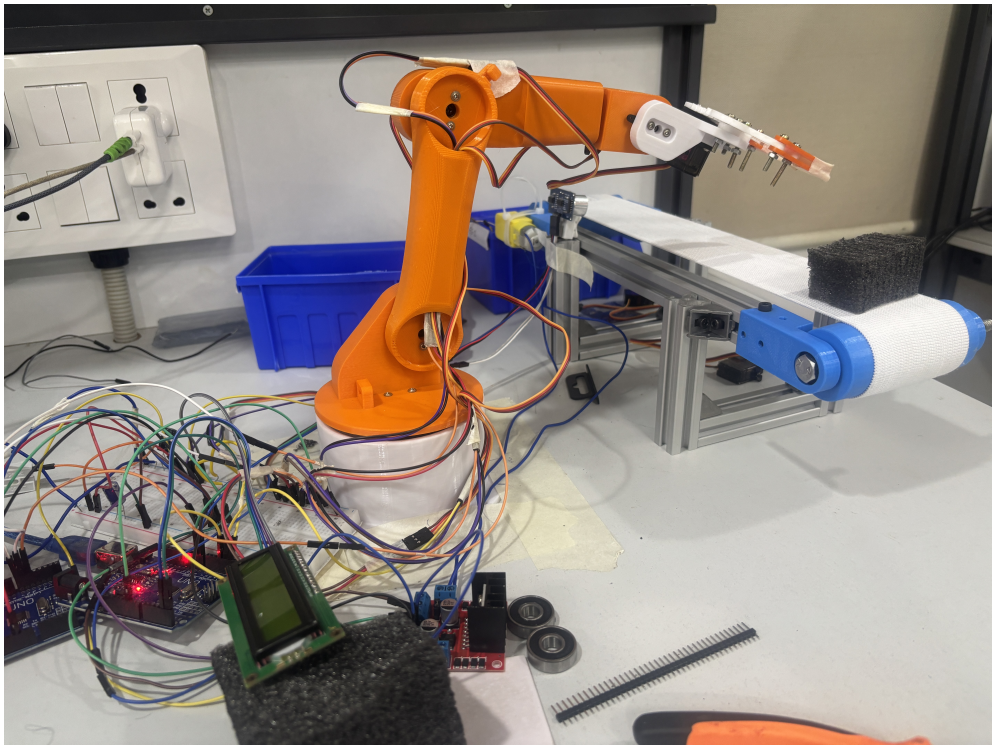
Indera Mission Control

Software, Control, and Automation Report

Project Theme: Augmenting the Indera base robotic arm into a software-driven sorting and demonstration platform using dual microcontrollers, a browser-based control stack, persistent runtime state, camera-based color detection, sensor-triggered automations, and a hardcoded closed-loop demonstration sequence.



Name 1	Harshit Singh
Roll Number 1	24B4506
Name 2	Vaibhav Negi
Roll Number 2	24B4503

April 19, 2026

Contents

1. Abstract

This report documents the software and systems engineering work carried out on the base Indera robotic arm platform. The starting point was a functional robotic arm mechanism and conveyor setup. Our contribution was to transform that base machine into an integrated cyber-physical system that can be manually operated through a modern web dashboard, taught configurable poses, assembled into configurable workflows, triggered through sensor events, monitored through an onboard LCD, and executed in a repeatable hardcoded demonstration loop that branches based on detected object color.

The final system uses two Arduino microcontrollers, a Python Flask backend, a browser-based operator interface, an ultrasonic distance sensor, an I2C LCD, a conveyor drive motor, and a camera-based OpenCV color detector. The software architecture deliberately separates low-level hardware actuation from high-level orchestration. Arduino firmware remains focused on deterministic device-level control, while the backend owns workflows, automation rules, demo sequencing, state persistence, and event routing.

2. Introduction

The base Indera robot provided the mechanical foundation: a multi-joint robotic arm, a conveyor mechanism, and a bench-top integration environment. However, the base system required substantial software augmentation to become useful for a live, interactive automation task. The goal of our work was not merely to move the arm, but to build a complete control and automation stack around it.

