WonkyBot

An Autonomous Differential-Drive Robot for Color-Based Ball Sorting

Lillian Slaton · Edward Fields · Matthew Garrett
Department of Engineering — University of Central Arkansas — Senior Design 2025–2026
lslaton@uca.edu · efields@uca.edu · mgarrett@uca.edu

Abstract

This notebook documents WonkyBot, a fully autonomous differential-drive robot built for the 2026 Autonomous Vehicle Competition (AVC) at the University of Central Arkansas. The robot locates colored balls in an open arena, picks them up with a dual-stepper lead-screw arm, and deposits each ball into its matching-color bucket without human input. The system runs ROS 2 Jazzy on a Raspberry Pi 5, uses a classical HSV color-detection pipeline with a four-feature shape classifier for perception, and a stepper-position state machine where the physical position of the arm IS the system state. Nav2 with RTAB-Map SLAM was developed and validated over four months, then deliberately replaced by an odometry-based delivery system better suited to the open, featureless competition arena. This document covers the mechanical design, electrical wiring, software architecture, and an honest account of what worked, what didn't, and what we would do differently.

TABLE OF CONTENTS

I. Project Overview 2
II. Composite Drawing and Description 2–3
III. Electrical Schematic and Description 3–4
IV. Code Feature: Stepper-Position State Machine 4–5
V. Successes and Failures 5–7
References 7

LIST OF FIGURES

Fig. 1. Initial concept CAD — early lead-screw design (M. Garrett)2
Fig. 2. Physics simulation — horizontal arm layout (M. Garrett)2
Fig. 3. Linear rail shaft mount — 3D-printed PLA-CF (M. Garrett)2
Fig. 4. Front plate and gripper connections CAD3
Fig. 5. Final stepper motor bracket with NEMA-17 (M. Garrett)3
Fig. 6. Camera mount housing — RealSense D455 (M. Garrett)3
Fig. 7. WonkyBot fully assembled — front view (lab, Apr. 2026)3
Fig. 8. WonkyBot fully assembled — side view (lab, Apr. 2026)3
Fig. 9. KiCad electrical schematic — Pico 2, motor driver, servos (E. Fields)3
Fig. 10. Pico 2 breakout board with motor driver — wiring photo4
Fig. 11. TMC2209 stepper driver breakout board — wiring photo4
Fig. 12. RViz SLAM map — first successful Nav2 navigation, Feb. 20265
Fig. 13. HSV color detection — balls and buckets labeled in real-time5

I. PROJECT OVERVIEW

WonkyBot is a fully autonomous differential-drive robot built for the 2026 AVC. The competition places colored balls and matching-color buckets in an open arena; the robot must find each ball, pick it up, and deposit it in the correct bucket without any human input once started.

The platform runs ROS 2 Jazzy on a Raspberry Pi 5 for high-level decision-making, with a Raspberry Pi Pico 2 handling low-level motor control. An Intel RealSense D455 RGB-D camera provides color images for ball and bucket detection and depth data for position estimation.

The core design idea: rather than tracking state in a software variable that could drift out of sync with hardware, the physical position of the arm stepper motor IS the state. If the arm is down, the robot grabs. If the arm is up with the gripper closed, it delivers. This eliminated an entire class of hard-to-debug failures.

A. Team roles

Lillian Slaton led all software — ROS 2 nodes, HSV detection, state machine, and autonomy integration. Edward Fields led electrical design, KiCad schematics, wiring, and hardware bring-up. Matthew Garrett led mechanical design, 3D-printed all custom parts in PLA-CF, and built the lead screw arm assembly using Onshape.

B. System overview

Domain	What it does
Perception	RealSense D455 — color image + aligned depth + IMU. Quadrature encoders on both DC motors.
Decision	Raspberry Pi 5: ROS 2 nodes for detection, state machine, EKF odometry fusion.
Action	Pico 2: PID-controlled DC motors, two NEMA-17 steppers for arm, two servos for wrist.
Power	14.8 V LiPo (drive + compute). 7.4 V LiPo (steppers). 7.4 V LiPo → BEC 6 V (servos).
Pi ↔ Pico	USB serial 115,200 baud. Commands down, odometry up at 60 Hz.

Table II. System overview.

