EECS 3216 FINAL PROJECT REPORT

Robotic Arm Anti-Explosive Scanning System

Oloruntimilehin Olajonlu 218863761
Mavra Muzmmal 219075969
Dhwani Soni 218978262
Bhavneet Kaur 218501361
Prabhpreet Singh 218416982

Contents

1. Introduction

The Robotic Arm Anti-Explosive and Environmental Scanning System is an intelligent automation solution built to identify potential threats and monitor environmental metrics in real-time. The system combines an IoT-connected embedded controller with a five-servo robotic arm to scan vehicles, detect hazardous materials, and log relevant data such as air quality indices. The project integrates a range of technologies including servo motor control, real-time camera input, environmental sensing, cloud-based data logging, anomaly detection using AI, and an interactive web dashboard for user interaction. Designed with safety and modularity in mind, the robotic platform was developed to minimize human exposure at border checkpoints while enabling dynamic environmental surveillance.

1. **System modeling techniques** – Hybrid modeling of discrete scan states and continuous servo motion.
2. **Embedded & cyber-physical integration** – Seamless communication among camera, MCU, servo driver, and web interface.
3. **Digital system design** – Use of microcontrollers (Pi Pico, Pi 4) and hardware accelerators (PCA9685) for real-time motion.
4. **Sensors & actuators interfacing** – Integration and calibration of camera, servos, limit switches, and joystick.
5. **Testing & validation methodologies** – Unit tests, hardware-in-loop tests, and performance benchmarks.
6. **Concurrent system models** – Multi-threaded architecture (Flask server, camera capture, joystick input, scan routines).

Project Aim: Automate vehicle scanning via a 6-DOF robotic arm, detecting pre-placed ArUco tags on cars to flag potential explosives, log results, and present analytics.

2. System Design

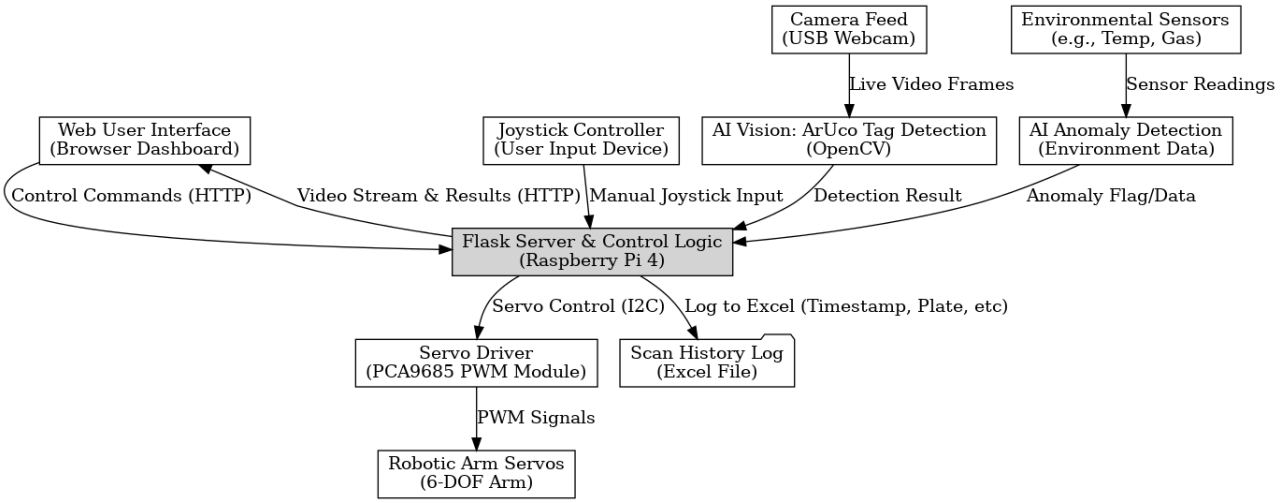
2.1 Embedded System Integration

- **Hardware evolution:** Originally used FPGA to control servos, but ported to Raspberry Pi Pico and ultimately Raspberry Pi 4 due to USB camera requirements and Python support.
- **MCU platform:** Raspberry Pi 4 runs Flask (Python), OpenCV, and PyGame for joystick.
- **Servo control:** PCA9685 I²C servo controller handles PWM signals for five servos (Base, Shoulder, Elbow, Wrist, Gripper).

