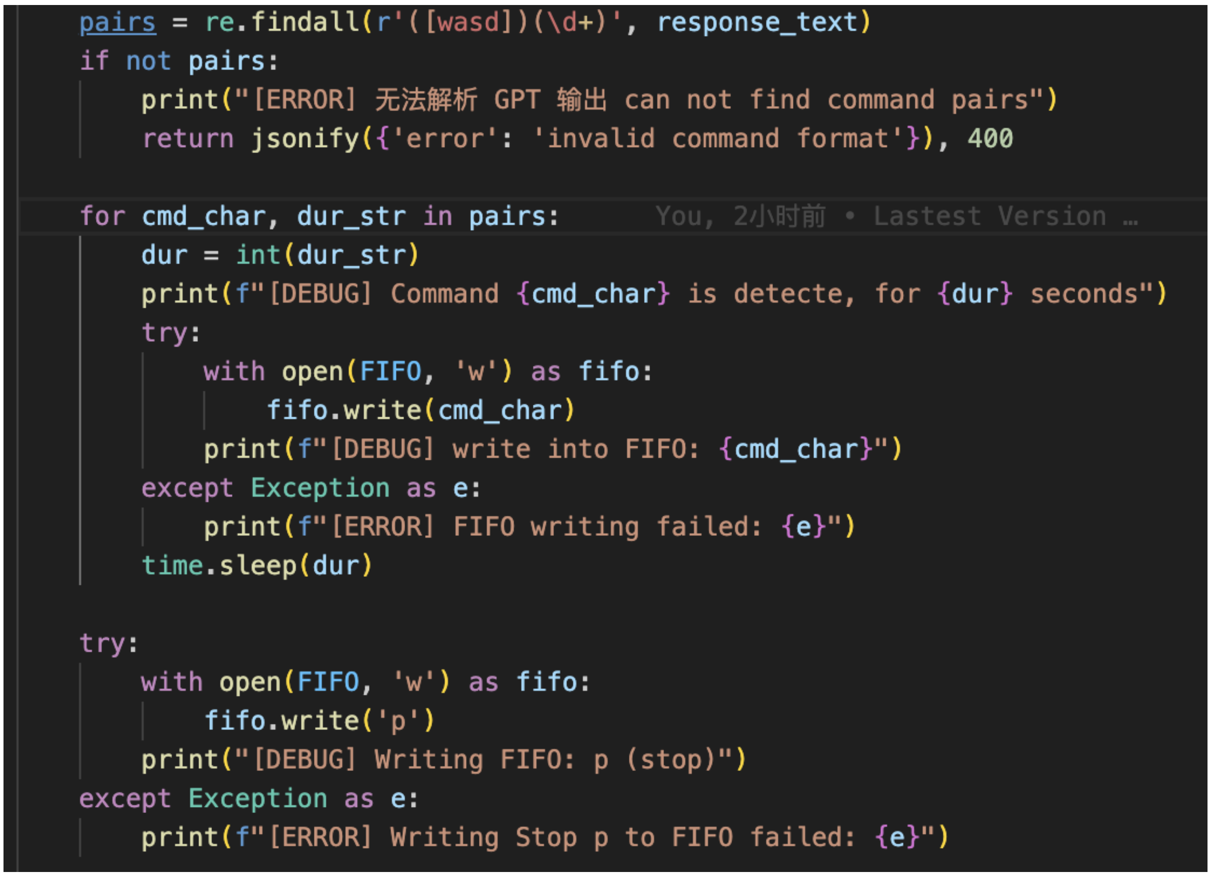
```
transcription = None
try:
    client = openai.OpenAI()
    print("[DEBUG] Using client.audio.transcriptions.create to have Whisper process...")
    with open(temp_path, 'rb') as audio_fp:
        transcript_resp = client.audio.transcriptions.create(
            model="whisper-1",
            file=audio_fp,
            language="en",
        )      You, 6天前 • Local Voice Control Success …
    transcription = transcript_resp.text
    print(f"[DEBUG] Whisper Result: {transcription}")
except Exception as e:
    print(f"[ERROR] Whisper transcript failed: {e}")
    return jsonify({'error': f'Whisper recognition failed: {e}'}), 500
```

3. **Intent parsing** – the transcription is inserted into a chat request together with an external system prompt read from ***gpt_prompt.txt***. GPT-4.1 nano replies with a command string such as ***w5 a3 p***.



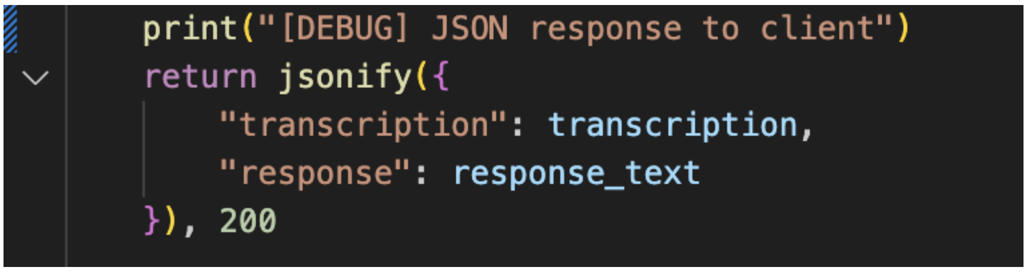
```
response_text = None
try:
    print(f"[DEBUG] Starting to use GPT, input text:{transcription}")
    chat_resp = client.chat.completions.create(
        model="gpt-4.1-nano",
        messages=[
            {"role": "system", "content": gpt_prompt},
            {"role": "user",   "content": transcription}
        ],
        temperature=1,
        max_tokens=150
    )
    response_text = chat_resp.choices[0].message.content.strip()
    print(f"[DEBUG] GPT Response: {response_text}")
    print("[DEBUG] Voice output incoming")
```

4. **Timed dispatch** – a regular expression splits that string into ***(char, duration)*** pairs. For each pair the code writes the character to the FIFO, sleeps for the requested number of seconds, and finally issues a compulsory ***p*** stop.



```
    pairs = re.findall(r'([wasd])(\d+)', response_text)
    if not pairs:
        print("[ERROR] 无法解析 GPT 输出 can not find command pairs")
        return jsonify({'error': 'invalid command format'}), 400

    for cmd_char, dur_str in pairs:        You, 2小时前 • Lastest Version …
        dur = int(dur_str)
        print(f"[DEBUG] Command {cmd_char} is detecte, for {dur} seconds")
        try:
            with open(FIFO, 'w') as fifo:
                fifo.write(cmd_char)
            print(f"[DEBUG] write into FIFO: {cmd_char}")
        except Exception as e:
            print(f"[ERROR] FIFO writing failed: {e}")
        time.sleep(dur)

    try:
        with open(FIFO, 'w') as fifo:
            fifo.write('p')
        print("[DEBUG] Writing FIFO: p (stop)")
    except Exception as e:
        print(f"[ERROR] Writing Stop p to FIFO failed: {e}")
```

5. **JSON response** – the original transcription and the raw GPT command string are returned to the browser, where they are shown to the user and spoken aloud through the Web Speech API.



