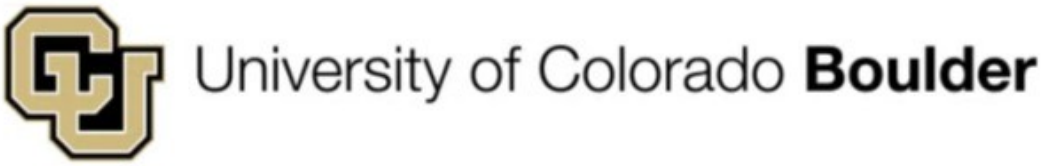

ECEN5623: FINAL PROJECT AUTOMATIC WASTE SEGREGATOR

Abhirath Koushik and Karthik R

May 3, 2025

Contents

Cover Page

1 Introduction

This project presents the design and implementation of an intelligent waste segregation system that leverages sensor integration and image processing to automate the classification and sorting of waste. The system initiates operation through an ultrasonic sensor, which detects the presence of an object on a motorized conveyor belt. Upon detection, a camera module is triggered to capture an image of the object, which is then processed using a trained image classification model to determine whether the waste is biodegradable or non-biodegradable. Based on the classification output, servo motors actuate to direct the object into the corresponding disposal bin. To enhance safety and reliability, a gas sensor is incorporated into the system to continuously monitor for the presence of smoke or harmful gases. If the sensor detects any hazardous condition, the conveyor and all motor operations are immediately halted to prevent damage or escalation. By combining real-time sensing, embedded control, and machine vision, this project offers a scalable solution for efficient and automated waste management with built-in safety mechanisms.

1.1 Github Link

RTES Final Project Repository

2 Design Overview

The waste segregation system is built around a modular architecture comprising four core layers: hardware, firmware, software and mechanical, all working in coordination under real-time constraints to ensure deterministic performance during object detection, classification, and actuation.

2.1 Hardware Components

- **Raspberry Pi 4B:** Acts as the central processing unit, managing sensor input, image processing, decision-making, and actuation.
- **Ultrasonic Sensor (USS):** Detects the presence of an object on the conveyor belt and serves as the initial trigger for activating the camera.
- **Camera:** Logitech C270 Webcam - Captures an image of the waste object upon detection by the USS.
- **DC Motors:** Conveyor belt control, moving the waste along the path for classification.
- **Servo Motors:** Direct waste into the appropriate bin based on classification output.
- **Gas Sensor:** MQ2 Sensor - Continuously monitors air quality. If smoke or gas is detected, it triggers an emergency stop sequence.

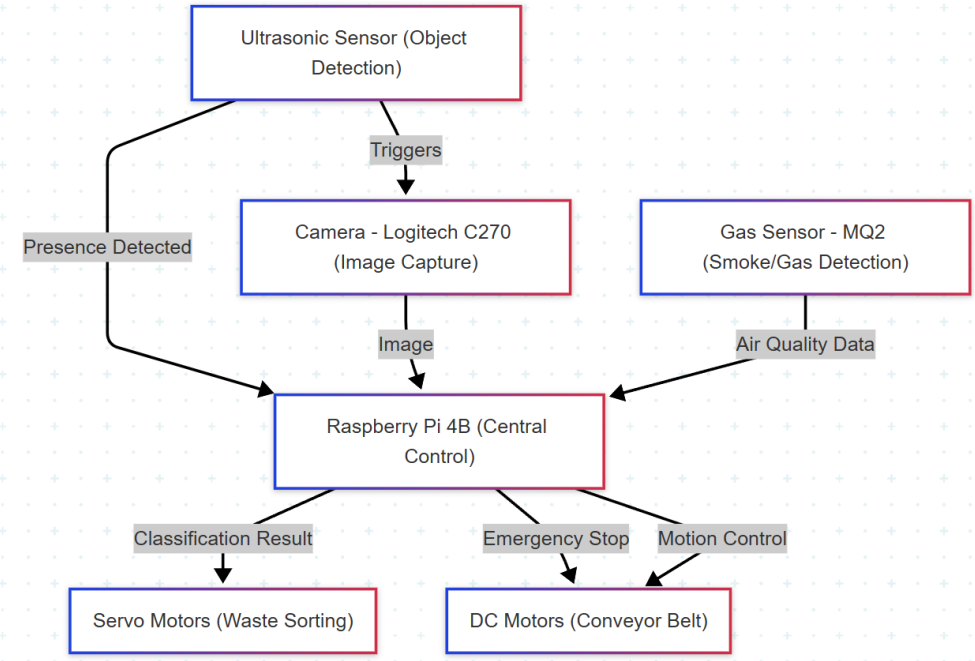

Figure 1: Hardware Block Diagram

2.2 Firmware Design

- Developed in C++ to support image processing.
- Handles low-level interaction with sensors and actuators.
- Implements interrupt-driven logic for real-time response to sensor events: The ultrasonic sensor is polled at regular intervals.Upon detection, an interrupt sets a flag to trigger the camera module.
- Gas sensor readings are sampled periodically through the scheduler. If the threshold is crossed, a non-maskable emergency interrupt halts the system (including the Conveyer Belt and Servos).

2.3 Software Design

- Image Processing Module (Python/OpenCV/TensorFlow Lite):
  - Captures and preprocesses the object image.
  - Passes the image to a trained ML model to classify waste.
  - Returns a binary decision: biodegradable or non-biodegradable.
- Decision Logic:
  - Maps classification output to GPIO controls for activating the corresponding servo motor.
  - Integrates feedback loops to confirm sorting success.
- Safety Module:
  - Continuously evaluates gas sensor input in the background.

– Overrides all subsystems in case of emergency detection, stopping DC motor (controlling the conveyer belt) and Servo Motors.

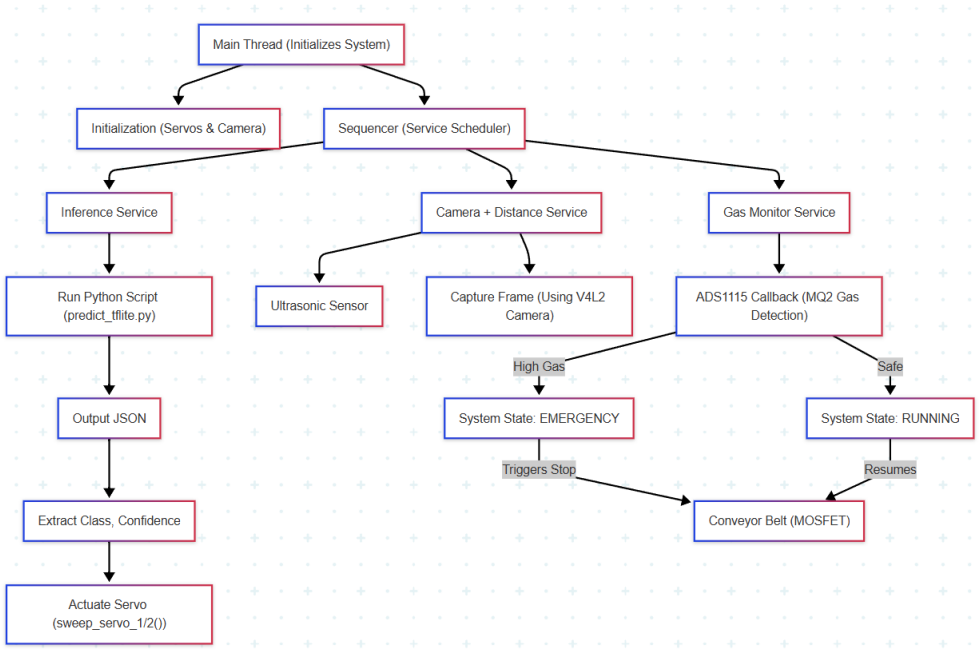

Figure 2: Software Block Diagram

2.4 Mechanical Implementation

- Constructed a waste seperator box along with a conveyor belt using wooden and plastic pipes.
- This involved implementation and learnings of mechanical world on how to counter act the friction and how to minimize the load maintaining the rotation of the belt.
- Learnt the requirements of drilling holes, and strategically planning to get a model that works.