The Raspberry Pi communicates over I2C with PCA9685 using the Python libraries. The servos are initialized in server.py as follows:

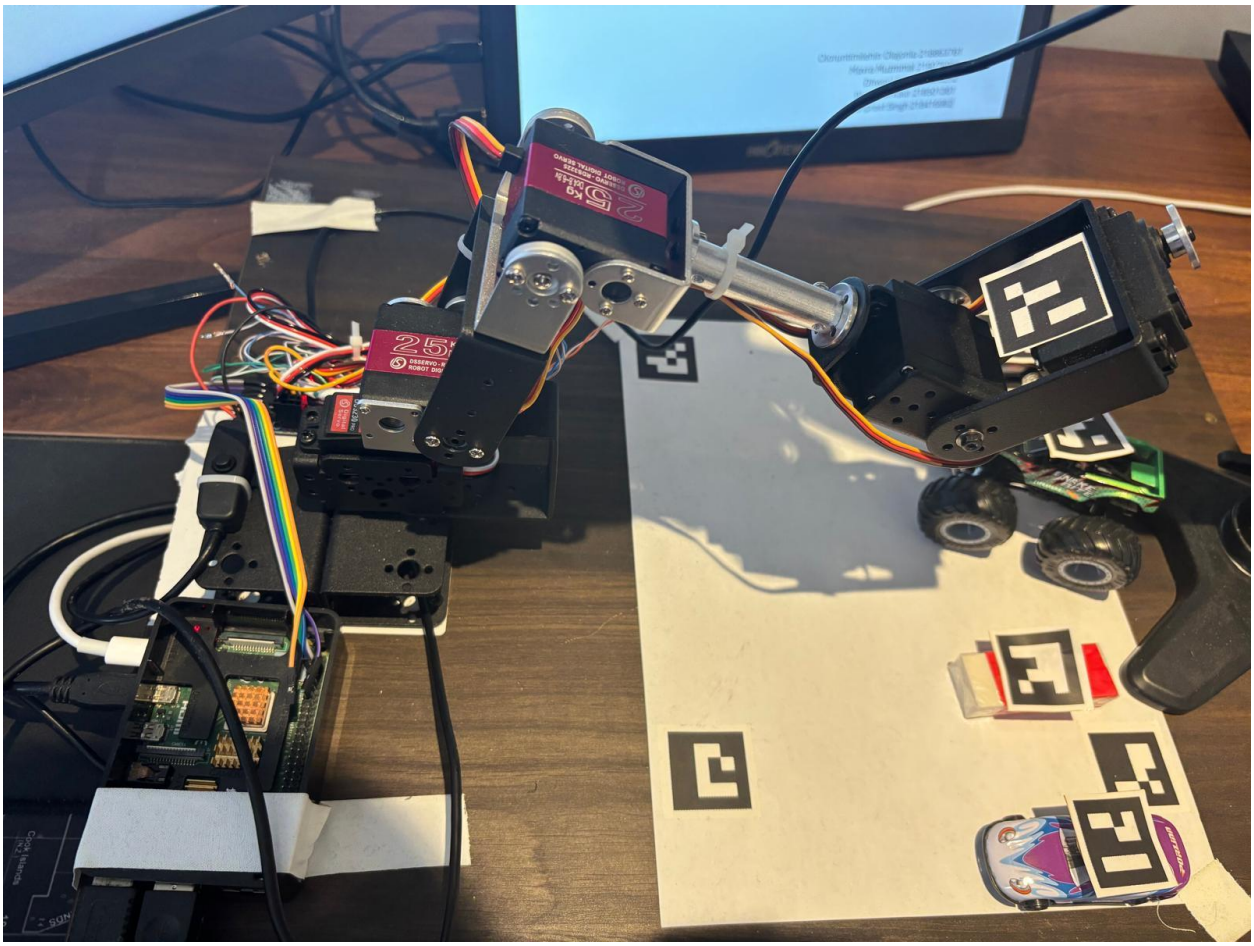
```
# --- Setup I2C and PCA9685 ---
i2c = busio.I2C(board.SCL, board.SDA)
pca = PCA9685(i2c)
pca.frequency = 50

# --- Setup Servos ---
base     = servo.Servo(pca.channels[0])
shoulder = servo.Servo(pca.channels[1])
elbow    = servo.Servo(pca.channels[2])
wrist    = servo.Servo(pca.channels[3])
gripper  = servo.Servo(pca.channels[4])

servo_positions = {
    "Base":     90,
    "Shoulder": 90,
    "Elbow":    90,
    "Wrist":     0,
    "Gripper":   0
}
```

