Duke University
Department of Mechanical Engineering & Materials Science (MEMS)

Final Project

RoboBarista

An Autonomous Robotic Barista System

Submitted by

Team 7

Alexey Khotimsky (Ph.D. in MEMS)
Mohammad Afrazi (Ph.D. in MEMS)
Celia Pan (M.S. in MEMS)
Yongjae Lee (M.S. in MEMS)
Zhe Feng (M.S. in MEMS)

Course: ECE 383 / ME 555 – Introduction to Robotics
Instructor: Dr. Siobhan Rigby Oca

December 08, 2025

Contents

List of Figures

1 Introduction

The deployment of service robots in unstructured environments requires capabilities beyond simple pick-and-place operations. Specifically, tasks involving the manipulation of fluids or granular materials demand precise, dynamic trajectory control to ensure spill-free transport and even distribution. This project explores the development of an autonomous "Robotic Barista" system designed to bridge the gap between static automation and dynamic human service tasks.

Our system utilizes a **Kinova Gen3 Lite** manipulator driven by a **ROS 2** (Robot Operating System) architecture. The primary objective was to implement a closed-loop control system that allows a human operator to command specific pouring profiles using natural hand gestures. By integrating computer vision with real-time motion planning, we demonstrate a robust framework for Human-Robot Collaboration (HRC).

The core technical contribution of this work is the development of a **parametric trajectory generator** for the pouring motion. Rather than relying on static waypoints, the system calculates a continuous circular path—mimicking the technique of a professional barista—to ensure uniform extraction and distribution of the coffee grounds. This report details the kinematic derivation, software architecture, and experimental validation of the system.

In our physical experiments, we use uncooked rice as a granular stand-in for coffee to simplify cleanup and enable precise gravimetric measurements on the scale.

Figure 1: RoboBarista at Duke! (The image created by ChatGPT and edited by the team.)

2 Documentation of Robot Actions

2.1 System Architecture

The system integration is illustrated in Figure 2. The communication pipeline flows from the vision sensors to the central supervisor node, which dispatches motion tasks to the manipulator.

In the perception layer, a camera stream is processed by a YOLOv8-based gesture recognizer that publishes discrete commands on the `/gesture` topic. The `GestureControl` node subscribes to this topic and selects either the *Simple Gravimetric Pour* or the *Barista Style Circular Pour* behavior node. Both pouring nodes use MoveIt2 to plan and execute Cartesian motions for the Kinova Gen3 Lite arm.

For the gravimetric pour, we additionally integrate a digital scale composed of a load cell, an amplifier board, and an Arduino microcontroller. The Arduino streams the measured weight over a serial connection to a ROS 2 node, which publishes the real-time mass on a `/weight` topic. The SimplePourBot monitors this topic during the slow tilting motion and stops pouring as soon as the measured mass crosses a 100 g threshold, then returns the cup to an upright pose. This closes the loop around the poured mass, while the barista mode remains open loop and focuses on generating a smooth circular trajectory without using the weight feedback.

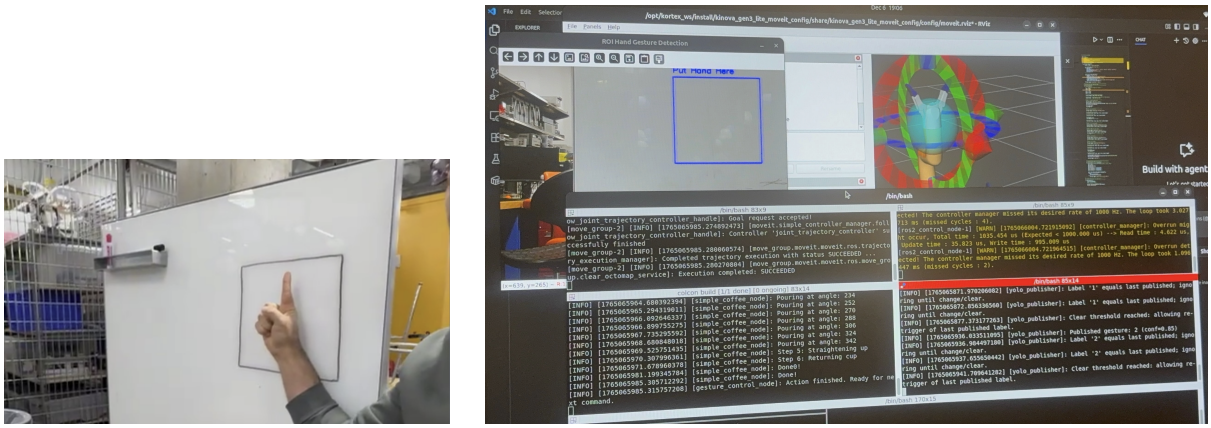

Figure 2: System-level view of RoboBarista.
Left: System diagram showing data flow: Camera → YOLO → Gesture Control → MoveIt2.
Right: runtime screenshot showing ROI hand detection, RViz visualization, and ROS 2 console logs that confirm the gesture-triggered pouring sequence.

2.2 Visual Perception and Gesture Recognition

2.2.1 Model Selection and Dataset

For the gesture recognition task, we selected the YOLOv8n architecture. This model was chosen for its lightweight footprint and high inference speed, which is essential for real-time robotic interaction without significant latency.

We utilized an open-source Hand Gesture Recognition Dataset sourced from Open Source Toolkit[1], which contains labeled images for numerical gestures ranging from 0 to 9. The model was trained on the full dataset to ensure robust feature extraction and generalization capability across all hand signs. For this specific application, we implemented a filtering logic in the ROS node, where the system exclusively monitors for classes “1” and “2”, mapping them to the robot’s “simple Pour (Simple Gravimetric Pour)” and “Barista Pour (Barista Style Circular Pour)” modes respectively.

[1]Dataset available at: https://gitcode.com/open-source-toolkit/2c695

2.2.2 Training Configuration

The model was trained using the Ultralytics framework via a transfer learning approach. We initialized the model with pretrained COCO weights (yolov8n.pt) to leverage learned feature extractors, significantly reducing the training time required for our custom dataset.

The training parameters were configured as follows:

- Epochs: 30
- Batch Size: 16
- Image Size: 640 × 640 pixels
- Optimizer: SGD (Stochastic Gradient Descent)
- Hardware: Training was performed on a CPU-based environment

2.2.3 Inference and ROI Optimization

A key challenge in computer vision for open environments is background noise (e.g., detecting a face or a hand in the background as a command). To mitigate this, we implemented a Region of Interest (ROI) logic in the inference script.

Instead of processing the entire camera frame, the system defines a specific active zone (200x200 pixels) located in the top-right quadrant of the video feed. The workflow is as follows:

- Crop: The script extracts only the ROI area from the main frame.
- Inference: The YOLOv8 model runs detection only on this cropped section.
- Mapping: If a gesture is detected with a confidence score above 0.5, the local coordinates are mapped back to the global frame for visualization.

This approach effectively acts as a safety filter, ensuring the robot only responds when the user deliberately places their hand in the designated "command zone."