2.5 High-Level Design Diagram

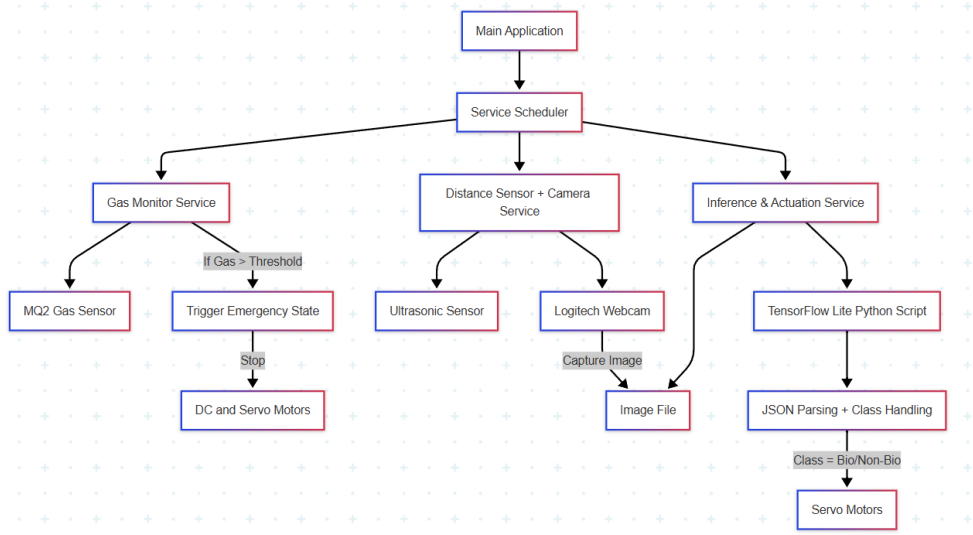

Figure 3: High-Level Design Diagram

- The 3 Services scheduled are Gas Monitor Service , Distance Sensor + Camera capture Service and Inference and Actuation Service.
- Gas Monitor Service independently monitors for the smoke levels. If there is any smoke level detected above the threshold, then it activates the Emergency State which stops the Overall system (including the DC motor controlling the conveyer belt and Servo motors)
- The Distance Thread has an Ultrasonic Sensor that keeps checking the distance values. If the distances decrease below a threshold, then it activates the Camera which captures a picture and saves it to the disk.
- The Inference and Actuation Thread uses this image from the disk to run it through a pre-trained Machine learning model and obtain the classification result (biodegradable or non-biodegradable).
- Based on the Inference thread result, a Servo motor moves the object from the conveyer belt into either sides of the setup, segregating the object into biodegradable and non-biodegradable.

2.6 Core Allotment

We ensure that Core-1 is used for both Gas Monitor Service and Distance & Camera Service, with the Gas Monitor thread as the Highest Priority. Core-2 is used for the Inference & Actuation Service with the Highest Priority on its core.

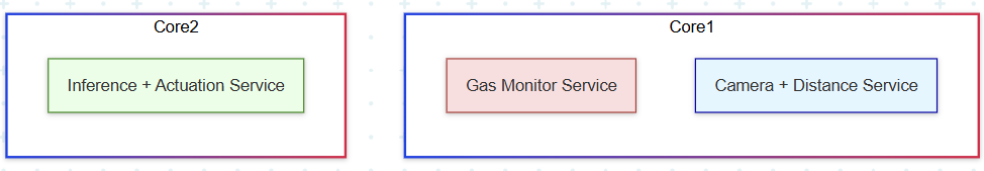

Figure 4: Core Allotment

2.7 Real-Time Considerations

The waste segregator system incorporates real-time processing to achieve:

1. Differentiation and segregation of biodegradable and non-biodegradable waste using a pre-trained ML model, actuated by DC and servo motors.
2. An emergency stop mechanism activated upon smoke detection, immediately halting the conveyor belt and servo motors.

Real-Time Requirements

1. **Deterministic Task Scheduling:**
   A real-time scheduler is used to define fixed periodic tasks such as ultrasonic polling and gas sensor sampling as well as dependent tasks such as classification decision windows. Each task has a well-defined execution budget and deadline, ensuring predictable system behavior.
2. **Event-Driven Interrupt Handling:**
   Sensor events like object detection from the USS and smoke alerts from the gas sensor are handled via GPIO interrupts with prioritization. The gas sensor thread is given the highest priority to preempt all other threads and safely shut down the conveyor and servos, preserving system integrity.
3. **Deadline Monitoring & Logging:** Each real-time task logs its worst-case execution time (WCET), deadline misses, and response latency. This aids in verifying the real-time behavior during both simulation and deployment.

3 System Requirements

Based on our Final Project Porposal, we were able to complete both the Minimum Requirement and Target Requirement objectives of our project. They are listed below.

3.1 Minimum Requirements (M)

- Real-time image processing to classify waste types (Biodegradable and Non-Biodegradable)
- Immediate Emergency Stop overriding all other tasks (using Gas Sensor)
- Real-time Actuation Control (Servo Motors)

3.2 Target Requirements (T)

- Real-time image processing to classify waste types (Biodegradable and Non-Biodegradable)
- Immediate Emergency Stop overriding all other tasks (using Gas Sensor)
- Real-time Actuation Control (Servo Motors)

4 Detailed Technical Topics

4.1 Services Descriptions

4.1.1 Gas Monitor Service

- **Purpose:** Monitors gas concentration via the MQ-7 sensor using the ADS1115 ADC.
- **Trigger:** Periodic, every 10ms.
- **Safety:** If gas voltage > 1.9V, sets system state to EMERGENCY and disables the output MOSFET to stop motor operation.
- **Restore Condition:** Gas voltage < 1.7V reverts system state to RUNNING.
- **Implementation:** Callback interface MQ7Callback in gas_service() using ADS1115rpi API.
- **Thread Affinity:** Core 1.

The gas monitoring service is responsible for detecting carbon monoxide (CO) levels using an MQ-7 sensor. The MQ-7 is an analog semiconductor gas sensor that requires a cyclical heating procedure for accurate CO measurements[**?**]. . According to the MQ-7 datasheet (and verified by experimental guides), the sensor's tin dioxide sensing element must be heated at 5V for 60 seconds (high-temperature cleaning phase) and then at 1.4V for 90 seconds (low-temperature sensing phase) in a repeating cycle [1] . During the low-temperature phase, the sensor's resistance changes in proportion to CO concentration, yielding a measurable analog voltage across a load resistor. The Pi 4 cannot directly drive the MQ-7's heater voltage or read analog values, so the design incorporates an ADS1115 16-bit ADC and a MOSFET driver circuit:

- **ADS1115 ADC (I^2C):** The MQ-7's analog output (voltage across the sensor's load resistor) is fed into an ADS1115 analog-to-digital converter. The ADS1115 connects to the Raspberry Pi via the I^2C bus and provides 16-bit resolution readings of the sensor voltage [2] . This allows the Pi to measure gas sensor analog values with high precision. The ADC can be configured with a suitable gain (PGA) to match the sensor's output range for CO levels of interest.
- **CO Level Measurement and Threshold:** The gas service periodically reads the ADC (e.g., at 1 Hz or appropriate intervals during the 90 s low-heat window) to obtain the sensor's voltage, which is converted to a resistance ratio and CO concentration (using calibration curves or formulas) [3] . A calibration routine can translate the ADC reading to an approximate CO ppm value by using the MQ-7's resistance vs. CO concentration characteristics [3].
- **MOSFET Shutdown Logic:** In normal operation, the MOSFET is toggled for heater control as described. In an emergency (CO alarm), the same MOSFET can be used to shut down parts of the system. The software can immediately turn the MOSFET off to remove power from the MQ-7 heater (eliminating a potential ignition source and preserving the sensor). Additionally, the system could be designed to use this MOSFET (or another GPIO-controlled power switch) to cut off power to non-critical loads or even the Pi itself after executing a safe shutdown sequence. For example, if the Pi is powered through a controlled MOSFET, the gas service could request a safe software shutdown of the Pi and then have an external microcontroller or circuit cut the main power via MOSFET.

