Client Code (Python)
This code runs on a computer (or Raspberry Pi, etc.) that:

Captures video from an ESP32-CAM stream.

Uses a YOLO model to perform object detection.

Determines motion commands based on object positions.

Sends these commands over Wi-Fi to the ESP32 server controlling the USV (Unmanned Surface Vehicle).

1. Imports and Configuration
python
Copy
Edit
```
import socket
import cv2
import math
import time
import numpy as np
from ultralytics import YOLO
```
socket: Used to establish TCP connections to the ESP32 server.

cv2: OpenCV library for video processing (frame capture, drawing boxes, etc.).

math: Provides mathematical functions (like calculating distances).

time: For delays and timestamp calculations.

numpy: Useful for image data manipulation.

ultralytics.YOLO: Loads and uses the YOLO object detection model.

python
Copy
Edit
```
# ---------------- Configuration ----------------
ESP32_IP = "192.168.43.66"           # IP for sending commands
ESP32_PORT = 80
ESP32_CAM_URL = "http://192.168.43.165:81/stream"
```
ESP32_IP & ESP32_PORT: Define where to send commands. This is the server’s IP and port.

ESP32_CAM_URL: URL to access the ESP32-CAM’s video stream.

python
Copy
Edit
```
FRAME_WIDTH, FRAME_HEIGHT = 640, 480
FRAME_CENTER = (FRAME_WIDTH // 2, FRAME_HEIGHT // 2)
STOP_THRESHOLD = 100
FRAME_SKIP = 5  # Process every 5th frame
```
FRAME_WIDTH & FRAME_HEIGHT: Set the resolution for processing the video frames.

FRAME_CENTER: Calculates the midpoint of the frame; used to determine object deviation.

STOP_THRESHOLD: A distance threshold (in pixels) – if an object’s center is closer than this to the frame center, the vehicle stops.

FRAME_SKIP: Only every 5th frame is processed for performance reasons.

python

Copy
Edit

```
# Recyclable Waste Classes
RECYCLABLE_CLASSES = {
    "recyclable", "aluminum can", "cardboard", "glass bottle",
    "paper", "plastic bottle", "plastic bag", "tin", "zip plastic bag"
}
```

RECYCLABLE_CLASSES: A set of strings representing objects that are classified as recyclable. This helps decide which conveyor belt to trigger.

python
Copy
Edit

```
# ---------------- YOLO Model ----------------
model = YOLO("weights/best_model.pt")
```

YOLO Model Initialization: Loads a pre-trained YOLO model from a local file. This model will detect waste objects in the video frames.

2. Connection Function
python
Copy
Edit

```
def send_command(command):
    """Open a new connection to ESP32, send command, then close the
connection."""
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.settimeout(10)
            s.connect((ESP32_IP, ESP32_PORT))
            s.sendall(f"{command}\n".encode('utf-8'))
            print(f"📤 Sent Command: {command}")
            # Brief delay to allow the ESP32 to process the command before
closing
            time.sleep(1)
    except Exception as e:
        print(f"⚠️ Error sending command '{command}': {e}")
```

Purpose: Opens a TCP connection, sends a command string, and then closes the connection.

Socket creation and timeout: Sets a 10-second timeout so that if the connection hangs, it doesn’t block indefinitely.

Connection and send: Connects to the ESP32’s IP and port, sends the command followed by a newline.

Delay: Allows time for the server to process the command before the connection closes.

Error Handling: Catches exceptions and prints an error message.

3. Utility Functions
python
Copy
Edit

```
def calculate_distance(p1, p2):
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
```

calculate_distance: Uses the Euclidean distance formula to determine how far two points (e.g., the frame center and object center) are from each other.

python
Copy
Edit

```
def classify_waste(detected_class):
    """Determine if detected waste is recyclable or non-recyclable."""
```

```
    if detected_class.lower() in RECYCLABLE_CLASSES:
        return "Recyclable", "Recyclable Belt"
    else:
        return "Non-Recyclable", "Non-Recyclable Belt"
```

classify_waste: Checks if the detected class (converted to lowercase for consistency) is in the set of recyclable classes. It returns a tuple with:

A descriptive label (“Recyclable” or “Non-Recyclable”).

The command string for the corresponding conveyor belt.

4. Video Processing and Detection Loop

python
Copy
Edit

```
def video_detection_loop():
    """Main loop to capture video, run object detection, and send commands."""
    cap = cv2.VideoCapture(ESP32_CAM_URL)
    if not cap.isOpened():
        print("❌ Error: Cannot access ESP32-CAM")
        return
```

Video Capture Initialization: Opens the video stream from the ESP32-CAM.