The central idea of the project was to evolve the robot into a demonstration-ready smart sorting platform with the following properties:

- live browser-based manual control,
- configurable saved positions,
- configurable workflows made from reusable motion blocks,
- trigger-based actions driven by sensor input,
- a visible machine-side status display,
- persistent state and recoverable software behavior,
- color-aware branching for sorting behavior.

The result is a software-centered robotic system whose intelligence sits primarily in the backend, while the microcontrollers are used for safe and practical low-level interfacing.

3. Project Objectives

3.1. Primary Objective

To convert the base Indera robot into a controllable and automatable robotic sorting platform capable of:

- receiving operator commands through a web interface,
- detecting object presence using ultrasonic sensing,
- detecting object color using computer vision,
- executing configurable workflows,
- triggering actions based on sensor conditions,
- demonstrating closed-loop conveyor plus arm behavior.

3.2. Secondary Objectives

- keep Arduino firmware relatively simple and robust,
- centralize orchestration in the backend,
- persist runtime state across page refreshes and restarts,
- create a structure that supports future extension to richer sorting logic.

4. Base Platform and Hardware Augmentation

4.1. Base Platform

The project started from the Indera base robotic arm. Mechanically, the arm already provided the structural degrees of freedom required for pick-and-place type motion. A conveyor was added as the object transport mechanism. The software work focused on making the robot practically usable and demonstrable.

4.2. Hardware Used

Component	Role in System
Arduino 1	Controls base, shoulder, elbow; reads ultrasonic sensor; drives LCD
Arduino 2	Controls wrist pitch, wrist roll, gripper, and conveyor motor
Servo motors	Actuate arm joints
DC motor + L298N driver	Drives conveyor belt motion
HC-SR04 ultrasonic sensor	Detects object presence / distance on conveyor
I2C 16x2 LCD	Displays machine-side status and execution messages
Web camera	Performs color detection through computer vision
Laptop / host computer	Runs Flask backend, OpenCV pipeline, and operator dashboard

Table 1: Hardware used in the final system

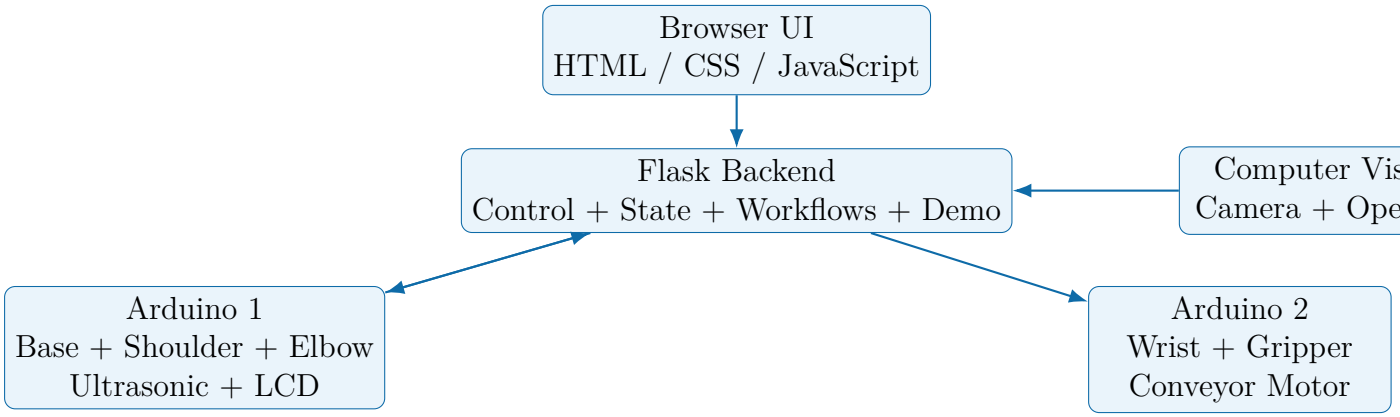
4.3. Why Two Arduinos Were Used

We deliberately used two Arduinos to separate responsibilities and reduce integration risk. One controller handles the structural side of the robot along with the local sensing/display devices. The other handles the end-effector and conveyor motor. This decision improved modularity, simplified pin usage, and made debugging easier because each controller had a clearer role.