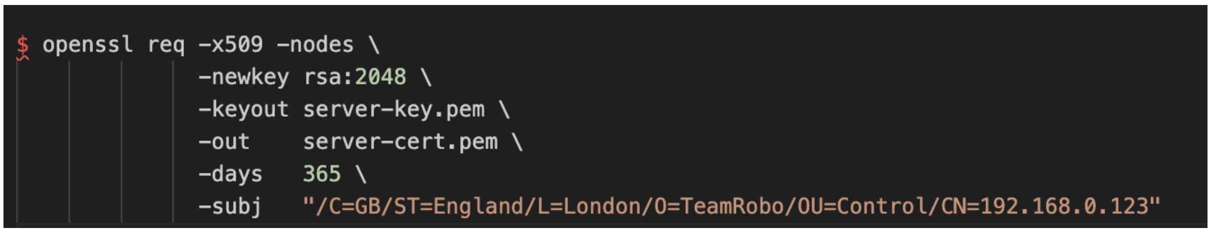
```
    print("[DEBUG] JSON response to client")
    return jsonify({
        "transcription": transcription,
        "response": response_text
    }), 200
```

Throughout the function, every external call--file IO, Whisper, GPT-4.1-nano sits inside its own ***try–except***. A failure at any stage produces a 5xx reply with an explanatory payload yet never crashes the Flask process.

4.4E - OPENSSL Certificates

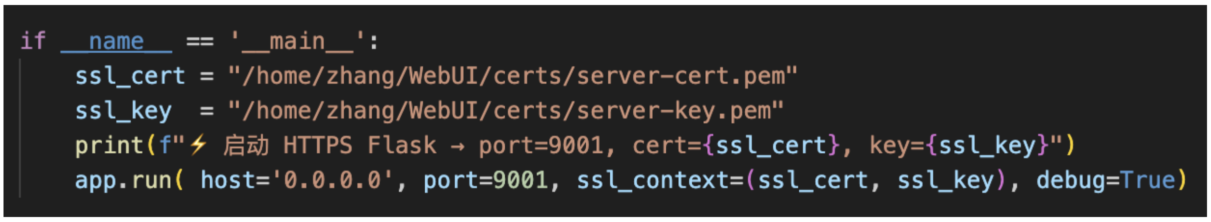
To provide a genuine HTTPS context - required by modern browsers before they will grant microphone access - we elected to generate a self-signed RSA certificate directly on the Raspberry Pi. The procedure is deliberately minimal so that any teammate can reproduce it in under a minute.

We invoked a single OpenSSL command that produces both a private key and a public X.509 certificate in PEM format:



```
$ openssl req -x509 -nodes \
            -newkey rsa:2048 \
            -keyout server-key.pem \
            -out    server-cert.pem \
            -days   365 \
            -subj   "/C=GB/ST=England/L=London/O=TeamRobo/OU=Control/CN=192.168.0.123"
```

The Flask launcher loads the files through *Werkzeug's* built-in SSL wrapper:



```
if __name__ == '__main__':
    ssl_cert = "/home/zhang/WebUI/certs/server-cert.pem"
    ssl_key  = "/home/zhang/WebUI/certs/server-key.pem"
    print(f"⚡ 启动 HTTPS Flask → port=9001, cert={ssl_cert}, key={ssl_key}")
    app.run( host='0.0.0.0', port=9001, ssl_context=(ssl_cert, ssl_key), debug=True)
```

Finally, because the certificate embeds an IP address rather than a domain name, no external DNS is necessary. Should the Pi move to a different subnet, we simply regenerate the certificate with the new address and restart the Flask service - a ten-second maintenance task that keeps the development workflow friction-free.

4.4F - Security Considerations

The application refuses any **/api/command** payload longer than a single character, mitigating the risk of command injection. HTTPS is mandatory; if either the certificate or the key is missing, aborts the launch rather than running in plain HTTP. Environment variables supply the OpenAI API key so that no credential is hard-coded. The front-end never receives raw keys, only pre-processed JSON.

4.4G - Error Logging and Recoverability

All exceptions are printed with a ***[DEBUG]*** or ***[ERROR]*** tag and a human-readable message. Because the robot may disconnect or the network may stall, the design philosophy is “fail soft”: the backend returns partial data or an error JSON, and the front-end degrades gracefully. During testing we repeatedly killed the Wi-Fi interface; once connectivity resumed, both telemetry and voice control recovered without a manual restart.

4.5 - Communication Process Between Frontend and Backend

4.5A - Front-back communication stack

All traffic rides a single HTTPS connection on port 9001. The browser issues three REST calls:

- **POST /api/command –** one printable byte → Flask → written to ***/tmp/robot-cmd*** FIFO → UART task on ESP32 decodes the byte within a few ms.
- **GET /api/telemetry –** 5 Hz poll → Flask streams ***/tmp/telemetry.json***, which the ESP32 rewrites every control cycle; missing file returns {} so the dashboard never flashes.
- **POST /api/voice –** WebM blob → Whisper (speech-to-text) → GPT-4 prompt → flat string; regex splits into *(char, duration)* pairs, each byte piped to the FIFO and timed with ***sleep()***, final ***p*** ensures stop.

Manual D-Pad taps, feature toggles (***l/k, o/i***) and AI commands all converge on the same single-character stream, so the firmware parses exactly one byte protocol.

TLS is terminated by Werkzeug with a self-signed RSA cert; because CN = Pi's LAN IP the browser trusts the origin after a single click and therefore releases ***getUserMedia()***.
All ***fetch*** calls include promise-level error logging; when Wi-Fi drops, polling pauses silently and resumes without a page refresh.