Error Check: If the stream cannot be opened, an error is printed and the loop exits.

python
Copy
Edit

```
    frame_count = 0
    last_detected_class = None
    last_action_time = time.time()
```

frame_count: Keeps track of frames processed to implement skipping.

last_detected_class & last_action_time: Track the last waste type processed and time of the action to avoid repeatedly processing the same waste too frequently.

python
Copy
Edit

```
    while cap.isOpened():
        ret, frame = cap.read()
        if not ret:
            print("🔄 Reconnecting to camera...")
            cap.release()
            time.sleep(2)
            cap = cv2.VideoCapture(ESP32_CAM_URL)
            continue
```

Frame Read Loop: Continuously reads frames from the camera.

Reconnect Logic: If reading a frame fails (e.g., network issue), the code releases the stream, waits, and tries to reconnect.

python
Copy
Edit

```
        frame_count += 1
        if frame_count % FRAME_SKIP != 0:
            continue  # Skip frames for performance
```

Frame Skipping: Only process every 5th frame to reduce processing load.

python
Copy
Edit

```
        # Preprocess frame
        frame = cv2.resize(frame, (FRAME_WIDTH, FRAME_HEIGHT))
        movement_direction = "FORWARD"
        nearest_distance = float('inf')
        detected_class = None
```
Frame Resize: Ensures each frame is at the specified resolution.

Initial Settings: Sets a default movement direction ("FORWARD"), initializes the nearest distance as infinity, and resets the detected class.

python
Copy
Edit
```
        # Object detection using YOLO
        results = model.predict(frame, conf=0.5, imgsz=640, device="cpu")
```
YOLO Prediction: Runs object detection on the frame with a confidence threshold of 0.5. The image size and device (CPU) are specified.

python
Copy
Edit
```
        for result in results:
            for box in result.boxes:
                x1, y1, x2, y2 = map(int, box.xyxy[0])
                x_center = (x1 + x2) // 2
                y_center = (y1 + y2) // 2
                distance = calculate_distance(FRAME_CENTER, (x_center,
y_center))
```
Iterating Over Detections: For each detected object, extract the bounding box coordinates.

Center Calculation: Finds the center of the detected bounding box.

Distance Calculation: Computes how far the detected object is from the frame center.

python
Copy
Edit
```
                # Draw bounding box
                cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
```
Visual Feedback: Draws a green rectangle around the detected object on the video frame.

python
Copy
Edit
```
                if distance < nearest_distance:
                    nearest_distance = distance
                    detected_class = model.names[int(box.cls[0])]
                    deviation_x = x_center - FRAME_CENTER[0]
                    if abs(deviation_x) > 50:
                        movement_direction = "RIGHT" if deviation_x > 0 else
"LEFT"
                    else:
                        movement_direction = "FORWARD"
```
Finding the Closest Object: Only the nearest object (smallest distance) is considered.

Detection and Class Retrieval: The detected class name is extracted.

Directional Decision:

The deviation from the center (in x-axis) is used to decide if the vehicle should steer left or right.

If the deviation is small (less than 50 pixels), the vehicle continues forward.

python
Copy
Edit

```
        # When object is close enough, process waste detection
        if detected_class and nearest_distance < STOP_THRESHOLD:
            movement_direction = "STOP"
            detected_waste_type, belt_action = classify_waste(detected_class)
            # Only process if a new waste type is detected or enough time has
passed
            if detected_class != last_detected_class or time.time() -
last_action_time > 5:
                print(f"♻️ Processing {detected_waste_type} waste
({detected_class})")
                send_command("STOP")
                send_command(belt_action)
                time.sleep(3)  # Wait for sorting to complete
                send_command("FORWARD")
                last_detected_class = detected_class
                last_action_time = time.time()
```

Object Proximity Check: If an object is detected and its distance is below the threshold:

The command is changed to “STOP”.

The waste type is classified.

To avoid repeated commands, it checks if the waste type is new or if more than 5 seconds have passed.

It then sends a series of commands:

STOP: Halts the vehicle.

Belt Command: Activates the appropriate conveyor belt (recyclable or non-recyclable).

Delay: Waits 3 seconds for sorting.

FORWARD: Resumes motion.

