1 System implementation

1.1 Overall System Architecture

AI chatbot:

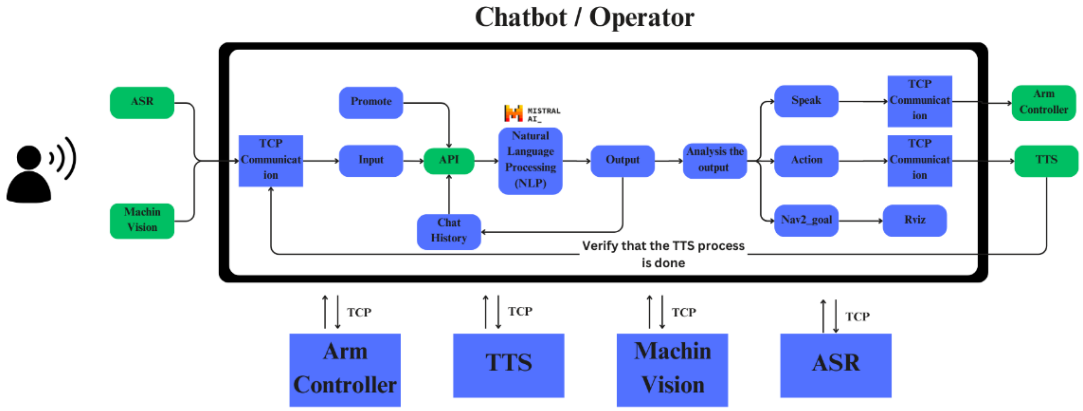

Figure 1.1 AI chatbot Architecture

This block diagram the chatbot system architecture responsible for transfer the user voice commands to easy command that triggering robot actions accordingly. The chatbot, named ELA Chat, is structured as an operator where each component communicates via TCP sockets, allowing flexible integration and separation of processing units.

System Overview

As shown in the figure, the system starts with two types of input:

1) Voice input using ASR (Automatic Speech Recognition)
2) Gesture input using Machine Vision

Both types of input are sent to the chatbot through TCP.

Once a user speaks or do a gesture, the input is sent to the Input Handler, which formats it for the Natural Language Processing (NLP) system. The core of the chatbot relies on the Mistral API, which processes the input using a prompt and generates a structured response. The special prompt made for including chat history to ensure good interaction for the user.

Processing Flow

Also the prompt made to give a response from the chatbot to includes these three parts:

- talk: what the robot want to say
- action: which gesture or arm movement to perform

- nav2_goal: where the robot should go

Each of these is handled separately:

- The talk part is sent to the TTS (Text-to-Speech) module.
- The action is sent to the Arm Controller.
- The nav2_goal is published in ROS2 to RViz for navigation.

All of these are sent over separate TCP connections. This setup helps each module work on its own without delay or interference.

There's also a if condition to check and to ensure that the text-to-speech (TTS) module has done speaking before the robot takes the next step. This prevents speech and movement interference.

Flexible Integration

The system uses a modular design, which makes it easy to upgrade, maintain and debug. It works with several external modules through TCP including:

1) ASR for voice input
2) Machine Vision for gesture input
3) Arm Controller for moving the robot's arm
4) TTS for converting text to speech

Each one of these modules run alone and that helps the system to stay responsive and stable. This setup helps to keep the communication between parts and the chatbot clear and also help to avoids delays during interaction.

Arm Controller GUI:

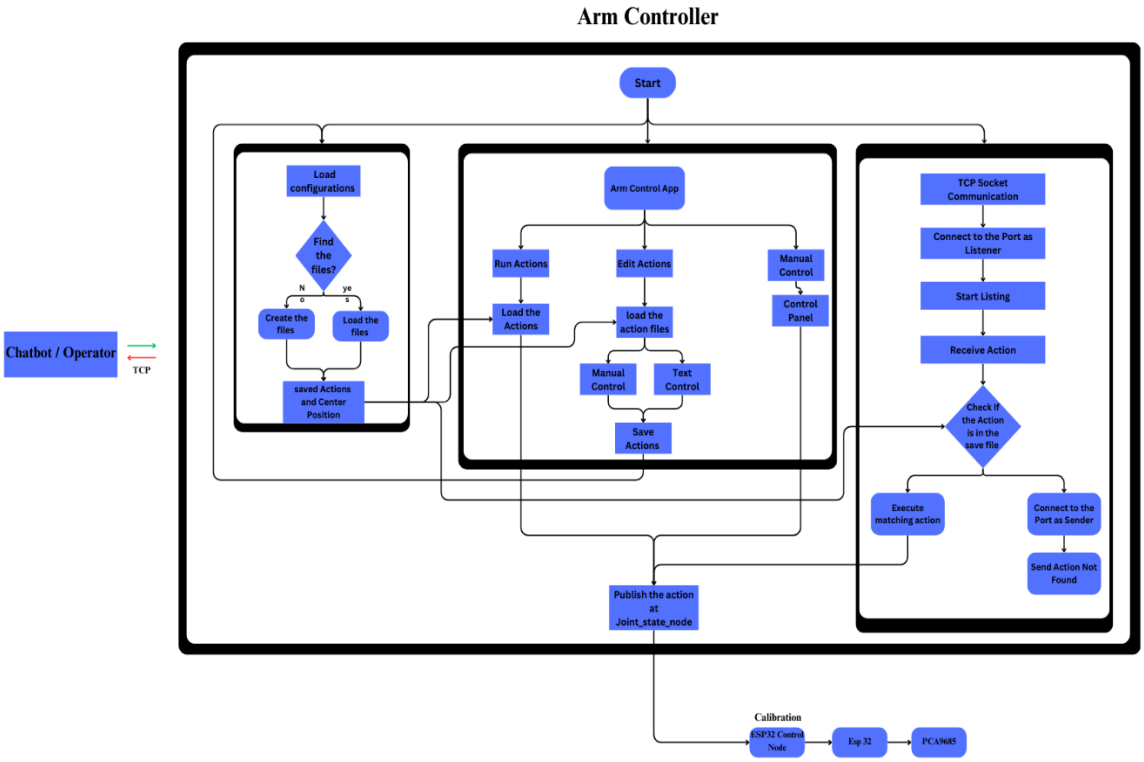

Figure 1.2 Arm Controller GUI Architecture

Controlling the robot's arm is not just about moving the joints. It means turning the chatbot's commands into clear and accurate motions that help the robot move it is own arms. The Arm Control System shown in the Figure is a modular setup that connects stored actions, real-time control, and TCP commands to make the robot move it is arm in the real world.

System Initialization and Configuration

The main program is the Arm Control GUI. When it starts, it checks for configuration files that store saved joint positions and motion actions. If the files are not found, like on a new device or first run, it creates them with default values. This makes sure the arm can go back to it is center point when it's idle or resetting.

Control Modes and Interface

The system supports two control modes:

1) Manual control using a GUI, where the user can move the joints with sliders. This is useful for testing or edit the positions.
2) Text-based control or manual controller where joint values can be saved under names like "wave_left" or "salute." These actions can be continues or with limit amount of movement and run when needed.

Once an action is selected, it is sent to the joint_state_node in ROS2. From there, the data goes to the ESP32, which creates PWM signals using the PCA9685 to move the motors. This setup allows both manual controller and automatic controller inside the full robot system.

TCP Command Triggering

The arm controller listens for commands sent from the chatbot over TCP. For example, if it receives "wave_left," it checks if that action is saved. If it is saved the arm moves. If not saved a message is sent back saying the action was not found. This will help of avoiding any errors and debugging.

Motor Communication and Calibration

The ESP32 is connected by USB. It listens for joint data and sends PWM signals to the motors. Calibration is also available through the GUI, where the user can adjust the angle of each joint to make sure the movement is smooth and correct.

The arm control system is not just a fixed setup. It can be updated with new actions tested with the GUI, and triggered by the chatbot. This helps the robot to be more humanly and match its physical motion with the way it talks.

Machine vision

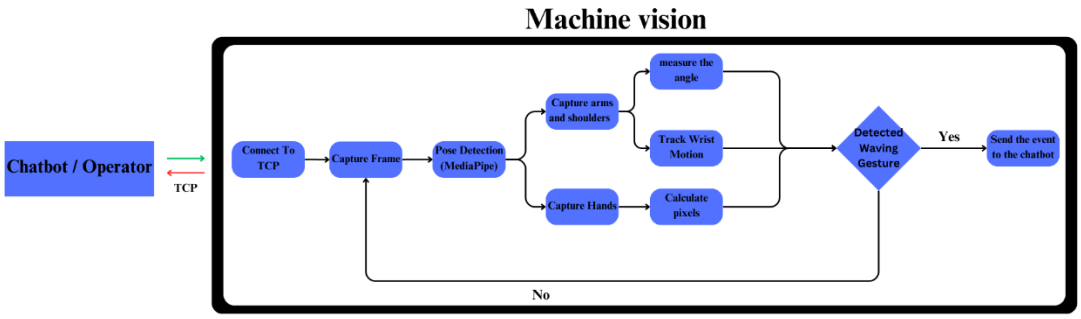

Figure 1.3 Machine vision Architecture

Besides using voice the robot also has eyes can be used in understand human gestures. This lets the user interact with the robot by moving their hands or body. The Machine Vision Module shown in the Figure uses a camera to detect gestures like waving and handshaking. These eyes let the robot to be more like a human.

Pose Detection with MediaPipe

The system starts by making a TCP connection with the chatbot. Then, it keeps reading video from the webcam. The frames are sent to MediaPipe, which checks for body parts like the shoulders, elbows, wrists, and hands.

After getting these points, the system looks at how they move over time to figure out which gesture is being made.

Gesture Detection Rules

- Waving Gesture: The system watches how the wrist moves left and right. It also checks if the arm is raised by looking at the angle between the shoulder and upper arm. If the hand is moving side to side and the arm is raised above the shoulder, the system detect it as a wave. When this happens, it sends a “wave” message to the chatbot through TCP. The chatbot can then tell the robot to wave back.
- Handshake Detection: The system also checks if someone is trying to shake hands. This is done by checking how fare the hand is and since it is hard to detect distance using 2D camera. Smart if logic help in detecting the distance by check the number of the pixels of the hand. If the hand is more than 200 pixels, it means the hand is close to the camera, like reaching out. When this happens, a “shake_hand” message is sent to the chatbot.

Scanning and Fallback

If no wave or handshake is seen, the system keeps checking the next frame. This keeps the system running without stopping or delays.