2.3 Granular Description of Efforts

The robot's operation is divided into four distinct phases:

1. **Initialization Phase:** The robot moves to a safe "Home" configuration, opens the gripper, and waits in an idle state with the cup placed on the table.
2. **Perception & Command Phase:** The vision system monitors the region of interest and publishes a gesture label on the `/gesture` topic. Upon receiving a valid command, the `GestureControl` node selects the corresponding pouring behavior, and the arm computes inverse kinematics to move to a pre-grasp pose and securely grasp the cup.
3. **Dispensing Phase:** After grasping the cup, the system executes one of two pouring behaviors:
    - *Simple Gravimetric Pour (Command "1"):* The arm moves the cup above the bowl and calls a slow-tilt routine that interpolates between the upright and pouring orientations over 20 small quaternion steps. During this motion, the system reads the weight from the load cell and Arduino-based scale in real time. When the measured mass crosses a 100 g threshold, the node stops the tilt, returns the cup to an upright pose, and prepares to place it back on the table.

- *Barista Style Circular Pour (Command "2"):* The arm tilts the cup once to a fixed angle and then follows a parametric circular trajectory of radius $r = 3\,\text{cm}$ above the bowl. This mode ignores the scale feedback and focuses on producing a smooth, visually pleasing circular pour pattern similar to a manual barista pour.

4. **Return Phase:** The robot brings the cup back to the table, opens the gripper to release it, and then returns to the "Home" pose, resetting the system for the next gesture command.

2.4 Robot Execution Image

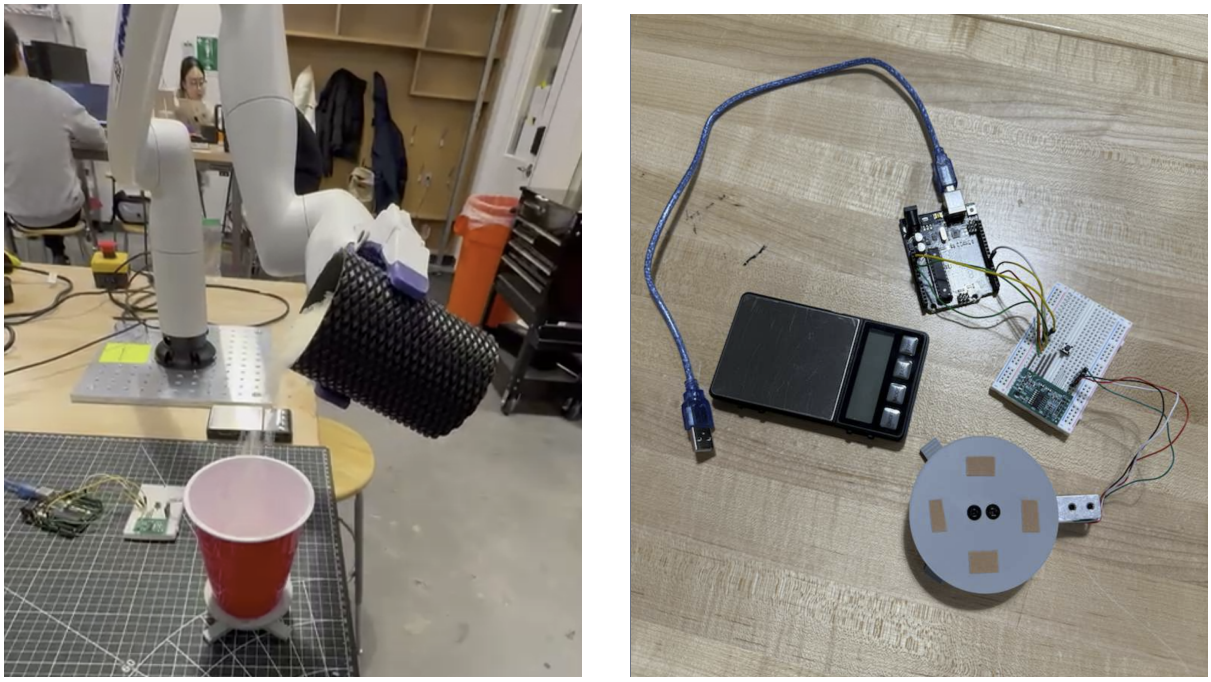

Figure 3: RoboBarista hardware setup.
Left: the Kinova Gen3 Lite performing the parametric circular pouring task.
Right: the custom digital scale with a load cell and Arduino-based electronics used for closed-loop gravimetric pouring.

2.5 Code Explanation

The software is architected using a modular **Supervisor–Worker** pattern implemented through three ROS 2 components responsible for perception, supervision, and execution. The primary control signal flows through the `/gesture` topic, while an additional `/weight` topic connects the digital scale node to the gravimetric pouring behavior. We detail each component below.

2.5.1 YOLO Detection Node (Perception Layer)

The YOLO gesture recognizer is implemented as an independent ROS 2 publisher node (`detect_pub.py`). Rather than invoking robot behaviors directly, it performs real-time inference on the camera feed and publishes a filtered gesture label (e.g., "pour", "barista") on the `/gesture` topic using `std_msgs/String`.

To ensure that only intentional and stable human gestures trigger robot motion, the node implements several lightweight filtering mechanisms:

- **Multi-frame debouncing**: A gesture is accepted only after appearing consistently for several frames.

- **Gesture-change rule**: The node stores the last published label and only republishes when a new, different gesture is detected. This prevents repeated activations while the user holds the same gesture steady.
- **Clear-frame reset**: After a sequence of "no-detection" frames, the previous gesture is cleared, allowing the same gesture to be triggered again later.

These mechanisms operate above the ROS 2 publish–subscribe layer and ensure that only stable, meaningful gesture commands propagate to the supervisory control layer.

2.5.2 GestureControl (Supervisor Node)

This is the central decision-making node. It subscribes to the `/gesture` topic and acts as a dispatcher. Upon receiving a command string, it selects the appropriate behavior:

- **Input "1" / "pour"**: Triggers the `SimplePourBot` thread for a standard static pour.
- **Input "2" / "barista"**: Triggers the `BaristaPourBot` thread for the advanced circular pouring task.

It utilizes a `MultiThreadedExecutor` to spawn these actions in background threads, ensuring that the vision callbacks are never blocked by robot motion. A busy-lock mechanism ensures that only one motion routine executes at a time, even though all components reside in the same ROS package.

2.5.3 Motion Bots (Execution Layer)

Two motion scripts implement the robot's physical behaviors:

- `coffee_pourer.py`: Implements the closed-loop gravimetric pour. The arm moves the cup above the bowl, then calls a slow tilt routine that interpolates between the upright and pouring orientations over 20 small quaternion steps while monitoring the `/weight` topic.
- `coffee_pourer_v2.py`: Implements the open loop barista-style circular pour, where the cup is tilted once to a fixed angle and then follows a circular trajectory.

The circular pour routine computes a sequence of 20 waypoints according to:

$$x_i = x_c + r\cos(\theta_i), \qquad y_i = y_c + r\sin(\theta_i) \tag{1}$$

From the supervisor node's perspective, both behaviors are encapsulated as worker classes, and the node simply instantiates the appropriate bot class and executes its `run_my_code()` method when triggered.
Overall, this architecture cleanly separates perception, supervisory control, and execution responsibilities. The YOLO node provides stable gesture intent, the GestureControl node handles behavior selection and concurrency, and the motion bots perform the corresponding robotic actions through MoveIt 2 and the gripper interface.

2.5.4 Digital Scale Node

In addition to the motion bots, a dedicated Arduino scale was designed that utilized a load cell and an HX711 amplifier and reads serial data from the Arduino based digital scale and sends the current mass a a serial message to `/weight`. This node parses the weight readings from `/dev/ttyACM0` and provides real-time feedback to the `SimplePourBot`, which uses the stream to stop the slow tilting motion once the 100 g threshold is reached.