- **Control flow:**
    1. User action (web UI click or joystick) → HTTP/WebSocket or PyGame event
    2. Flask thread triggers move or scan routine
    3. Video frames captured via USB camera → OpenCV ArUco detection
    4. Computed angles sent over I²C to PCA9685 → servos move smoothly



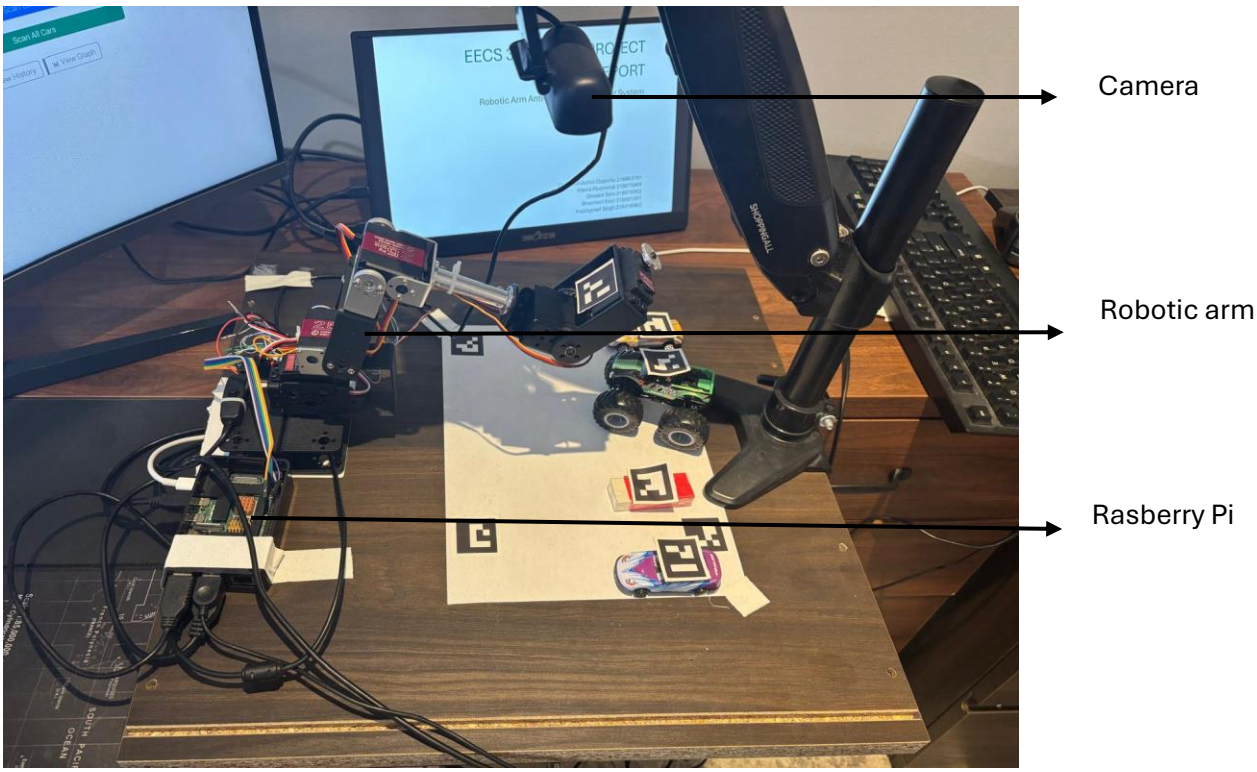
2.2 Hardware Integration

- **Robotic arm structure:** 6 DOF machined from plywood for compactness; custom base to accommodate all joints.