  Operation: The gas monitoring thread runs with a defined period. It maintains a state machine for the heater: e.g., state = HIGH_HEAT for 60 s then state = LOW_HEAT for

90 s. The service reads the ADS1115 at a suitable rate and filters/averages the readings to reduce noise. It computes the CO concentration from the sensor resistance. If the concentration stays below the alarm threshold, the service simply logs the value or prints it for monitoring. If the threshold is exceeded, the service immediately sets an emergency flag and broadcasts an alert to the rest of the system. This triggers the safety responses: for instance, instructing the inference and camera services to stop further actions, moving servos to a safe position, and shutting off the heater MOSFET. The gas service is considered safety-critical, so it will preempt other activities if a dangerous condition is detected.

4.1.2 Capture Service

- **Purpose:** Purpose: Measures object proximity via HC-SR04 sensor. If an object is detected within 30 cm, captures an image using a USB/CSI camera.
- **Trigger:** Periodic, every 200ms.
- **Sensor Handling:** TRIG and ECHO GPIO pins to measure distance.
- **Camera:** Gas voltage < 1.0V reverts system state to RUNNING.
- **State Check:** Avoids duplicate capture if processing_in_progress is true.
- **Thread Affinity:** Core 1.

The camera + distance service handles object detection in front of the device by using an ultrasonic range sensor to trigger a camera capture. It effectively merges two functions: continuously measuring distance to any obstacle and capturing an image with the Pi Camera when an object is very close (within 20 cm).

- Ultrasonic Sensor (HC-SR04): The system uses an ultrasonic rangefinder to measure distance to the nearest object. The HC-SR04, for example, sends out an ultrasonic pulse and measures the time until the echo returns [5] . The service issues a trigger pulse (a 10 µs HIGH signal) on a GPIO output pin, which causes the sensor to emit a 40 kHz ultrasonic burst. Then it listens on an GPIO input pin for the echo pulse; the length of this echo pulse is proportional to the distance (the pulse stays HIGH until the echo is received). The Python code measures the duration of the echo pulse (using time.time() or hardware timers) and computes distance using the speed of sound (approximately 343 m/s) [5] The formula used is:

$$d = \frac{1}{2} \times t_{\text{echo}} \times 34300\,\text{cm/s}$$

- where t_{echo} is the round-trip time. A typical implementation might wait for the echo pin to go HIGH, record the timestamp, then wait for it to go LOW and record the timestamp again; the difference gives t_{echo}. In our system, the distance service performs this measurement periodically (e.g. every 100 ms, i.e. 10 Hz). If no echo is received within a timeout (e.g. 30 ms), it assumes no object in range (or out of max range 4–5 m).

- Distance Threshold and Camera Trigger: The service defines a threshold of 30 cm for triggering the camera. This threshold is chosen based on the use-case (perhaps the camera only needs to capture detailed images when an object is very close). On each distance measurement, the service checks the measured distance. If the distance is less than or equal to 0.2 m, the service will initiate a camera capture event. This could be done either in the same thread or by signaling a dedicated camera thread. In our design, we treat the camera capture as part of the same service for simplicity, but logically it can be a separate task triggered by the distance sensor task.

4.1.3 Capture Service

- **Purpose:** Purpose: Measures object proximity via HC-SR04 sensor. If an object is detected within 30 cm, captures an image using a USB/CSI camera.
- **Trigger:** Periodic, every 200ms.
- **Sensor Handling:** TRIG and ECHO GPIO pins to measure distance.
- **Camera:** Gas voltage < 1.0V reverts system state to RUNNING.
- **State Check:** Avoids duplicate capture if processing_in_progress is true.
- **Thread Affinity:** Core 1.

The Camera service is responsible for capturing images from the Raspberry Pi Camera Module when triggered. In the updated design, it uses a Persistent V4L2 Camera interface instead of OpenCV's VideoCapture. A custom class (referred to as PersistentV4L2Camera) opens the Video4Linux2 device (e.g. /dev/video0) once and maintains the camera stream, so frames can be grabbed on demand with minimal delay. This replaces the previous OpenCV VideoCapture approach, which had higher overhead (especially on Raspberry Pi's libcamera stack) and incurred re-initialization delays. By accessing the V4L2 API directly and persistently, we achieve better performance and lower latency in frame capture [6] . (Using OpenCV's high-level interface could add unnecessary layers or even use GStreamer internally, slowing down capture [6] .
The Camera service waits for the distance trigger from the sensor service. In the new design, it does not wait on a condition variable; instead, it periodically checks an atomic trigger flag (or uses C++20 atomic wait/notify if available) that the sensor service sets when an object is within 20 cm. Once the flag is detected, the camera thread immediately grabs a frame from the live camera feed. The persistent V4L2 connection ensures the camera is already warmed up and streaming, so capturing a frame is quick (on the order of the frame interval, e.g. 30 ms or less for 320x240 to 640x480 resolution). This is a significant improvement over the previous implementation where the camera might have been opened or reconfigured on each trigger, which could incur 100+ ms delays.

After capturing the image, the Camera service then clears or resets the trigger flag (to prevent repeated captures from one event) and uses another atomic flag to signal the Inference service that a new frame is available. The captured image frame is passed to the inference thread through a shared buffer or pointer. Because the synchronization is done via atomic flags, memory consistency is handled such that when frame_ready is set, the inference thread can safely read the frame data (the atomic operation acts as a release-acquire barrier ensuring the image memory is updated before the flag). The camera thread then returns to an idle wait state (polling the trigger flag). By using an atomic-based wait (busy-wait with brief sleep or the C++20 atomic_wait primitive), the camera thread can react to triggers with very low latency while still yielding CPU when idle. This design avoids the context-switch cost of a condvar signal, aligning with the need for real-time responsiveness. [7].
Additionally, the construction of the Camera service thread has been refined using a lambda function. The camera thread is spawned with a lambda that captures the camera device handle and the atomic flag references, encapsulating its behavior within a single defined routine. This modern C++ approach improves code clarity and avoids global variables or static thread entry functions – the lambda provides a clean closure with access to needed resources, simplifying thread initialization and service construction.

4.1.4 Inference Service

The Inference service handles real-time image processing using a pre-trained machine learning model. When the frame_ready flag is set by the Camera service, the inference thread wakes and obtains the new image frame for analysis. The system performs inference on the Raspberry Pi 4.

The model processes the frame to produce a classification result. The code changes have introduced detailed inference result logging: for each processed frame, the system now records the predicted class label, the confidence score, and the inference_time_ms.

These logs are written to a log file on the Pi's storage (or printed to a console) to allow later review of the system's decisions and performance. The inference service uses high-resolution timers to measure the model execution duration, then it packages the results into a log entry. The logging is implemented carefully to minimally impact the real-time performance: buffered I/O is used, appending text to a file, which will typically be cached in memory and flushed infrequently so that each log write is very fast. The inference thread performs the logging after the critical detection is done, and at its own priority, so this extra step does not interfere with sensor sampling or camera capture timing.

With the removal of condition variables, the Inference service no longer waits on a condvar for a frame – instead it checks the atomic frame_ready flag or uses an atomic wait. This polling/wait loop is efficient because frames are not arriving at a high rate continuously; the thread will be blocked (or yielding) most of the time until a trigger occurs. Once a frame is ready, the inference thread proceeds to run the model. During inference, the thread runs at a lower priority, meaning that if the sensor service needs CPU (for its next 860 Hz sample) or the camera thread needs to capture another frame, those will preempt the inference process. This design choice ensures that high-priority real-time tasks (sampling and capture) are never delayed by the compute-intensive inference. The inference itself is a soft real-time task – it should be as fast as possible on average, but a slight delay in obtaining the classification is acceptable compared to missing a sensor sample or a camera trigger.

Typical inference times on the Raspberry Pi 4 for modest models are on the order of tens of milliseconds. For instance, an object detection model like MobileNet SSD can run in 80 ms on the Pi 4's CPU (unaccelerated) [8] . Our system's measured inference_time_ms values are in this range (tens to low-hundreds of milliseconds depending on model complexity). These timing measurements are now part of the logged data. By logging inference_time_ms, we can continuously monitor the performance of the ML model on-device and ensure it stays within expected bounds. If the model execution time were to spike (due to CPU contention or anomalous inputs), it would be visible in the logs, aiding in performance tuning. The class and confidence logging also help evaluate the system's decision-making accuracy over time.