Engineering justification: We optimized for robustness and modularity rather than minimum component count. For a prototype integrating multiple servos, a conveyor motor, sensor interfaces, LCD messaging, and serial control, splitting responsibilities was the safer and more maintainable architecture.

5. System Architecture

5.1. Layered View



5.2. Software Components

File / Module	Responsibility
`app.py`	Main backend controller; serial routing; workflows; automations; hardcoded demo logic; API endpoints
`state_store.py`	Persistent runtime state in JSON form
`main_1.ino`	Structural joint firmware + ultrasonic + LCD behavior
`main_2.ino`	Effector joint firmware + motor control
`templates/index.html`	Dashboard markup
`static/script.js`	Frontend logic and UI behavior
`static/style.css`	Dashboard styling
`cv_detector.py`	Reusable computer vision detection logic
`cv.py`	Standalone preview utility for camera-based color detection

Table 2: Key software modules

6. Methodology and Development Approach

6.1. Stage 1: Manual Control Foundation

The first software layer implemented was direct browser-based control of the robot joints and conveyor motor. This created the minimum viable ability to command the robot from a host computer. Serial commands were mapped to the correct Arduino based on which actuator was being addressed.

6.2. Stage 2: Saved Positions

Manual control alone is insufficient for repeatable robotic behavior. Therefore, we introduced named positions, stored in `positions.json`. This allowed the robot to be taught specific poses such as pick, lift, and drop configurations.