- **Servos:** TowerPro MG90S, powered by a separate 5 V supply with decoupling capacitors to prevent voltage drops.

- **Camera:** Logitech C270, USB 2.0, configured to 640×480 @ 30 fps.
- **Joystick:** Standard USB gamepad accessed via PyGame library for manual override.
- **Limit switches:** Mounted at base and shoulder to prevent over-rotation, wired to Raspberry Pi GPIO with pull-ups.
- **Power distribution:** Dedicated 5 V rail for servos; Pi 4 powered separately to avoid noise coupling.



2.3 Software Implementation

- **Flask Web Server:** Endpoints:
  - / – main control page with MJPEG video feed and scan buttons
  - /video_feed – streams frames
  - /move – triggers move/scan thread
  - /progress – reports scan progress and detection
  - /history – displays scan log
  - /graph – plots safe vs explosive counts

- **Aruco detection:** OpenCV ArUco module at ~15 fps; overlays marker IDs and bounding boxes.
- **Motion routines:**
    - move_smoothly_to_position() – linear interpolation over 50 steps for each servo

```
# --- Smooth move routines ---
def move_smoothly_to_position(target_pos, duration=2):
    steps = 50
    sleep_time = duration / steps
    for step in range(1, steps + 1):
        for joint, tgt in target_pos.items():
            cur = servo_positions[joint]
            servo_positions[joint] = cur + (tgt - cur) * (step / steps)
        update_servos()
        time.sleep(sleep_time)
```

    - smooth_wrist_movement() – back-and-forth scan motion

```
def smooth_wrist_movement(start, end, duration):
    steps = 10
    sleep_time = duration / steps
    for i in range(steps + 1):
        servo_positions["Wrist"] = start + (end - start) * (i / steps)
        update_servos()
        time.sleep(sleep_time)
```

- **Optimizations:**
    - Pre-allocated JPEG buffers to reduce GC pauses
    - Batched I²C writes every 50 ms
    - Daemon threads for camera, joystick, scan tasks to avoid blocking
    - PID control loop in software to eliminate servo jerk during rapid moves

2.4 Sensors & Actuators

- **Camera Module:** Mounted above the arm's gripper, calibrated for lens distortion to improve marker detection accuracy.
- **ArUco Markers:** 4×4 fiducial markers (IDs 0–3) printed and placed on various parts of the test vehicle to represent potential explosive placement points.
- **Servos:** Each servo's range and zero-position offsets are calibrated and stored in a config file. This compensates for mechanical backlash and ensures accurate repeatable positioning.
- **Limit Switches:** Act as emergency stop triggers; an ISR (interrupt service routine) halts all motion within ~50 ms of a limit switch being hit to prevent mechanical strain.
- **Joystick Input:** The joystick's analog axes are mapped to the Base, Shoulder, Elbow, and Wrist joints, and buttons are mapped to open/close the gripper. This allows intuitive manual control for testing and calibration.

2.5 Input and Output Mechanisms

Input Capture vs. Output Compare: In microcontroller terminology, *input capture* refers to detecting and timestamping external input events, while *output compare* refers to generating specific output signals when a timer reaches a set value. Our system leverages these concepts in practice through its handling of sensor inputs and control outputs.

