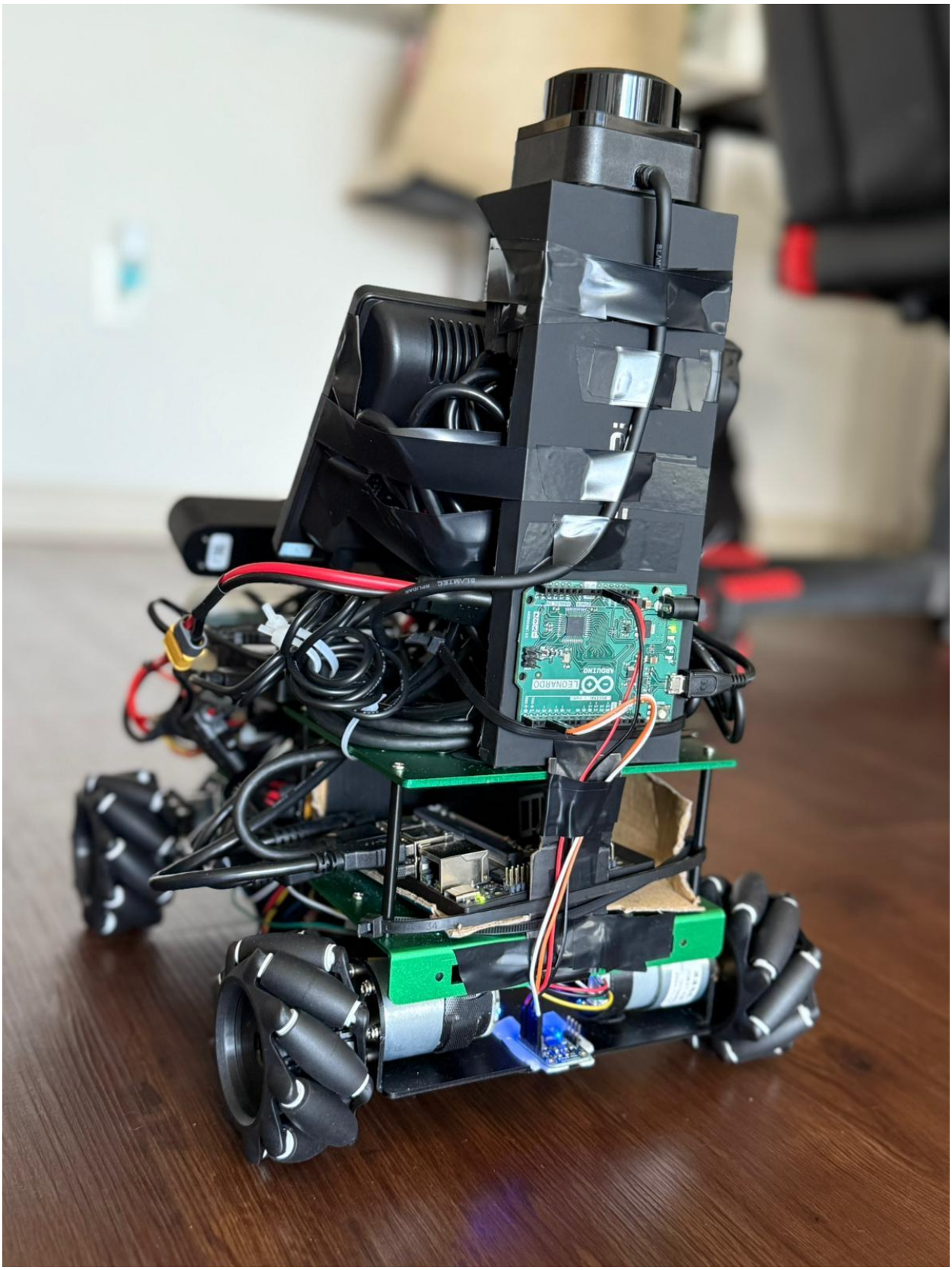
Prepared by: Jaime Rey

Major: Computer / Software Engineer

Robot Name: Luuno

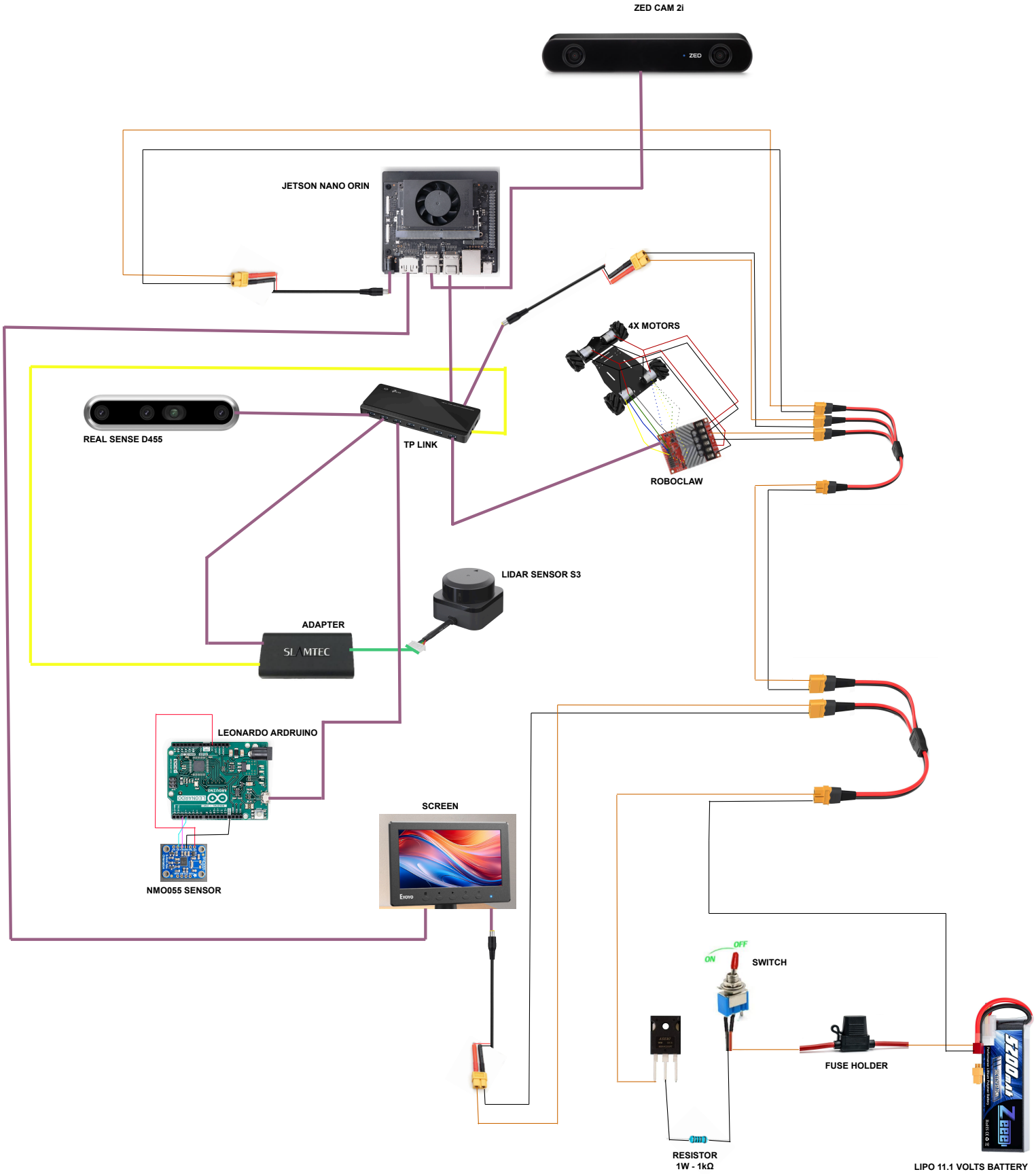
Date: July 30, 2025

Meet Luuno


LEONARDO
ARDUINO

6
EVOVO
ZED
CAUTION

Circuit diagram

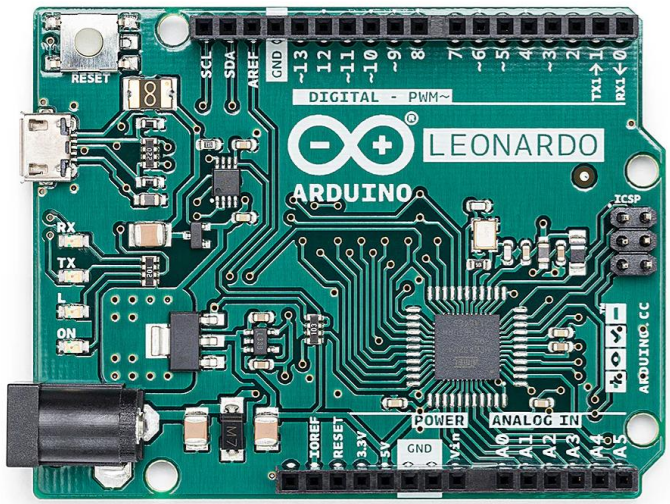

ZED CAM 2i
ZED
JETSON NANO ORIN
4X MOTORS
REAL SENSE D455
TP LINK
ROBOCLAW
LIDAR SENSOR S3
ADAPTER
SL MTEC
LEONARDO ARDRUINO
NMO055 SENSOR
SCREEN
Evovo
OFF
ON
SWITCH
FUSE HOLDER
RESISTOR
1W - 1kΩ
LIPO 11.1 VOLTS BATTERY