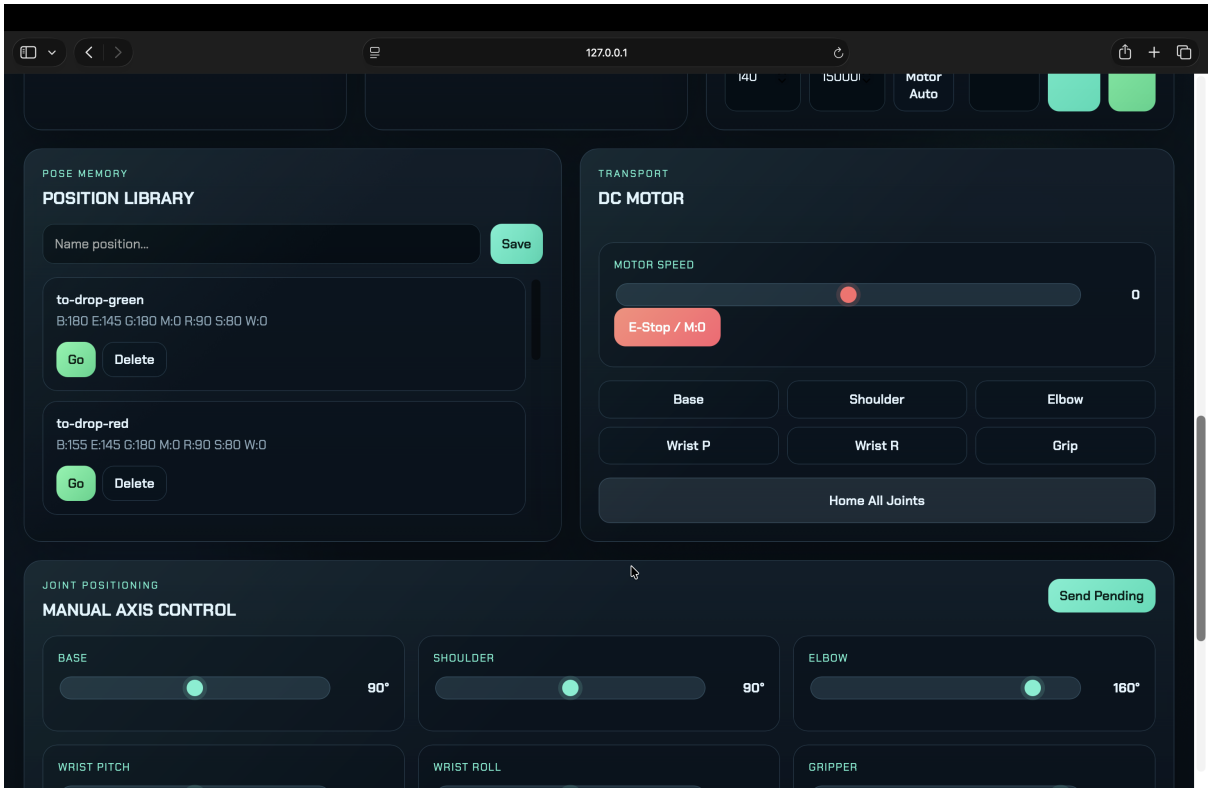


Figure 1: Position Library and Manual Control

6.3. Stage 3: Configurable Workflows

Once named positions were available, the next step was workflow composition. Users can create workflows from reusable blocks:

- `move` to a saved position,
- `wait` for a given duration,
- `motor_run` for a speed-dependent conveyor segment.

These workflows are stored in `workflows.json` and are executed by the backend rather than by the browser.

6.4. Stage 4: Trigger-Based Automations

With motion primitives available, we introduced sensor-triggered behavior. The ultrasonic sensor continuously reports distance. If configured conditions are satisfied, the backend schedules an automation rule that can stop the motor, home the arm, or run a workflow after a configurable delay.