2.6 End-Effector Cup Design for Stable Grasping

To ensure a robust grasp during fast motions and circular pouring trajectories, we designed a custom cup tailored to the Kinova Gen3 Lite gripper. We first measured the inner width and finger travel of the gripper and matched the cup diameter so that a fully closed grasp neither crushes the cup nor leaves excessive slack.
Using Autodesk Fusion, we created three surface variants: a smooth-walled cup, a cup with vertical ribs, and a cup with a diamond lattice texture. Each design was 3D printed and tested under real pouring motions to evaluate slip, vibration, and the risk of losing the grasp.
Experiments showed that the smooth cup was prone to slipping, and the vertically ribbed cup reduced slip but still allowed small shifts during dynamic motions. The diamond-patterned cup provided the most stable grasp, maximizing contact area and friction between the gripper pads and the cup surface. As a result, we selected the diamond-patterned design for all final gravimetric and barista pouring trials. Figure 4 shows the CAD models and the corresponding 3D printed prototypes.

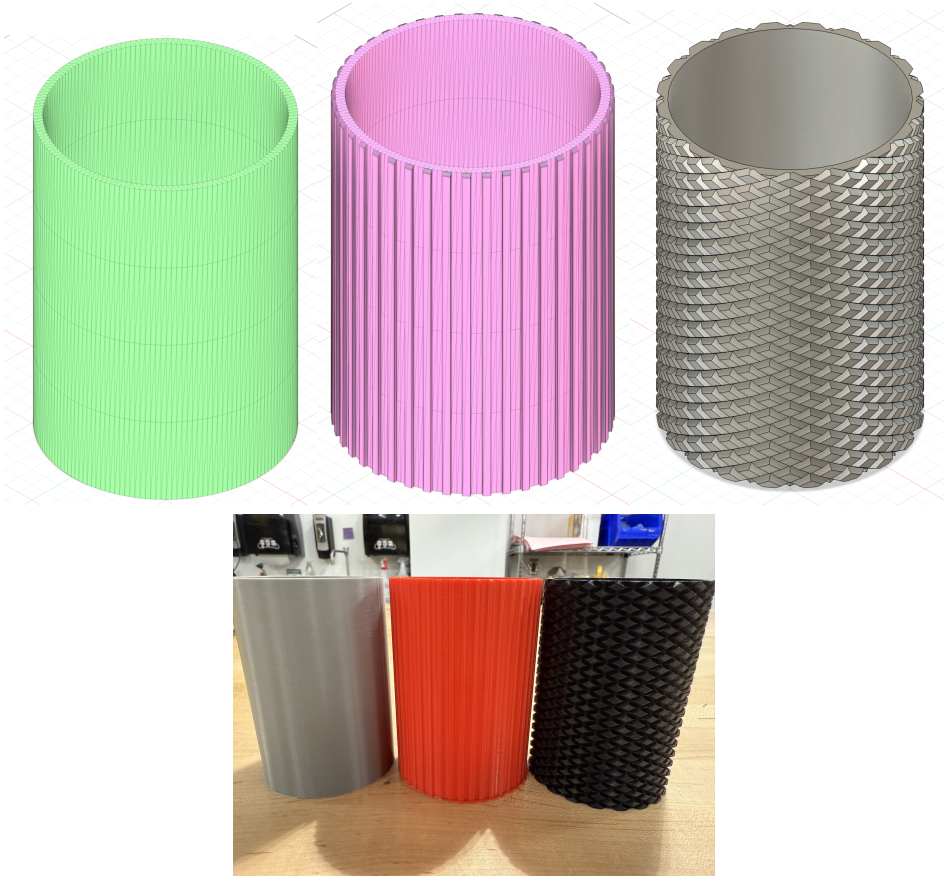

Figure 4: Custom cup designs and prototypes.
Top: CAD models of the smooth, ribbed, and diamond patterned cups.
Bottom: 3D printed prototypes used to evaluate grasp stability. The diamond patterned cup provided the most robust grasp and was selected for all final trials.

3 Human Integration

The system is designed for high-risk or sterile environments where physical contact is undesirable. The human operator interacts via a **non-contact visual interface**.
As illustrated in Figure 5, the interaction follows a safety protocol:

1. **Command Initiation:** Operator displays a hand gesture.
2. **Verification:** The node filters noise by requiring the signal to persist for 1 second.
3. **Execution & Lockout:** Once triggered, the system enters a `BUSY` state, ignoring all inputs until the task is complete.

4 Challenges

One significant challenge we faced was that YOLO could not run inside the default Docker environment provided for the course. Many required dependencies for ultralytics and OpenCV were missing, and direct installation attempts resulted in version conflicts. To resolve this, we created an extended Dockerfile based on the course image and manually added the necessary system libraries and Python packages. After rebuilding the container, we were finally able to run YOLO detection both offline and through the camera.

Another practical challenge involved the behavior-triggering mechanism for gesture control. Since the gesture topic continuously publishes the same detection result frame-by-frame, the robot would repeatedly execute the same action unless we introduced a trigger condition. To address this, we implemented a state-change mechanism: the robot only executes a motion when the detected gesture changes relative to the previous frame. This ensured stable and predictable control when integrating the vision system with the Kinova Gen3 arm.

A third challenge arose in the pouring dynamics and accuracy. In our initial implementation, the arm moved directly from the upright pose to a 90° tilt in a single motion. This caused a sudden surge of rice, often spilling outside the bowl and overshooting the 100 g target by a large margin, with final readings frequently in the 110–120 g range. We addressed this by redesigning the motion as a multi-step slow tilt. The cup orientation is interpolated between the start and end quaternions over 20 small steps while monitoring the weight. This modification made the motion visibly smoother and reduced the typical overshoot, so that the final mass after triggering the 100 g threshold stop was around 103 g.

5 Future Work

If provided more time, we would extend the gravimetric closed loop control that we implemented for the simple pour to the barista style circular pour. In the simple mode, a load cell, amplifier, and Arduino based scale already stream weight to ROS, and the robot stops pouring and returns the cup once the measured mass crosses a 100 g threshold while executing a 20 step, slowly tilting motion. However, in the barista pour mode the system still runs open loop. The arm tilts the cup once to a fixed angle and follows a circular trajectory without using the scale feedback, which can lead to over or under pouring. As future work, we would integrate the same weight feedback into the barista pour, modulate the circular path and tilt angle based on the real time mass reading, and redesign the motion as a multi step trajectory so that the circular pour remains smooth while stopping close to the desired target mass.