C. ROS 2 node map

Node	Key topics (sub → pub)
realsense2_camera	→ /color/image_raw · /aligned_depth · /imu
detector_node	color+depth → /detected_objects (JSON with depth_m)
detection_3d_node	/detected_objects + /odom → /detected_objects_3d
sorting_master	/detected_objects + /arm_state → /cmd_vel + /arm_command
octo_pilot	/cmd_vel + /arm_command → /odom + /arm_state + serial
robot_localization	/odom + /imu → /odometry/filtered (EKF)
all_color_detector	color+depth → /ball_distance + /ball_offset
autonomy_master	Voice trigger → nav to detect zone → grab loop → deliver

Table VI. ROS 2 node communication map.

II. COMPOSITE DRAWING AND DESCRIPTION

The following figures document the mechanical evolution of WonkyBot from initial concept sketches through the final assembled robot. All CAD work was done in Onshape by Matthew Garrett. The final robot is shown in Figs. 7 and 8.

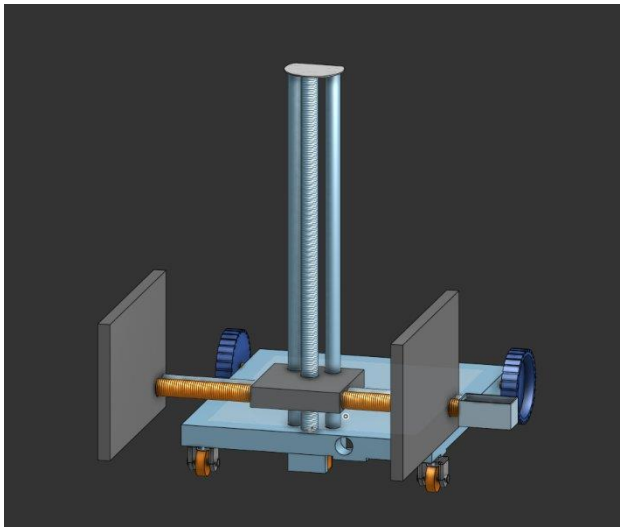

Fig. 1. Initial concept CAD — early lead-screw arm design (Onshape, M. Garrett).

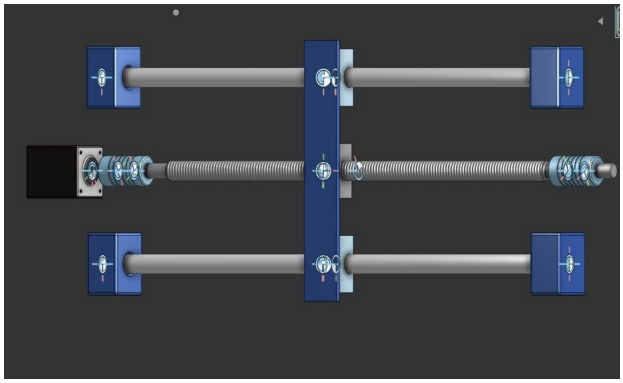

Fig. 2. Physics simulation — horizontal arm showing lead screw, guide rails, and bearing blocks.

A. Chassis and drivetrain

The chassis uses 2020 aluminum extrusion in a double-decker configuration. The lower deck holds the drive motors, motor controller, Pico 2, and batteries. The upper deck supports the lead screw arm. All brackets,

motor housings, and the camera mount are 3D-printed in PLA-CF for stiffness without added weight.

Two DC gear motors (70:1) with quadrature encoders drive the rear wheels through a belt-and-pulley transmission. Direct-shaft coupling was abandoned early after shaft couplers slipped under load — the belt drive fixed this completely. Wheel separation is 0.477 m.

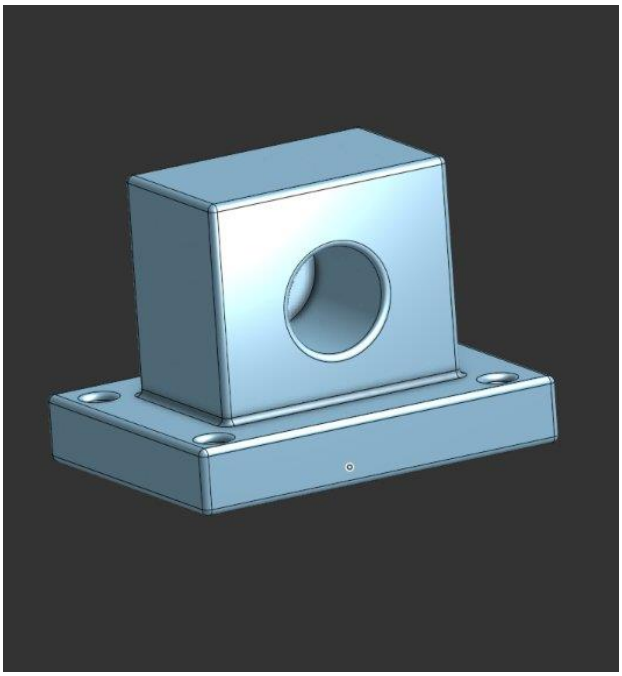

Fig. 3. Linear rail shaft mount — 3D-printed PLA-CF (M. Garrett).