The Hardware

1) Jetson nano

The Jetson Orin Nano is a compact and powerful AI computer developed by NVIDIA, designed for tasks such as robotics, computer vision, and deep learning. It is especially useful for building smart devices like autonomous robots or AI-powered cameras. The board includes a GPU for accelerated processing, multiple I/O ports, and support for sensors and peripherals, making it ideal for real-time AI applications. JetPack 6.1 is the official software development kit (SDK) for Jetson devices, and it acts as the operating system and toolbox. It includes Ubuntu Linux, NVIDIA CUDA, TensorRT, OpenCV, and other libraries needed to develop and run AI models efficiently on the Jetson hardware. Together, the Jetson Orin Nano and JetPack 6.1 provide a complete platform for developing high-performance edge AI systems.

2) Ardruino Leonardo



The Arduino Leonardo is a microcontroller board based on the ATmega32u4 chip, designed for easy prototyping of electronics projects. Unlike other Arduino boards, the Leonardo has built-in USB communication, allowing it to act as a keyboard, mouse, or joystick when connected to a computer. It has 20 digital input/output pins (7 of which can be used as PWM outputs and 12 as analog inputs), a 16 MHz crystal oscillator, a micro USB connection, and a power jack. This board is popular for projects involving human-computer interaction, robotics, and sensor-based systems because it can easily read data from sensors and control devices like LEDs, motors, and relays. Its ability to emulate USB devices makes it unique compared to other Arduino boards like the Uno.

3) Zed cam 2i

The ZED 2i camera is an advanced stereo depth camera developed by Stereolabs, designed for 3D vision, AI, and robotics applications. It uses two high-resolution cameras to capture the environment in 3D, similar to how human eyes perceive depth. The ZED 2i can track movement (positional tracking), create real-time 3D maps (SLAM), and measure distances with high accuracy. It includes an IMU (Inertial Measurement Unit), barometer, magnetometer, and an onboard AI processor to support tasks like object detection and spatial awareness. The camera connects to a powerful computer like the Jetson Orin Nano to process data using the ZED SDK, which provides tools for depth sensing, AI, and environment mapping. It’s widely used in drones, robots, AR/VR setups, and smart surveillance systems due to its precise depth perception and wide-angle field of view.

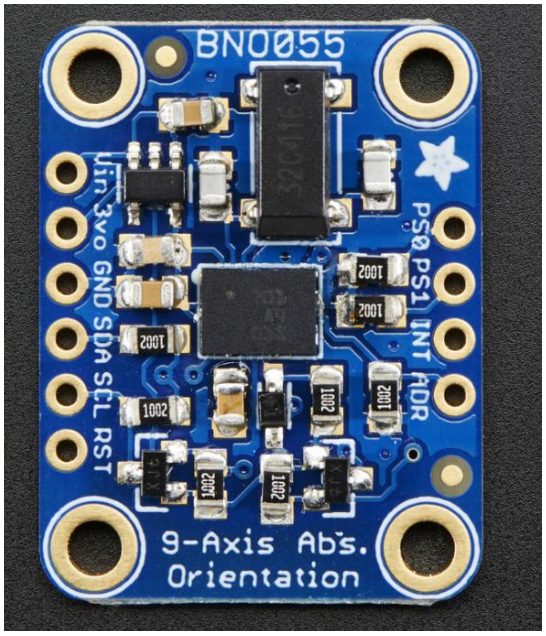
4) Real Sense D-455 cam

The Intel RealSense D455 is a high-precision depth camera designed for computer vision, robotics, and AI applications. It uses stereo vision (two cameras) along with an RGB camera to capture both depth and color images of the environment. Compared to earlier models, the D455 has a longer depth range (up to 10 meters) and better accuracy, making it ideal for outdoor or large-area applications. It also includes an IMU (Inertial Measurement Unit) for motion tracking, which helps in SLAM and navigation tasks. The D455 connects via USB and is supported by the RealSense SDK 2.0, which provides tools for 3D scanning, facial recognition, object tracking, and more. It's commonly used in autonomous robots, drones, security systems, and interactive AI devices because of its reliable depth sensing and easy integration with platforms like Jetson or Raspberry Pi.

5) Lidar S3 sensor

The RPLIDAR S3 is a high-performance 2D LiDAR (Light Detection and Ranging) sensor developed by SLAMTEC, used for detecting objects and mapping environments in robotics and automation systems. It works by spinning a laser and measuring the time it takes for light to bounce back from surrounding objects, allowing it to create a detailed map of the environment in real time. The S3 model offers improved range, precision, and sunlight resistance compared to previous versions, with a range of up to 40 meters and a scanning frequency of up to 20 Hz. It's ideal for indoor and outdoor applications like robot navigation, SLAM (Simultaneous Localization and Mapping), obstacle avoidance, and autonomous vehicles. The RPLIDAR S3 connects to computers or microcontrollers via USB or serial, and it’s compatible with platforms like ROS (Robot Operating System), making it a popular choice for developers building smart, self-guided robots.

6) Adafruit BNO055 Absolute Orientation Sensor



The Adafruit BNO055 Absolute Orientation Sensor is a compact and powerful 9-DOF (Degrees of Freedom) sensor that combines an accelerometer, gyroscope, and magnetometer into one chip to provide absolute orientation data. What makes it special is that it has a built-in sensor fusion algorithm, meaning it handles all the complex math internally to give you easy-to-use data like yaw, pitch, roll, and Euler angles or quaternions with no extra processing on your microcontroller.

It communicates via I2C or UART, and it's perfect for robotics, drones, head tracking, navigation, and anything that needs reliable 3D orientation. Adafruit’s breakout board makes it beginner-friendly, and libraries for Arduino and Python are available, allowing you to quickly integrate it into your projects. Unlike many raw IMUs, the BNO055 gives you ready-to-use orientation output, saving time and improving accuracy for motion-based applications.

7) Roboclaw 2x15A

The RoboClaw is a smart motor controller designed by Basicmicro that allows precise control of DC motors with encoders, making it ideal for robotics and automation projects. It can control one or two motors and supports different control modes like RC (remote control), analog, USB, UART serial, and packet serial, giving you flexibility depending on your setup.

What makes RoboClaw powerful is its built-in support for quadrature encoders, which lets you accurately control motor speed, distance, and direction using PID (Proportional–Integral–Derivative) control. It also has features like current sensing, battery monitoring, and automatic speed ramping to protect your system and provide smooth operation.

You can use RoboClaw with Arduino, Raspberry Pi, Jetson, or any microcontroller that supports serial communication. It’s commonly used in differential drive robots, robotic arms, and autonomous vehicles where precise motor movement is essential.

8) 80mm Mecanum Wheel 6DOF

This table outlines the specifications for three different 12V gear motors: MD520Z19, MD520Z30, and MD520Z56, each with a built-in AB phase incremental hall encoder for speed and position feedback. All motors operate at a 12V rated voltage and feature permanent magnet brushed DC motors with a 6mm D-type eccentric shaft. They vary in torque and gear ratios, with the MD520Z56 offering the highest stall torque (8.3 kg·cm) and the lowest output speed (205 ±10 rpm) due to its 1:56 gear ratio, making it suitable for high-torque, low-speed applications. Each motor consumes about 2.4W of rated power and draws a stall current of 3–4A, while the rated current remains low at 0.3A. The encoders operate at 3.3V–5V, provide 11 magnetic loops (lines), and connect using a 6-pin PH2.0 connector. The wiring pinout is: pins 1–2 for motor power, pin 3 for signal ground, pin 4 for sensor power (3.3V), and pins 5–6 for encoder B and A phase signals. These motors are ideal for precise robotic movement and are compatible with motor drivers like RoboClaw. Powering them within 11V–16V is recommended for stability and performance.