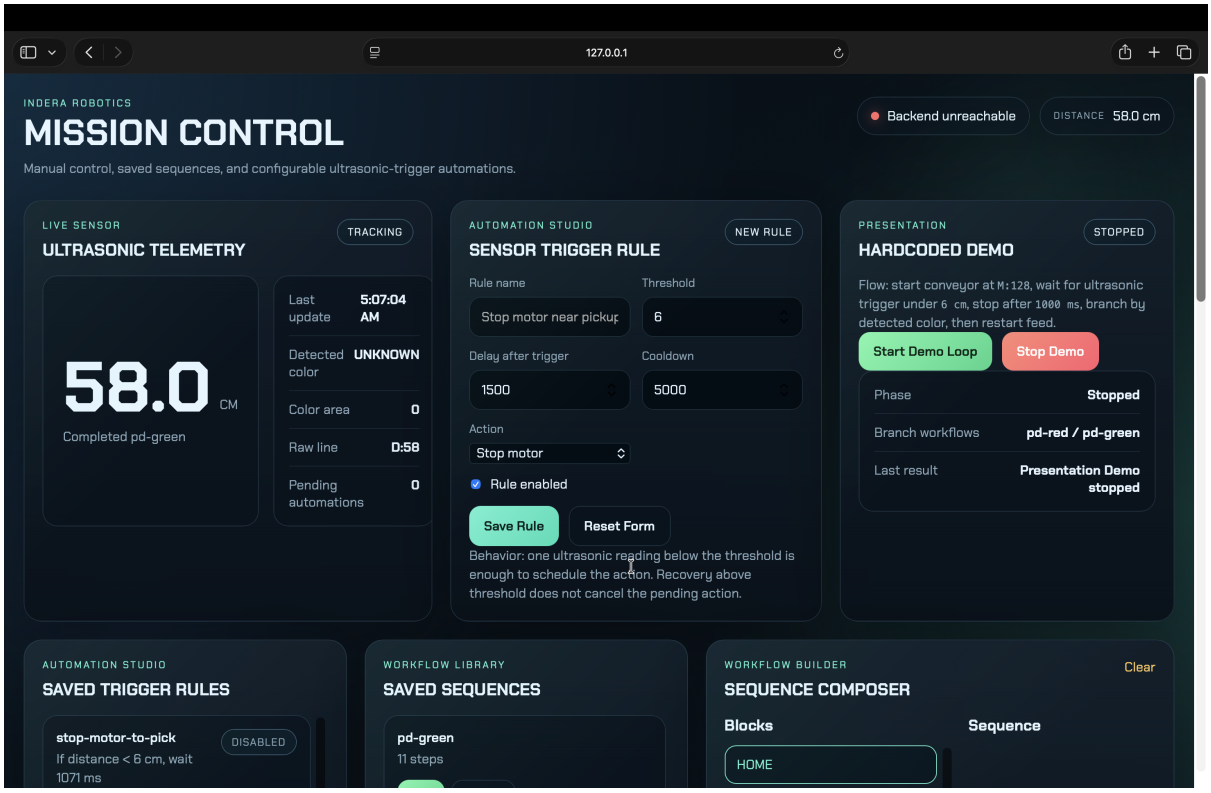


Figure 2: Sensor Display, Triggers and Demo

6.5. Stage 5: Machine-Side Status Display

To improve observability, an I2C LCD was added to Arduino 1. The backend can send compact status banners to the LCD so the system displays useful execution state even without looking at the browser window.

6.6. Stage 6: Persistent Runtime State

As the system grew more complex, frontend-only state became unsafe and insufficient. We therefore implemented a backend-owned runtime state store in `runtime_state.json`. This stores execution state, last commanded joints, sensor values, demo state, and recent commands.

6.7. Stage 7: Color-Aware Demonstration Loop

Finally, the system was extended into a hardcoded demonstration loop where an object is transported on the conveyor, detected by ultrasonic sensing, classified by computer vision color detection, and routed into the correct branch of behavior.

7. Serial Communication Design

The backend and Arduinos communicate using lightweight text commands over serial at 9600 baud.

Command	Meaning	Example
`B:angle`	Set base servo angle	`B:90`
`S:angle`	Set shoulder servo angle	`S:120`
`E:angle`	Set elbow servo angle	`E:145`
`W:angle`	Set wrist pitch angle	`W:90`
`R:angle`	Set wrist roll angle	`R:90`
`G:angle`	Set gripper angle	`G:120`
`M:speed`	Set conveyor motor speed	`M:-200`
`H`	Home robot section	`H`
`D:distance`	Ultrasonic reading from Arduino 1	`D:5`
`LCD:ms:text`	LCD banner/status message	`LCD:2500:MOTOR STOPPED`

Table 3: Serial protocol used in the system

8. Arduino Firmware Design

8.1. Arduino 1 Firmware

`main_1.ino` performs the following tasks:

- receives servo commands for base, shoulder, and elbow,
- homes the structural joints,
- periodically measures ultrasonic distance,
- publishes ultrasonic data over serial,
- drives the I2C LCD,
- receives LCD banner commands from the backend.

8.2. Arduino 2 Firmware

`main_2.ino` performs the following tasks:

- receives servo commands for wrist pitch, wrist roll, and gripper,
- receives conveyor motor speed commands,
- homes the end-effector section.

8.3. Firmware Philosophy

We intentionally kept firmware logic relatively lean. The firmware focuses on:

- direct actuation,
- sensor reading,
- simple command interpretation.

Higher-level logic such as sequencing, conditional branching, and runtime orchestration is done in Python.

9. Backend Design

9.1. Why the Backend Is the Control Brain

The backend is responsible for almost all high-level intelligence:

- routing commands to the correct Arduino,
- normalizing and executing workflows,
- scheduling trigger-based automations,
- managing demo mode,
- maintaining persistent runtime state,
- reading color information from the camera pipeline.

This design made the system significantly easier to evolve during development because behavioral changes could be made in Python without reflashing firmware repeatedly.

9.2. Important Backend Endpoints

Endpoint	Purpose
`/send_command`	Send a manual joint or motor command
`/home`	Home both robot sections
`/positions`	Fetch saved positions
`/save_position`	Save a new named pose
`/workflows`	Fetch saved workflows
`/save_workflow`	Save a workflow
`/run_workflow`	Run a saved workflow
`/automations`	Fetch saved automations
`/save_automation`	Save an automation rule
`/state`	Fetch the complete runtime state snapshot
`/demo_mode/start`	Start the hardcoded demo loop
`/demo_mode/stop`	Request demo loop stop

Table 4: Major backend API endpoints

10. Persistent Runtime State

One of the important engineering corrections during development was to make the backend, not the browser, the owner of runtime state.