6 Takeaways

- **Concurrency in Robotics:** We learned that for any real-time interactive robot, single-threaded execution is insufficient. Mastering ROS 2 `Executors` and `CallbackGroups` is essential for creating responsive systems.
- **Parametric Trajectories:** We learned that simple point-to-point motion (PTP) is often unnatural for manipulation tasks. Generating trajectories using mathematical functions

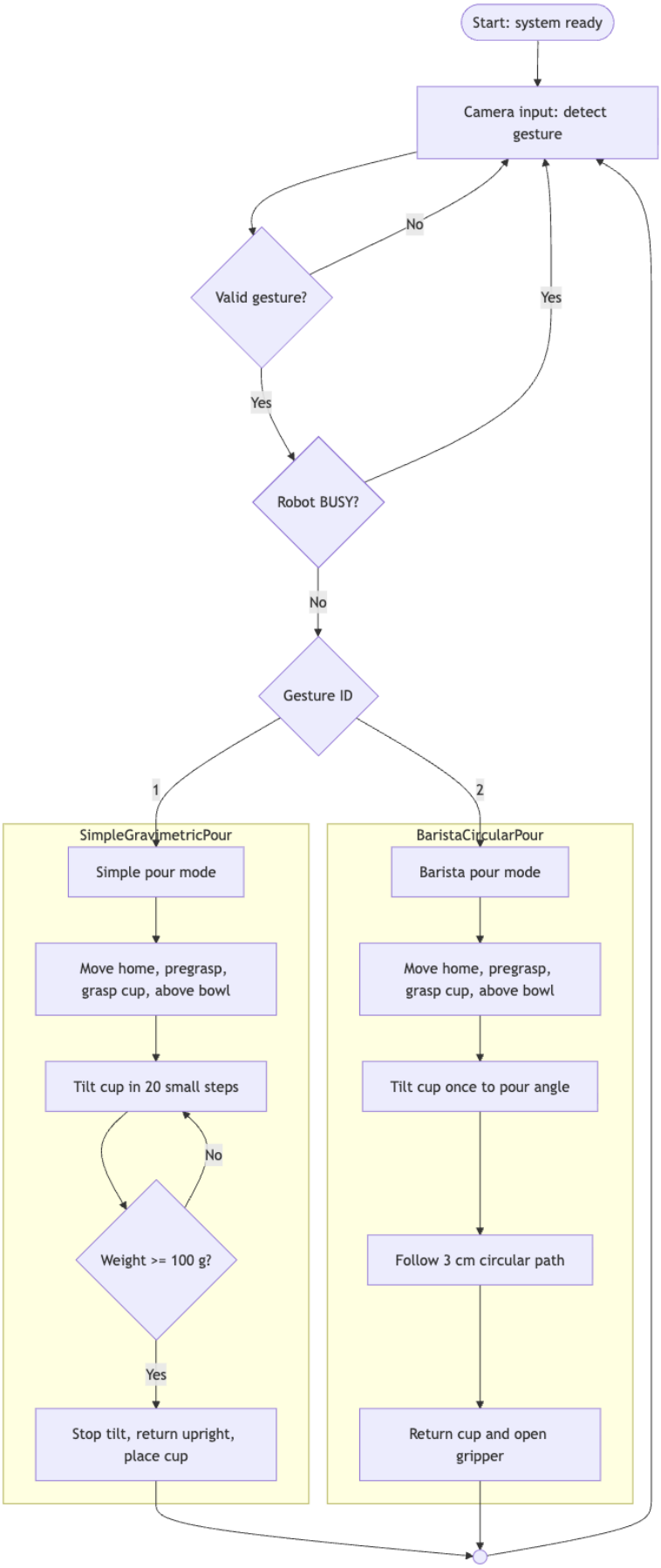

Figure 5: Flowchart of the Human-Robot Interaction Logic.

(like our circular pour) yields much smoother and more effective physical results than hard-coded waypoints.

- **Feedback and End-Effector Design:** We learned that achieving precise pouring requires both closed loop feedback and careful end effector design. The gravimetric pour showed how real time weight measurements reduce overshoot, and the custom diamond patterned cup significantly improved grasp stability during dynamic motions.

A Code Appendix

A.1 Gesture Detection Node

This node serves as the perception front-end. It performs real-time gesture recognition and publishes the resulting gesture label as a string message on the `/gesture` topic