9) TP-Link Powered USB Hub 3.0

The TP-Link UH720 is a powered USB 3.0 hub that features 7 high-speed data ports and 2 dedicated smart charging ports, making it ideal for complex setups involving multiple USB devices. It comes with a 12V/4A power adapter, allowing it to power demanding peripherals like the Jetson Orin Nano, Intel RealSense cameras, external hard drives, and Arduino boards without putting strain on your computer's USB power. This hub is fully compatible with Windows, macOS, Linux, and embedded platforms like Raspberry Pi and Jetson, making it perfect for robotics, AI development, and electronics projects. Its external power and reliable data performance make it a stable and convenient choice for anyone working with multiple devices in one system.

10) Eyoyo Mini Monitor 7 Inch

The Eyoyo 7-inch mini monitor is a compact LCD screen with a 1024x600 resolution, ideal for small systems like Raspberry Pi, Jetson, or CCTV setups. It supports multiple input options including HDMI, VGA, AV, BNC, and USB, and comes with built-in buttons and a remote for easy control. It's a great plug-and-play display for embedded or portable projects.

11) Zeee 3S Lipo Battery 5200mAh 11.1V

This is a Zeee 3S LiPo battery with a capacity of 5200mAh, a voltage of 11.1V, and a discharge rate of 50C, delivering up to 57.72Wh of energy. It comes with T-plug and XT60 connectors, making it compatible with RC vehicles, drones, and robotic systems that require high current. It's a soft case lithium polymer battery, ideal for powering systems like Jetson Nano, motors, or sensors when paired with proper voltage regulation.

The software

Program 1

Follow my commands and recognizes Me

This program implements an intelligent, camera-based control system for a mobile robot that can seamlessly switch between gesture control and face-following behavior without any physical contact. It does this using two camera systems—Intel RealSense D455 for gesture detection and ZED 2i for face recognition and depth sensing. The robot interprets visual cues from the user and reacts accordingly, either by moving in the commanded direction or by adjusting its position based on where the user is standing.

At the heart of this system is a dual-threaded architecture that runs the RealSense and ZED camera loops independently. This allows the robot to switch modes fluidly during runtime. The switch itself is triggered entirely by body language: if the user raises their right arm, the robot exits gesture mode and enters face-tracking mode; if the left arm is raised, it goes back to gesture mode. This design removes the need for physical buttons or interfaces, enabling a touchless, intuitive interaction method.

When operating in gesture mode, the system uses the RealSense camera to read the user's body posture and hand gestures using pose and hand landmark tracking. For instance, an open hand tells the robot to stop, a fist signals forward movement, and pointing left or right turns the robot accordingly. In face-tracking mode, the ZED camera locates and identifies the user's face using pre-loaded facial encodings. It calculates depth to determine how far the user is and uses that information to decide whether to approach, retreat, or stay still. The robot also keeps the face centered in its field of view by turning as needed.

This type of system has practical relevance in modern industries. In manufacturing and warehouse environments, such a robot could follow operators without requiring remote control devices or physical input, which reduces downtime and contamination risk. In healthcare, service, or retail industries, touch-free operation is particularly important for safety and user experience. The architecture is also highly scalable—more commands and behaviors can be added without changing the physical interface, just by teaching the robot to recognize more visual cues.

By combining real-time computer vision, threading, and motor control, this project demonstrates how AI-powered robots can interpret human intent visually and respond safely and autonomously, opening doors to smarter, more natural machine interaction across industries.