The primary additions are the logging and ensuring the inference thread's synchronization uses atomic flags exclusively. Overall, the Inference service remains an independent thread that runs the ML model and records the outcome, operating in a pipeline after the Camera service.

4.2 Rate Monotonic Scheduling and Timing Estimates

The system is structured as a set of periodic or sporadic tasks scheduled under a Rate Monotonic (RM) real-time scheduling policy. Each service described above corresponds to one main thread-/task with a fixed priority. We assign thread priorities according to their rate (frequency) and criticality, consistent with RM scheduling (higher frequency = higher priority for periodic tasks). Table 1 summarizes the updated tasks with their periods (or trigger rates), worst-case execution

times (WCET), and priority assignments:

Task	Period (T)	WCET (C)	Priority (1=Highest)
Gas Sensor	100ms	0.2 ms	1
Camera Capture	200ms	85ms	2

Figure 5: Enter Caption

In our design, we utilize the Rate Monotonic (RM) scheduling approach, which assigns higher priorities to tasks with shorter periods. RM scheduling provides a theoretical guarantee of schedulability if the system's total CPU utilization remains below a calculated Least Upper Bound (LUB). For a set of nn periodic real-time tasks, the RM schedulability criterion defined by Liu and Layland (1973) states:

Where U is the total CPU utilization computed as the sum of the execution time (CC) divided by the period (TT) for each task.

Initially, analyzing all tasks (Gas Monitoring, Camera Capture, and ML Inference) running simultaneously on a single CPU core yields a high utilization (¿100%), indicating potential infeasibility under strict RM analysis. However, recognizing the multi-core capability of the Raspberry Pi 4, we strategically assigned the computationally heavy inference task to a dedicated separate core. This isolates the inference task's heavy load, allowing RM analysis to focus exclusively on the high-frequency, time-critical tasks (Gas Monitoring and Camera Capture) running on a single core.

RM Feasibility with Inference Task on Separate Core

Considering only the Gas Monitoring and Camera Capture tasks, the updated utilization calculation is:

- Gas Sensor Task:

$$U_{\text{gas}} = \frac{0.2\,\text{ms}}{100\,\text{ms}} = 0.2\% \tag{1}$$

- Camera Capture Task:

$$U_{\text{camera}} = \frac{85\,\text{ms}}{200\,\text{ms}} = 42.5\% \tag{2}$$

- Total CPU Utilization (single-core):

$$U_{\text{total}} = 0.2\% + 42.5\% = 42.7\% \tag{3}$$

For two real-time tasks (n=2n=2), the RM LUB is calculated as:

$$U_{\text{LUB}} = 2\left(2^{\frac{1}{2}} - 1\right) \approx 82.84\% \tag{4}$$

At 42.7% total CPU utilization, the system is well within the RM feasibility threshold, comfortably meeting the schedulability criteria.

Cheddar Analysis

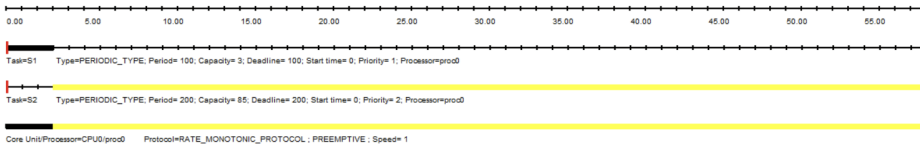

Figure 6: Cheddar Analysis

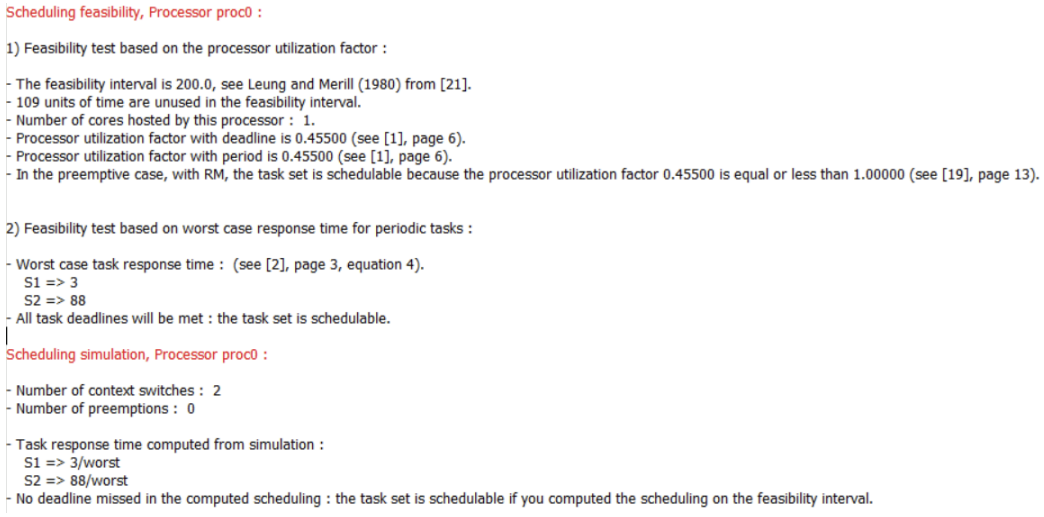

Figure 7: feasibilty test for RM

As you can see in the above images, we ran a Cheddar Analysis and the calculated values and the analysis are almost the same.

4.3 Safety Margin and Real-Time Constraints

With the updated configuration, we revisited the system's safety margins to ensure reliable real-time operation. Safety margin here refers to the buffer between the system's worst-case load and the point of overload or missed deadlines. Several changes have impacted this margin in different ways:

- **Faster Sensor Sampling:** Increasing the ADS1115 sampling rate from 8 Hz to 860 Hz markedly reduces the time between samples, which tightens the timing requirements on the sensor service. Previously, the system had 125 ms to comfortably read the sensor and trigger the camera; now it must do so every 1.16 ms. This reduces the margin for error or delay in the sensing loop. However, the actual sampling and processing are very fast (sub-millisecond), and our timing analysis shows the sensor thread still has slack in each period. We also mitigate the risk of overloading the CPU by the relatively low cost of each sample. The CPU utilization of the sensor task rose (to 17% of a core), but remains well below 100%, leaving room for other tasks.

- Tighter Camera Trigger (20 cm): Lowering the distance threshold means the system waits longer (object comes closer) before activating the camera. This can actually reduce the number of trigger events in scenarios where many objects pass by at 25 cm, for example – those would no longer trigger the camera. Thus, on average, the camera and inference

services will be activated less frequently, which increases the average computational slack. The system is focusing on nearer objects that are presumably more critical to capture. When an object does come within 20 cm, the expectation might be that it's an important event (for instance, something very close to a hazard). The system will then capture and process it with full attention.
Reaction time: Since the sensor reading is every 1.16 ms, the worst-case delay from an object crossing 20 cm to setting the trigger flag is <1.16 ms. The camera thread then responds. Thus, within a few milliseconds the camera will start capturing. Given a frame period (30 ms for 30 fps), the total time from threshold crossing to having an image is on the order of 30–40 ms. In summary, the change to 20 cm improves the computational safety margin (fewer triggers) while still meeting the real-time requirement of capturing important events.

- Task Sequencing and Priority Adjustments: The code changes included slight tweaks to thread sequencing and service construction (e.g., using a lambda for camera thread). These have minor implications on start-up order and how tasks coordinate their first iterations. By constructing the Camera service thread with a lambda after the sensor service is up, we ensure that by the time the camera thread starts checking the trigger flag, the sensor is already running and able to set that flag. This prevents any race condition at startup where the camera might miss a trigger because it wasn't ready. The priorities have been confirmed such that Sensor > Camera > Inference. We verified that no priority inversion can occur (no locks held by lower-priority threads that higher-priority ones need). The removal of condition variable means we removed one potential priority inversion scenario (a lower priority thread waking a higher one while holding a lock).