Fast Response and Easy Setup

The gesture events are sent to the chatbot through TCP. Since this system works on its own, the chatbot can still do other jobs like talking or navigation at the same time. This makes the system flexible and easy to add more features later.

TTS (Text To Speech)

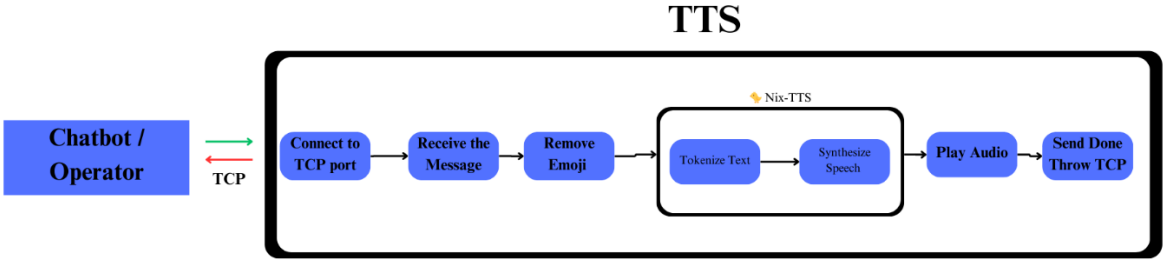

Figure 1.4 TTS (Text To Speech) Architecture

The TTS system gives the robot a voice. It turns the chatbot reply into speech so ELA can talk. This is how the robot can greet users or confirm commands or respond with voice in a human way.

TCP-Based Design

The TTS works as a separate part of the system. It connects to the chatbot over TCP. When the chatbot wants to speak, it sends the message as plain text through the connection.

Before making the speech, the system cleans the text. It removes any emoji or special characters that could cause problems. This step makes sure the message is ready for the voice engine to pronounces.

Speech Synthesis with Nix-TTS

The TTS uses Nix-TTS to make the voice. First, the cleaned text is broken into smaller parts so the voice engine can understand how to say it correctly. after that, the engine will generate the audio and then it play it through the speaker.

Feedback to Chatbot

When the robot finishes speaking the TTS system sends a "Done" message to the chatbot. This is important so the robot doesn't speak again or pick it is own voice as a user input. It helps all parts of the robot stay in sync and respond at the right time.

ASR (Automatic Speech Recognition)

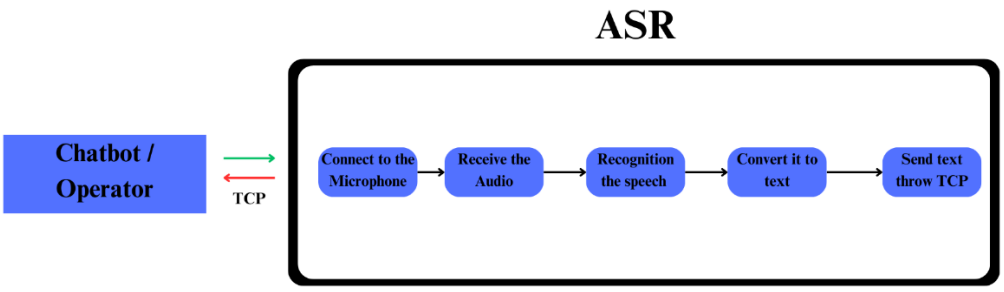

Figure 1.5 ASR (Automatic Speech Recognition) Architecture

For the robot to feel more natural, it must be able to listen. The ASR system shown in the Figure lets the robot hear and turn voice into text so the chatbot can understand and reply.

Voice Input in Real Time

The system starts by connecting to the microphone. Usually, this is a far-field mic placed on the robot's body. Once it's connected, the system keeps listening for any voice around the robot.

Speech Detection and Text Conversion

When the system hears someone speaking, the audio is sent to the speech recognition engine. It uses real-time tools like the speech_recognition library in Python or Google's recognizer to understand what is being said. The process includes:

1) Picking out human speech from background noise
2) Matching sounds with known word patterns
3) Converting the speech into full sentences

After that, the result is turned into clean text. This is the part the chatbot uses to understand the user's message.

Sending the Text to the Chatbot

Once the speech is changed into text, it is sent right away to the chatbot using TCP. The ASR system does not store anything and keeps working all the time without slowing down. This makes the voice input fast and ready for real-time use.

1.2 Hardware Integration

USB Hub

The hardware was designed to support modular development. It also keeps all parts running on one processing unit. The Jetson Orin Nano is the main board. It handles computing, perception, and control.

A USB hub is connected to the Jetson Nano. It does add more USB ports and gives power to devices that need it. The components that are plugged into the hub:

1) A microphone to record the input voice for the ASR system.
2) An ESP32 microcontroller gets joint angle data and sends PWM signals to servo motors using the PCA9685.
3) A Logitech C270 camera to record video for gesture detection using MediaPipe.
4) A speaker plays audio from the TTS system.
5) A USB power cable for the speaker.

Figure 1.6 the physical setup of the USB hub

Some peripherals are connected directly to the Jetson's own USB ports. These include a LiDAR sensor, an Intel RealSense D435i, and an Arduino board that drives the base motors. Other team members added these parts, but they also run on the same system during use.