B. Arm assembly

The grabber uses two NEMA-17 steppers driven by TMC2209 V2.0 drivers. Stepper 1 (vertical) drives a T8 lead screw 75,000 counts to lower the arm to ball level. Stepper 2 (horizontal) closes the gripper jaw 31,000 counts. Both axes run on 16 mm carbon-fiber guide rails. Two servos (GP14, GP15) control wrist angle: wide during descent to scoop, rotated up to secure the ball, mid-position to tip it into the bucket.

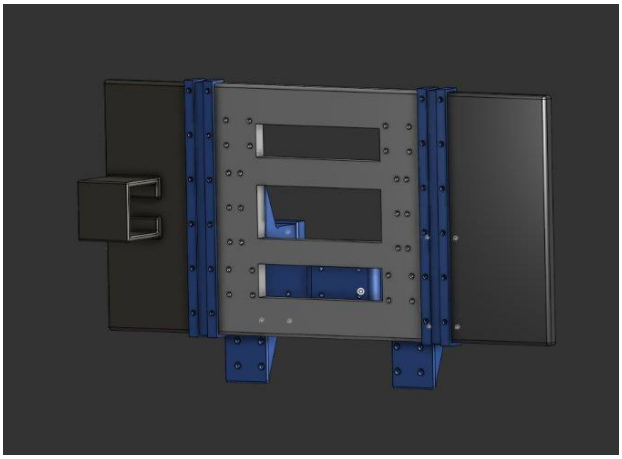

Fig. 4. Front plate and gripper connections — rail attachment points and jaw geometry.

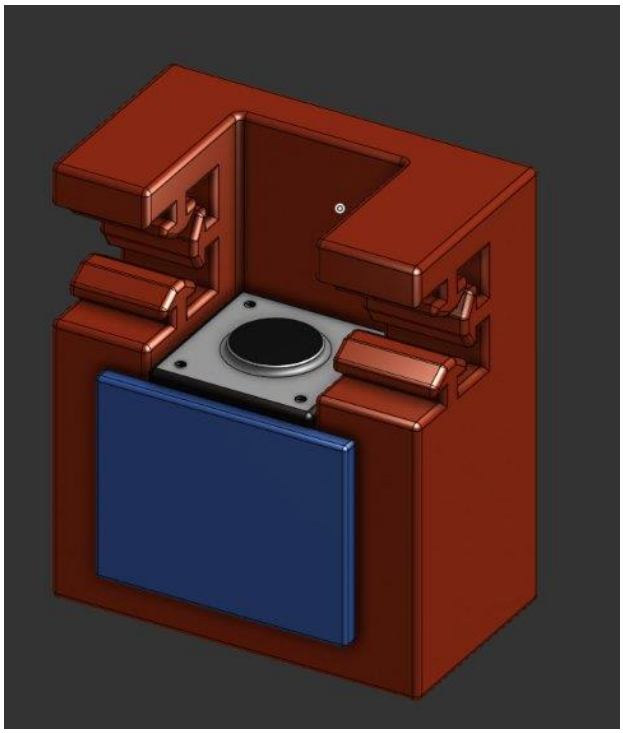

Fig. 5. Final stepper motor bracket with NEMA-17. Multiple print iterations were needed before this was rigid enough under load.

Parameter	Value
Vertical travel (S1)	75,000 steps
Gripper travel (S2)	31,000 steps
Servo GP15	DOWN=180° MID=120° UP=60°
Servo GP14	DOWN=80° MID=140° UP=200°
Camera offset	x=+0.43 m z=+0.10 m
Wheel separation	0.477 m
Chassis length	~600 mm
Tire diameter	~160 mm

Table VII. Key mechanical dimensions.

C. Camera mount

The RealSense D455 is housed in a custom 3D-printed mount (M. Garrett). It positions the camera 0.43 m forward and 0.10 m above base_link, facing forward. The offset is published as a static TF transform so all depth data is correctly referenced to the robot frame. The mount protects the ribbon cable and holds the correct tilt angle.