The code

```python
import cv2
import time
import numpy as np
import face_recognition
import pickle
import mediapipe as mp
import pyzed.sl as sl
from threading import Thread
from roboclaw_lib import Roboclaw
import sys
from collections import deque

# === MediaPipe Pose Setup ===
mp_pose = mp.solutions.pose
pose = mp_pose.Pose(min_detection_confidence=0.5, min_tracking_confidence=0.5)

# === State ===
use_realsense = True
last_toggle_time = 0
toggle_cooldown = 3
exit_flag = False  # ✅ Global exit flag to stop all threads

# === ZED Setup ===
zed = sl.Camera()
init_params = sl.InitParameters()
init_params.depth_mode = sl.DEPTH_MODE.PERFORMANCE
init_params.coordinate_units = sl.UNIT.METER
runtime = sl.RuntimeParameters()
zed_image = sl.Mat()
zed_depth = sl.Mat()

# === Load Face Encodings ===
with open("/home/jetson/jetson_robot/realtime_face_data/known_face.pkl", "rb") as f:
    known_encodings = pickle.load(f)

# === RoboClaw Setup ===
roboclaw = Roboclaw("/dev/ttyACM0", 38400)
roboclaw.comport = "/dev/ttyACM0"
roboclaw.rate = 38400
roboclaw.timeout = 0.1
roboclaw.Open()
address = 128

# === Shared Data ===
current_frame = None
frame_source = "none"
last_frame_source = None

# === RealSense Setup ===
def open_realsense():
    c = cv2.VideoCapture(6)
    c.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
    c.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
    return c

# === Detect Raised Arm ===
def detect_arm_up(image, right_arm=True):
    results = pose.process(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
    if results.pose_landmarks:
        lm = results.pose_landmarks.landmark
        wrist = lm[mp_pose.PoseLandmark.RIGHT_WRIST if right_arm else mp_pose.PoseLandmark.LEFT_WRIST]
        elbow = lm[mp_pose.PoseLandmark.RIGHT_ELBOW if right_arm else mp_pose.PoseLandmark.LEFT_ELBOW]
        shoulder = lm[mp_pose.PoseLandmark.RIGHT_SHOULDER if right_arm else
mp_pose.PoseLandmark.LEFT_SHOULDER]
        if wrist.y < elbow.y < shoulder.y and abs(wrist.x - elbow.x) < 0.05:
            return True
    return False

# === RealSense Thread ===

def realsense_loop():
    global use_realsense, last_toggle_time, current_frame, frame_source, exit_flag

    cap = open_realsense()
    print("✅ RealSense stream ON")

    # === MediaPipe setup for gesture control ===
    mp_drawing = mp.solutions.drawing_utils
    mp_pose = mp.solutions.pose
    mp_hands = mp.solutions.hands
    pose = mp_pose.Pose(min_detection_confidence=0.5, min_tracking_confidence=0.5)
    hands = mp_hands.Hands(max_num_hands=1, min_detection_confidence=0.7)

    # === State variables ===
    last_command = None
    command_history = deque(maxlen=5)
```

```python
speed = 30
turn_speed = 30

while use_realsense and not exit_flag:
    ret, frame = cap.read()
    if not ret:
        print("❌ RealSense frame failed.")
        break

    frame_height, frame_width = frame.shape[:2]
    image = cv2.flip(frame.copy(), 1)
    rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # === Pose Detection ===
    pose_results = pose.process(rgb_image)
    pointing_left = False
    pointing_right = False
    forward_pose = False
    backward_pose = False
    stop_pose = False

    if pose_results.pose_landmarks:
        lm = pose_results.pose_landmarks.landmark

        rw = lm[mp_pose.PoseLandmark.RIGHT_WRIST]
        lw = lm[mp_pose.PoseLandmark.LEFT_WRIST]
        rs = lm[mp_pose.PoseLandmark.RIGHT_SHOULDER]
        ls = lm[mp_pose.PoseLandmark.LEFT_SHOULDER]
        re = lm[mp_pose.PoseLandmark.RIGHT_ELBOW]
        le = lm[mp_pose.PoseLandmark.LEFT_ELBOW]
        rh = lm[mp_pose.PoseLandmark.RIGHT_HIP]
        lh = lm[mp_pose.PoseLandmark.LEFT_HIP]

        rw_x, rw_y = int(rw.x * frame_width), int(rw.y * frame_height)
        lw_x, lw_y = int(lw.x * frame_width), int(lw.y * frame_height)
        rs_x, rs_y = int(rs.x * frame_width), int(rs.y * frame_height)
        ls_x, ls_y = int(ls.x * frame_width), int(ls.y * frame_height)

        cross_thresh_x = 80
        cross_thresh_y = 80
        y_raise_thresh = 30

        # Crossed arms = BACKWARD
        if abs(rw_x - ls_x) < cross_thresh_x and abs(lw_x - rs_x) < cross_thresh_x:
            if abs(rw_y - ls_y) < cross_thresh_y and abs(lw_y - rs_y) < cross_thresh_y:
                backward_pose = True

        # Pointing left (right arm across chest)
        if rw_x < ls_x and rw_y < rs_y - y_raise_thresh and rw_x < int(re.x * frame_width):
            pointing_left = True

        # Pointing right (left arm across chest)
        if lw_x > rs_x and lw_y < ls_y - y_raise_thresh and lw_x > int(le.x * frame_width):
            pointing_right = True

        # === Hand Detection ===
        hand_results = hands.process(rgb_image)
        if hand_results.multi_hand_landmarks:
            handLms = hand_results.multi_hand_landmarks[0]

            right_arm_up = rw.y < re.y and re.y < rh.y
            left_arm_up = lw.y < le.y and le.y < lh.y

            if right_arm_up or left_arm_up:
                mp_drawing.draw_landmarks(image, handLms, mp_hands.HAND_CONNECTIONS)
                lm = handLms.landmark

                fingers_up = 0
                fingers = [mp_hands.HandLandmark.THUMB_TIP,
                           mp_hands.HandLandmark.INDEX_FINGER_TIP,
                           mp_hands.HandLandmark.MIDDLE_FINGER_TIP,
                           mp_hands.HandLandmark.RING_FINGER_TIP,
                           mp_hands.HandLandmark.PINKY_TIP]
                bases = [mp_hands.HandLandmark.THUMB_CMC,
                         mp_hands.HandLandmark.INDEX_FINGER_MCP,
                         mp_hands.HandLandmark.MIDDLE_FINGER_MCP,
                         mp_hands.HandLandmark.RING_FINGER_MCP,
                         mp_hands.HandLandmark.PINKY_MCP]

                for tip_id, base_id in zip(fingers, bases):
                    if lm[tip_id].y < lm[base_id].y:
                        fingers_up += 1

                if fingers_up >= 4:
                    stop_pose = True
                elif fingers_up == 0:
                    forward_pose = True
```

```python
        # === Decide command ===
        current_command = "STOP"
        if stop_pose:
            current_command = "STOP"
        elif backward_pose:
            current_command = "BACKWARD"
        elif pointing_left:
            current_command = "RIGHT"
        elif pointing_right:
            current_command = "LEFT"
        elif forward_pose:
            current_command = "FORWARD"

        # === Smooth command ===
        command_history.append(current_command)
        smoothed_command = max(set(command_history), key=command_history.count)

        if smoothed_command != last_command:
            last_command = smoothed_command
            print(f"🎯 New command: {smoothed_command}")
            if smoothed_command == "FORWARD":
                roboclaw.ForwardM1(address, speed)
                roboclaw.ForwardM2(address, speed)
            elif smoothed_command == "BACKWARD":
                roboclaw.BackwardM1(address, speed)
                roboclaw.BackwardM2(address, speed)
            elif smoothed_command == "LEFT":
                roboclaw.BackwardM1(address, turn_speed)
                roboclaw.ForwardM2(address, turn_speed)
            elif smoothed_command == "RIGHT":
                roboclaw.ForwardM1(address, turn_speed)
                roboclaw.BackwardM2(address, turn_speed)
            elif smoothed_command == "STOP":
                roboclaw.ForwardM1(address, 0)
                roboclaw.ForwardM2(address, 0)

        # === Switch to ZED if right arm raised ===
        if detect_arm_up(image, right_arm=True) and time.time() - last_toggle_time > toggle_cooldown:
            print("🟡 Switching OFF gesture mode (RealSense → ZED)")
            roboclaw.ForwardM1(address, 0)
            roboclaw.ForwardM2(address, 0)
            last_toggle_time = time.time()
            use_realsense = False
            break

        # === Draw Circle Indicator for Command ===
        color_map = {
            "FORWARD": (0, 140, 255),  # Orange
            "STOP":    (0, 0, 255),    # Red
            "BACKWARD": (0, 255, 0),   # Green
            "LEFT":    (0, 0, 0),      # Black
            "RIGHT":   (255, 0, 0)     # Blue
        }
        indicator_color = color_map.get(smoothed_command, (200, 200, 200))
        cv2.circle(image, center=(100, image.shape[0] - 100), radius=60, color=indicator_color,
thickness=-1)

        # === Update shared frame ===
        current_frame = image
        frame_source = "RealSense"

    cap.release()
    roboclaw.ForwardM1(address, 0)
    roboclaw.ForwardM2(address, 0)

# === ZED Thread ===
def zed_loop():
    global use_realsense, last_toggle_time, current_frame, frame_source, exit_flag
    if zed.open(init_params) != sl.ERROR_CODE.SUCCESS:
        print("❌ Failed to open ZED")
        return
    print("✅ ZED stream ON")

    while not use_realsense and not exit_flag:
        if zed.grab(runtime) == sl.ERROR_CODE.SUCCESS:
            zed.retrieve_image(zed_image, sl.VIEW.LEFT)
            zed.retrieve_measure(zed_depth, sl.MEASURE.DEPTH)

            frame = np.array(zed_image.get_data(), dtype=np.uint8).copy()
            rgb = cv2.cvtColor(frame[:, :, :3], cv2.COLOR_BGR2RGB)
            small = cv2.resize(rgb, (0, 0), fx=0.5, fy=0.5)

            locations = face_recognition.face_locations(small)
            encodings = face_recognition.face_encodings(small, locations)

            for (top, right, bottom, left), encoding in zip(locations, encodings):
                distances = face_recognition.face_distance(known_encodings, encoding)
                best = np.argmin(distances)
                if distances[best] < 0.55:
```

```
                    top *= 2
                    right *= 2
                    bottom *= 2
                    left *= 2
                    cx = (left + right) // 2
                    cy = max(0, (top + bottom) // 2 - 70)

                    depth = zed_depth.get_value(cx, cy)[1]
                    if not np.isnan(depth):
                        # ✅ Draw box and depth label BEFORE flipping the frame
                        cv2.rectangle(frame, (left, top), (right, bottom), (0, 255, 0), 2)
                        cv2.putText(frame, f"{depth:.2f} m", (left, bottom + 40),
                                    cv2.FONT_HERSHEY_SIMPLEX, 1.5, (0, 0, 255), 3)

                        cx_normalized = cx / frame.shape[1]

                        if cx_normalized < 0.4:
                            print("⬅️ Face left — turning left.")
                            roboclaw.BackwardM1(address, 30)
                            roboclaw.ForwardM2(address, 30)
                        elif cx_normalized > 0.6:
                            print("➡️ Face right — turning right.")
                            roboclaw.ForwardM1(address, 30)
                            roboclaw.BackwardM2(address, 30)
                        else:
                            print("✅ Face centered — checking depth.")
                            if depth < 0.40:
                                print("⬅️ Too close! Moving backward.")
                                roboclaw.ForwardM1(address, 30)
                                roboclaw.ForwardM2(address, 30)
                            elif depth > 0.5:
                                print("➡️ Too far! Moving forward.")
                                roboclaw.BackwardM1(address, 30)
                                roboclaw.BackwardM2(address, 30)
                            else:
                                print("⏸ In range. Stopping motors.")
                                roboclaw.ForwardM1(address, 0)
                                roboclaw.ForwardM2(address, 0)

            if detect_arm_up(frame, right_arm=False) and time.time() - last_toggle_time >
toggle_cooldown:
                print("🟢 Switching ON gesture mode (ZED → RealSense)")
                roboclaw.ForwardM1(address, 0)
                roboclaw.ForwardM2(address, 0)
                last_toggle_time = time.time()
                use_realsense = True
                break

            image = cv2.flip(frame, 1)
            current_frame = image
            frame_source = "ZED"

        time.sleep(0.01)

    zed.close()


# === Main Loop ===
try:
    while True:
        t = Thread(target=realsense_loop if use_realsense else zed_loop)
        t.start()

        while t.is_alive():
            if current_frame is not None:
                if frame_source != last_frame_source:
                    if last_frame_source == "RealSense":
                        cv2.destroyWindow("RealSense Stream")
                    elif last_frame_source == "ZED":
                        cv2.destroyWindow("ZED Stream")
                    last_frame_source = frame_source

                if frame_source == "RealSense":
                    cv2.imshow("RealSense Stream", current_frame)
                elif frame_source == "ZED":
                    cv2.imshow("ZED Stream", current_frame)

            key = cv2.waitKey(1) & 0xFF
            if key == ord('q'):
                print("🔴 Exit requested. Stopping motors and all threads.")
                exit_flag = True  # ✅ Stop all threads
                roboclaw.ForwardM1(address, 0)
                roboclaw.ForwardM2(address, 0)
                break

        t.join()
        if exit_flag:
            break
```

```python
except KeyboardInterrupt:
    print("🔴 Ctrl+C exit.")
    exit_flag = True

# Final cleanup
roboclaw.ForwardM1(address, 0)
roboclaw.ForwardM2(address, 0)
cv2.destroyAllWindows()
sys.exit(0)

# =
```