Conclusion of Safety Margin: After the updates, the system remains within schedulable limits and has adequate timing slack to accommodate occasional bursts of activity. By using efficient synchronization, focusing processing only on crucial events (20 cm triggers), and leveraging the performance of the Raspberry Pi 4, we maintain real-time performance with a margin for unforeseen delays. We will continuously use the logged data (especially inference timing and trigger frequency) to validate this margin during operation. If logs indicate the system running close to its limits (e.g., consistently high inference times or very frequent triggers), we can consider optimizations such as further lowering resolution, or adjusting priorities/affinities. So far, the design meets all deadlines with the changes implemented, and each change was made with careful attention to preserving or improving real-time guarantees.

4.4 Hardware/Software Architecture Diagrams

4.5 Hardware Architecture

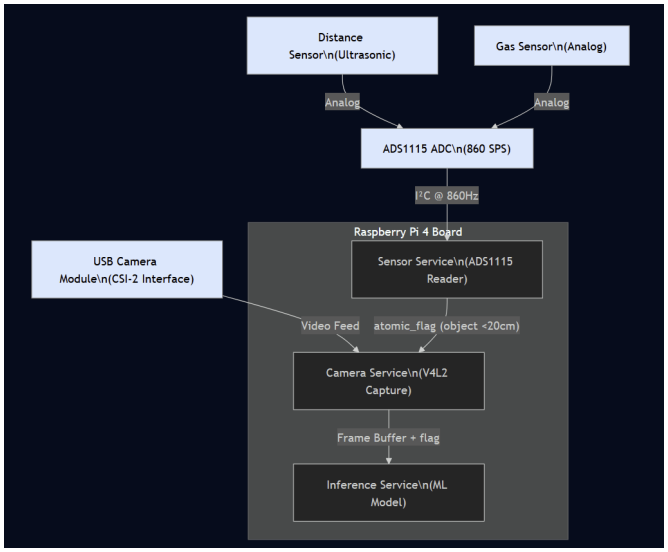

Figure 8: Hardware Architecture

4.6 Real-Time Thread Interaction (Sequence)

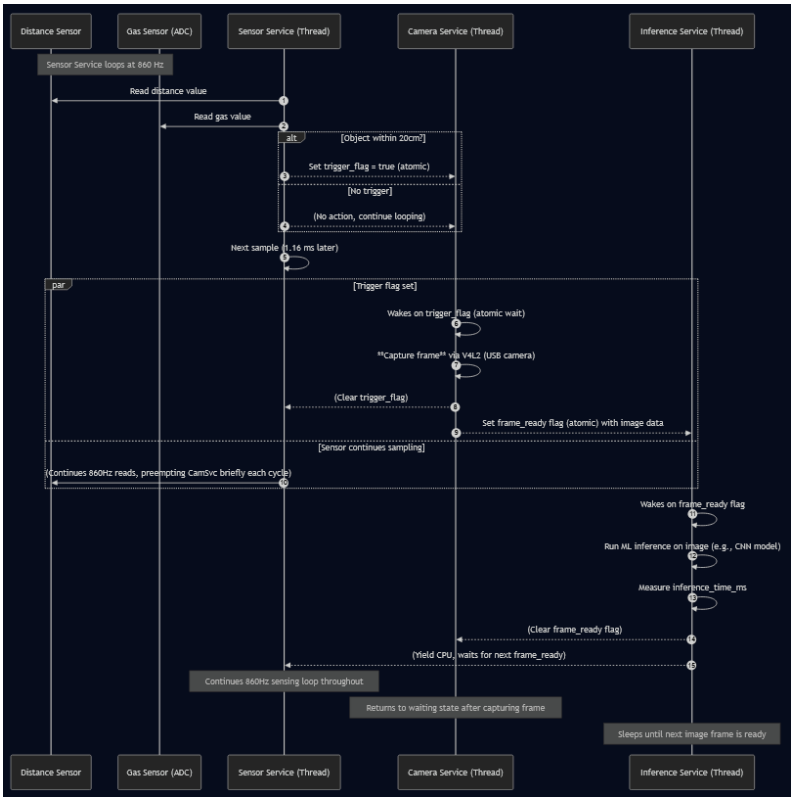

Figure 9: Real Time Sequence

These diagrams together provide a clear picture of the system's structure and behavior after the updates. The hardware/software diagram shows the static architecture and data pathways, and the sequence diagram shows the dynamic real-time operation and synchronization using atomic

flags. They have been updated to reflect the new threshold (20 cm in the SensorService logic), the use of the persistent V4L2 camera path.

4.7 Thread Design and Inter-Service Communication

Each service runs in its own thread, and all inter-service communication is achieved via shared atomic flags and shared data buffers, eliminating the need for complex locking or condition variables. Here we detail the thread functions and how they communicate with each other:

- Sensor Service Thread: This thread executes a tight loop to continuously read sensor data:

```
void gas_service() {
    static MQ7Callback cb;
    static ADS1115rpi reader;
    static bool initialized = false;

    if (!initialized) {
        ADS1115settings settings;
        settings.channel = ADS1115settings::AIN0;
        settings.pgaGain = ADS1115settings::FSR2_048;
        settings.samplingRate = ADS1115settings::FS860HZ;
        reader.registerCallback(&cb);
        reader.start(settings);
        initialized = true;
    }
```

Figure 10: Gas Service thread

The first thread ("Gas Monitor") executes every 100 milliseconds with high priority (99). It continuously monitors gas levels using an ADS1115 ADC through the gas_service() function. This thread initializes the ADC once with a callback mechanism (MQ7Callback) that evaluates the gas concentration. If the gas sample exceeds a threshold of 1.9V, the system transitions into an emergency state (SystemState::EMERGENCY) and activates a MOSFET to stop the operation (digitalWrite(MOSFET_WPI_PIN, HIGH)). If the gas level drops back below 1.7V, the system returns to normal operation, turning the MOSFET off.

- Camera Service Thread: The camera thread waits for the trigger_flag from the sensor. Without condition variables, we have two possible waiting strategies that we considered:

```cpp
void capture_frames(PersistentV4L2Camera& camera) {
    // auto start = std::chrono::steady_clock::now();

    if (processing_in_progress) return;
    float distance = measure_distance();
    std::cout << "Measured distance: " << distance << " cm\n";
    if (distance < 20.0) {
        if (camera.captureToFile(saved_image_path)) {
            frame_ready = true;
            processing_in_progress = true;
            std::cout << "Captured " << saved_image_path << "\n";
        } else {
            std::cerr << "Failed to capture frame\n";
        }
    }
    // std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    // processing_in_progress = false;
}
```

Figure 11: Camera capture thread

The second thread ("Camera + Distance") runs every 200 milliseconds at a slightly lower priority (98). It calls the capture_frames() function, which measures distance using an ultrasonic sensor (measure_distance()). If an object is detected within 20 cm, it triggers a camera capture using a PersistentV4L2Camera object and saves the captured image to "capture.jpg". It then signals that a new frame is ready for processing by setting atomic flags (frame_ready, processing_in_progress).

- Inference Thread: The third thread ("Inference") is scheduled every 300 milliseconds with priority 99. It invokes the inference_service() function, responsible for running image classification inference on the captured frame. It first checks whether a new frame is ready, executes a Python script (predict_tflite.py) via the command line, and parses the script output to identify the class of the detected object. Depending on the class ("biodegradable" or "non-biodegradable"), it actuates the corresponding servo (sweep_servo_1() or sweep_servo_2()) to physically separate waste items. The thread also measures and outputs the total inference duration in milliseconds for logging purposes.

```cpp
void inference_service() {
    if (!frame_ready) return;

    {
        std::lock_guard<std::mutex> lock(frame_mutex);
        frame_ready = false;
    }

    auto start = std::chrono::high_resolution_clock::now();
    std::string output = run_python_script(saved_image_path);
    if (!output.empty()) {
        std::string detected_class = extract_json_field(output, "class");
        std::string confidence = extract_json_field(output, "confidence");
        std::string inference_time = extract_json_field(output, "inference_time_ms");
        if (detected_class == "biodegradable") sweep_servo_1();
        else if (detected_class == "nonbiodegradable") sweep_servo_2();
        else std::cout << "Unknown detection result!\n";

        std::cout << "Detected Class   : " << detected_class << "\n";
        std::cout << "Confidence       : " << confidence << "\n";
        std::cout << "Inference Time   : " << inference_time << " ms\n";

        processing_in_progress = false;
    }
    auto end = std::chrono::high_resolution_clock::now();
    auto duration_ms = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
    std::cout << "Time taken for Inference: " << duration_ms << " ms\n";
}
```