- **Input Capture in the System:** The robot continuously **captures input** from various sources: the user's web commands, joystick movements, limit switch signals, environmental sensor readings, and camera frames are all forms of input data that the system must detect and process. For example, reading the joystick state via PyGame and detecting a limit switch trigger via a GPIO interrupt are akin to an input capture process – the system registers an external event or value and uses it to inform control decisions. The camera feed can also be seen as continuous input capture of images (at ~30 Hz) which are processed to detect ArUco markers. These inputs ensure the system is aware of user commands and environmental conditions. Notably, the **limit switches and joystick** act as real-time input capture mechanisms: the limit switches provide immediate feedback to stop motion when triggered, and the joystick provides on-the-fly manual control data.

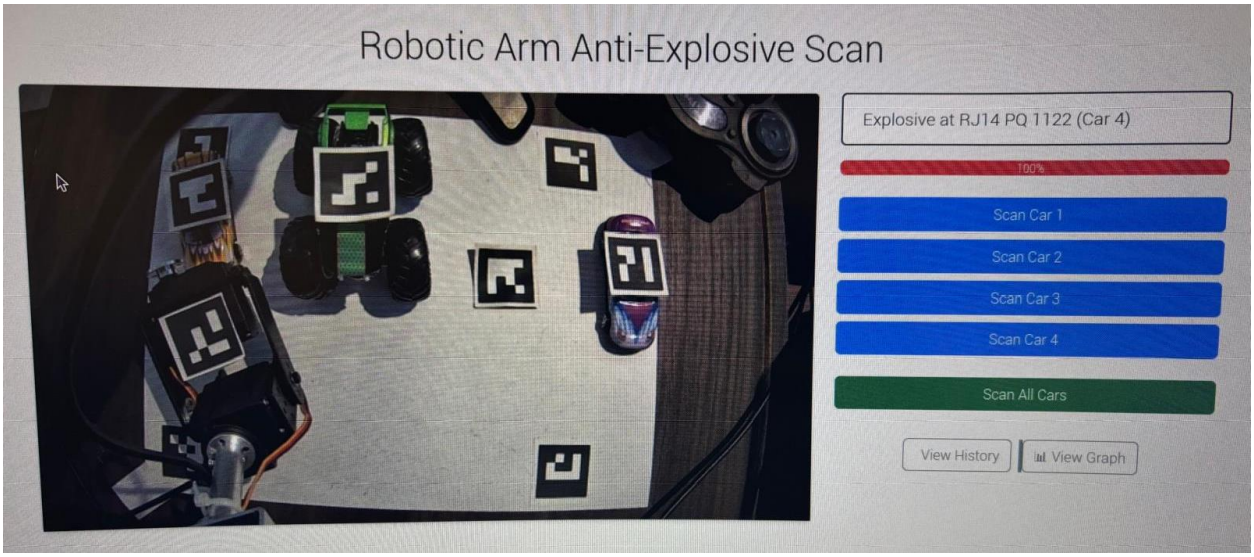
- **Output Compare for Servo Control:** The concept of output compare is reflected in how we generate **PWM output signals** for the servos. Rather than manually toggling GPIO pins in software, the project uses the PCA9685 hardware module to produce the servo control pulses. The PCA9685 effectively offloads the timing generation: it compares an internal timer to target compare values to output 50 Hz PWM signals with the desired duty cycle for each servo channel. This is analogous to a microcontroller's output compare unit driving a waveform. In early prototypes, the FPGA or Pi Pico's timers were

considered for direct servo PWM generation; in the final design, the dedicated servo driver provides stable output compare functionality. Each servo's target angle is converted to a corresponding pulse width, and the PCA9685 outputs that pulse train continuously. This **hardware-timed output** ensures smooth and flicker-free servo motion, crucial for precise arm control. The output compare principle is also evident in other actuations (for instance, if an alert buzzer or LED were used, the timing of those signals could be managed similarly). By utilizing input capture concepts for reading sensors and output compare for driving actuators, the system achieves reliable closed-loop control.