Listing 1: Detect and Publish

```
1
2  #!/usr/bin/env python3
3  # predict_pub.py - debounced publisher that only republishes when label
       changes
4  import os
5  import sys
6  import time
7  import cv2
8  from ultralytics import YOLO
9
10 # ROS2 imports
11 import rclpy
12 from rclpy.node import Node
13 from std_msgs.msg import String
14
15 # -------------------------
16 # Config (can override via env)
17 # -------------------------
18 DEFAULT_MODEL = r"weights/best.pt"
19 MODEL_PATH = os.environ.get("YOLO_MODEL_PATH", DEFAULT_MODEL)
20 YOLO_VIDEO = os.environ.get("YOLO_TEST_VIDEO", "")  # if set, use video
       file
21 CAMERA_INDEX = int(os.environ.get("YOLO_VIDEO_INDEX", "0"))
22 CONF_THRESHOLD = float(os.environ.get("YOLO_CONF", "0.5"))
23 ROI_WIDTH = int(os.environ.get("YOLO_ROI_W", "200"))
24 ROI_HEIGHT = int(os.environ.get("YOLO_ROI_H", "200"))
25 DEBOUNCE_FRAMES = int(os.environ.get("YOLO_DEBOUNCE", "3"))
26 CLEAR_FRAMES = int(os.environ.get("YOLO_CLEAR_FRAMES", "8"))  #
       consecutive "no-detection" frames to clear last_published_label
27 # Optional pause (not required with this method)
28 PAUSE_SEC = float(os.environ.get("YOLO_PAUSE_SEC", "0.0"))
29 # -------------------------
30
31 def load_model(path):
32     try:
33         print(f"[INFO] Loading YOLO model from: {path}")
34         model = YOLO(path)
35         return model
36     except Exception as e:
37         print(f"[ERROR] Could not load model: {e}", file=sys.stderr)
38         raise
39
40 class YoloPublisher(Node):
41     def __init__(self, model, video_source, camera_index):
42         super().__init__('yolo_publisher')
43         self.pub = self.create_publisher(String, 'gesture', 10)
44         self.model = model
45         self.cap = None
46         if video_source:
```

```
47              self.get_logger().info(f"Opening␣video␣file:␣{video_source}
                    ")
48              self.cap = cv2.VideoCapture(video_source)
49          else:
50              self.get_logger().info(f"Opening␣camera␣index:␣{
                    camera_index}")
51              self.cap = cv2.VideoCapture(camera_index)
52
53          if not self.cap or not self.cap.isOpened():
54              self.get_logger().error("Unable␣to␣open␣video/camera␣source
                    .")
55              raise RuntimeError("Unable␣to␣open␣video/camera␣source.")
56
57          # state for debounce and change-only logic
58          self.last_seen_label = None         # last label seen (for
                debounce)
59          self.debounce_counter = 0
60
61          self.last_published_label = None    # last label we actually
                published
62          self.clear_none_counter = 0         # counts consecutive frames
                with no detection (to allow re-publish same label)
63
64      def busy_check_and_publish(self, label, conf):
65          # -------------------------
66          # Called when we have a debounced label candidate.
67          # Publish only if label != last_published_label.
68          # After publishing, reset clear counter.
69          # -------------------------
70          if label is None:
71              return False
72
73          # If label equals last published, ignore (require clear or
                different label)
74          if self.last_published_label == label:
75              self.get_logger().info(f"Label␣'{label}'␣equals␣last␣
                    published;␣ignoring␣until␣change/clear.")
76              return False
77
78          # Publish now
79          msg = String()
80          msg.data = label
81          self.pub.publish(msg)
82          self.get_logger().info(f"Published␣gesture:␣{label}␣(conf={conf
                :.2f})")
83          # update state
84          self.last_published_label = label
85          self.clear_none_counter = 0
86          return True
87
88      def process_frame_and_publish(self):
89          ret, frame = self.cap.read()
90          if not ret:
91              # if video ended, rewind (useful for testing with short
                    videos)
92              try:
93                  self.cap.set(cv2.CAP_PROP_POS_FRAMES, 0)
94              except Exception:
```

```
                pass
            return

        h_frame, w_frame = frame.shape[:2]
        center_x = int(w_frame * 0.75)
        center_y = int(h_frame * 0.25)
        roi_x1 = center_x - (ROI_WIDTH // 2)
        roi_y1 = center_y - (ROI_HEIGHT // 2)
        roi_x1 = max(0, min(roi_x1, w_frame - ROI_WIDTH))
        roi_y1 = max(0, min(roi_y1, h_frame - ROI_HEIGHT))
        roi_x2 = roi_x1 + ROI_WIDTH
        roi_y2 = roi_y1 + ROI_HEIGHT

        # draw guide box
        cv2.rectangle(frame, (roi_x1, roi_y1), (roi_x2, roi_y2), (255,
            0, 0), 2)
        cv2.putText(frame, "Put␣Hand␣Here", (roi_x1, roi_y1 - 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 0, 0), 2)

        roi_frame = frame[roi_y1:roi_y2, roi_x1:roi_x2]

        # Run inference (single ROI)
        try:
            results = self.model(roi_frame, stream=True, verbose=False)
        except Exception as e:
            self.get_logger().error(f"Inference␣error:␣{e}")
            return

        # Get best detection (label + confidence) or None
        best_label = None
        best_conf = 0.0
        for result in results:
            for box in result.boxes:
                conf = float(box.conf[0].item())
                if conf < CONF_THRESHOLD:
                    continue
                cls_idx = int(box.cls[0].item())
                name = self.model.names.get(cls_idx, str(cls_idx)) if
                    hasattr(self.model, 'names') else str(cls_idx)
                if conf > best_conf:
                    best_conf = conf
                    best_label = name

        # --- Clear logic: if no detection in this frame, increment
            clear counter
        if best_label is None:
            self.clear_none_counter += 1
        else:
            self.clear_none_counter = 0

        # --- Debounce logic: require DEBOUNCE_FRAMES consecutive same
            label to consider it real
        if best_label is None:
            # reset last seen label and debounce counter
            self.last_seen_label = None
            self.debounce_counter = 0
        else:
            if best_label == self.last_seen_label:
```

```
                self.debounce_counter += 1
            else:
                self.last_seen_label = best_label
                self.debounce_counter = 1

        # When debounced candidate ready:
        if self.debounce_counter >= DEBOUNCE_FRAMES:
            # Only publish if label differs from last published OR if last_published_label is None
            published = self.busy_check_and_publish(self.last_seen_label, best_conf)
            # optional small hard pause after publish
            if published and PAUSE_SEC > 0:
                self.get_logger().info(f"Hard pause for {PAUSE_SEC} seconds after publish.")
                time.sleep(PAUSE_SEC)
            # reset debounce so we don't immediately re-evaluate same frames
            self.debounce_counter = 0
            self.last_seen_label = None

        # If last published label exists and we've seen enough consecutive empty frames, clear it so same label can re-trigger
        if self.last_published_label is not None and self.clear_none_counter >= CLEAR_FRAMES:
            self.get_logger().info("Clear threshold reached: allowing re-trigger of last published label.")
            self.last_published_label = None
            self.clear_none_counter = 0

        # show frame (optional)
        cv2.imshow('ROI Hand Gesture Detection', frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            raise KeyboardInterrupt()

    def shutdown(self):
        try:
            self.cap.release()
        except Exception:
            pass
        cv2.destroyAllWindows()

def main(args=None):
    # load model
    model = load_model(MODEL_PATH)

    # choose video or camera
    video_src = YOLO_VIDEO if YOLO_VIDEO else None

    # init ROS
    rclpy.init(args=args)
    node = YoloPublisher(model=model, video_source=video_src, camera_index=CAMERA_INDEX)

    try:
        node.get_logger().info("Starting detection loop. Press 'q' in window to quit.")
```

```
197             while rclpy.ok():
198                 node.process_frame_and_publish()
199                 rclpy.spin_once(node, timeout_sec=0)
200         except KeyboardInterrupt:
201             node.get_logger().info("Keyboard␣interrupt,␣shutting␣down.")
202         finally:
203             node.shutdown()
204             node.destroy_node()
205             rclpy.shutdown()
206 
207 if __name__ == "__main__":
208     main()
```

A.2 Gesture Control (Supervisor Node)

This node handles the state machine and multi-threading logic.

Listing 2: Gesture Control Logic

```
#!/usr/bin/env python3

import rclpy
import threading
from rclpy.node import Node
from rclpy.executors import MultiThreadedExecutor
from std_msgs.msg import String
import time

# CRITICAL: These files must NOT have their own main() and rclpy.init() blocks.
# They are imported here as service classes (Nodes).
from .coffee_pourer import SimpleCoffeeBot as SimplePourBot
from .coffee_pourer_v2 import SimpleCoffeeBot as BaristaPourBot


class GestureControl(Node):
    # -------------------------
    # Central node that manages robot action nodes and listens for commands
    # From the YOLO vision publisher. Uses MultiThreadedExecutor for stability.
    # -------------------------
    def __init__(self, action_node_executor):
        super().__init__("gesture_control_node")

        # --- State Management ---
        self.busy = False
        self.executor = action_node_executor

        # --- Action Node Instantiation (Created only ONCE) ---
        # These objects are the actual robot control nodes that talk to MoveIt2.
        self.simple_bot = SimplePourBot()
        self.barista_bot = BaristaPourBot()

        # Add the action nodes to the shared Executor so they can process ROS topics
        self.executor.add_node(self.simple_bot)
        self.executor.add_node(self.barista_bot)

        # --- ROS Subscription ---
        # Ensure this topic name matches the one published by your YOLO node
        self.sub = self.create_subscription(String, "gesture", self.gesture_callback, 10)

        self.get_logger().info("GestureControl and Robot Managers are ready.")
        self.get_logger().info("Listening to /gesture topic for commands (1 or 2)...")

```

```
45
46      def gesture_callback(self, msg):
47          """Processes␣incoming␣gesture␣commands."""
48          label = msg.data.strip()
49          self.get_logger().info(f"Received␣gesture:␣{label}")
50
51          if self.busy:
52              self.get_logger().warn("Robot␣busy␣-␣ignoring␣gesture.")
53              return
54
55          # --- Modified Logic: Check for '1' or '2' ---
56          if label == "1":
57              # Pass the run_my_code method of the pre-initialized bot
                    instance
58              self.start_thread(self.simple_bot.run_my_code)
59              self.get_logger().info("Triggering:␣Simple␣Pour␣(V1)")
60
61          elif label == "2":
62              self.start_thread(self.barista_bot.run_my_code)
63              self.get_logger().info("Triggering:␣Barista␣Pour␣(V2)")
64
65          else:
66              self.get_logger().warn(f"Unknown␣gesture:␣{label}")
67
68      # ======================================================
69      # Thread management
70      # ======================================================
71      def start_thread(self, target_fn):
72          """Starts␣the␣robot␣action␣in␣a␣separate␣thread␣to␣prevent␣ROS␣
                callbacks␣from␣blocking."""
73          thread = threading.Thread(target=self.safe_run, args=(target_fn
                ,), daemon=True)
74          thread.start()
75
76      def safe_run(self, target_fn):
77          """Wrapper␣function␣to␣manage␣the␣robot's␣busy␣state."""
78          self.busy = True
79          self.get_logger().info(f"Starting␣action:␣{target_fn.__name__}"
                )
80          try:
81              # Execute the actual motion function
82              target_fn()
83          except Exception as e:
84              self.get_logger().error(f"Error␣executing␣motion:␣{e}")
85          finally:
86              self.busy = False
87              self.get_logger().info("Action␣finished.␣Ready␣for␣next␣
                    command.")
88
89
90  def main(args=None):
91      rclpy.init(args=args)
92
93      # 1. Create MultiThreaded Executor to handle callbacks and MoveIt2
            actions concurrently
94      executor = MultiThreadedExecutor(num_threads=4)
95
96      # 2. Create the main control node (passing the executor)
```

```
    control_node = GestureControl(executor)

    # 3. Add the control node to the executor
    executor.add_node(control_node)

    try:
        # 4. Start the ROS event loop
        executor.spin()
    except KeyboardInterrupt:
        pass
    except Exception as e:
        control_node.get_logger().error(f"Executor error: {e}")
    finally:
        rclpy.shutdown()


if __name__ == "__main__":
    main()
```