Updates the last detected class and timestamp.

python
Copy
Edit

```
        else:
            # If no waste is detected or object is not close enough, send
movement commands
            send_command(movement_direction)
```

Movement Control: If no sorting action is needed, the command corresponding to the calculated direction is sent.

python
Copy
Edit

```
        # Overlay information on frame
        cv2.putText(frame, f"Move: {movement_direction}", (20, 40),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
        if detected_class:
            cv2.putText(frame, f"Detected: {detected_class}", (20, 80),
```

```
                        cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)
            cv2.putText(frame, f"Distance: {int(nearest_distance)}", (20, 120),
                        cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)
```
Display Overlays: Puts text on the video frame showing the current movement direction, detected class, and distance for easier debugging and monitoring.

python
Copy
Edit
```
        cv2.imshow("Waste Detection", frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
```
Video Display: The processed frame is shown in a window named “Waste Detection.”

Exit Condition: If the ‘q’ key is pressed, the loop breaks, ending the program.

python
Copy
Edit
```
    cap.release()
    cv2.destroyAllWindows()
```
Cleanup: Releases the video capture object and closes any OpenCV windows when the loop ends.

python
Copy
Edit
```
# ---------------- Main ----------------
if __name__ == '__main__':
    video_detection_loop()
```
Entry Point: When the script is executed directly, it starts the video detection loop.

Server Code (Arduino/C++)
This code runs on an ESP32-based system that:

Manages Wi-Fi and listens for commands on a server.

Controls motors for movement.

Operates conveyor belts for sorting recyclable vs. non-recyclable waste.

Supports both autonomous and manual modes (using Dabble for Bluetooth control).

1. Includes and Wi-Fi Setup
cpp
Copy
Edit
```
#include <Arduino.h>
#include <WiFi.h>
#include <WiFiClient.h>
#include <WiFiServer.h>
#define CUSTOM_SETTINGS
#define INCLUDE_GAMEPAD_MODULE
#include <DabbleESP32.h>
```
Arduino.h & WiFi Libraries: Provide core functionality for the ESP32 and Wi-Fi communication.

DabbleESP32: A library that supports gamepad input via Bluetooth for manual control.

CUSTOM_SETTINGS & INCLUDE_GAMEPAD_MODULE: Preprocessor definitions that customize the Dabble library behavior.

cpp
Copy
Edit
```cpp
// ===== WiFi Credentials =====
const char* ssid = "najil";
const char* password = "project123";
WiFiServer server(80);
```
WiFi Credentials: SSID and password for connecting to the local network.

WiFiServer: Sets up a TCP server on port 80 to receive commands from the client.

2. Motor and Conveyor Belt Pin Definitions
cpp
Copy
Edit
```cpp
// ===== Motor Driver Pins (L298N) =====
const int enableA = 5, motorLeft1 = 19, motorLeft2 = 21;
const int enableB = 18, motorRight1 = 22, motorRight2 = 23;
```
Motor Pins: Define the pins connected to the L298N motor driver for controlling the left and right motors.

cpp
Copy
Edit
```cpp
// ===== Conveyor Belt Pins (L298N) =====
const int enableBeltA = 33, beltLeft1 = 25, beltLeft2 = 26;
const int enableBeltB = 4, beltRight1 = 27, beltRight2 = 32;
```
Belt Pins: Define the pins for controlling two separate conveyor belts (one for recyclable and one for non-recyclable waste).

cpp
Copy
Edit
```cpp
bool processingWaste = false;
```
processingWaste: A flag to indicate if the system is currently processing waste (to prevent overlapping commands).

3. Mode and Motor State Management
cpp
Copy
Edit
```cpp
// ===== Mode Selection =====
enum Mode { AUTO_MODE, MANUAL_MODE };
Mode currentMode = AUTO_MODE;
```
Mode Enum: Defines two operating modes:

AUTO_MODE: Receives commands via Wi-Fi.

MANUAL_MODE: Uses Dabble’s gamepad input.

currentMode: Starts in AUTO_MODE by default.

cpp
Copy
Edit
```cpp
// ===== Motor State Tracking =====
bool motorsStopped = true;
```
motorsStopped: Tracks whether the motors are currently stopped, preventing redundant commands and logging.

4. Wi-Fi Reconnect Management
cpp
Copy
Edit

```
// ===== WiFi Reconnect Management =====
unsigned long lastWifiReconnectAttempt = 0;
const unsigned long wifiReconnectInterval = 10000; // 10 seconds
```
Wi-Fi Reconnect Variables: These variables help manage and space out Wi-Fi reconnection attempts if the connection is lost.