4.5B - Server-side status page

A second Flask micro-service exposes a minimalist "gateway" UI that protects the main dashboard from unsolicited hits. The Python script (redirect_service.py) binds to HTTPS on port 443 and holds an in-memory Boolean called ***connected***. Every second the static page **index.html** issues ***fetch('/status'***); the JSON reply toggles the DOM label between *Not Connected* (red) and *Connected* (green) and enables the "Jump to Target Website" link only when the flag is true. Any background process - our FRP tunnel watchdog, for instance - can flip that flag through a simple ***POST /set_connected call***, after which external users follow the hyperlink and land on the full Robot Web-UI that runs internally on port 9001.

All endpoints inherit the same self-signed RSA certificate chain used by the main application, so the browser remains in a single TLS session from status probe to final dashboard, and no plain-text port is ever exposed.

5 - AI-Based Control Interface for Voice-Driven Robot Interaction

To provide a more intuitive and natural method of interacting with the self-balancing vehicle, an AI-driven voice control module was developed. Instead of relying solely on traditional manual interfaces, this module allows users to issue high-level spoken commands that are interpreted and executed by the robot through a speech-to-command pipeline.

Unlike conventional microphone setups where a recording device is directly connected to the Raspberry Pi, this system avoids such an approach due to the inconsistent audio quality and limited compatibility across different hardware. Instead, microphone access is implemented via the Web UI, allowing the use of the user's local device microphone through browser APIs. This design improves audio quality and ensures cross-platform compatibility. The Web UI's specific implementation is covered in detail in a separate section.

Once the audio is captured on the client side, it is transmitted to the server and processed using OpenAI's cloud-based models. The Whisper model is used for speech-to-text transcription, and the transcribed text is subsequently passed to ChatGPT-4o Nano, which converts the natural language command into a structured sequence of low-level movement instructions suitable for the robot.

A major challenge in this workflow is the latency introduced by cloud inference and sequential processing. To address this, multiple models were evaluated - including standard GPT-4, GPT-3.5-turbo, and O4 mini - in terms of both inference speed and consistency of structured output. ChatGPT-4o Nano was ultimately selected for its balanced trade-off between responsiveness and control over formatting.

In addition, several rounds of prompt optimisation were conducted to reduce latency and improve response determinism. The prompt was iteratively refined to:

- Remove unnecessary explanation requests or role-play instructions
- Use concise role definitions such as "You are a robot assistant"
- Constrain output to a simple syntax: e.g., "W:3, L:2" (meaning move forward for 3 seconds, then turn left for 2 seconds)

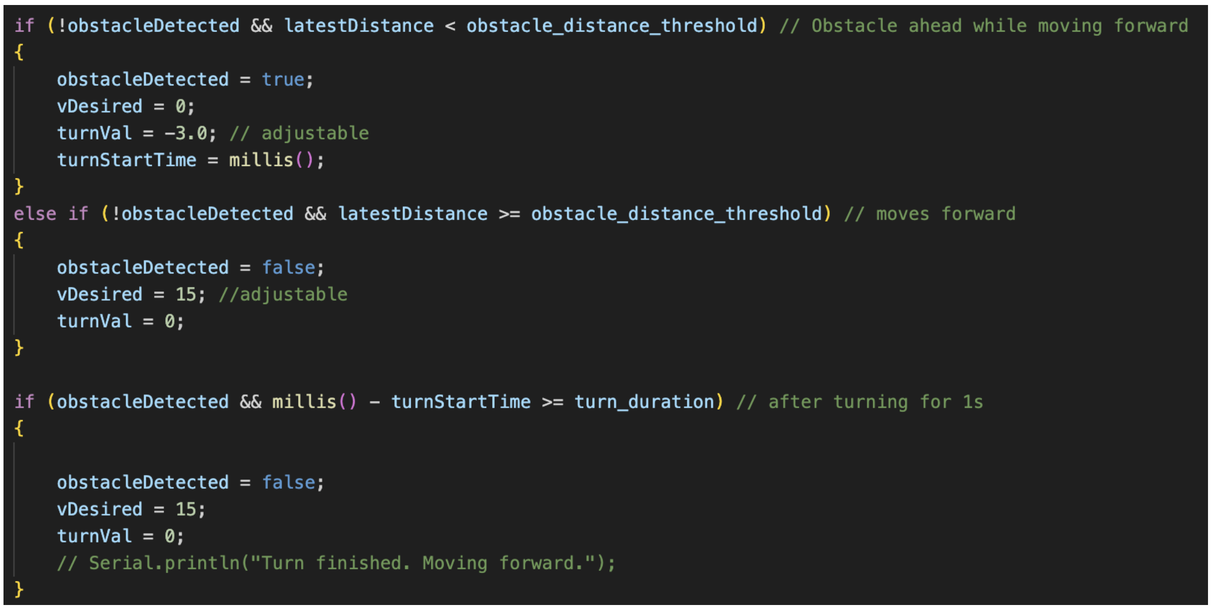
This structured output format was specifically designed to facilitate machine parsing. Each command consists of a capital letter denoting movement direction (W=forward, L=left, R=right, S=stop, etc.), followed by a colon and duration in seconds. A custom parser on the server using regex reads this sequence and translates it into serial control commands, which are then streamed into the FIFO interface. This enables the Raspberry Pi to pass instructions to the ESP32 asynchronously and with minimal runtime decision-making overhead.

By bundling multiple time-tagged instructions into a single response, the system avoids the need for continuous interaction with the language model. In practical tests, the complete workflow - from initial voice capture to command execution - achieved an average **round-trip time** of approximately **5 seconds**, enabling practical non-critical autonomous control through natural spoken language.

6 – Ultrasonic Sensor-based Obstacle Avoiding

A single forward-facing ultrasonic sensor is sufficient to provide the robot with basic obstacle avoidance. A voltage divider circuit was required to step down the sensor's 5V logic-level output to be compatible with the ESP32's 3.3V input pins. The control logic is straightforward: if the calculated distance to an object is less than 20cm, the robot initiates a pre-programmed "turn left" manoeuvre.

A critical implementation detail in the code was the switch from using the standard pulseIn() function to an Interrupt Service Routine (ISR). The pulseIn() function is blocking, meaning it pauses all other code while waiting for the ultrasonic echo. This delay, however brief, was catastrophic for the high-frequency balance loop, causing severe oscillation. By moving the pulse measurement to an ISR, the main processor is freed. It can continue executing the critical balance code uninterrupted, only briefly jumping to the ISR when the echo pulse actually arrives. This architectural change was essential for successfully integrating an autonomous behaviour without compromising the robot's core stability.