Servo Motor Layout and Channel Mapping

The robot's left and right arms each use five servo motors. These are controlled by a PCA9685 16-channel PWM driver. The driver connects to the ESP32 using I2C. Each servo channel matches a specific joint. The servo setup is listed below:

Table 1.1 servo channel for each joint

Arm Side	Joint	PCA9685 Channel
Left	Upper Arm X	0
	Upper Arm Y	1
	Forearm Z	2
	Forearm X	3
	Hand/Finger	4
Right	Upper Arm X	8
	Upper Arm Y	9
	Forearm Z	10
	Forearm X	11
	Hand/Finger	12

This setup lets the system control each servo one by one. It can run gestures or custom movements. Figure 4.7 shows a simple wiring diagram with each servo ID marked.

ESP32 to PCA9685 Wiring and Power Supply

The PCA9685 connects to the ESP32 with I2C. The wiring is done as follows:

Table 1.2 connection for PCA9685

PCA9685	ESP32
GND on PCA9685	GND on ESP32
SCL	D22
SDA	D21
VCC	3.3V on ESP32
V+ (Motor Power)	11.1V battery

The PCA9685 has two power inputs. One is for logic level communication (VCC). The other is for motor power (V+). The logic part runs on 3.3V, which fits the ESP32. The motors need more power, so an 11.1V Li-ion battery is used.

This setup gives enough power to the servo motors. It also protects the logic side by using safe voltage levels. All components share a common ground. This keeps the signal reference stable across the battery, PCA9685, and ESP32.

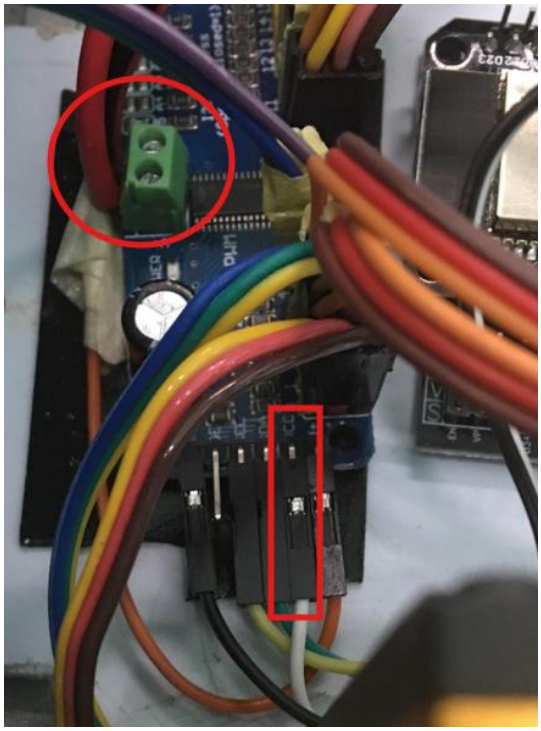

Figure 1.7 dual-power wiring configuration

Microphone and camera placement

The microphone is placed inside the robot's head near the ear area. This spot was chosen to match common human interaction zones. It also helps to jeep the design simple and easy to use. Placing the mic here help to reduce motor noise. It also creates a more natural point for voice interaction.
The camera is fixed near the robot's left shoulder. This gives a wide and steady view of the area around the robot. It works well for users who are waving far away or near want to shake hands. This position also keeps the camera view clear from the robot's moving arms.

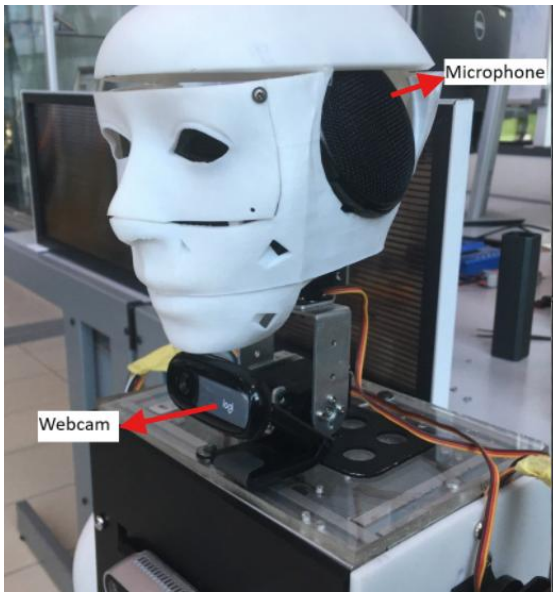

Figure 1.8microphone and camera are mounted on the robot's frame

1.3 Software Design

The software design is split into modular components that interface through ROS 2 and TCP sockets. The design focuses on two primary domains:

1) Arm Control Software Design
2) Chatbot, TTS, and ASR Integration

Arm Control Software Design

The arm control system is used to turn joint values into real movement. It is made of a few main parts:

A. ROS2 Launch and Controller Setup

Launch Files

The launch files are used to start the robot's state publisher, the joint broadcaster, and tools like RViz or Gazebo. These files also load the robot's URDF file using xacro. The URDF shows the robot's body parts and joint setup.