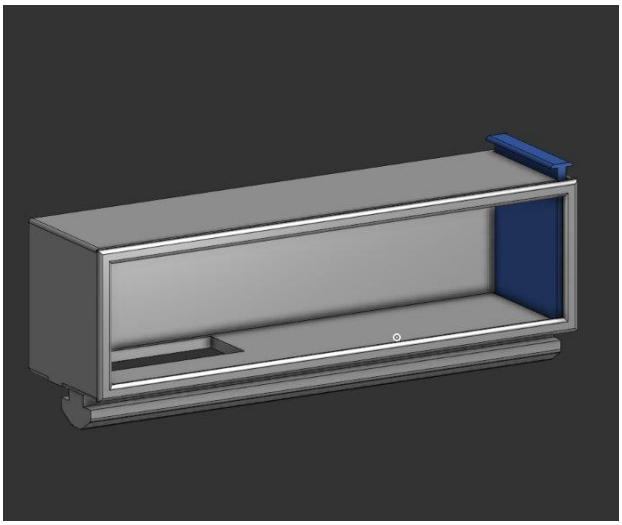

Fig. 6. Camera mount housing — RealSense D455 at +0.43 m forward, +0.10 m up.

D. Assembled robot

Figures 7 and 8 show the final assembled WonkyBot as it appeared in the lab during final integration testing in April 2026. The orange and red 3D-printed components are PLA-CF; the silver uprights are 2020 aluminum extrusion. The RealSense camera is visible at the front base.

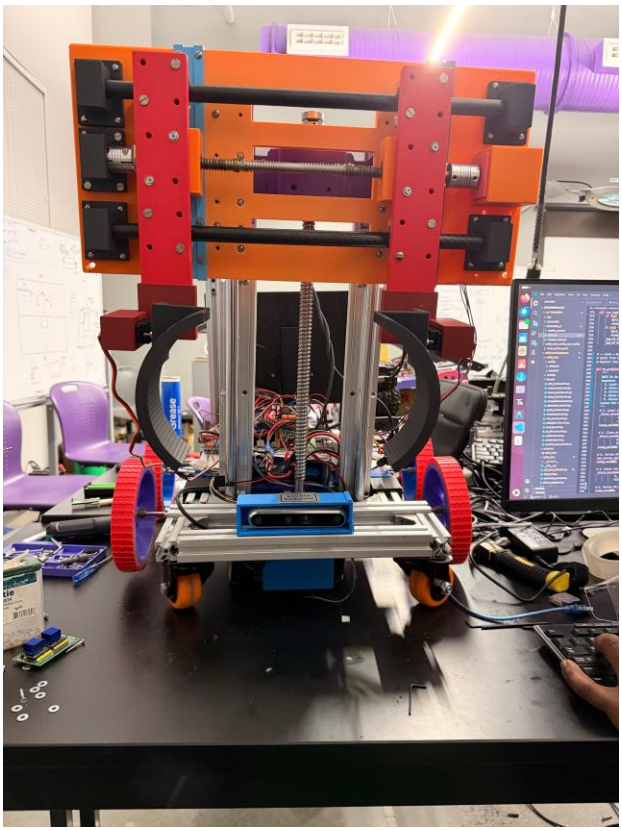

Fig. 7. WonkyBot fully assembled — front view, April 2026. Lead screw arm and gripper visible at top. RealSense D455 mounted at base.

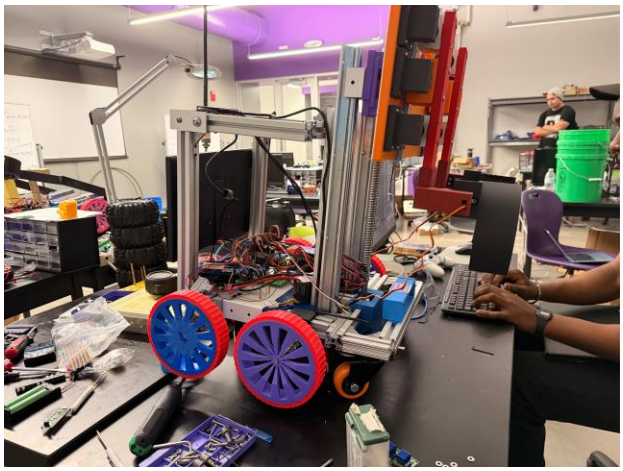

Fig. 8. WonkyBot fully assembled — side view, April 2026. Belt-drive transmission, aluminum extrusion chassis, and wiring visible.