A.3 Simple Pour Node (Static)

This node handles the basic Pick-and-Pour logic without trajectory modification.

Listing 3: Simple Coffee Pouring

```
#!/usr/bin/env python3

import time
import threading
import math
import serial
import rclpy
from rclpy.node import Node
from rclpy.executors import MultiThreadedExecutor
from geometry_msgs.msg import Pose
from pymoveit2 import MoveIt2
from pymoveit2.gripper_interface import GripperInterface


class SimpleCoffeeBot(Node):
    def __init__(self):
        super().__init__("simple_coffee_node")

        # --- SERIAL SETUP ---
        # Initialize the serial connection when the node starts
        self.serial_port_name = "/dev/ttyACM0"  # Check this device name
        self.baud_rate = 9600
        self.ser = None

        try:
            self.ser = serial.Serial(self.serial_port_name, self.baud_rate, timeout=1)
            self.get_logger().info(f"Connected to serial device on {self.serial_port_name}")
        except serial.SerialException as e:
            self.get_logger().error(f"Failed to connect to serial: {e}")

        # --- ROBOT SETUP ---
        self.arm = MoveIt2(
            node=self,
            joint_names=["joint_1", "joint_2", "joint_3", "joint_4", "joint_5", "joint_6"],
            base_link_name="base_link",
            end_effector_name="end_effector_link",
            group_name="arm",
        )

        # Gripper setup
        self.gripper = GripperInterface(
            node=self,
            gripper_joint_names=["right_finger_bottom_joint"],
            open_gripper_joint_positions=[0.01],
            closed_gripper_joint_positions=[0.8],
            gripper_group_name="gripper",
```

```
            gripper_command_action_name="/gen3_lite_2f_gripper_controller/gripper_cmd",
            ignore_new_calls_while_executing=False,
        )

        self.get_logger().info("Robot is ready!")

    # ------------------------------------------------------------
    # Quaternion helpers
    # ------------------------------------------------------------
    def lerp_quat(self, q0, q1, t: float):
        """Linear interpolation between two quaternions followed by normalization."""
        q = [q0[i] + t * (q1[i] - q0[i]) for i in range(4)]
        norm = math.sqrt(sum(v * v for v in q))
        if norm == 0.0:
            return q0[:]  # Fallback to start quaternion if something goes wrong
        return [v / norm for v in q]

    def tilt_slowly(self, x, y, z, q_start, q_end, steps: int = 10):
        # ----------------------
        # Gradually tilt the cup from q_start to q_end in multiple small steps.
        # This makes the pouring motion look slow and smooth instead of sudden.
        # ----------------------
        for i in range(1, steps + 1):
            t = i / steps  # interpolation factor from 0 to 1
            q = self.lerp_quat(q_start, q_end, t)
            self.move_to(x, y, z, q)

    # ------------------------------------------------------------
    # Main sequence
    # ------------------------------------------------------------
    def run_my_code(self):
        # --- COORDINATES ---
        cup_x = 0.40
        cup_y = -0.20
        cup_z = 0.0001

        pour_x = 0.40
        pour_y = 0.20
        pour_z = 0.22

        # Orientation: cup held straight
        orient_straight = [0.0, 0.7071, 0.0, 0.7071]

        # Orientation: tilted for pouring (from RViz calibration)
        orient_tilted = [0.5373, 0.4869, 0.4869, 0.4869]  # ~115 degrees

        # ------------------------------------------------
        # STEP 1: Go to a safe home pose and open gripper
        # ------------------------------------------------
        self.get_logger().info("Step 1: Go to home and open gripper")
        self.arm.move_to_configuration(
            joint_positions=[0.0, -0.28, 1.3, 1.0, -1.0, 0.0]
```

```
        )
        self.arm.wait_until_executed()

        self.gripper.open()
        # If you want to be safer, you can also call self.gripper.
            wait_until_executed()
        time.sleep(1.0)

        # ------------------------------------------------
        # STEP 2: Pick up the cup
        # ------------------------------------------------
        self.get_logger().info("Step 2: Picking up the cup")

        # Move above the cup
        self.move_to(cup_x, cup_y, cup_z + 0.15, orient_straight)
        # Move down to the cup
        self.move_to(cup_x, cup_y, cup_z, orient_straight)

        # Close gripper to grab the cup
        self.gripper.close()
        # Optionally: self.gripper.wait_until_executed()
        time.sleep(1.0)

        # Lift the cup up
        self.move_to(cup_x, cup_y, cup_z + 0.15, orient_straight)

        # ------------------------------------------------
        # STEP 3: Move to the pouring position
        # ------------------------------------------------
        self.get_logger().info("Step 3: Moving to the pouring position"
            )
        self.move_to(pour_x, pour_y, pour_z, orient_straight)

        # ------------------------------------------------
        # STEP 4: Pour (tilt slowly and then wait for serial)
        # ------------------------------------------------
        self.get_logger().info("Step 4: Pouring slowly...")
        # Slowly tilt from straight orientation to tilted orientation
        self.tilt_slowly(
            pour_x,
            pour_y,
            pour_z,
            orient_straight,
            orient_tilted,
            steps=20,
        )

        # Wait until the sensor value exceeds a threshold (e.g., weight
            > 100)
        if self.ser and self.ser.is_open:
            self.get_logger().info("Reading serial data... waiting for
                value > 100")
            # Clear any old data from the buffer so we don't read stale
                values
            self.ser.reset_input_buffer()

            while True:
                try:
```

```
                    # Read a line, decode from bytes to string, strip whitespace
                    line = self.ser.readline().decode("utf-8", errors="ignore").strip()

                    if line:
                        try:
                            value = float(line)
                            self.get_logger().info(f"Sensor Value: {value}")
                            if value > 100:
                                self.get_logger().info("Target reached! Stopping pour.")
                                break
                        except ValueError:
                            # Ignore non-numeric messages, but log them for debugging
                            self.get_logger().warn(f"Received non-numeric data: {line}")

                except Exception as e:
                    self.get_logger().error(f"Serial read error: {e}")
                    break
        else:
            self.get_logger().warn("Serial not connected! Defaulting to 2 second wait.")
            time.sleep(2.0)

        # ------------------------------------------------
        # STEP 5: Straighten the cup again
        # ------------------------------------------------
        self.get_logger().info("Step 5: Straightening the cup")
        self.move_to(pour_x, pour_y, pour_z, orient_straight)

        # ------------------------------------------------
        # STEP 6: Return the cup to its original position
        # ------------------------------------------------
        self.get_logger().info("Step 6: Returning the cup")
        # Move above the original cup position
        self.move_to(cup_x, cup_y, cup_z + 0.15, orient_straight)
        # Move down to place the cup
        self.move_to(cup_x, cup_y, cup_z, orient_straight)

        # Open gripper to release the cup
        self.gripper.open()
        # Optionally: self.gripper.wait_until_executed()
        time.sleep(1.0)

        # Move arm up again
        self.move_to(cup_x, cup_y, cup_z + 0.15, orient_straight)

        # Go back to a neutral joint configuration
        self.get_logger().info("Done! Moving back to neutral configuration.")
        self.arm.move_to_configuration(
            joint_positions=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
        )
        self.arm.wait_until_executed()
```

```
203
204     # ---------------------------------------------------------
205     # Helper: move to a Cartesian pose
206     # ---------------------------------------------------------
207     def move_to(self, x, y, z, orientation):
208         """Plan and execute a Cartesian move to the given pose."""
209         pose = Pose()
210         pose.position.x = x
211         pose.position.y = y
212         pose.position.z = z
213         pose.orientation.x = orientation[0]
214         pose.orientation.y = orientation[1]
215         pose.orientation.z = orientation[2]
216         pose.orientation.w = orientation[3]
217
218         plan = self.arm.plan(pose=pose, cartesian=True)
219         if plan:
220             self.arm.execute(plan)
221             self.arm.wait_until_executed()
222         else:
223             self.get_logger().error("Could not find a path!")
224
225
226 # ---------------------------------------------------
227 #  Standard ROS2 boilerplate
228 # ---------------------------------------------------
229 def main(args=None):
230     rclpy.init(args=args)
231     node = SimpleCoffeeBot()
232
233     executor = MultiThreadedExecutor(num_threads=2)
234     executor.add_node(node)
235     thread = threading.Thread(target=executor.spin, daemon=True)
236     thread.start()
237
238     try:
239         # Small delay to let everything start up
240         time.sleep(2.0)
241         node.run_my_code()
242     except KeyboardInterrupt:
243         pass
244     finally:
245         # Close serial port cleanly on exit
246         if node.ser and node.ser.is_open:
247             node.ser.close()
248         rclpy.shutdown()
249         thread.join()
```