For example, in display.launch.py, the robot URDF is loaded and the state publisher is started like this:

```
model_arg = DeclareLaunchArgument(
    name="model",
    default_value=os.path.join(
        get_package_share_directory("ela2_arms"),
        "urdf",
        "ela2arms.urdf.xacro"
    ),
    description="Absolute path to the robot URDF file"
)

# Parse the xacro file and set it as the robot description
robot_description = ParameterValue(
    Command(["xacro ", LaunchConfiguration("model")])
)

# Robot State Publisher node
robot_state_publisher = Node(
    package="robot_state_publisher",
    executable="robot_state_publisher",
    parameters=[{"robot_description": robot_description}],
)
```

Figure 1.9 display.launch.py

This part of the code sets the URDF path, processes it using xacro, and starts the robot state publisher. There is also a gazebo.launch.py file. It sets the simulation path using gazebo_model_path and spawns the robot in Gazebo. These launch files are used to start the system for both real hardware and simulation.

Controller Setup

The controller setup is written in a YAML file. It defines the joint trajectory controllers for both arms and also the joint state broadcaster. The file includes joint names and settings for each controller.

Here's a part from ros2_controllers.yaml:

```
left_arm_controller:
  action_ns: follow_joint_trajectory
  type: FollowJointTrajectory
  default: True
  joints:
    - left_upper_arm_y_joint
    - left_upper_arm_x_joint
    - left_forearm_z_joint
    - left_forearm_x_joint
    - left_finger_joint

right_arm_controller:
  action_ns: follow_joint_trajectory
  type: FollowJointTrajectory
  default: True
  joints:
    - right_upper_arm_y_joint
    - right_upper_arm_x_joint
    - right_forearm_z_joint
    - right_forearm_x_joint
    - right_finger_joint

joint_state_broadcaster
  type: "joint_state_broadcaster/JointStateBroadcaster"
```

Figure 1.10 ros2_controllers.yaml

This configuration connects the right joints to the correct controllers and sets the broadcaster to send joint data.

To run the controllers there's launch file called controller_launch.py. This file starts the state publisher and the controller spawner nodes. For example:

```python
# Robot state publisher node
robot_state_publisher = Node(
    package='robot_state_publisher',
    executable='robot_state_publisher',
    parameters=[{'robot_description': robot_description}]
)

# Joint state broadcaster spawner
joint_state_broadcaster_spawner = Node(
    package='controller_manager',
    executable='spawner',
    arguments=['joint_state_broadcaster', '--controller-manager', '/controller_manager']
)

# Left Arm Controller Spawner
left_arm_controller_spawner = Node(
    package='controller_manager',
    executable='spawner',
    arguments=['left_arm_controller', '--controller-manager', '/controller_manager']
)

# Right Arm Controller Spawner
right_arm_controller_spawner = Node(
    package='controller_manager',
    executable='spawner',
    arguments=['right_arm_controller', '--controller-manager', '/controller_manager']
)
```

Figure 1.11 controller_launch.py

This part shows how ROS2 nodes are made to launch the controllers and the broadcaster. It makes sure the robot knows which joints to control and how to control it.

B. ESP32 and Hardware Connection

The ESP32ControlNode which is part of the esp32_controller package, listens to the /joint_states topic. From there, it gets the latest joint angles. It then changes these angles into PWM signals based on the calibration. These signals are sent to the ESP32 over serial to control the motors.

One example of this conversion is shown in the code below:

```python
def position_to_pwm(self, position, joint):
    """
    Converts a joint position (in radians) into a PWM command.
    PWM = servo_center + (scale * position)
    """
    params = self.calibration.get(joint)
    pwm_value = int(round(params['servo_center'] + params['scale'] * position))
    # Clamp the PWM value between min and max values
    min_pwm, max_pwm = 100, 500
    return max(min_pwm, min(max_pwm, pwm_value))
```

Figure 1.12 PWM Conversion in ESP32ControlNode

This code snapshot shows how the node uses calibration values like servo center and scaling to change the joint angles into PWM signals. It also makes sure the values stay safe and do not go too far.

C. ESP32 PWM Calibration for Realistic Arm Control

To achieve proper and realistic arm movements on the physical robot, a calibration had to be conducted to convert ROS2 /joint_states topic joint angles into appropriate PWM values demanded by the servos. Transmitting raw simulated angles directly to the servos caused significant gaps between intended arm positions and movement. This was due to mechanical tolerances, servo motor non-linearities and gravitational on the arm.

The calibration process introduced two parameters for each joint:

- servo_center: the PWM signal value to place the real joint at 0 radians (neutral center position).
- scale: the increment in PWM signal for each radian of joint rotation.

The formula used to convert simulation angles to PWM signals was:

$$PWM = servo\_center + (scale \times \theta)$$

with θ being the joint angle in radians obtained from the /joint_states topic.

The following table summarizes the servo calibration settings used in the system:

Table 1.3 Servo Calibration settings

Joint Name	Servo Center (PWM)	Scale (PWM/rad)
left_upper_arm_x_joint	266	-160
left_upper_arm_y_joint	500	-180
left_forearm_z_joint	149	70
left_forearm_x_joint	330	-160
right_upper_arm_x_joint	400	160
right_upper_arm_y_joint	90	-180
right_forearm_z_joint	149	70
right_forearm_x_joint	400	160

These values were determined through manual calibration by aligning the physical arm movements with the RViz simulation at several key angles, including 0 rad, ±0.5 rad, and ±1.0 rad.

The calibration logic was implemented inside the ESP32ControlNode package in the following way.