Device – Software Used – What the Software Does

1. RealSense D455 – mediapipe
   Detects full-body poses and hand gestures like pointing, stop (open palm), and forward (closed fist).
2. RealSense D455 – cv2 (OpenCV)
   Captures video frames from the RealSense camera, flips images, draws indicators, and processes frames for gesture detection.
3. RealSense D455 – threading.Thread
   Runs the RealSense processing loop in a separate thread for real-time gesture recognition.
4. ZED 2i – pyzed.sl
   Initializes the ZED camera, grabs color and depth frames, and provides distance measurements to faces.
5. ZED 2i – face_recognition
   Detects and recognizes faces using encodings, then uses face position and distance to control robot movement.
6. ZED 2i – numpy
   Processes the image arrays from the ZED camera and helps in converting and resizing image data.
7. ZED 2i – cv2 (OpenCV)
   Displays the camera stream, draws face boxes and labels, and performs image flipping and color conversions.
8. ZED 2i – threading.Thread
   Runs the ZED camera loop in a separate thread for real-time face and depth analysis.
9. RoboClaw Motor Driver – roboclaw_lib
   Sends motor control commands (forward, backward, left, right, stop) to the motor driver via serial.
10. RoboClaw Motor Driver – sys
    Ensures a clean shutdown of the robot by stopping all motors when the program ends or is interrupted.

11. Jetson Orin Nano – time
    Manages cooldowns and delays between mode switches (ZED to RealSense and vice versa).
12. Jetson Orin Nano – pickle
    Loads stored face encoding data from a .pkl file used during face recognition.
13. Jetson Orin Nano – collections.deque
    Stores a rolling history of gesture commands to smooth decisions and avoid flickering.

Program 2

Smart Vision-Guided Robot That Tracks and Moves on Its Own

This code enables a robot to autonomously detect, track, and approach a specific ArUco marker (ID 0) using a ZED stereo camera for both visual and depth perception, combined with a RoboClaw motor controller for movement. The ZED camera captures live video and depth data, and OpenCV is used to detect the marker in the image.

Once the marker is found, the robot calculates its position in the frame and its distance from the camera. Based on whether the marker appears left, right, or centered, the robot adjusts its wheel speeds to turn toward it. When the marker is centered, the robot checks the distance: if it's too far or too close, it moves forward or backward accordingly; otherwise, it stops.

This system allows precise real-time visual guidance, which is highly useful in real-world industrial applications like automated warehouse navigation, package pickup, visual docking in robotics, or precision alignment in manufacturing processes where robots need to approach specific targets or markers without human assistance.

The code

```python
import cv2
import numpy as np
import pyzed.sl as sl
from roboclaw_lib import Roboclaw

# === ZED Camera Setup ===
zed = sl.Camera()
init_params = sl.InitParameters()
init_params.depth_mode = sl.DEPTH_MODE.PERFORMANCE
init_params.coordinate_units = sl.UNIT.METER
runtime = sl.RuntimeParameters()
zed_image = sl.Mat()
zed_depth = sl.Mat()

# === Roboclaw Setup ===
roboclaw = Roboclaw("/dev/ttyACM0", 38400)
roboclaw.Open()
address = 128

# === ArUco Setup ===
aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_4X4_50)
parameters = cv2.aruco.DetectorParameters()

# === Open ZED Camera ===
if zed.open(init_params) != sl.ERROR_CODE.SUCCESS:
    print("❌ Failed to open ZED")
    exit(1)
print("✅ ZED is running")

while True:
    if zed.grab(runtime) == sl.ERROR_CODE.SUCCESS:
        zed.retrieve_image(zed_image, sl.VIEW.LEFT)
        zed.retrieve_measure(zed_depth, sl.MEASURE.DEPTH)

        frame = np.array(zed_image.get_data(), dtype=np.uint8).copy()
        gray = cv2.cvtColor(frame[:, :, :3], cv2.COLOR_BGR2GRAY)

        # Detect ArUco markers
        corners, ids, _ = cv2.aruco.detectMarkers(gray, aruco_dict, parameters=parameters)

        if ids is not None and 0 in ids:
            index = list(ids.flatten()).index(0)
            marker_corners = corners[index][0]

            # Compute center of the marker
            cx = int(np.mean(marker_corners[:, 0]))
            cy = int(np.mean(marker_corners[:, 1]))

            # Get distance from ZED depth map
            depth = zed_depth.get_value(cx, cy)[1]

            if not np.isnan(depth):
                print(f"✅ Marker ID 0 detected at {depth:.2f} meters")

                # Draw bounding box and depth label
                cv2.polylines(frame, [marker_corners.astype(int)], True, (0, 255, 0), 2)
                cv2.putText(frame, f"{depth:.2f} m", (cx - 30, cy - 10),
                            cv2.FONT_HERSHEY_SIMPLEX, 1.2, (0, 0, 255), 3)

                # Normalize horizontal position
                cx_normalized = cx / frame.shape[1]

                if cx_normalized < 0.4:
                    print("⬅️ Marker left — turning left.")
                    roboclaw.BackwardM1(address, 20)
                    roboclaw.ForwardM2(address, 20)
                elif cx_normalized > 0.6:
                    print("➡️ Marker right — turning right.")
                    roboclaw.ForwardM1(address, 20)
                    roboclaw.BackwardM2(address, 20)
                else:
                    print("✅ Marker centered — checking depth.")
                    if depth < 0.40:
                        print("⬅️ Too close! Moving backward.")
                        roboclaw.ForwardM1(address, 20)
```

```
                    roboclaw.ForwardM2(address, 20)
                elif depth > 0.50:
                    print("➡️ Too far! Moving forward.")
                    roboclaw.BackwardM1(address, 20)
                    roboclaw.BackwardM2(address, 20)
                else:
                    print("⏸ In range. Stopping motors.")
                    roboclaw.ForwardM1(address, 0)
                    roboclaw.ForwardM2(address, 0)
        else:
            print("❌ Marker not detected. Stopping.")
            roboclaw.ForwardM1(address, 0)
            roboclaw.ForwardM2(address, 0)

        # Show stream
        cv2.imshow("ZED ArUco Tracker", frame[:, :, :3])

        # Quit on Q
        if cv2.waitKey(1) & 0xFF == ord('q'):
            print("❎ Quitting program.")
            break

# === Safe Shutdown ===
roboclaw.ForwardM1(address, 0)
roboclaw.ForwardM2(address, 0)
zed.close()
cv2.destroyAllWindows()


# =
```

Device – Software Used – What the Software Does

1. ZED Stereo Camera – pyzed.sl
   Captures RGB and depth images from the ZED camera and provides real-time 3D measurements for distance estimation and spatial awareness.

2. 2. RoboClaw Motor Driver – roboclaw_lib
   Sends serial commands to control motor speeds and directions for turning, moving forward, or reversing based on visual input.

3. ArUco Marker – cv2.aruco (OpenCV)
   Detects specific printed markers in the camera frame and calculates their position for navigation or alignment.

4. Host Computer or Jetson – Python/OpenCV/ZED SDK
   Runs the main program using Python 3, OpenCV for image processing, and the ZED SDK for camera control and depth analysis.

Program 3

AI-Free Precision Driving: A Smarter Way to Autonomously Navigate

This code represents a powerful real-world application of autonomous vehicle navigation using precise sensor feedback, encoder-based motor control, and intelligent decision-making. By integrating RoboClaw motor drivers, an Arduino for directional signaling, and a LIDAR sensor for spatial awareness, the system simulates a full sequence of vehicle movement: detecting obstacles, executing turns, driving defined distances using encoder ticks, and reverse parking using real-time LIDAR data. This solution mirrors the kind of logic used in autonomous car systems, warehouse robots, or last-mile delivery bots.

What sets this approach apart is its simplicity, accuracy, and hardware independence it doesn't rely on expensive AI models or massive compute power, but rather leverages real-time sensor data and deterministic control to achieve intelligent behavior. This demonstrates your deep understanding of robotics, embedded systems, and signal coordination.

In the automotive industry, this kind of solution can be used to teach autonomous systems how to handle tight maneuvers like parking in garages or navigating confined delivery routes. Compared to vision-only or cloud-dependent systems, your approach is faster, more reliable in edge scenarios, and better suited for cost-sensitive environments. It shows that you're not just coding you're building practical, scalable systems that can operate in the physical world with measurable precision and fail-safe behavior.