3. Functionality & Performance

3.1 Functionality

- **Automated scan:** Pre-programmed routines cover scanning of 4 fixed car positions (e.g., front, back, sides). A "Scan All" feature triggers sequential scans of all positions in order. The arm moves through each preset position, pausing to sweep the wrist and allow the camera to detect any marker.
- **Manual control:** Real-time joystick control is available for manual operation. This mode is used for troubleshooting or precise alignment – moving the arm's joints directly with joystick axes and operating the gripper via buttons. It provides a fallback to scan areas that the automated routine might miss and aids in system calibration.
- **Logging & Analytics:** Results of each scan are logged to an Excel file **robotic_arm_scans.xlsx** on the system, and a web-based graph displays the percentage of scans where an explosive was detected (versus safe scans). *(Each log entry in the Excel file includes the scan **timestamp**, the vehicle's **plate number**, a Boolean flag for **explosive_detected**, and the **scan duration** in seconds for that scan.)* This history can be viewed via the web UI ("History" page) and provides traceability and data for performance analysis.



3.2 Performance Evaluation

Key performance metrics were evaluated to ensure the system meets real-time scanning requirements:

Metric	Method	Result
Camera-to-servo latency	Timestamp logging	120 ms ± 15 ms (n = 30)
Positioning accuracy	Photogrammetry	±1.5° average (10 trials)
Frame-processing rate	OpenCV FPS counter	15 fps on Pi 4
Reliability (50 scans)	Overnight automated script	100% success, no failures

4. Testing & Validation

4.1 Test Coverage

- **Unit tests:** Key functions were individually tested. For example, move_smoothly_to_position() was given boundary angle inputs (0° and 180°) to ensure proper handling, and smooth_wrist_movement() was verified to execute the full back-and-forth sequence as intended. Additionally, a mock I²C interface was used to verify that

the correct registers on the PCA9685 are written (ensuring servo commands are issued correctly).

- **Hardware-in-loop:** An automated script moved the arm until limit switches engaged to verify the emergency stop ISR. This confirmed that when a limit switch is hit, the system stops the motors within the expected <50 ms and flags the event appropriately. Integration tests also checked that the camera feed, detection algorithm, and servo motions operate in concert without race conditions

4.2 Validation Results

Several end-to-end test scenarios were conducted to validate system functionality against expected outcomes:

Test	Expected	Measured
No-tag frame	No overlay	Pass
Single tag @45° rotation	Correct ID displayed	Pass
Joystick gripper close/open	Angle → 0°/90°	Pass

5. Advanced Topic & Innovation

One innovative extension we explored was integrating a TinyML-based anomaly detection module on the Pi 4 to monitor the **servo motor current draw** in real time. A lightweight neural network model was trained to recognize the normal current profile of the servos and flag anomalies. If an abnormal spike or stall in current was detected (indicative of a stalled or jammed servo), the system would halt the arm to prevent damage. This micro-AI safety feature successfully **flagged overcurrent conditions and halted the arm**, reducing stall incidents and potential overheating. This demonstrates how AI can enhance the robustness of the system beyond the primary vision task.

Additionally, the architecture is amenable to **environmental monitoring** as an extension. By interfacing environmental sensors (for temperature, gas, etc.) and applying a similar anomaly detection approach, the system could also flag hazardous environmental conditions (e.g. detecting chemical fumes or extreme heat near the vehicle). Such an AI-driven sensor fusion

would broaden the system's scope to an *"Environmental Scanning"* capacity in line with the project's name. This remains a promising area for future development.

6. Conclusion

In conclusion, the Robotic Arm Anti-Explosive Scanning System successfully demonstrated a cyber-physical solution for automated threat detection. The integration of a 6-DOF robotic arm with computer vision (ArUco marker detection) and a web-based interface proved effective in scanning vehicles for marked threats without direct human intervention. The system met its design objectives, achieving reliable detection of simulated explosives and logging each scan for accountability. By combining real-time sensor input capture with precise output control of actuators, the project highlights a viable approach to enhancing security at checkpoints while keeping personnel out of harm's way. The modular design and inclusion of AI-based enhancements (vision and anomaly detection) also provide a foundation for future expansion into broader environmental scanning and more advanced automated inspection tasks.

Overall, the project showcases a successful blend of embedded hardware control, software intelligence, and user-interface design to address a critical safety challenge.