10.1. Why This Was Necessary

Frontend-only memory caused a serious problem: old commanded state could be reapplied in ways that did not necessarily reflect the robot's real physical pose after interruptions or restarts. For an open-loop robotic arm, this is unsafe.

10.2. What Is Stored

`runtime_state.json` stores:

- connection status of both Arduinos,
- last commanded joint targets,
- pose trust status,
- ultrasonic sensor values,
- detected color and color confidence surrogate (area),
- workflow execution status,
- automation scheduling state,
- demo mode status and phase,
- recent command history.

10.3. Important Safety Idea

Persisted state is useful for software synchronization, but it is not blindly treated as physical truth. In other words, the software tracks the last commanded state, not guaranteed real pose feedback.

11. Computer Vision Color Detection

11.1. Why Camera-Based Detection Was Used

The system initially explored onboard color-sensor logic, but color detection was shifted to the host side because:

- camera-based tuning is easier,
- HSV thresholds are easier to modify in software,
- repeated firmware reflashing is avoided,

- backend integration with sorting logic is simpler.

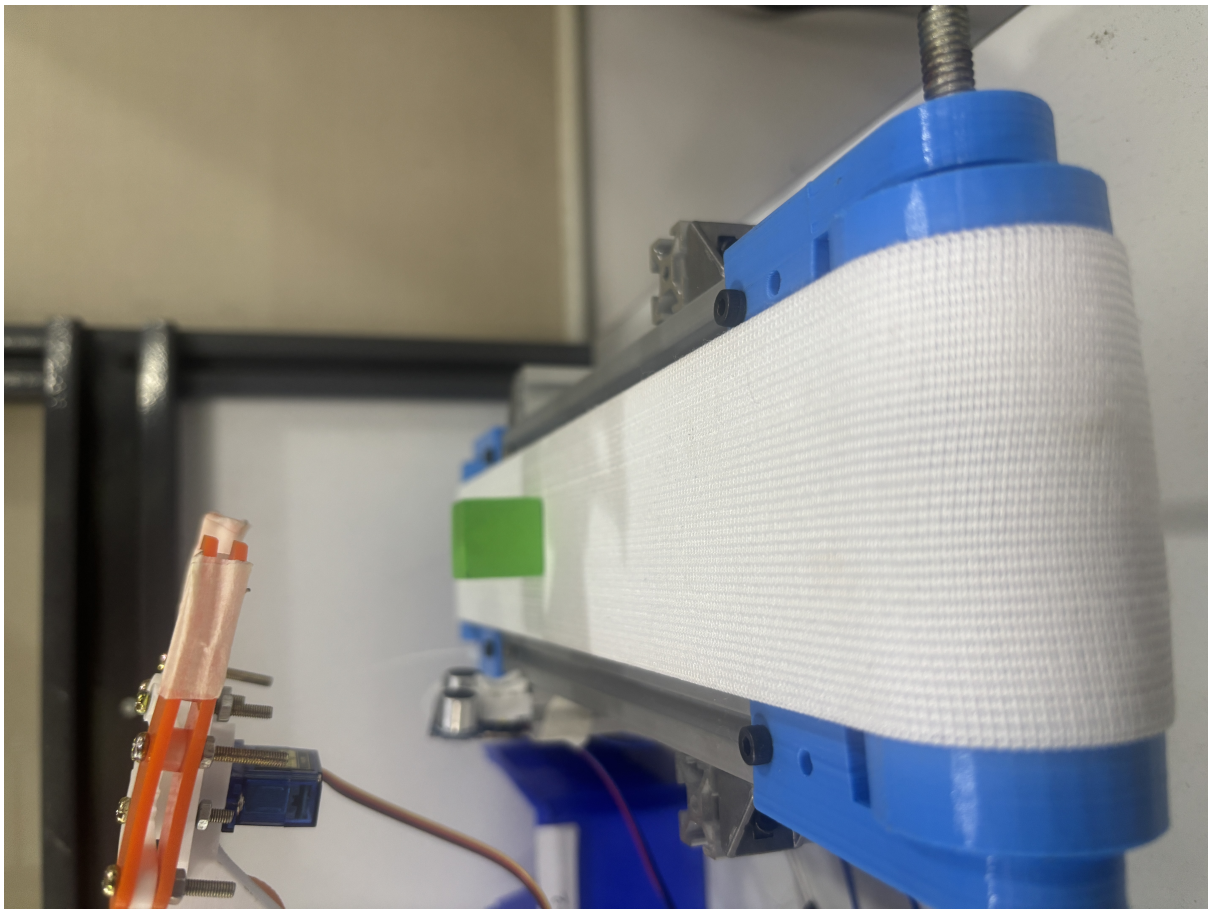


Figure 3: Color Detection Camera View

11.2. Implementation

The file `cv_detector.py` defines HSV ranges and determines the dominant detected color within a center region of interest (ROI). This module is used by:

- the backend camera monitoring thread,
- the standalone preview tool `cv.py`.

11.3. Detected Colors

The detector supports:

- red,
- green,
- blue,
- yellow,
- black,
- white.

11.4. OpenCV-Based Flow

```
frame -> ROI extraction -> HSV conversion -> mask generation
     -> contour extraction -> largest valid contour
     -> dominant color label -> backend runtime state
```

Listing 1: Simplified computer vision flow

12. Workflow System

12.1. Motivation

A robotic arm becomes much more useful when repeatable behaviors can be authored once and replayed multiple times. Workflows provide this capability.

12.2. Configurable Positions

Positions are saved snapshots of the current joint targets. Examples include:

- pick pose,
- lift pose,