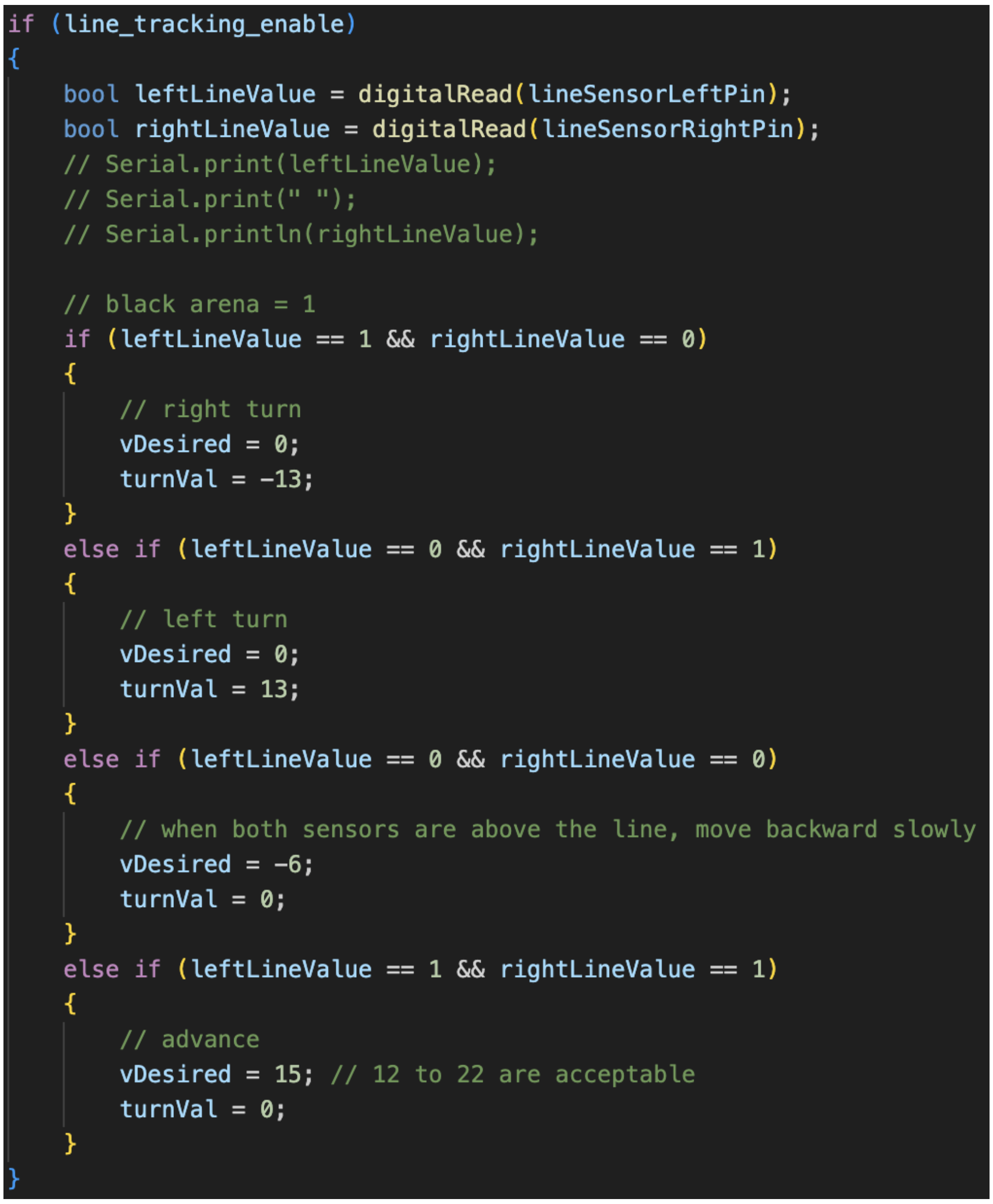
```
if (!obstacleDetected && latestDistance < obstacle_distance_threshold) // Obstacle ahead while moving forward
{
    obstacleDetected = true;
    vDesired = 0;
    turnVal = -3.0; // adjustable
    turnStartTime = millis();
}
else if (!obstacleDetected && latestDistance >= obstacle_distance_threshold) // moves forward
{
    obstacleDetected = false;
    vDesired = 15; //adjustable
    turnVal = 0;
}

if (obstacleDetected && millis() - turnStartTime >= turn_duration) // after turning for 1s
{

    obstacleDetected = false;
    vDesired = 15;
    turnVal = 0;
    // Serial.println("Turn finished. Moving forward.");
}
```

As shown in the figure above, while moving forward, the robot continuously checks for obstacles ahead and turns for one second when one is detected.

If time permits, autonomous maze navigation will also be explored.

7 – Infrared Sensor-based Line Tracking

The robot was equipped with autonomous line-tracking capability using two refractive light sensors. This two-sensor configuration was a trade-off; a single sensor was not robust enough to handle sharp turns, while three introduced excessive physical vibration. The system operates on a simple binary logic (black surface = 1, yellow line = 0) to determine its position relative to the line, as shown in the figure below. Also, a demonstration of the robot tracking a curved line path is included in **Appendix 9.3**.



```
if (line_tracking_enable)
{
    bool leftLineValue = digitalRead(lineSensorLeftPin);
    bool rightLineValue = digitalRead(lineSensorRightPin);
    // Serial.print(leftLineValue);
    // Serial.print(" ");
    // Serial.println(rightLineValue);

    // black arena = 1
    if (leftLineValue == 1 && rightLineValue == 0)
    {
        // right turn
        vDesired = 0;
        turnVal = -13;
    }
    else if (leftLineValue == 0 && rightLineValue == 1)
    {
        // left turn
        vDesired = 0;
        turnVal = 13;
    }
    else if (leftLineValue == 0 && rightLineValue == 0)
    {
        // when both sensors are above the line, move backward slowly
        vDesired = -6;
        turnVal = 0;
    }
    else if (leftLineValue == 1 && rightLineValue == 1)
    {
        // advance
        vDesired = 15; // 12 to 22 are acceptable
        turnVal = 0;
    }
}
```

The tuning process was iterative. The distance between the sensors was physically adjusted to minimise oscillation when centred on the line. The robot's turning speed was reduced to ensure it had sufficient time to react at corners without overshooting, while its speed on straight sections was increased for efficiency. While functional at present, future work could involve implementing a PID controller for this feature to achieve faster, smoother line following.