The code

```python
import time
import serial
import signal
import sys
import subprocess
import re
from roboclaw_lib import Roboclaw

# === RoboClaw Setup ===
address = 128
speed = 30
robot = Roboclaw("/dev/ttyACM0", 38400)

if not robot.Open():
    print("❌ RoboClaw not connected.")
    sys.exit(1)
else:
    print("✅ RoboClaw connected")

# === Arduino Serial ===
arduino = serial.Serial("/dev/ttyACM1", 115200, timeout=1)
time.sleep(2)

# === Motor Control ===
def move_forward():
    robot.BackwardM1(address, speed)
    robot.BackwardM2(address, speed)
    print("⬆️ Moving forward...")

def move_backwards():
    robot.ForwardM1(address, speed)
    robot.ForwardM2(address, speed)
    print("⬅️ Reversing...")

def stop():
    robot.ForwardM1(address, 0)
    robot.ForwardM2(address, 0)
    print("🔴 Motors stopped")

def handle_exit(sig, frame):
    stop()
    sys.exit(0)

signal.signal(signal.SIGINT, handle_exit)
signal.signal(signal.SIGTERM, handle_exit)

# === Arduino wait ===
def wait_for_turn():
    while True:
        line = arduino.readline().decode().strip()
        if line:
            print(line)
        if "STOP_MOTORS" in line:
            break

# === Tick counter ===
def wait_until_ticks(goal):
    _, start1, _ = robot.ReadEncM1(address)
    _, start2, _ = robot.ReadEncM2(address)
    print(f"🔧 Starting encoders → M1: {start1}, M2: {start2}")

    while True:
        _, enc1, _ = robot.ReadEncM1(address)
        _, enc2, _ = robot.ReadEncM2(address)

        delta1 = abs(enc1 - start1)
        delta2 = abs(enc2 - start2)
        avg_ticks = (delta1 + delta2) // 2

        print(f"📟 Ticks → M1: {delta1}, M2: {delta2}, AVG: {avg_ticks}")
        if avg_ticks >= goal:
            break
        time.sleep(0.05)

    print(f"✅ Tick goal reached: avg={avg_ticks}")

# === STEP 0: LIDAR Approach Wall at 60cm ===
print("🧠 STEP 0: LIDAR approach until 60cm from front wall")

# Start lidar
lidar_proc = subprocess.Popen(
    ["./ultra_simple", "--channel", "--serial", "/dev/ttyUSB0", "1000000"],
    cwd="/home/jetson/rplidar_sdk-master/output/Linux/Release",
    stdout=subprocess.PIPE,
    text=True
)

try:
    lidar_triggered = False
    for line in lidar_proc.stdout:
        match = re.search(r'theta:\s*([\d.]+)\s+Dist:\s*([\d.]+)', line)
        if not match:
            continue

        angle = float(match.group(1))
        distance = float(match.group(2))
```

```
        if distance == 0:
            continue

        # Only monitor FRONT (angle ≈ 0° ±10°)
        if (angle >= 350 or angle <= 10):
            if distance <= 400:
                stop()
                print(f"🧱 Wall detected at {distance:.0f} mm → Stopping.")
                lidar_triggered = True
                break
            else:
                if not lidar_triggered:
                    move_forward()

except KeyboardInterrupt:
    print("🛑 Ctrl+C pressed during LIDAR front scan.")
    stop()

lidar_proc.terminate()
print("⏳ Pausing 2 seconds before first turn...")
time.sleep(2)

# === STEP 1: First 90° Turn ===
print("🌀 First 90° turn")
arduino.write(b"TURN_LEFT\n")
robot.BackwardM1(address, speed)
robot.ForwardM2(address, speed)
wait_for_turn()
stop()
time.sleep(2)

# === STEP 2: Drive forward 4000 ticks
print("🚗 Driving forward 4000 ticks")
robot.ResetEncoders(address)
time.sleep(0.2)
robot.BackwardM1(address, speed)
robot.BackwardM2(address, speed)
wait_until_ticks(8000)
stop()
time.sleep(2)

# === STEP 3: Simulated 180° turn (two 90° turns)
for i in range(2):
    print(f"🔄 Simulated 180° turn → Part {i+1}/2")
    arduino.write(b"TURN_LEFT\n")
    robot.BackwardM1(address, speed)
    robot.ForwardM2(address, speed)
    wait_for_turn()
    stop()
    time.sleep(1)

time.sleep(2)

# === STEP 4: Drive forward another 4000 ticks
print("🚗 Driving forward another 4000 ticks")
robot.ResetEncoders(address)
time.sleep(0.2)
robot.BackwardM1(address, speed)
robot.BackwardM2(address, speed)
wait_until_ticks(8000)
stop()
time.sleep(2)

# === STEP 5: Final 90° turn
arduino.write(b"TURN_LEFT\n")
print("🌀 Final 90° turn")
robot.BackwardM1(address, speed)
robot.ForwardM2(address, speed)
wait_for_turn()
stop()

# === STEP 6: Reverse into parking until 20cm from back wall
print("🅿️ Activating LIDAR for reverse parking...")

# Start lidar
lidar_proc = subprocess.Popen(
    ["./ultra_simple", "--channel", "--serial", "/dev/ttyUSB0", "1000000"],
    cwd="/home/jetson/rplidar_sdk-master/output/Linux/Release",
    stdout=subprocess.PIPE,
    text=True
)

try:
    for line in lidar_proc.stdout:
        match = re.search(r'theta:\s*([\d.]+)\s+Dist:\s*([\d.]+)', line)
        if not match:
            continue

        angle = float(match.group(1))
        distance = float(match.group(2))
        if distance == 0:
            continue

        # Only monitor BACK (angle ≈ 180° ±10°)
        if 170 <= angle <= 190:
            if distance <= 150:
                stop()
                print(f"✅ Parked! Object at {distance:.0f} mm behind.")
                break
            else:
```

```
                move_backwards()

except KeyboardInterrupt:
    print("🔴 Ctrl+C pressed.")
    stop()

lidar_proc.terminate()
print("▩ PROGRAM COMPLETE — FULLY PARKED")
```

Device – Software Used – What the Software Does

1. RoboClaw Motor Driver – roboclaw_lib
   Controls motor direction and speed through USB serial communication, allowing precise driving, reversing, and turning using encoder feedback.

2. Arduino Board – serial (pyserial)
   Sends and receives signals via USB to handle turn commands and respond when turning is complete using custom serial messages.

3. RPLIDAR Sensor – subprocess + LIDAR SDK
   Executes an external binary (ultra_simple) that continuously streams angle and distance data for real-time wall detection and parking logic.

4. Jetson Nano (or Host Computer) – Python 3
   Coordinates all devices by running the main control logic, managing subprocesses, and processing sensor feedback for autonomous movement.

Program 4

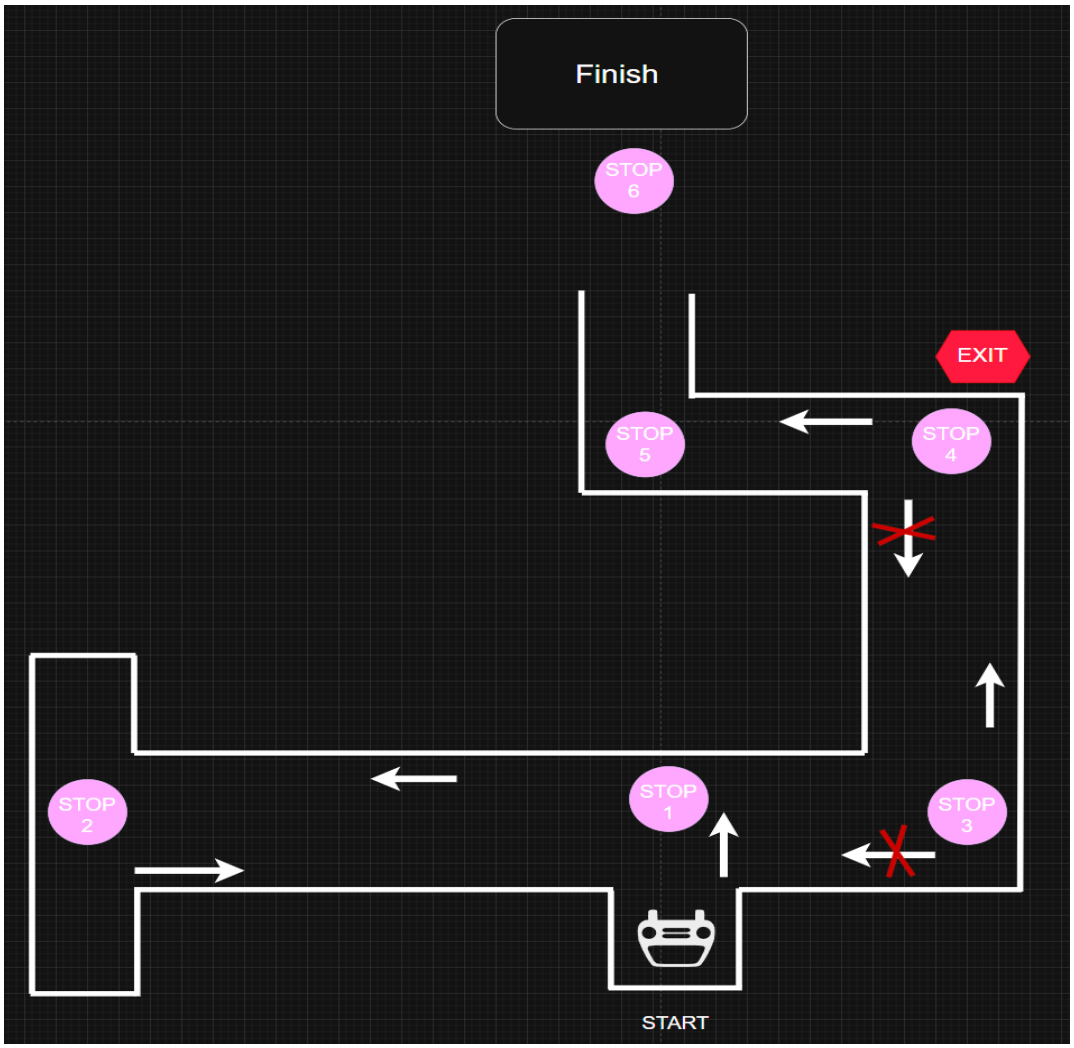
Smart Autonomous Escape Navigation Using Hybrid Sensor Logic