A.4 Barista Pour Node (Parametric)

This node implements the circular path planning algorithm.

Listing 4: Barista Circular Path

```
#!/usr/bin/env python3

import time
import threading
import rclpy
import math
from rclpy.node import Node
from rclpy.executors import MultiThreadedExecutor
from geometry_msgs.msg import Pose
from pymoveit2 import MoveIt2
from pymoveit2.gripper_interface import GripperInterface


class SimpleCoffeeBot(Node):
    def __init__(self):
        super().__init__("simple_coffee_node")

        # --- ROBOT SETUP (DON'T TOUCH THIS) ---
        self.arm = MoveIt2(
            node=self,
            joint_names=["joint_1", "joint_2", "joint_3", "joint_4", "
                joint_5", "joint_6"],
            base_link_name="base_link",
            end_effector_name="end_effector_link",
            group_name="arm",
        )

        # Setup the Gripper (The fingers)
        self.gripper = GripperInterface(
            node=self,
            gripper_joint_names=["right_finger_bottom_joint"],
            open_gripper_joint_positions=[0.01],   # Open
            closed_gripper_joint_positions=[0.8], # Closed
            gripper_group_name="gripper",
            gripper_command_action_name="/
                gen3_lite_2f_gripper_controller/gripper_cmd",
            ignore_new_calls_while_executing=False,
        )
        self.get_logger().info("Robot␣is␣ready!")

    # ============================================================
    #  YOUR CODE GOES HERE (READ THIS PART)
    # ============================================================
    def run_my_code(self):

        # --- COORDINATES (Change these numbers!) ---
        cup_x = 0.40
        cup_y = -0.20
        cup_z = 0.01

        pour_x = 0.40
        pour_y = 0.20
```

```
        pour_z = 0.22

        # Orientation: Normal (Holding cup straight)
        orient_straight = [0.0, 0.7071, 0.0, 0.7071]

        # Orientation: Tilted (Pouring)
        orient_tilted   = [0.5, 0.5, 0.5, 0.9]

        # ----------------------------------------
        # STEP 1: Go Home and Open Hand
        # ----------------------------------------
        self.get_logger().info("Step 1: Start")
        # Move joints to a safe starting position
        self.arm.move_to_configuration(joint_positions=[0.0, -0.28,
            1.3, 1.0, -1.0, 0.0])
        self.arm.wait_until_executed()

        self.gripper.open()
        time.sleep(1.0)

        # ----------------------------------------
        # STEP 2: Pick up the Cup
        # ----------------------------------------
        self.get_logger().info("Step 2: Picking up cup")

        # A. Go above the cup (Hover)
        self.move_to(cup_x, cup_y, cup_z + 0.15, orient_straight)

        # B. Go down to the cup
        self.move_to(cup_x, cup_y, cup_z, orient_straight)

        # C. Grab it
        self.gripper.close()
        time.sleep(1.0) # Wait for grip

        # D. Lift it up
        self.move_to(cup_x, cup_y, cup_z + 0.15, orient_straight)

        # ----------------------------------------
        # STEP 3: Go to Pouring Spot
        # ----------------------------------------
        self.get_logger().info("Step 3: Moving to pour spot")
        # Move to the pour location, still holding it straight
        self.move_to(pour_x, pour_y, pour_z, orient_straight)

        # ----------------------------------------
        # STEP 4: POUR! (Tilt the robot) old version
        # ----------------------------------------
        # self.get_logger().info("Step 4: Pouring...")
        # # Stay in same spot, change orientation to TILTED
        # self.move_to(pour_x, pour_y, pour_z, orient_tilted)

        # time.sleep(2.0) # Wait for coffee to pour out...

        # ----------------------------------------
        # STEP 4: The "Barista" Circular Pour
        # ----------------------------------------
        self.get_logger().info("Step 4: Barista Mode Initiated")
```

```
109
110         # 1. Move to the START of the circle (Angle 0)
111         # We start at x + radius, so we don't crash into the side
                immediately
112         radius = 0.03  # 3cm radius circle
113         start_x = pour_x + radius
114         start_y = pour_y
115
116         # Tilt the cup first at the starting point
117         self.move_to(start_x, start_y, pour_z, orient_tilted)
118
119         # 2. Perform the Circle
120         # We divide the circle into 20 steps
121         steps = 20
122
123         for i in range(steps):
124             # Calculate the angle for this step (0 to 360 degrees)
125             angle = (i / steps) * 2.0 * math.pi
126
127             # Calculate the new X and Y based on the circle formula
128             # Note: We keep Z and Orientation CONSTANT
129             next_x = pour_x + (radius * math.cos(angle))
130             next_y = pour_y + (radius * math.sin(angle))
131
132             self.get_logger().info(f"Pouring␣at␣angle:␣{int(math.
                    degrees(angle))}")
133
134             # Move to the next point on the circle
135             self.move_to(next_x, next_y, pour_z, orient_tilted)
136
137
138
139         # ----------------------------------------
140         # STEP 5: Stop Pouring
141         # ----------------------------------------
142         self.get_logger().info("Step␣5:␣Straightening␣up")
143         # Change orientation back to STRAIGHT
144         self.move_to(pour_x, pour_y, pour_z, orient_straight)
145
146         # ----------------------------------------
147         # STEP 6: Put Cup Back
148         # ----------------------------------------
149         self.get_logger().info("Step␣6:␣Returning␣cup")
150
151         # A. Go above the original spot
152         self.move_to(cup_x, cup_y, cup_z + 0.15, orient_straight)
153
154         # B. Go down
155         self.move_to(cup_x, cup_y, cup_z, orient_straight)
156
157         # C. Let go
158         self.gripper.open()
159         time.sleep(1.0)
160
161         # D. Move away (Up)
162         self.move_to(cup_x, cup_y, cup_z + 0.15, orient_straight)
163
164         self.get_logger().info("Done0!")
```

```
        self.arm.move_to_configuration(joint_positions=[0.0, 0.0, 0.0,
            0.0, 0.0, 0.0])
        self.arm.wait_until_executed()

        self.get_logger().info("Done!")


    # ============================================================
    #  HELPER FUNCTION (Makes moving easier)
    # ============================================================
    def move_to(self, x, y, z, orientation):
        # This function handles all the boring math to create a "Pose"
        pose = Pose()
        pose.position.x = x
        pose.position.y = y
        pose.position.z = z
        pose.orientation.x = orientation[0]
        pose.orientation.y = orientation[1]
        pose.orientation.z = orientation[2]
        pose.orientation.w = orientation[3]

        # Tell the robot to plan and execute the path
        plan = self.arm.plan(pose=pose, cartesian=True)
        if plan:
            self.arm.execute(plan)
            self.arm.wait_until_executed()
        else:
            self.get_logger().error("Could not find a path!")

# ------------------------------------------------------
#  Standard Boilerplate (Just copy this)
# ------------------------------------------------------
def main(args=None):
    rclpy.init(args=args)
    node = SimpleCoffeeBot()

    # Run the node in a background thread
    executor = MultiThreadedExecutor(num_threads=2)
    executor.add_node(node)
    thread = threading.Thread(target=executor.spin, daemon=True)
    thread.start()

    try:
        time.sleep(2.0) # Wait for startup
        node.run_my_code() # <--- This runs your code above
    except KeyboardInterrupt:
        pass
    finally:
        rclpy.shutdown()
        thread.join()
```