IMPERIAL

8 – Battery Monitoring

8.1 – Methods

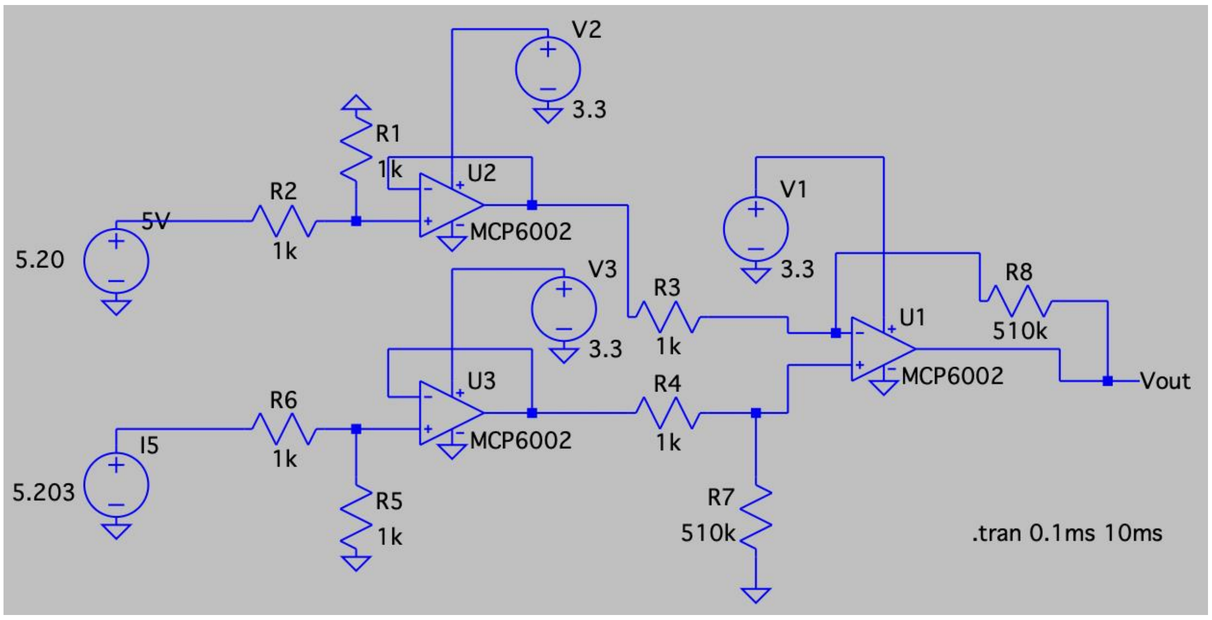
Research identifies three main approaches to monitoring a battery: tracking voltage over time, measuring energy consumption, and measuring charge consumption. The voltage-over-time method was discarded due to its variability under different discharge rates. Measuring either energy or charge used are functionally equivalent; however, the latter was chosen as it requires only current measurement, making it simpler to implement.

8.2 – Circuit Design

An early challenge was implementing accurate battery current monitoring. The provided sensing resistors were problematic: one produced two voltages (~15V) far too high for the ESP32's ADC with a maximum input voltage of 4.096V, while the other produced a voltage drop (~3.3mV) so small that it was indistinguishable from noise.

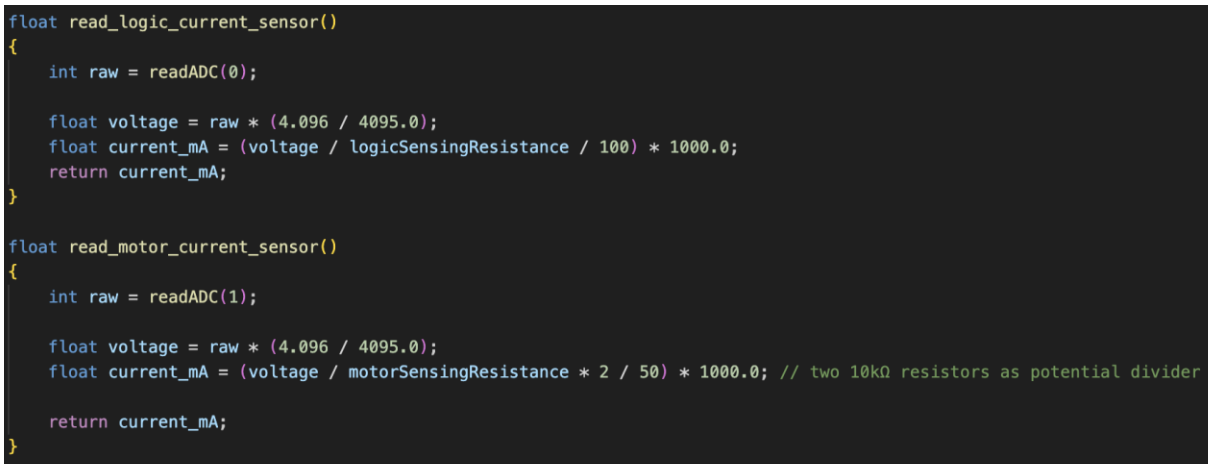
Several solutions were prototyped:

- **Voltage Dividers**: While simple, this approach to scale the 15V signal introduced an unacceptable amount of noise, making the readings and hence the voltage difference unreliable.
- **Custom Differential Amplifier**: A more complex circuit using MCP6002 op-amps was designed and simulated, as shown below. However, both the simulated and physical prototypes exhibited a significant offset voltage of approximately 1.3 mV, making it unsuitable for accurately calculating the difference between the input voltages using the output voltage.