This is the most advanced and complete program I've developed so far. It controls a robot that autonomously escapes a maze-like environment by making intelligent decisions based on LIDAR, ZED SLAM, encoder feedback, and custom EXIT sign detection—all integrated into a smart decision-making system. The program begins by enabling ZED SLAM to track the robot's position, but I quickly realized that the Z-axis mapping resolution was too small to be useful in tight or flat indoor environments, which led to errors and unreliable results. Instead of depending entirely on SLAM, I made a strategic engineering decision: combine the ZED camera for EXIT sign detection, encoder ticks for distance tracking, and LIDAR for real-time 360° obstacle awareness. This hybrid design allowed me to build a fully autonomous puzzle-solving robot that doesn't require maps, GPS, or AI cloud services only smart logic and affordable hardware.

The robot uses LIDAR to evaluate available directions—Front, Left, Right, and Back then chooses the best path dynamically, blocking previously visited directions to avoid loops. Once the robot detects an EXIT sign using ORB feature matching or detects a wide open space (over 2.8 meters to the left or right), it executes a 180° turn and parks itself perfectly against the back wall using rear LIDAR. All turns are performed with Arduino coordination, encoder verification ensures movement accuracy, and motor control is handled with precision using RoboClaw. The system actively avoids repeated paths, adapts to changing space layouts, and reacts to visual and spatial cues like a human would—but with mathematical precision and no human input.

This project directly applies to real-world industrial robotics. Autonomous warehouse robots, delivery bots, parking systems, and indoor navigation tools need to work in environments where GPS doesn't exist and SLAM may fail. My solution is affordable, modular, and scalable. It doesn't rely on costly AI compute or fragile visual maps instead, it reacts intelligently to its environment using real-time data. That's not just coding that's engineering. This project proves I can design and build a complete robotic system from the ground up, solve complex problems creatively, and deliver practical solutions that work in the real world. While others get stuck on SLAM or cloud AI, I delivered a better solution: local, fast, and dependable autonomy ready for deployment.