Figure 12: Inference thread

Python code to understand how we trained the model:

- This step loads and prepares the input image to the correct size and format required by the TensorFlow Lite model (MobileNetV2 preprocessing).

```
# ---- Load image and preprocess ----
img = Image.open(IMAGE_PATH).convert('RGB')
img = img.resize(IMAGE_SIZE)
img = np.array(img).astype(np.float32)
img = img / 127.5 - 1.0  # MobileNetV2 preprocess_input
img = np.expand_dims(img, axis=0)
```

Figure 13: How the image is loaded into the model

- This part runs the TensorFlow Lite model to obtain predictions and measures inference execution time.

```
# ---- Run inference with timing ----
start_time = time.time()
interpreter.set_tensor(input_index, img)
interpreter.invoke()
preds = interpreter.get_tensor(output_index)[0]
end_time = time.time()

# ---- Output prediction ----
top = np.argmax(preds)
confidence = float(preds[top])  # <-- Convert to Python float
elapsed_time_ms = float((end_time - start_time) * 1000)  # <-- Also convert
```

Figure 14: How the inference is done

5 System Testing

5.1 Testing Plan

The objective of system testing was to ensure that all components of the waste classification and actuation system function correctly in isolation and in integration. Specific goals included:

- Validating individual sensor inputs (ultrasonic and gas detection).
- Ensuring image capture and TensorFlow Lite inference work under real-time conditions.
- Verifying correct servo actuation based on classified output.
- Confirming that multithreaded execution (using a sequencer) maintains data consistency and timing correctness.

5.2 Methodology

- **Unit Testing**: Each module was tested individually (sensor reading, camera frame capture, Python inference, and servo control) using separate testing codes.
- **Integration Testing**: The full system was operated under different input conditions (gas levels, object distances, object types) to evaluate end-to-end functionality. We tested with varying object distances as well as different biodegradable objects (banana, orange peels, apple, pizza slices) as well as non-biodegradable objects (soda tin can, plastic bowl).
- **Real-Time Testing**: Timing delays between object detection and corresponding servo actuation were measured.

5.3 Testing Results and Analysis

Component	Test Scenario	Expected Outcome	Actual Result
Ultrasonic Sensor	Object placed at varying distances	Accurate distance measurements	Passed
Gas Sensor (MQ7 + ADS)	Simulated high and low CO levels, tested by blowing air and lighting an incense stick	Emergency mode activation and recovery	Passed
Camera Capture	Trigged when object less than 25cm.	Capture and save the picture. Overwrite the previous picture.	Passed
Python Inference	Run classification script on captured image	Correct class, confidence, and inference time	Passed
Servo Actuation	Classification result triggers servo	Appropriate servo sweeps based on class	Passed

Component Testing Results

Figure 15: Component Testing Results

The table below is for object-based testing analysis,

Object-based Inference Testing

Test Object	Expected Class	Detected Class	Confidence (%)	Inference Time (ms)	Result
Banana	Biodegradable	Biodegradable	98.2	85	Passed
Apple	Biodegradable	Biodegradable	99.1	82	Passed
Tin Can	Non-Biodegradable	Non-Biodegradable	99.5	83	Passed
Plastic Bowl	Non-Biodegradable	Non-Biodegradable	98.5	86	Passed

Figure 16: Object-Based Testing

6 Conclusion

The objective of this project was to design and implement a real-time, automatic waste segregation system using a Raspberry Pi that integrates sensor data, image classification, and servo-based actuation. Our system successfully meets the outlined goals by leveraging a modular architecture that combines embedded C++, Python-based machine learning inference, and precise real-time task scheduling.

Through the integration of key hardware components such as the ultrasonic distance sensor, MQ7 gas sensor with ADS1115 ADC, camera module, and servos motors, we achieved synchronized decision-making in real time. The ultrasonic sensor effectively detects proximity, triggering the image capture module when an object is within range. The camera captures the object frame, which is then passed to a TensorFlow Lite-based Python script that classifies the object as either biodegradable or non-biodegradable. Based on the result, the corresponding servo mechanism directs the object to the correct bin. Simultaneously, the gas sensor continuously monitors for dangerous levels of CO, with emergency override logic built in to stop normal operations when unsafe conditions are detected.

This project demonstrates the effectiveness of combining embedded systems programming with lightweight machine learning models for edge computing applications. The sequencer and multithreaded service architecture ensured deterministic timing, making the system responsive and efficient. In conclusion, the project stands as a practical and scalable approach to smart waste segregation and offers a robust foundation for further development toward full-scale deployment in smart city or industrial automation contexts.

7 References

References

[1] Mq7_Sensor Arduino.*Arduino CO Monitor Using MQ-7 Sensor*.

[2] Adafruit library for the Gas Sensor. *Using ADS1115 ADC with Raspberry Pi*.

[3] Calibration_and_Implementation_of_Heat_Cycle_Requirement_of_MQ-7_Semiconductor_Sensor_for_Detection_of_Carbon_Monoxide_Concentrations.*CO concentration documentation*.

[4] raspberry-pi-analog-to-digital-converters.*raspberry-pi-analog-to-digital-converters*.

[5] Using USS with Raspberry Pi*Using USS with Raspberry Pi*.

[6] fastest-way-to-capture-4k-video. *fastest-way-to-capture-4k-video*.

[7] performancecomparison-of-condition-variables-and-atomics *performancecomparison-of-condition-variables-and-atomics*.

[8] benchmarking-machine-learning-on-the-new-raspberry-pi-4-model. *benchmarking-machine-learning-on-the-new-raspberry-pi-4-model-*.