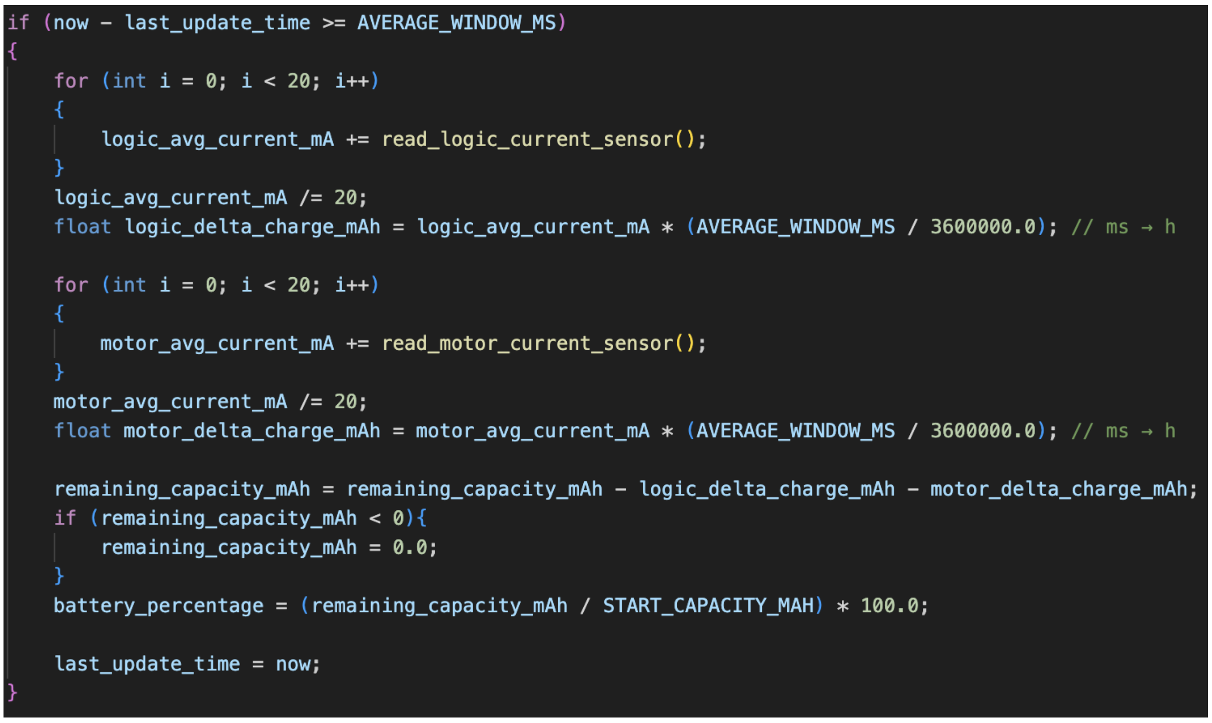
- **Specialized IC Solution**: Success was finally achieved using INA180A3 and A2 current-sense amplifiers. These integrated differential amplifiers are specifically designed for this purpose to accept common-mode voltages from –0.2 V to +26 V independent of the supply voltage, and they output clean, stable, and reliable signals that were well within the ADC's measurable range.

8.3 – Software



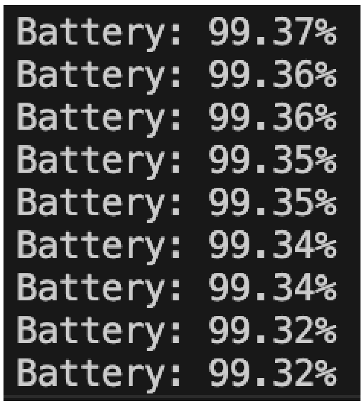
```
float read_logic_current_sensor()
{
    int raw = readADC(0);

    float voltage = raw * (4.096 / 4095.0);
    float current_mA = (voltage / logicSensingResistance / 100) * 1000.0;
    return current_mA;
}

float read_motor_current_sensor()
{
    int raw = readADC(1);

    float voltage = raw * (4.096 / 4095.0);
    float current_mA = (voltage / motorSensingResistance * 2 / 50) * 1000.0; // two 10kΩ resistors as potential divider

    return current_mA;
}
```

With a reliable hardware signal, a C++ program was implemented on the ESP32 to manage the battery data, as shown above. The name logic refers to ESP32, Raspberry Pi and all connected sensors, 100 and 50 are the gains of INA180 current-sense amplifiers, and a potential divider was used to scale the output voltage to within 4.096V input range of the ADC.



```
if (now - last_update_time >= AVERAGE_WINDOW_MS)
{
    for (int i = 0; i < 20; i++)
    {
        logic_avg_current_mA += read_logic_current_sensor();
    }
    logic_avg_current_mA /= 20;
    float logic_delta_charge_mAh = logic_avg_current_mA * (AVERAGE_WINDOW_MS / 3600000.0); // ms → h

    for (int i = 0; i < 20; i++)
    {
        motor_avg_current_mA += read_motor_current_sensor();
    }
    motor_avg_current_mA /= 20;
    float motor_delta_charge_mAh = motor_avg_current_mA * (AVERAGE_WINDOW_MS / 3600000.0); // ms → h

    remaining_capacity_mAh = remaining_capacity_mAh - logic_delta_charge_mAh - motor_delta_charge_mAh;
    if (remaining_capacity_mAh < 0){
        remaining_capacity_mAh = 0.0;
    }
    battery_percentage = (remaining_capacity_mAh / START_CAPACITY_MAH) * 100.0;

    last_update_time = now;
}
```

The code finds the average voltage reading at the start of every second and sums it over intervals of one second to estimate charge consumption. Based on the battery's known 2000mAh capacity, this allows the system to calculate and display a real-time battery percentage on the Web UI. The software estimated a battery life of approximately 1 hour under normal usage, which was consistent with real-world performance, as shown below.



```
Battery: 99.37%
Battery: 99.36%
Battery: 99.36%
Battery: 99.35%
Battery: 99.35%
Battery: 99.34%
Battery: 99.34%
Battery: 99.32%
Battery: 99.32%
```

9 - Speaker System Design and Implementation

This chapter describes the design, development, and evaluation of a speaker system designed specifically for audio playback functionality within the robotic platform. It explains the initial motivations behind the audio system, outlines detailed hardware and software decisions, evaluates system performance, discusses encountered design limitations due to hardware constraints, and suggests potential improvements for future iterations.

9.1 - Motivation and Requirements

Initially, the robotic platform successfully implemented speech recognition and command execution capabilities based on ChatGPT integration. To further enhance user interaction and provide more immersive experiences, a speaker system was proposed for audio feedback and multimedia playback.

The primary motivations behind this audio feature included:

Enabling the robot to broadcast audio cues, general feedback sounds, or music, enriching user-robot interactions.

Adding entertainment value by playing pre-recorded audio files, thus contributing to a more intuitive user experience.