- drop pose,
- home pose.

12.3. Workflow Step Types

Step Type	Purpose
`move`	Move to a saved pose
`wait`	Pause execution for a fixed duration
`motor_run`	Run conveyor motor with speed-dependent stop timing

Table 5: Supported workflow step types

12.4. Example Workflow Representation

```
[
  {"type": "move", "name": "to-pick"},
  {"type": "wait", "duration": 1.0},
  {"type": "motor_run", "speed": 128, "stop_factor": 150000, "wait_ms": 1171},
  {"type": "move", "name": "to-drop-red"}
]
```

Listing 2: Example workflow structure

12.5. Speed-Dependent Motor Step

To make workflows more expressive, we introduced a motor step where stop timing depends on motor speed. The backend computes:

$$wait\_ms = \frac{150000}{|speed|}$$

This allows a single workflow definition to remain meaningful across different conveyor speeds.

13. Trigger-Based Automation

13.1. Goal

The purpose of automation rules is to let sensor events cause behavior without operator intervention.

13.2. Current Trigger Source

The current trigger source is the ultrasonic sensor. Example rule format:

- if distance is below threshold,
- wait a configured delay,
- then stop motor / home arm / run workflow.

13.3. Why This Matters

This architecture moved the system from pure teleoperation to event-driven behavior, which is the first step toward autonomous sorting.

14. LCD Status System

The LCD provides local observability on the robot side. This is useful during demonstrations because the machine can communicate its internal state even when the audience is not looking at the laptop screen.

14.1. Displayed Information

The LCD is used to display:

- ultrasonic detection state,
- workflow execution banners,
- demo mode status,
- automation countdowns,
- motor stop / homing messages.

14.2. Backend-to-LCD Command Example

```
command = f"LCD:{duration_ms}:{compact_lcd_text(message)}\n"
```

Listing 3: Example LCD status message from backend

15. Hardcoded Demonstration Loop

15.1. Why a Hardcoded Demo Was Added

While configurable workflows and automations are important, a demonstration requires repeatability and minimal operator burden. Therefore, we added a hardcoded loop tailored to the final presentation behavior.