The critical function for converting joint angles to PWM signals was:

```
def position_to_pwm(self, position, joint):
    params = self.calibration.get(joint)
    if params is None:
        min_pwm = 100
        max_pwm = 500
        min_position = -1.57
        max_position = 1.57
        pwm_value = int((position - min_position) / (max_position - min_position) * (max_pwm - min_pwm)
+ min_pwm)
        return max(min_pwm, min(max_pwm, pwm_value))
    pwm_value = int(round(params['servo_center'] + params['scale'] * position))
    min_pwm = 100
    max_pwm = 500
    return max(min_pwm, min(max_pwm, pwm_value))
```

Figure 1.13 position to pwm

This process, initially, checks if there is a calibration for the joint and then calculates the corresponding PWM value. It also caps the output PWM to stay within safe ranges (100-500).

Once calculated, the PWM is passed on via serial communication to the ESP32:

```
def send_to_esp32(self, channel, pwm_value):
    last_value = self.last_pwm.get(channel, None)
    if last_value is not None and abs(pwm_value - last_value) < self.command_threshold:
        return
    self.last_pwm[channel] = pwm_value
    command = f"{channel},{pwm_value}\n"
    try:
        self.ser.write(command.encode('utf-8'))
        self.get_logger().info(f"Sent to ESP32: {command.strip()}")
    except Exception as e:
        self.get_logger().error(f"Error sending command to ESP32: {e}")
```

Figure 1.14 send to esp32

This ensures that PWM values are updated only if the change is significant, which helps prevent unnecessary motor movement and extends servo life.

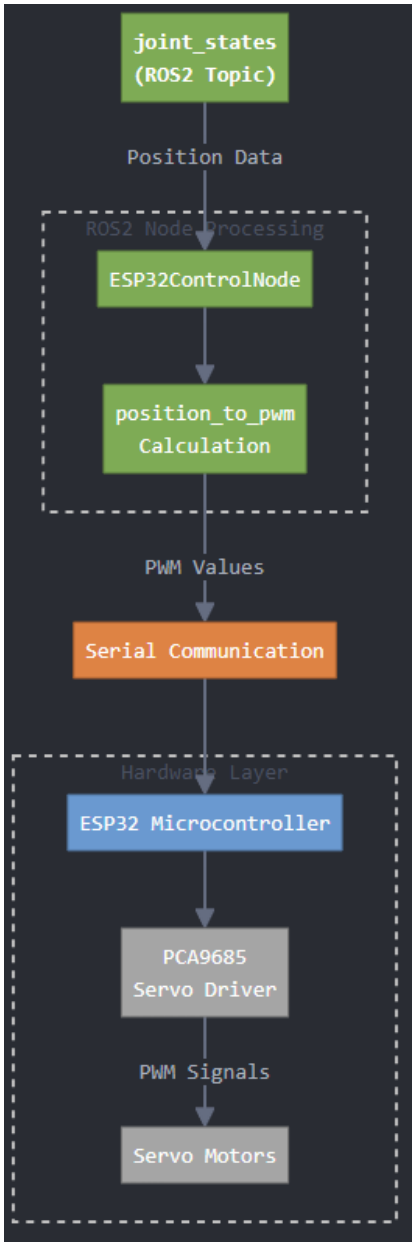

Figure 1.15 Block Diagram of the Arm PWM Calibration Flow

The above diagram illustrates the complete data and control flow for robotic arm movement using simulated joint angles. The flow begins from the /joint_states topic in ROS 2, which publishes all the available joints' current angles from the simulation. This is subscribed to by the ESP32ControlNode.

Inside the node, position_to_pwm employs a calibration formula to convert the simulated joint angle to its PWM equivalent. The translation ensures that all the servo motors turn to the correct physical position, putting the simulated pose as close as possible.

Once the PWM values are computed, they are passed from the ESP32 microcontroller via serial communications. ESP32 then acts as an interface between ROS and the

physical actuators. ESP32 receives the PWM and interfaces to the PCA9685 servo driver via an I2C interface. The PWM is then sent from the PCA9685 to the servo actuators and closes the control loop.

It separates simulation logic, data calibration and hardware control within this layered architecture to deliver dependable, scalable, and accurate motion across the robot's joints.

Several real-world challenges were encountered during the calibration phase. Gravity caused slight downward sag on the arm, especially at horizontal positions (0.5 rad to 1.0 rad), which made fine alignment difficult. Servo backlash also introduced small non-linearities; at low speeds, some joints did not react until a larger PWM shift was applied. Moreover, mechanical play in the forearm segments amplified small errors at the wrist and hand joints.

To minimize these issues, calibration was focused around the most commonly used motion ranges (0 to ±0.5 rad), where most gestures and movements happen.