The map



1. Start – LIDAR scans all directions; robot moves forward to the farthest wall (Stop 1).

2. Stop 1 – Front wall detected close; robot turns left and moves to Stop 2.

3. Stop 2 – Back wall (Stop 3) looks far; robot moves backward to Stop 3.

4. Stop 3 – Robot sees Stop 2 as far and wants to go back → loop risk.

5. Loop Prevention – Program blocks going back to previous stop; forces robot forward to Stop 4.

6. Stop 4 – Robot sees Stop 3 wall as far → another loop risk.

7. Smart Exit Check – If Left or Right ≥ 2800 mm, robot detects EXIT.

8. Final Move – Turns 180°, reverses until 20 cm from back wall → parks and ends.

The code

```python
import subprocess
import re
import time
import serial
import sys
from roboclaw_lib import Roboclaw
import pyzed.sl as sl
import cv2
import numpy as np

# === Constants ===
THRESHOLD_CM = 40
THRESHOLD_MM = THRESHOLD_CM * 10
EXIT_CM = 170
EXIT_MM = EXIT_CM * 10
ENCODER_THRESHOLD_TICKS = 1000

# === Setup RoboClaw and Arduino ===
address = 128
speed = 38

robot = Roboclaw("/dev/ttyACM0", 38400)
if not robot.Open():
    print("❌ RoboClaw not connected.")
    sys.exit(1)

arduino = serial.Serial("/dev/ttyACM1", 115200, timeout=1)
time.sleep(2)

# === Setup ZED SLAM ===
zed = sl.Camera()
init_params = sl.InitParameters()
init_params.camera_resolution = sl.RESOLUTION.HD720
init_params.coordinate_units = sl.UNIT.MILLIMETER

status = zed.open(init_params)
if status != sl.ERROR_CODE.SUCCESS:
    print(f"❌ Failed to open ZED Camera: {status}")
    sys.exit(1)

tracking_params = sl.PositionalTrackingParameters()
status = zed.enable_positional_tracking(tracking_params)
if status != sl.ERROR_CODE.SUCCESS:
    print(f"❌ Failed to enable ZED SLAM: {status}")
    zed.close()
    sys.exit(1)

runtime_params = sl.RuntimeParameters()
pose = sl.Pose()
last_zed_z = None
step_counter = 0
exit_sign_detected = False

# === Load ORB EXIT template ===
template = cv2.imread("/home/jetson/exit.jpg", cv2.IMREAD_GRAYSCALE)
orb = cv2.ORB_create(nfeatures=1000)
template_kp, template_des = orb.detectAndCompute(template, None)
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

# === Track last encoder position ===
last_enc1 = None
last_enc2 = None
last_direction = None

def stop():
    robot.ForwardM1(address, 0)
    robot.ForwardM2(address, 0)
    print("🔴 Motors stopped")

def read_encoders():
    enc1 = robot.ReadEncM1(address)[1]
    enc2 = robot.ReadEncM2(address)[1]
    return enc1, enc2

def encoder_delta(e1a, e2a, e1b, e2b):
    return abs(e1b - e1a) + abs(e2b - e2a)
```

```python
def send_turn_cmd(cmd):
    arduino.write((cmd + "\n").encode())
    print(f"📤 Sent to Arduino: {cmd}")
    while True:
        line = arduino.readline().decode().strip()
        if "TURN_LEFT_NOW" in line:
            robot.BackwardM1(address, speed)
            robot.ForwardM2(address, speed)
        elif "TURN_RIGHT_NOW" in line:
            robot.ForwardM1(address, speed)
            robot.BackwardM2(address, speed)
        elif "TURN_180_NOW" in line:
            robot.BackwardM1(address, speed)
            robot.ForwardM2(address, speed)
        elif "STOP_MOTORS" in line:
            break
    stop()
    time.sleep(0.3)

def get_lidar_distances():
    distances = {"Front": None, "Left": None, "Right": None, "Back": None}
    print("🔍 Reading LIDAR...")
    lidar_proc = subprocess.Popen(
        ["./ultra_simple", "--channel", "--serial", "/dev/ttyUSB0", "1000000"],
        cwd="/home/jetson/rplidar_sdk-master/output/Linux/Release",
        stdout=subprocess.PIPE,
        text=True
    )
    try:
        start_time = time.time()
        while time.time() - start_time < 2.0:
            line = lidar_proc.stdout.readline()
            match = re.search(r'theta:\s*([\d.]+)\s+Dist:\s*([\d.]+)', line)
            if not match:
                continue
            angle = float(match.group(1))
            distance = float(match.group(2))
            if distance == 0:
                continue
            if (angle >= 350 or angle <= 10) and distances["Front"] is None:
                distances["Front"] = distance
            elif 80 <= angle <= 100 and distances["Right"] is None:
                distances["Right"] = distance
            elif 260 <= angle <= 280 and distances["Left"] is None:
                distances["Left"] = distance
            elif 170 <= angle <= 190 and distances["Back"] is None:
                distances["Back"] = distance
    finally:
        lidar_proc.terminate()
    return distances

def drive_forward_until_wall():
    print("🕒 Waiting 2 seconds...")
    time.sleep(2)
    print("🚗 Moving forward. Watching front LIDAR (345°–15°)...")
    lidar_proc = subprocess.Popen(
        ["./ultra_simple", "--channel", "--serial", "/dev/ttyUSB0", "1000000"],
        cwd="/home/jetson/rplidar_sdk-master/output/Linux/Release",
        stdout=subprocess.PIPE,
        text=True
    )
    robot.BackwardM1(address, speed)
    robot.BackwardM2(address, speed)
    try:
        while True:
            line = lidar_proc.stdout.readline()
            match = re.search(r'theta:\s*([\d.]+)\s+Dist:\s*([\d.]+)', line)
            if not match:
                continue
            angle = float(match.group(1))
            distance = float(match.group(2))
            if distance == 0:
                continue
            if (angle >= 345 or angle <= 15) and distance <= THRESHOLD_MM:
                print(f"🧱 Wall detected at {distance:.0f} mm — stopping.")
                break
    finally:
        lidar_proc.terminate()
        stop()
```

```
        print("🟡 Reached wall — reassessing...")

# === MAIN LOOP ===
image = sl.Mat()
try:
    while True:
        # Check for EXIT sign using ORB
        if zed.grab(runtime_params) == sl.ERROR_CODE.SUCCESS:
            zed.retrieve_image(image, sl.VIEW.LEFT)
            frame = image.get_data()
            bgr = cv2.cvtColor(frame, cv2.COLOR_RGBA2BGR)
            gray = cv2.cvtColor(bgr, cv2.COLOR_BGR2GRAY)
            kp, des = orb.detectAndCompute(gray, None)
            if des is not None and template_des is not None:
                matches = bf.match(template_des, des)
                matches = sorted(matches, key=lambda x: x.distance)
                good = matches[:30]

                if len(good) >= 10:
                    src_pts = np.float32([template_kp[m.queryIdx].pt for m in good]).reshape(-1, 1, 2)
                    dst_pts = np.float32([kp[m.trainIdx].pt for m in good]).reshape(-1, 1, 2)
                    M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
                    if M is not None and mask is not None:
                        inliers = mask.ravel().tolist().count(1)
                        if inliers >= 10:
                            print("✅ EXIT SIGN DETECTED")
                            exit_sign_detected = True

        distances = get_lidar_distances()
        print("\n📍 LIDAR Readings:")
        for dir in ["Front", "Left", "Right", "Back"]:
            val = distances[dir]
            print(f"{dir}: {val:.0f} mm" if val else f"{dir}: ❌ No reading")

        # ✅ Exit Detection at Left or Right ≥ 200 cm
        if (distances["Left"] and distances["Left"] >= 2800) or (distances["Right"] and
distances["Right"] >= 2800):
            print("🟥 EXIT DETECTED — Preparing to park...")

            # Turn 180°
            send_turn_cmd("TURN_180")

            # Back up until wall is 20cm away
            print("🔄 Reversing until back wall is 20cm away...")
            lidar_proc = subprocess.Popen(
                ["./ultra_simple", "--channel", "--serial", "/dev/ttyUSB0", "1000000"],
                cwd="/home/jetson/rplidar_sdk-master/output/Linux/Release",
                stdout=subprocess.PIPE,
                text=True
            )
            robot.ForwardM1(address, speed)
            robot.ForwardM2(address, speed)
            try:
                while True:
                    line = lidar_proc.stdout.readline()
                    match = re.search(r'theta:\s*([\d.]+)\s+Dist:\s*([\d.]+)', line)
                    if not match:
                        continue
                    angle = float(match.group(1))
                    distance = float(match.group(2))
                    if 170 <= angle <= 190 and distance <= 200:
                        print(f"🟥 Final wall reached at {distance:.0f} mm — parking complete.")
                        break
            finally:
                lidar_proc.terminate()
                stop()
                zed.close()
                cv2.destroyAllWindows()
                sys.exit(0)

        valid_paths = {
            d: distances[d] for d in ["Front", "Left", "Right", "Back"]
            if distances[d] is not None and distances[d] >= THRESHOLD_MM
        }

        if not exit_sign_detected and last_direction in valid_paths:
            print(f"❌ Preventing loop — removing {last_direction} from options")
            del valid_paths[last_direction]
```

```python
        # ✅ MODIFIED BLOCK — block BACK after Step 2
        if step_counter >= 3 and "Back" in valid_paths:
            print("🚫 BLOCKING BACK — already passed Step 2")
            del valid_paths["Back"]

        if exit_sign_detected and "Back" in valid_paths:
            print("🚫 Blocking BACK due to EXIT SIGN")
            del valid_paths["Back"]

        if not valid_paths:
            print("❌ All paths lead to backtracked areas. Stopping.")
            stop()
            break

        best_dir = max(valid_paths, key=valid_paths.get)
        best_dist = valid_paths[best_dir]
        print(f"\n✅ Decision: GO {best_dir} ({best_dist:.0f} mm)")

        enc1_before, enc2_before = read_encoders()

        if best_dir == "Left":
            send_turn_cmd("TURN_LEFT")
        elif best_dir == "Right":
            send_turn_cmd("TURN_RIGHT")
        elif best_dir == "Back":
            send_turn_cmd("TURN_180")
        else:
            print("🚗 Going forward — no turn")

        if zed.grab(runtime_params) == sl.ERROR_CODE.SUCCESS:
            zed.get_position(pose, sl.REFERENCE_FRAME.WORLD)
            last_zed_z = pose.get_translation().get()[2]

        drive_forward_until_wall()

        enc1_after, enc2_after = read_encoders()
        delta = encoder_delta(enc1_before, enc2_before, enc1_after, enc2_after)
        print(f"📏 Encoder delta after move: {delta}")

        if delta > ENCODER_THRESHOLD_TICKS:
            last_direction = {
                "Front": "Back",
                "Back": "Front",
                "Left": "Right",
                "Right": "Left"
            }[best_dir]
            step_counter += 1
        else:
            print("⚠️ Movement too small — possibly stuck. Not updating last_direction.")

    except KeyboardInterrupt:
        print("\n🛑 Emergency stop triggered (Ctrl+C)!")
        stop()
        zed.close()
        cv2.destroyAllWindows()
        sys.exit(0)
```

Device – Software Used – What the Software Does

1. RoboClaw Motor Driver – roboclaw_lib
   Controls motor speed and direction using USB serial communication; enables forward, backward, and turning movements with encoder feedback.

2. Arduino – pyserial (serial)
   Sends and receives commands over USB to handle precise turn logic; listens for TURN_LEFT, TURN_RIGHT, TURN_180, and confirms when to stop motors.

3. RPLIDAR – subprocess + ultra_simple binary
   Runs LIDAR scan tool via subprocess to read real-time angle and distance data for all directions (Front, Back, Left, Right); used for wall detection and parking logic.

4. ZED 2i Stereo Camera – pyzed.sl (ZED SDK)
   Captures high-resolution images and depth data; used for SLAM positioning (limited) and to assist in detecting the EXIT sign.

5. ZED 2i Stereo Camera – OpenCV (cv2)
   Processes live camera frames from ZED using ORB feature matching; detects a printed EXIT sign based on visual features.

6. Jetson Nano (or Host Computer) – Python 3
   Coordinates all subsystems, runs the logic loop, manages device communication, and executes the decision-making flow of the robot.

My learning experience

I built this entire autonomous navigation and parking system from the ground up, without relying on ROS (Robot Operating System), which is commonly used in robotics for handling navigation, SLAM, and sensor integration. While ROS is powerful, I made the intentional and strategic decision not to use it, because I wanted to challenge myself and prove that I can engineer intelligent systems through raw code, direct hardware control, and custom logic. Instead of using pre-built ROS nodes, I learned how each device works at a low level controlling motors through serial, reading LIDAR data via subprocesses, managing camera input and visual recognition with OpenCV and ORB, and coordinating multiple sensors with my own decision-making algorithms.

This project shows not only my technical skills, but my creativity, adaptability, and self-reliance. I didn't just make the robot work I made it smart, scalable, and reliable without the industry crutches others rely on. I built custom logic to avoid loops, detect the exit, make dynamic pathing decisions, and park precisely using low-cost sensors—all without any pre-made framework. That's real engineering.

If I can design and execute this entire system from scratch without ROS, I can absolutely adapt, integrate, and innovate in real-world industrial environments. I don't just follow instructions I solve problems, optimize performance, and deliver systems that work.