III. ELECTRICAL SCHEMATIC AND DESCRIPTION

The electrical design was led by Edward Fields, who produced all schematics in KiCad and built the breakout board for the Pico 2. The KiCad schematic (Fig. 9) shows the complete Pico 2 pinout, motor driver connections, and servo headers.

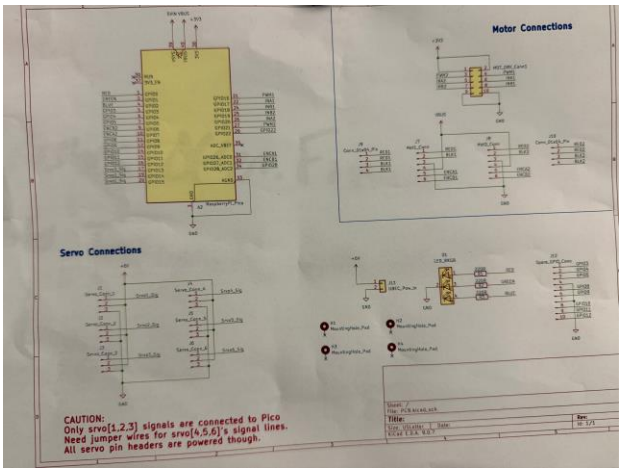

Fig. 9. KiCad electrical schematic — Pico 2 GPIO assignments, motor driver connections, servo headers, and encoder inputs (E. Fields).

Component	Role
Raspberry Pi 5 (16 GB)	Primary compute — ROS 2, vision pipeline
Raspberry Pi Pico 2	Motor PID, encoders, steppers, servos
Intel RealSense D455	RGB-D camera — detection + depth
TMC2209 × 2	Stepper drivers — STEP/DIR, current limiting
Servo motors × 2	Wrist angle — GP14 and GP15, 50 Hz PWM
DRV8833 H-bridge	DC drive motor controller — 2-ch PWM
Buck converters × 2	14.8 V → 5 V for Pi 5 and Pico 2
LiPo batteries × 3	14.8 V drive/compute · 7.4 V steppers · 7.4 V servos
BEC	7.4 V → 6 V regulated for servos
Optocoupler kill switch	Isolates Pi GPIO from motor supply

Table I. Component summary.

A. Power distribution

Three isolated battery domains prevent motor noise from corrupting logic rails. The 14.8 V 4-cell LiPo powers drive motors directly and steps down to 5 V for the Pi 5 via a PEB (Power Expansion Board). A 7.4 V 2-cell LiPo powers both TMC2209 stepper drivers. A separate 7.4 V pack feeds a BEC that outputs a clean 6 V rail for the servos.

Domain	Source → Load
Drive + compute	14.8 V LiPo → Motor driver (direct) + PEB 5 V → Pi 5 → Pico 2
Stepper supply	7.4 V LiPo → TMC2209 #1 → Stepper 1 (vertical arm)
Stepper supply	7.4 V LiPo → TMC2209 #2 → Stepper 2 (horiz/gripper)
Servo supply	7.4 V LiPo → BEC (6 V) → Servo GP14 + Servo GP15

Table III. Power distribution — three battery domains.

B. Pico 2 pin assignments

GPIO	Dir	Connected to
GP6/7	IN	Right motor encoder A/B
GP8	OUT	TMC2209 #1 — DIR (vertical stepper)
GP9	OUT	TMC2209 #1 — STEP (vertical stepper)
GP10	OUT	TMC2209 #2 — DIR (horiz stepper)
GP11	OUT	TMC2209 #2 — STEP (horiz stepper)
GP14	OUT	Servo 2 — 50 Hz PWM wrist
GP15	OUT	Servo 1 — 50 Hz PWM wrist
GP16–18	OUT	H-bridge — left motor PWM + dir A/B
GP19–21	OUT	H-bridge — right motor dir B/A + PWM
GP26/27	IN	Left motor encoder A/B
USB	BI	Pi 5 serial — 115,200 baud, 60 Hz

Table IV. Pico 2 GPIO pin assignments.

C. Breakout board wiring

The Pico 2 plugs into a custom prototyping breakout board that Edward wired to route all GPIO signals to the motor driver, stepper drivers, and servo headers. Figures 10 and 11 show the installed wiring inside the robot chassis.