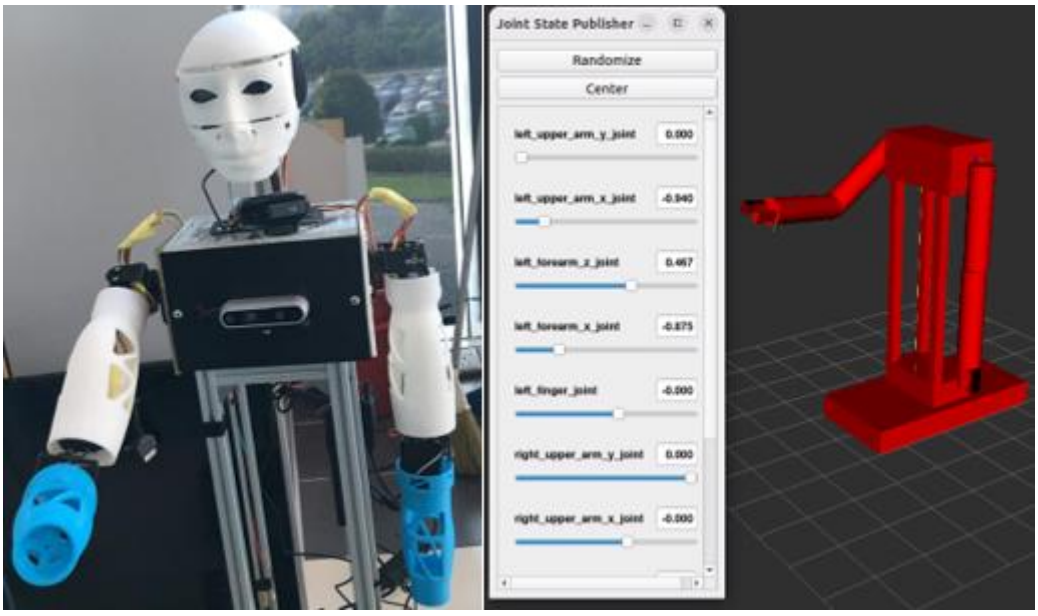

Figure 1.16 RViz pose vs real robot arm

Photos from RViz and the real robot were compared side-by-side during calibration to visually verify the accuracy of each joint across different poses.

D. GUI-Based Manual and Automated Control

The GUI is made using Tkinter package. It lets the user to control the arm by moving sliders or by running saved actions. When the user moves a slider or presses a button, the GUI sends joint angles through a temporary ROS node.

The function below is part of the GUI module. It shows how the joint angles from the sliders are sent out as JointState messages:

```
def publish_joint_state(self, joint):
    node = Node("manual_control")
    publisher = node.create_publisher(JointState, "/joint_states", 10)
    msg = JointState()
    msg.header.stamp = node.get_clock().now().to_msg()
    # Combines joint names for left and right arms
    msg.name = self.controller.ARM_JOINT_NAMES["left"] + self.controller.ARM_JOINT_NAMES["right"]
    msg.position = [self.joint_positions[j].get() for j in (self.controller.ARM_JOINT_NAMES["left"] +
self.controller.ARM_JOINT_NAMES["right"])]
    publisher.publish(msg)
    time.sleep(0.1)
    node.destroy_node()
```

Figure 1.17 Publishing Joint States from the GUI

This allows the arm to be controlled in real time based on the values the user sets in the GUI.

Chatbot, TTS, and ASR

The robot interaction system includes three main parts:

1) Chatbot: Uses Mistral API to understand and reply with natural language. It takes text input, keeps track of the conversation, and gives back structured output like talk, action, and nav2_pose.
2) TTS: Turns the chatbot's reply into voice using Nix-TTS.
3) ASR: Uses the Speech Recognition library to listen to speech and turn it into text then sends that to the chatbot.

All these parts use TCP sockets to send messages between each other. Also the chatbot use ROS2 topics for navigation.

a. Chatbot Module

The chatbot listens for TCP messages from other systems like ASR or vision. When it gets input, it logs it, sends it to Mistral API, and splits the reply into three parts: what to say, what action to do, and where to go.

Here's a part of the code that handles this:

```python
def handle_client(conn, addr):
    with tts_lock:
        if time.time() < ignore_input_until:
            log_to_file("Input ignored due to TTS suppression period.")
            conn.close()
            return

        data = conn.recv(4096)
        if not data:
            conn.close()
            return
        user_input = data.decode('utf-8').strip()
        log_to_file(f"Received input from {addr}: {user_input}")

        current_time = datetime.now().strftime("%H:%M")
        messages.append({"role": "user", "content": f"[Time: {current_time}] {user_input}"})

        response = client.chat.complete(model=model, messages=messages)
        reply = response.choices[0].message.content.strip()
        log_to_file(reply)
        messages.append({"role": "assistant", "content": reply})

        talk_text, action_text, nav2_pose_text = "", "", ""
        for line in reply.splitlines():
            if line.startswith("talk:"):
                talk_text = line.split("talk:", 1)[1].strip()
            elif line.startswith("action:"):
                action_text = line.split("action:", 1)[1].strip()
            elif line.startswith("nav2_pose:"):
                nav2_pose_text = line.split("nav2_pose:", 1)[1].strip()

        if not talk_text:
            talk_text = reply

        send_to_tts(talk_text)
        if action_text: send_action_to_gui(action_text)
        if nav2_pose_text: send_nav2_pose_to_ros2(nav2_pose_text)

        conn.sendall(reply.encode('utf-8'))
        conn.close()
```

Figure 1.18 part of Chatbot code

This function listens for messages, sends the input to the Mistral API, splits the response, and sends each part to where it needs to go either the TTS, GUI, or navigation.

b. TTS Module

The TTS part uses Nix-TTS to speak. It removes emojis, breaks the text into parts, and makes the audio using a voice model. Then it plays the sound using a speaker.