8 Appendix

Listing 1: Final Main file of Automated Waste Segregator program Implementation

```
#include <iostream>
#include <csignal>
#include <wiringPi.h>
#include <opencv2/opencv.hpp>
#include "ads1115rpi.h"
#include "servo.hpp"
#include "Sequencer.hpp"
#include <chrono>
#include <atomic>
#include <condition_variable>
#include "persistent_v4l2_camera.hpp"

#define MOSFET_WPI_PIN 6
#define TRIG_PIN 4
#define ECHO_PIN 5

std::atomic<bool> keepRunning{true};
std::atomic<bool> frame_ready(false);
std::atomic<bool> processing_in_progress(false);
std::atomic<bool> stop_threads(false);
std::mutex frame_mutex, mtx;
std::condition_variable cv_capture;
std::string saved_image_path = "capture.jpg";

enum class SystemState { RUNNING, EMERGENCY };
std::atomic<SystemState> systemState{SystemState::RUNNING};

void signalHandler(int signum) {
    std::cout << "\nSIGINT␣received.␣Stopping...\n";
    keepRunning = false;
    stop_threads = true;
}

class MQ7Callback : public ADS1115rpi::ADSCallbackInterface {
public:
    void hasADS1115Sample(float sample) override {
        if (sample > 1.9f && systemState != SystemState::EMERGENCY) {
            systemState = SystemState::EMERGENCY;
            std::cout << "ALERT:␣Gas␣level␣high!␣Emergency␣stop.\n";
        } else if (sample < 1.7f && systemState ==
           SystemState::EMERGENCY) {
            systemState = SystemState::RUNNING;
            std::cout << "Gas␣level␣safe.␣Resuming.\n";
        }
    }
};

void gas_service() {
    static MQ7Callback cb;
    static ADS1115rpi reader;
    static bool initialized = false;
```

```
    if (!initialized) {
        ADS1115settings settings;
        settings.channel = ADS1115settings::AIN0;
        settings.pgaGain = ADS1115settings::FSR2_048;
        settings.samplingRate = ADS1115settings::FS860HZ; // Changing
           sampling rate from 8 samples/sec to 860 samples/sec
        reader.registerCallback(&cb);
        reader.start(settings);
        initialized = true;
    }

    if (systemState == SystemState::EMERGENCY)
        digitalWrite(MOSFET_WPI_PIN, HIGH);
    else
        digitalWrite(MOSFET_WPI_PIN, LOW);
}

float measure_distance() {
    digitalWrite(TRIG_PIN, HIGH);
    delayMicroseconds(10);
    digitalWrite(TRIG_PIN, LOW);
    while (digitalRead(ECHO_PIN) == LOW);
    long start_time = micros();
    while (digitalRead(ECHO_PIN) == HIGH);
    long end_time = micros();
    return (end_time - start_time) * 0.0343 / 2.0;
}


void capture_frames(PersistentV4L2Camera& camera) {

    if (processing_in_progress) return;
    float distance = measure_distance();
    std::cout << "Measured␣distance:␣" << distance << "␣cm\n";
    if (distance < 20.0) {
        if (camera.captureToFile(saved_image_path)) {
            frame_ready = true;
            processing_in_progress = true;
            std::cout << "Captured␣" << saved_image_path << "\n";
        } else {
            std::cerr << "Failed␣to␣capture␣frame\n";
        }
    }
}

std::string run_python_script(const std::string& image_file) {
    std::string cmd =
       "/home/abhirathkoushik/RTES_files/RTES_final_project/myenv/bin/python3␣
       predict_tflite.py␣" + image_file;
    std::string result;
    char buffer[256];
    std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(cmd.c_str(),
       "r"), pclose);
```

```
    if (!pipe) return "";
    while (fgets(buffer, sizeof(buffer), pipe.get()) != nullptr) {
        result += buffer;
    }
    return result;
}

std::string extract_json_field(const std::string& json, const
   std::string& key) {
    size_t pos = json.find("\"" + key + "\"");
    if (pos == std::string::npos) return "";
    pos = json.find(":", pos);
    size_t start = json.find_first_not_of("␣\":", pos);
    size_t end = json.find_first_of(",}", start);
    std::string field = json.substr(start, end - start);
    field.erase(remove(field.begin(), field.end(), '"'), field.end());
    return field;
}

void inference_service() {
    if (!frame_ready) return;
    {
        std::lock_guard<std::mutex> lock(frame_mutex);
        frame_ready = false;
    }

    std::string output = run_python_script(saved_image_path);
    if (!output.empty()) {
        std::string detected_class = extract_json_field(output,
           "class");
        std::string confidence = extract_json_field(output,
           "confidence");
        std::string inference_time = extract_json_field(output,
           "inference_time_ms");
        if (detected_class == "biodegradable") sweep_servo_1();
        else if (detected_class == "nonbiodegradable") sweep_servo_2();
        else std::cout << "Unknown␣detection␣result!\n";

        std::cout << "Detected␣Class␣␣␣:␣" << detected_class << "\n";
        std::cout << "Confidence␣␣␣␣␣␣␣:␣" << confidence << "\n";
        std::cout << "Inference␣Time␣␣␣:␣" << inference_time << "␣
           ms\n";

        processing_in_progress = false;
    }
}

int main() {
    signal(SIGINT, signalHandler);
    wiringPiSetup();
    pinMode(MOSFET_WPI_PIN, OUTPUT);
    pinMode(TRIG_PIN, OUTPUT);
    pinMode(ECHO_PIN, INPUT);
    digitalWrite(MOSFET_WPI_PIN, LOW);
```

```
    digitalWrite(TRIG_PIN, LOW);

    init_servos();
    set_servo2_initial();
    set_servo1_initial();
    PersistentV4L2Camera camera("/dev/video0");
    Sequencer seq;
    seq.addService("Gas Monitor", gas_service, 1, 99, 100);
    seq.addService("Camera + Distance", [&camera]() {
       capture_frames(camera); }, 1, 98, 200);
    seq.addService("Inference", inference_service, 2, 99, 300);

    seq.startServices();
    std::cout << "Press Ctrl+C to stop...\n";

    while (keepRunning.load()) {
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
    seq.stopServices();
    std::cout << "System shutdown complete.\n";
    return 0;
}
```

Listing 2: Python inference program

```
import tflite_runtime.interpreter as tflite
from PIL import Image
import numpy as np
import sys
import time
import json

print("PYTHON:", sys.executable)

# ---- Config ----
IMAGE_PATH = sys.argv[1]
MODEL_PATH = "model_new_kaggle_dataset.tflite"
LABELS_PATH = "labels.txt"
IMAGE_SIZE = (224, 224)

# ---- Load labels ----
with open(LABELS_PATH, "r") as f:
    labels = [line.strip() for line in f.readlines()]

# ---- Load image and preprocess ----
img = Image.open(IMAGE_PATH).convert('RGB')
img = img.resize(IMAGE_SIZE)
img = np.array(img).astype(np.float32)
img = img / 127.5 - 1.0  # MobileNetV2 preprocess_input
img = np.expand_dims(img, axis=0)

# ---- Load TFLite model ----
interpreter = tflite.Interpreter(model_path=MODEL_PATH)
interpreter.allocate_tensors()

input_index = interpreter.get_input_details()[0]['index']
output_index = interpreter.get_output_details()[0]['index']

# ---- Run inference with timing ----
start_time = time.time()
interpreter.set_tensor(input_index, img)
interpreter.invoke()
preds = interpreter.get_tensor(output_index)[0]
end_time = time.time()

# ---- Output prediction ----
top = np.argmax(preds)
confidence = float(preds[top])  # <-- Convert to Python float
elapsed_time_ms = float((end_time - start_time) * 1000)  # <-- Also
   convert

# Standardize label
if labels[top] == "biodegradeable":
    label = "biodegradable"
elif labels[top] == "nonbio":
    label = "nonbiodegradable"
else:
    label = labels[top]
```

```
# Output as JSON
result = {
    "class": label,
    "confidence": confidence,
    "inference_time_ms": elapsed_time_ms
}
print(json.dumps(result))
```

Listing 3: Servo Motor program

```
#include <iostream>
#include <softPwm.h>
#include <unistd.h>
#include <wiringPi.h>
#include "servo.hpp"


void init_servos() {
    wiringPiSetup();

    if (softPwmCreate(SERVO1_GPIO, 0, 200) != 0) {
        std::cerr << "Failed to initialize servo 1\n";
        exit(1);
    }
    if (softPwmCreate(SERVO2_GPIO, 0, 200) != 0) {
        std::cerr << "Failed to initialize servo 2\n";
        exit(1);
    }
}

void set_servo2_initial()
{
    std::cout << "Setting Servo 2 at GPIO 27 to Initial
       Position"<<std::endl;
    softPwmWrite(SERVO2_GPIO, 17);
    sleep(1);

    softPwmWrite(SERVO2_GPIO, 0);
}

void sweep_servo_2()
{
    std::cout << "Sweeping Servo 2 on GPIO 27" << std::endl;
    for (int pulse = 17; pulse >= 9  ; --pulse) {
        softPwmWrite(SERVO2_GPIO, pulse);
        usleep(30000);
    }
    usleep(1000000);
    for (int pulse = 9; pulse <= 17; ++pulse) {
        softPwmWrite(SERVO2_GPIO, pulse);
        usleep(30000);
    }
    softPwmWrite(SERVO2_GPIO, 0);
}

void set_servo1_initial()
{
    std::cout << "Setting Servo 1 at GPIO 17 to Initial
       Position"<<std::endl;
    softPwmWrite(SERVO1_GPIO, 15);
    sleep(1);

    softPwmWrite(SERVO1_GPIO, 0);
```

```
}

void sweep_servo_1()
{
    std::cout << "Sweeping␣Servo␣1␣on␣GPIO␣17" << std::endl;
    for (int pulse = 15; pulse <= 23; ++pulse) {
        softPwmWrite(SERVO1_GPIO, pulse);
        usleep(30000);
    }
    usleep(1000000);
    for (int pulse = 23; pulse >= 15  ; --pulse) {
        softPwmWrite(SERVO1_GPIO, pulse);
        usleep(30000);
    }
    softPwmWrite(SERVO1_GPIO, 0);
}
```