However, given the computational limitations of the chosen controller—the Raspberry Pi 3A—we observed significant delays (ranging from ten seconds to tens of seconds) when performing local text-to-speech conversion. Such substantial latency negatively impacted real-time interaction quality, ultimately necessitating reconsideration and simplification of the speaker's intended use.

To avoid this issue within the timeframe and scope of the project, local text-to-speech conversion was not pursued beyond initial testing. Instead, audio playback was limited exclusively to pre-recorded standard audio files (such as music tracks), deferring potential real-time speech synthesis to future phases, possibly via cloud computation.

Thus, the final practical requirements included:

Playback of clear, audible music and pre-recorded audio files.

Sufficient audio loudness suitable for typical user distances (approximately 1–3 metres).

Simple hardware infrastructure easily integrable with the existing Raspberry Pi architecture.

Straightforward, automated software playback capability integrated via Python.

Complex on-device text-to-speech functionality, while desirable, was noted as a potential future enhancement rather than an immediate attainable goal.

9.2 - Hardware Component Selection and Design Decisions

The Raspberry Pi 3A used for our robotic platform includes a built-in analog stereo audio output accessible via a standard 3.5mm AV jack. However, due to its limited onboard amplifier power output, connecting a standalone speaker directly to the headphone jack would result in insufficiently loud audio.

Considering the project's requirement for louder speaker sound output for music playback, we selected the ABS-210-RC loudspeaker, offering a maximum output power of 1.5W. This choice allows sufficiently audible music playback at operational distances typical in our scenarios.

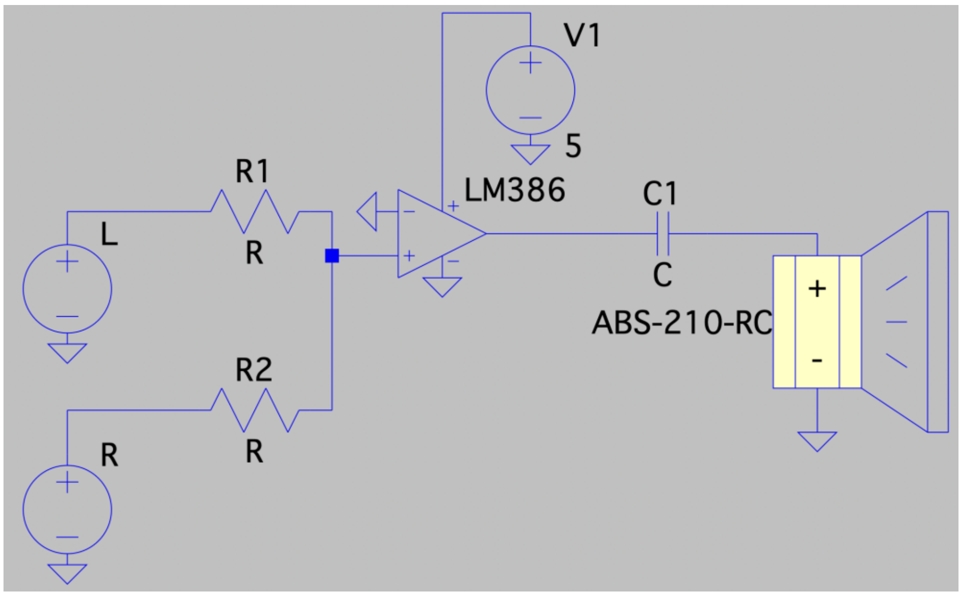
To effectively drive this speaker, we implemented a simple analog audio amplification circuit based on the LM386N1 IC amplifier, a cost-effective and readily available component. Providing a fixed internal gain of approximately 20, this amplifier suitably increases audio levels for playback purposes, powered conveniently from the existing ESP32 component's regulated +5V DC source.

Furthermore, the Raspberry Pi's 3.5mm stereo output presented a challenge: the chosen single speaker required mono input. Audio signals containing centered information (such as vocals) would lose clarity and intelligibility if only one channel (left or right) were connected. To overcome this, we employed a straightforward passive mixing circuit whereby the two stereo channels were combined using two separate 100Ω resistors placed in series. This approach effectively merges both channels into an acceptable mono output suitable for music playback, ensuring minimal signal quality loss and acceptable audio clarity.

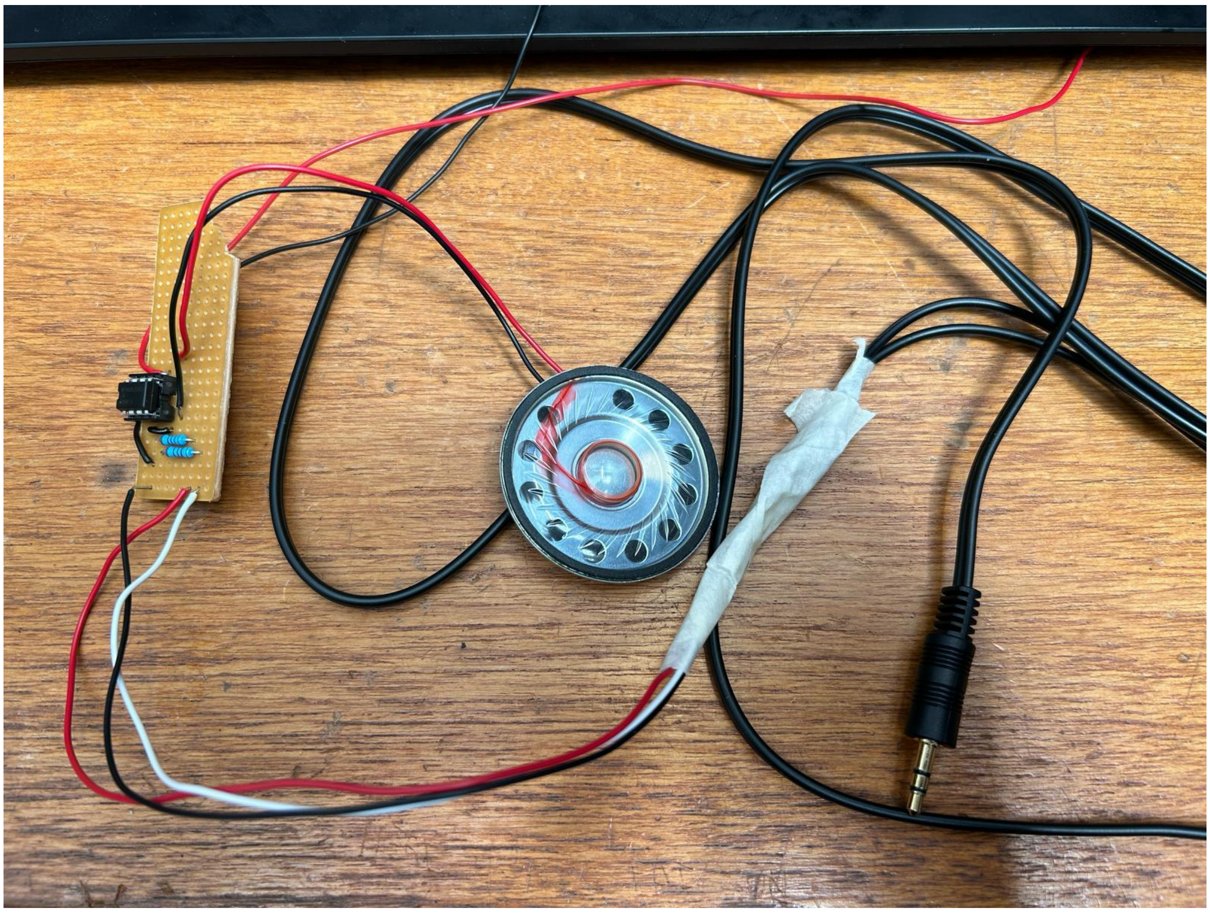
Hence, the final hardware configuration consisted of:

ABS-210-RC loudspeaker (1.5W rated output).

LM386N1 amplifier with a fixed gain of 20, powered via +5V DC from ESP32.



Simple passive stereo-to-mono mixing circuit comprised of two 100Ω resistors.



9.3 - Software Integration and Playback Implementation

With hardware setup completed, audio playback control was implemented through a software platform streamlined for performance given hardware constraints.

Due to observed Raspberry Pi 3A computational performance limitations precluding practical local speech synthesis, the adopted audio playback methodology relies entirely on pre-recorded mp3 audio files, typically music tracks or predefined feedback sounds.

The Linux-compatible multimedia playback tool "ffplay" (from the widely used FFmpeg suite) was selected. It offers dependable realtime audio playback with minimal computational overhead and no significant delay, fully appropriate given the project's current limitations and objectives.

The Python programming language—already extensively employed within project control systems—facilitated automated playback. This was seamlessly accomplished via Python's built-in OS library and straightforward shell command integration:

```
os.system(f"ffplay -nodisp -autoexit {file_path}")
```

This solution proved stable, straightforward, and effectively integrated into the project's overall software structure.

IMPERIAL

9.4 - System Testing and Evaluation

Extensive tests were conducted to validate audio amplifier circuit performance, sound quality, and playback reliability:

Amplifier circuit verification:

Input signals from a controlled waveform generator verified proper amplifier functionality and gain stability (20x) measured by oscilloscope observation.

Stereo-to-mono mixing effectiveness:

Assessed by feeding stereo test signals, ensuring successful mixing with no significant frequency attenuation or distortion audibly detected.

End-to-end playback testing:

Music audio files playback via Python-initiated ffplay commands repeatedly exhibited stable performance without system crashes.

Observations confirmed loudness adequate for typical operating conditions (1–3 meters).

Playback latency was measured consistently at under 100 ms during tests, indicating acceptable responsiveness for music playback scenarios.

Final evaluation feedback from group members qualitatively showed that the designed speaker system reliably and effectively achieved initial functional goals within the established Raspberry Pi performance limitations.

9.5 - Discussion and Reflection

Regrettably, the initial ambitious goal of achieving real-time text-to-speech conversion for robot feedback was significantly restricted by inherently limited hardware capabilities (computational performance of Raspberry Pi 3A). Preliminary experimentation confirmed that local speech synthesis incurred delays of tens of seconds—far beyond acceptable real-time constraints. Consequently, immediate real-time TTS implementation proved impractical. Nevertheless, realization of reliable music playback through simplified hardware and software solutions provided a successful, practical alternative within available timeframes and resource constraints.

Future development efforts can investigate shifting computationally intensive real-time tasks such as TTS into cloud computing environments, substantially reducing latency and processor load. Integration of cloud-based audio streaming might enable effective TTS utilization without hardware-imposed lag.

9.6 - Recommendations for Future Work

Based on the lessons learned during the implementation phase, several clear avenues for future system upgrades include:

Integrating cloud computing solutions (e.g., AWS, Azure, Google Cloud Platform) to offload real-time text-to-speech processing from the Raspberry Pi hardware, thereby enabling practical, real-time robot dialogue capability.

Adoption of higher-fidelity speakers or specialized digital amplifier modules (i.e., Class D amplifiers) for improved music playback quality, dynamic range, and efficiency.

Enhancing signal mixing hardware through active audio mixers or buffer amplifiers to better preserve original audio fidelity during stereo-to-mono conversion stages.

Introducing dynamic software-controlled volume adjustment components (digital potentiometers or digitally-controlled amplifier gain adjustments), providing enhanced interactive adjustments suitable for diverse operational settings.

Implementing such enhancements would markedly extend functional capabilities, usability, and audio performance, significantly elevating long-term project potential.

IMPERIAL

10 - Appendix

10.1 – GitHub

https://github.com/will03216/WhatdoUmean-Electronics_Design_Project_2_Balance_Robot

10.2 – Cost of Materials

Component	**Unit Cost**	**Quantity**	**Total Cost**
Male RCA x 2 to Male 3.5mm Stereo Jack RCA Cable	£1.70	2	£3.40
ABS-210-RC speaker	£2.28	1	£2.28
INA180A3IDBVT current sense amplifier	£0.76	2	£1.51
INA180A2IDBVT current sense amplifier	£0.74	2	£1.48
101020585 Sensor Board, ICM20600, AK09918	£9.39	1	£9.39
RPI 8MP CAMERA BOARD, Raspberry Pi Camera Board, Version 2	£11.28	1	£11.28
Joy-It Infrared Line Tracker	£1.85	2	£3.70
PSG04177 SENSOR, ULTRASONIC, 20MM-4.5M	£3.19	2	£6.38
Total Cost:			£39.42

10.3 - Video Links

[**2.4**] - 90 Degree precise turning

[**2.4**] - Precise position control

[**2.6**] - Fast acceleration and fast turning

[**3.5**] - WebUI control

[**6**] - Obstacle avoiding

[**7**] - Line tracking

[**9**] - Speaker Demonstration