15.2. Hardcoded Flow

1. Start conveyor at `M:128`
2. Wait until ultrasonic distance is below 6 cm
3. Wait 1000 ms
4. Stop the conveyor
5. Read the current color from the backend state
6. If red: run `pd-red`
7. If green: run `pd-green`
8. Otherwise: reverse motor at `M:-200` for 1000 ms
9. Restart conveyor
10. Repeat until stopped

15.3. Branching Logic

This was the first point at which the project genuinely behaved like a sorting system rather than a remote-controlled arm.

```
if color_name == "RED":
    run pd-red
elif color_name == "GREEN":
    run pd-green
else:
    reverse motor briefly and reject
```

Listing 4: Simplified demonstration branch logic

16. Frontend Dashboard

16.1. Purpose

The dashboard was designed to be both operationally useful and visually presentable. It brings together:

- manual control,
- saved positions,
- workflow creation,
- automation configuration,
- sensor telemetry,
- demo mode status.

16.2. Important Design Choice

The browser is intentionally not treated as the control brain. Instead, it is an operator-facing client that displays and manipulates backend-owned state.

17. Key Implementation Snippets

17.1. Serial Routing Logic

```
def route_serial_for_command(cmd_id):
    cmd_id = cmd_id.upper()
    if cmd_id in ["B", "S", "E"]:
        return ser1, "Arduino 1"
    if cmd_id in ["W", "R", "G", "M"]:
        return ser2, "Arduino 2"
    return None, None
```

Listing 5: Routing a command to the correct Arduino

17.2. Motor Stop Timing Formula

```
def compute_motor_wait_ms(speed, stop_factor=150000):
    return max(1, int(stop_factor / abs(speed)))
```

Listing 6: Speed-dependent motor wait time

17.3. Runtime State Persistence

```
temp_path = f"{self.path}.tmp"
with open(temp_path, "w") as file:
    json.dump(self.state, file, indent=4)
os.replace(temp_path, self.path)
```

Listing 7: Atomic JSON state persistence

18. Engineering Challenges and Corrections

18.1. Challenge 1: State Ownership

At one point, the software attempted to treat persisted joint values as though they were the real robot state. This was corrected because the arm is open-loop and has no encoder feedback. The correct interpretation is:

- persisted state = last commanded software state,
- not guaranteed physical truth.

18.2. Challenge 2: Color Classification Placement

Color logic initially drifted toward embedded-side handling. This was corrected by moving color interpretation to the backend / CV layer, where it is easier to tune and more natural to integrate with sorting logic.

18.3. Challenge 3: Demonstration Reliability

Generic features are useful, but live demonstrations require deterministic presentation behavior. Therefore, a hardcoded demo loop was added in parallel with the configurable workflow engine.

19. Results

The software system now supports:

- browser-based robot control,
- two-Arduino serial orchestration,
- configurable positions,
- configurable workflows,
- trigger-based ultrasonic automation,
- persistent runtime state,
- local LCD execution visibility,

- camera-based color detection,
- a complete hardcoded sorting demonstration loop.

In practical terms, the Indera base robot was transformed from a mechanical platform into a software-driven robotic sorting demonstrator.

20. Future Scope

The current system already supports color-aware branching, but the architecture is designed for further expansion. Future work can include:

- full configurable branch-on-color workflow steps,
- more robust CV calibration under variable lighting,
- sensor fusion between vision and distance,
- pick success verification,
- inventory counting and logging,
- additional trigger sources such as PIR or beam-break sensing,
- richer machine feedback through LEDs or stacked indicators.

21. Conclusion

This project demonstrates how a base robotic arm can be meaningfully transformed through software architecture. The primary contribution is not only that the robot moves, but that it does so through a layered, modular, and extensible control system. The use of two Arduinos, a Python orchestration backend, persistent runtime state, configurable workflows, sensor-triggered actions, LCD-based status visibility, and camera-driven color detection collectively created a much more capable platform than the original base machine.

The work shows a clear progression from manual actuation to structured automation and finally to branch-based demonstration behavior. In that sense, the project successfully bridges low-level mechatronics and higher-level software systems engineering.