5. Motor Control Functions
Each of the following functions directly controls the motor outputs using digital writes to the designated pins:

cpp
Copy
Edit
```cpp
void moveForward() {
  digitalWrite(enableA, HIGH); digitalWrite(enableB, HIGH);
  digitalWrite(motorLeft1, HIGH); digitalWrite(motorLeft2, LOW);
  digitalWrite(motorRight1, HIGH); digitalWrite(motorRight2, LOW);
  if (motorsStopped) {
    Serial.println("🚗 Moving Forward");
    motorsStopped = false;
  }
}
```
moveForward: Activates both motors to move the vehicle forward.

Sets the motor enable pins HIGH.

Configures the left motor to spin one direction (left1 HIGH, left2 LOW) and the right motor similarly.

Logs the movement if the motors were previously stopped.

cpp
Copy
Edit
```cpp
void moveBackward() { ... }
```
moveBackward: Similar to moveForward but reverses motor directions to drive backward.

cpp
Copy
Edit
```cpp
void moveLeft() { ... }
void moveRight() { ... }
```
moveLeft & moveRight: These functions set the motors to rotate the vehicle left or right by activating one motor forward and the other in reverse.

cpp
Copy
Edit
```cpp
void stopMotors() {
  if (!motorsStopped) {
    digitalWrite(enableA, LOW); digitalWrite(enableB, LOW);
    Serial.println("🛑 Motors Stopped");
    motorsStopped = true;
  }
}
```
stopMotors: Shuts off motor power by setting the enable pins LOW and updates the motor state.

6. Conveyor Belt Control Functions
cpp
Copy
Edit
```cpp
void stopBelts() {
```

```
  digitalWrite(enableBeltA, LOW); digitalWrite(enableBeltB, LOW);
}
```
stopBelts: Disables both conveyor belts.

cpp
Copy
Edit
```cpp
void runRecyclableBelt() {
  stopMotors();
  processingWaste = true;
  Serial.println("♻ Activating Recyclable Belt");

  digitalWrite(enableBeltA, HIGH);
  digitalWrite(beltLeft1, HIGH); digitalWrite(beltLeft2, LOW);
  delay(3000);
  stopBelts();
  processingWaste = false;
  moveForward();
}
```
runRecyclableBelt:

Stops the vehicle before processing.

Sets the processing flag to avoid command interference.

Activates the recyclable belt (via enableBeltA and the left belt motor pins).

Uses a 3-second delay to allow the belt to operate.

Stops the belt, resets the flag, and resumes forward motion.

cpp
Copy
Edit
```cpp
void runNonRecyclableBelt() { ... }
```
runNonRecyclableBelt: Similar to runRecyclableBelt but operates the belt designated for non-recyclable waste.

7. Command Processing Function
cpp
Copy
Edit
```cpp
void processCommand(String command) {
  command.trim();
  Serial.println("📩 Received: " + command);
  if (processingWaste) return;
```
processCommand:

Trims whitespace from the received command.

Logs the received command.

Checks if the system is busy processing waste; if so, it ignores new commands.

cpp
Copy
Edit
```cpp
  if (command == "FORWARD") moveForward();
  else if (command == "BACKWARD") moveBackward();
  else if (command == "LEFT") moveLeft();
  else if (command == "RIGHT") moveRight();
  else if (command == "STOP") stopMotors();
  else if (command == "Recyclable Belt") runRecyclableBelt();
  else if (command == "Non-Recyclable Belt") runNonRecyclableBelt();
```

```
}
```
Command Matching: Compares the incoming string to expected commands and calls the corresponding function.

8. Wi-Fi Handling Function

cpp
Copy
Edit
```cpp
void checkWiFi() {
  if (WiFi.status() != WL_CONNECTED) {
    unsigned long currentMillis = millis();
    if (currentMillis - lastWifiReconnectAttempt >= wifiReconnectInterval) {
      Serial.println("🔄 Attempting WiFi reconnection...");
      WiFi.disconnect();
      WiFi.begin(ssid, password);
      lastWifiReconnectAttempt = currentMillis;
    }
  }
}
```
checkWiFi: Periodically checks the Wi-Fi status.

If disconnected and the reconnect interval has passed, it attempts to reconnect.

This ensures that the ESP32 maintains a stable connection for receiving commands.

9. Setup Function

cpp
Copy
Edit
```cpp
void setup() {
  Serial.begin(115200);
  Serial.println("\n🚀 Starting USV...");
```
Serial Initialization: Sets up serial communication for debugging.