A.5 Arduino Code for Scale

This standalone script runs on Arduino, using a load cell and an HX711 amplifier to output the mass to the serial monitor, which can then be read by the ROS 2 script.

Listing 5: Arduino Code for Scale

```
1
2  #include <Arduino.h>
3  #include "HX711.h"
4
5  // circuit wiring
6  const int LOADCELL_DOUT_PIN = 2;
7  const int LOADCELL_SCK_PIN = 3;
8  const int BUTTON_PIN = 7;
9
10
11 long offset = 566200;
12 const float scale_factor = 420.58; // haha get it
13
14 HX711 scale;
15
16 /**
17  * @brief Performs a button-triggered tare operation.
18  * * It reads the HX711 multiple times over a 1-second period
19  * and calculates the average raw reading to set the new offset.
20  * * @return The new calculated average offset.
21  */
22  long button_tare() {
23   const int TARE_DURATION_MS = 1000; // Length of taring sequence (ms)
24   const int SAMPLE_DELAY_MS = 50; // Delay between samples
25   long sumOfReadings = 0; // Initialize running total
26   int count = 0;
27   unsigned long startTime = millis();
28
29   Serial.println("\n---␣Taring␣in␣progress␣(1␣second)...␣---");
30
31   // Read the scale for the specified duration
32   while (millis() - startTime < TARE_DURATION_MS) {
33     if (scale.is_ready()) {
34       sumOfReadings += scale.read();
35       count++;
36     }
37     delay(SAMPLE_DELAY_MS);
38   }
39
40   // Calculate the average offset
41   if (count > 0) {
42     long newOffset = sumOfReadings / count;
43     Serial.print("Tare␣complete.␣Averaged␣");
44     Serial.print(count);
45     Serial.print("␣readings.␣New␣Offset:␣");
46     Serial.println(newOffset);
47     return newOffset;
48   } else {
49     Serial.println("Error:␣Scale␣not␣ready␣during␣tare,␣keeping␣old␣
         offset.");
50     return offset; // Return the current offset if no readings were
         taken
```

```
  }
}

void setup() {
  Serial.begin(9600);
  Serial.println("HX711 Raw Reading Demo");

  // Initialize the scale with the DOUT and SCK pins
  scale.begin(LOADCELL_DOUT_PIN, LOADCELL_SCK_PIN);
  scale.tare();

  // Pin Setup
  pinMode(BUTTON_PIN, INPUT);
}

void loop() {
  if (digitalRead(BUTTON_PIN) == HIGH) {
    // Button is pressed, enter the taring state
    offset = button_tare();
  }

  // Check if the scale is ready to read

  if (scale.is_ready()) {
    // scale.read() returns the raw 24-bit reading from the ADC
    long reading = (scale.read() - offset) / scale_factor ;
    // Serial.print("Raw Reading: ");
    Serial.println(reading);
  } else {
    Serial.println("HX711 not found.");
  }

  // A small delay to keep the serial monitor readable
  delay(500);
}
```

A.6 YOLO Dataset Configuration (data.yaml)

This configuration file serves as the bridge between the YOLO model and the training data, defining the file paths, number of classes (nc), and the human-readable class names (names).

Listing 6: YOLO Dataset Configuration (data.yaml)

```
# Dataset root directory (Absolute path recommended)
path: '/path/to/dataset/gesture'

# Train and Validation image directories (Relative to 'path')
train: images\train
val: images\val

# Number of classes
nc: 10

# Class names
names: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

A.7 Computer Vision: Training Script

This script handles the transfer learning process, loading the pre-trained YOLOv8n weights and fine-tuning them on the hand gesture dataset.

Listing 7: YOLO Training Script

```
from ultralytics import YOLO
# Main entry point is required for Windows multiprocessing
if __name__ == '__main__':

    # --- Configuration ---
    # Path to your CORRECTED data.yaml file
    data_yaml_path = r"data.yaml"

    # Path to the pre-trained model (yolov8n.pt will be downloaded if
       not found)
    pre_model_path = "yolov8n.pt"

    # Project name for saving results
    project_name = "hand_gesture_train"

    # Device setup: 0 for GPU, 'cpu' for CPU (Assuming CPU for
       compatibility)
    device_id = 'cpu'

    # Training parameters
    NUM_EPOCHS = 30
    BATCH_SIZE = 16
    NUM_WORKERS = 2
    # ----------------------

    print("Loading␣model␣and␣transferring␣weights...")
    # Initialize the model and load pre-trained weights
    model = YOLO('yolov8n.yaml').load(pre_model_path)

    print("Starting␣training...")
    # Start training using the corrected configuration
    results = model.train(
```

```
        data=data_yaml_path,
        epochs=NUM_EPOCHS,
        imgsz=640,
        batch=BATCH_SIZE,
        workers=NUM_WORKERS,
        name=project_name,
        device=device_id
    )

    print(f"Training completed. Best weights saved in runs/detect/{
        project_name}/weights/best.pt")
```