Here's the TTS function:

```python
def tts_play(text):
    text = remove_emoji(text)
    print("TTS Service: Speaking ->", text)
    c, c_length, phoneme = nix.tokenize(text)
    xw = nix.vocalize(c, c_length)
    sd.play(xw[0, 0], tts_sample_rate)
    sd.wait()
    print("TTS Service: Done speaking.")
```

Figure 1.19 TTS function

This function is the core of the TTS system. It runs every time the chatbot sends a reply to be spoken. The system also listens on TCP to receive messages from the chatbot and pass them to this function.

c. ASR Module

The ASR listens to the user through a mic then turns the voice into text using Google Speech API, and sends it to the chatbot.

Here's the main part of the ASR loop:

```
with mic as source:
    print("Listening...")
    audio = recognizer.listen(source)
    try:
        text = recognizer.recognize_google(audio)
        print("Recognized:", text)
    except sr.UnknownValueError:
        print("Could not understand audio, please try again.")
        continue
    except sr.RequestError as e:
        print(f"Speech recognition error: {e}")
        continue

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(text.encode('utf-8'))
    response = s.recv(4096)
    if response:
        print("Chatbot Response:", response.decode('utf-8'))
```

Figure 1.20 ASR loop

This part listens for the user's voice, turns it into text, and sends it to the chatbot using TCP. When the chatbot replies, it prints it out.

Final Notes on Integration

TCP Communication: The Chatbot, TTS, and ASR all use TCP to send messages. This keeps the system modular and easy to update.

ROS2 Integration: The chatbot can also send navigation goals using a ROS2 publisher (like Nav2PosePublisher). This lets the robot move after understanding the command.

1.4 Working Principle and Flowchart

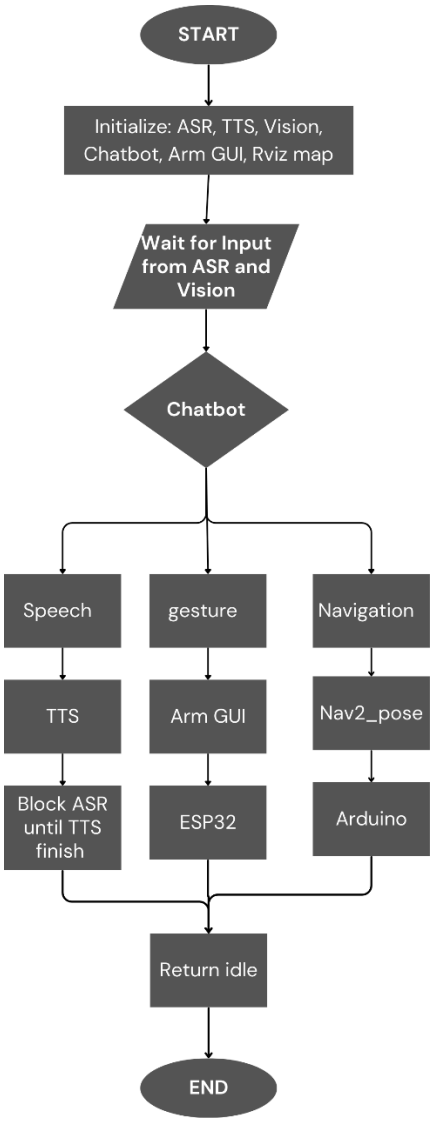

Figure 1.21 Flowchart of Robot Operation and Response Handling

The flowchart in Figure above shows the robot's main operating steps. It starts with initialization and continues through input handling, decision-making, and execution. The system supports different types of interaction. Each part works on its own, which makes the system easier to maintain and change.

START and Initialization

When the robot turns on, it starts all main modules. These include ASR for speech input, TTS for speech output, gesture detection using a vision system, the chatbot system, the Arm GUI for testing gestures, and the RViz map for navigation display. Each module runs on its own. The chatbot and ROS 2 handle coordination.

Waiting for Input

After startup, the robot begins listening. It watches two input sources:

- Audio from the microphone sent to the ASR system.
- Visual frames from the camera sent to the MediaPipe gesture recognizer.

This setup lets the robot handle both voice and gesture input at the same time.

Triggering the Chatbot

When the robot detects speech or a gesture, it sends the input to the Chatbot Operator System. This input is either a text transcript or a gesture label. The system sends it to the Mistral API. The API sends back a structured message with one or more of these fields:

- talk: text for speech output
- action: name of a gesture to perform
- nav2_pose: navigation target coordinates

Speech Execution (TTS Path)

If the talk: field is present, the text goes to the TTS engine. Before speaking, the system pauses the ASR. This prevents the robot from picking up its own voice. After speaking, the ASR resumes.

Gesture Execution (Arm Control Path)

If there is an action: field, the command goes to the Arm GUI and the ESP32 controller. The GUI helps with testing gestures by hand. The ESP32 sends signals to motors using the PCA9685 PWM driver. This setup works for both testing and live execution.

Navigation Execution (Nav2 Path)

If the chatbot gives a nav2_pose, the robot publishes it as a ROS 2 goal. The Arduino receives motor commands and moves the robot to the target.

Return to Idle

After finishing speech, gesture or navigation the robot goes back to idle. It waits for new input. No manual reset is needed.

System Design

The robot works in a loop with feedback. The system is modular, so new commands or actions can be added. Developers can change the chatbot prompt or expand the action handlers without changing the main control logic.