Startup Message: Logs the starting message.

cpp
Copy
Edit
```cpp
  // Initialize Dabble (for manual mode)
  Dabble.begin("USV");
  Serial.println("✅ Dabble Initialized. Connect via Bluetooth");
```
Dabble Initialization: Starts the Dabble library for manual control using a Bluetooth gamepad.

cpp
Copy
Edit
```cpp
  // Initialize Motor and Belt Pins
  pinMode(motorLeft1, OUTPUT); pinMode(motorLeft2, OUTPUT);
  pinMode(motorRight1, OUTPUT); pinMode(motorRight2, OUTPUT);
  pinMode(enableA, OUTPUT); pinMode(enableB, OUTPUT);
  pinMode(beltLeft1, OUTPUT); pinMode(beltLeft2, OUTPUT);
  pinMode(beltRight1, OUTPUT); pinMode(beltRight2, OUTPUT);
  pinMode(enableBeltA, OUTPUT); pinMode(enableBeltB, OUTPUT);
```
Pin Setup: Configures each motor and belt control pin as an output.

cpp
Copy
Edit
```cpp
  stopMotors();
  stopBelts();
```
Initial State: Ensures that motors and belts are not active on startup.

cpp
Copy
Edit
```
  // Connect to WiFi
  Serial.print("🔄 Connecting to WiFi...");
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("\n✅ WiFi Connected!");
  Serial.println(WiFi.localIP());
```
Wi-Fi Connection: Attempts to connect to the network, printing progress until a connection is established. Once connected, it prints the assigned IP address.

cpp
Copy
Edit
```
  server.begin();
  Serial.println("✅ Server Started. Ready for commands!");
}
```
Server Start: Initializes the TCP server on port 80 and logs that it is ready to receive commands.

10. Main Loop

cpp
Copy
Edit
```
void loop() {
  Dabble.processInput();  // Process Dabble input for manual control
  checkWiFi();
```
Loop Start: Continuously processes manual (Dabble) inputs and checks the Wi-Fi connection.

cpp
Copy
Edit
```
  // Toggle Mode using Joystick Select (debounced)
  static unsigned long lastModeSwitch = 0;
  unsigned long currentMillis = millis();
  if (GamePad.isSelectPressed() && (currentMillis - lastModeSwitch > 500)) {
    lastModeSwitch = currentMillis;
    currentMode = (currentMode == AUTO_MODE) ? MANUAL_MODE : AUTO_MODE;
    Serial.print("🔄 Switched to ");
    Serial.println((currentMode == AUTO_MODE) ? "AUTO Mode" : "MANUAL Mode");
  }
```
Mode Toggle:

Uses the “Select” button on the gamepad to toggle between AUTO_MODE and MANUAL_MODE.

Uses debouncing (500 ms delay) to avoid rapid toggling.

Logs the current mode.

cpp
Copy
Edit
```
  if (currentMode == AUTO_MODE) {
    // Auto Mode: Process WiFi commands
    WiFiClient client = server.available();
    if (client) {
      Serial.println("📡 Client Connected");
```

```
      String command = "";
      while (client.connected() && client.available()) {
        char c = client.read();
        if (c == '\n') {
          processCommand(command);
          break;
        } else if (c != '\r') {
          command += c;
        }
      }
      client.stop();
      Serial.println("🔌 Client Disconnected");
    }
  } else {
    // Manual Mode: Use Dabble GamePad controls
    if (GamePad.isUpPressed()) moveForward();
    else if (GamePad.isDownPressed()) moveBackward();
    else if (GamePad.isRightPressed()) moveRight();
    else if (GamePad.isLeftPressed()) moveLeft();
    else stopMotors();

    if (GamePad.isTrianglePressed()) runRecyclableBelt();
    if (GamePad.isCirclePressed()) runNonRecyclableBelt();
  }
```
AUTO_MODE Operation:

Checks if a Wi-Fi client has connected.

Reads the incoming command character by character until a newline is found.

Calls processCommand with the complete command and then disconnects the client.

MANUAL_MODE Operation:

Reads gamepad inputs to directly control the movement.

Checks for directional button presses and corresponding belt commands.

cpp
Copy
Edit
```
  delay(10); // Prevent watchdog triggers
}
```
Final Delay: A short delay (10 ms) helps prevent the watchdog timer from triggering due to a busy loop.

Summary
Client Code:

Connects to an ESP32-CAM stream.

Uses YOLO to detect objects (waste).

Calculates the object’s position relative to the frame center.

Decides movement commands (left/right/forward/stop).

Sends commands to the server via TCP.

Processes waste sorting commands based on detection and distance thresholds.

Server Code:

Sets up Wi-Fi and a TCP server to receive commands.

Controls motors for vehicle movement.

Operates two conveyor belts for waste sorting.

Supports both autonomous (Wi-Fi command based) and manual (Bluetooth gamepad via Dabble) modes.

Includes reconnection logic for Wi-Fi and debouncing for mode switching