Listing 4: The header file used for capturing and saving the image and opening and closing of the camera

```
// persistent_v4l2_camera.hpp
#ifndef PERSISTENT_V4L2_CAMERA_HPP
#define PERSISTENT_V4L2_CAMERA_HPP

#include <linux/videodev2.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <opencv2/opencv.hpp>
#include <string>
#include <stdexcept>
#include <cstring>
#include <iostream>

class PersistentV4L2Camera {
public:
    PersistentV4L2Camera(const std::string& device = "/dev/video0",
       int width = 640, int height = 480)
        : fd(-1), buffer(nullptr), buffer_length(0), WIDTH(width),
          HEIGHT(height)
    {
        open_device(device);
    }

    ~PersistentV4L2Camera() {
        if (fd >= 0) {
            v4l2_buf_type type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
            ioctl(fd, VIDIOC_STREAMOFF, &type);
            if (buffer) munmap(buffer, buffer_length);
            close(fd);
        }
    }

    bool captureToFile(const std::string& filename) {
        v4l2_buffer buf{};
        buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        buf.memory = V4L2_MEMORY_MMAP;
        buf.index = 0;

        if (ioctl(fd, VIDIOC_QBUF, &buf) < 0) return false;

        fd_set fds;
        FD_ZERO(&fds);
        FD_SET(fd, &fds);
        timeval tv = {2, 0};

        if (select(fd + 1, &fds, NULL, NULL, &tv) <= 0) {
            std::cerr << "Timeout␣waiting␣for␣frame\n";
            return false;
        }
```

```
        if (ioctl(fd, VIDIOC_DQBUF, &buf) < 0) return false;

        cv::Mat yuyv(HEIGHT, WIDTH, CV_8UC2, buffer);
        cv::Mat bgr;
        cv::cvtColor(yuyv, bgr, cv::COLOR_YUV2BGR_YUYV);
        return cv::imwrite(filename, bgr);
    }

private:
    int fd;
    void* buffer;
    size_t buffer_length;
    const int WIDTH, HEIGHT;

    void open_device(const std::string& device) {
        fd = open(device.c_str(), O_RDWR);
        if (fd < 0) throw std::runtime_error("Failed to open device");

        v4l2_format fmt{};
        fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        fmt.fmt.pix.width = WIDTH;
        fmt.fmt.pix.height = HEIGHT;
        fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
        fmt.fmt.pix.field = V4L2_FIELD_NONE;

        if (ioctl(fd, VIDIOC_S_FMT, &fmt) < 0)
            throw std::runtime_error("VIDIOC_S_FMT failed");

        v4l2_requestbuffers req{};
        req.count = 1;
        req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        req.memory = V4L2_MEMORY_MMAP;

        if (ioctl(fd, VIDIOC_REQBUFS, &req) < 0)
            throw std::runtime_error("VIDIOC_REQBUFS failed");

        v4l2_buffer buf{};
        buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        buf.memory = V4L2_MEMORY_MMAP;
        buf.index = 0;

        if (ioctl(fd, VIDIOC_QUERYBUF, &buf) < 0)
            throw std::runtime_error("VIDIOC_QUERYBUF failed");

        buffer_length = buf.length;
        buffer = mmap(NULL, buf.length, PROT_READ | PROT_WRITE,
           MAP_SHARED, fd, buf.m.offset);
        if (buffer == MAP_FAILED)
            throw std::runtime_error("mmap failed");

        v4l2_buf_type type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        if (ioctl(fd, VIDIOC_STREAMON, &type) < 0)
            throw std::runtime_error("VIDIOC_STREAMON failed");
    }
```

```
};

#endif
```

Listing 5: I2C for ADC

```
#include "ads1115rpi.h"

#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>


void ADS1115rpi::start(ADS1115settings settings) {
        ads1115settings = settings;

        char gpioFilename[20];
        snprintf(gpioFilename, 19, "/dev/i2c-%d", settings.i2c_bus);
        fd_i2c = open(gpioFilename, O_RDWR);
        if (fd_i2c < 0) {
            char i2copen[] = "Could␣not␣open␣I2C.\n";
#ifdef DEBUG
            fprintf(stderr,i2copen);
#endif
            throw i2copen;
        }

        if (ioctl(fd_i2c, I2C_SLAVE, settings.address) < 0) {
            char i2cslave[] = "Could␣not␣access␣I2C␣adress.\n";
#ifdef DEBUG
            fprintf(stderr,i2cslave);
#endif
            throw i2cslave;
        }

#ifdef DEBUG
        fprintf(stderr,"Init.\n");
#endif
        // Enable RDY
        i2c_writeWord(reg_lo_thres, 0x0000);
        i2c_writeWord(reg_hi_thres, 0x8000);

        unsigned r = (0b10000000 << 8); // kick it all off
        r = r | (1 << 2) | (1 << 3); // data ready active high &
           latching
        r = r | (settings.samplingRate << 5);
        r = r | (settings.pgaGain << 9);
        r = r | (settings.channel << 12) | 1 << 14; // unipolar
        i2c_writeWord(reg_config,r);

#ifdef DEBUG
        fprintf(stderr,"Receiving␣data.\n");
#endif

        chipDRDY = gpiod_chip_open_by_number(settings.drdy_chip);
        pinDRDY = gpiod_chip_get_line(chipDRDY,settings.drdy_gpio);

        int ret = gpiod_line_request_rising_edge_events(pinDRDY,
```

```
            "Consumer");
        if (ret < 0) {
#ifdef DEBUG
            fprintf(stderr,"Request␣event␣notification␣failed␣on␣pin␣
               %d␣and␣chip␣
               %d.\n",settings.drdy_chip,settings.drdy_gpio);
#endif
            throw "Could␣not␣request␣event␣for␣IRQ.";
        }

        running = true;

        thr = std::thread(&ADS1115rpi::worker,this);
}


void ADS1115rpi::setChannel(ADS1115settings::Input channel) {
        unsigned r = i2c_readWord(reg_config);
        r = r & (~(3 << 12));
        r = r | (channel << 12);
        i2c_writeWord(reg_config,r);
        ads1115settings.channel = channel;
}


void ADS1115rpi::dataReady() {
        float v = (float)i2c_readConversion() / (float)0x7fff *
           fullScaleVoltage();
        for(auto &cb: adsCallbackInterfaces) {
            cb->hasADS1115Sample(v);
        }
}


void ADS1115rpi::worker() {
    while (running) {
        const struct timespec ts = { 1, 0 };
        gpiod_line_event_wait(pinDRDY, &ts);
        struct gpiod_line_event event;
        gpiod_line_event_read(pinDRDY, &event);
        dataReady();
    }
}


void ADS1115rpi::stop() {
    if (!running) return;
    running = false;
    thr.join();
    gpiod_line_release(pinDRDY);
    gpiod_chip_close(chipDRDY);
    close(fd_i2c);
}
```

```
// i2c read and write protocols
void ADS1115rpi::i2c_writeWord(uint8_t reg, unsigned data)
{
        uint8_t tmp[3];
        tmp[0] = reg;
        tmp[1] = (char)((data & 0xff00) >> 8);
        tmp[2] = (char)(data & 0x00ff);
        long int r = write(fd_i2c,&tmp,3);
        if (r < 0) {
#ifdef DEBUG
                fprintf(stderr,"Could␣not␣write␣word␣from␣%02x.␣
                   ret=%d.\n",ads1115settings.address,r);
#endif
                throw "Could␣not␣write␣to␣i2c.";
        }
}

unsigned ADS1115rpi::i2c_readWord(uint8_t reg)
{
        uint8_t tmp[2];
        tmp[0] = reg;
        write(fd_i2c,&tmp,1);
        long int r = read(fd_i2c, tmp, 2);
        if (r < 0) {
#ifdef DEBUG
                fprintf(stderr,"Could␣not␣read␣word␣from␣%02x.␣
                   ret=%d.\n",ads1115settings.address,r);
#endif
                throw "Could␣not␣read␣from␣i2c.";
        }
        return (((unsigned)(tmp[0])) << 8) | ((unsigned)(tmp[1]));
}

int ADS1115rpi::i2c_readConversion()
{
        const int reg = 0;
        char tmp[3];
        tmp[0] = reg;
        write(fd_i2c,&tmp,1);
        long int r = read(fd_i2c, tmp, 2);
        if (r < 0) {
#ifdef DEBUG
                fprintf(stderr,"Could␣not␣read␣ADC␣value.␣
                   ret=%d.\n",r);
#endif
                throw "Could␣not␣read␣from␣i2c.";
        }
        return ((int)(tmp[0]) << 8) | (int)(tmp[1]);
}
```