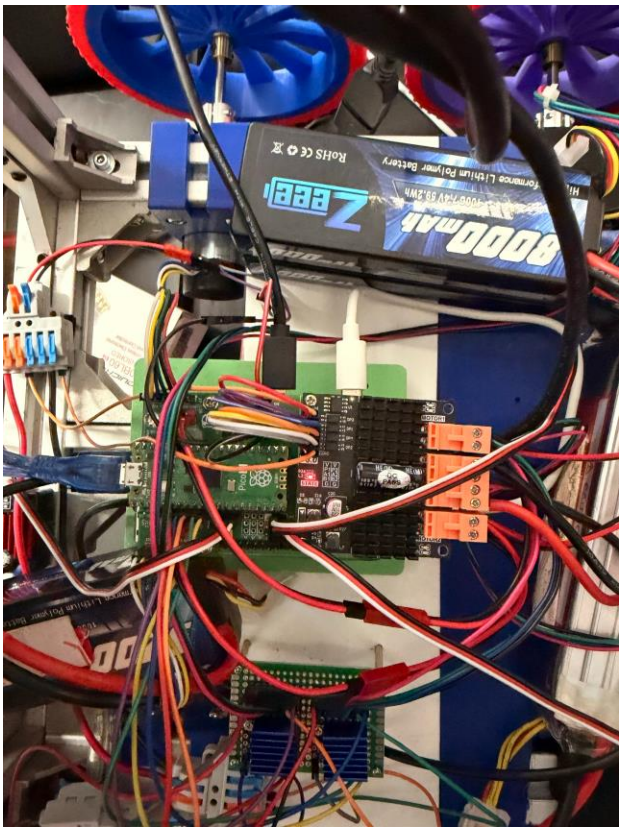

Fig. 10. Pico 2 breakout board with motor driver installed in chassis. 7.4 V LiPo visible at top.

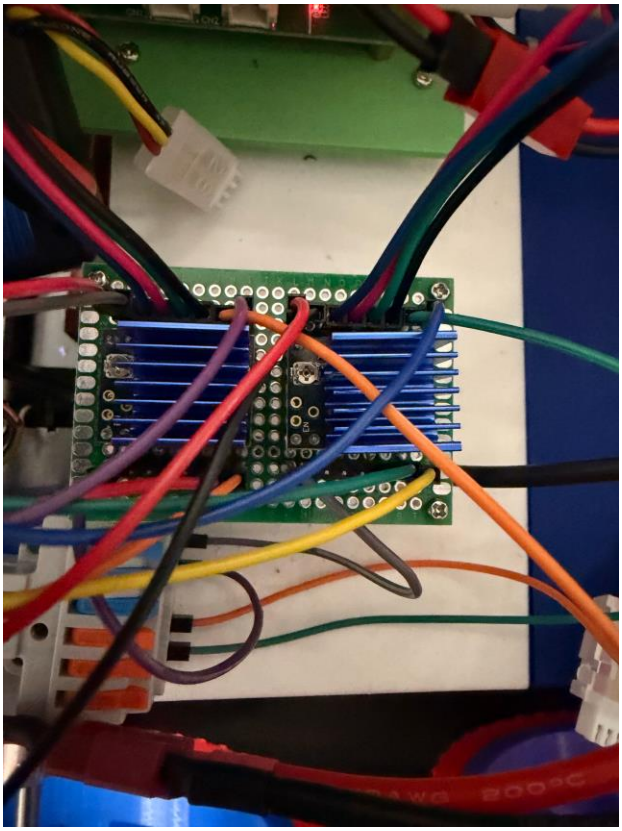

Fig. 11. TMC2209 stepper driver breakout board — shows STEP/DIR wiring to both stepper axes.

D. Serial communication protocol

Five command types travel Pi → Pico over USB serial: a 2-value drive command (lin, ang); a 4-value command (lin, ang, s1, s2) for simultaneous drive and arm; a jog command (J, s1, s2) for continuous timer-driven arm motion; PICKUP which runs the full blocking grab sequence (lower arm → close gripper → servos up → raise arm); and RELEASE which opens the gripper. The Pico replies at 60 Hz with measured velocities. A 500 ms watchdog stops all motion if the Pi goes silent.

E. Motor control and the timer-pause fix

Each drive wheel is controlled by a PID velocity loop (RegulatedWheel, 50 Hz). The most important firmware discovery of the project: PID timers and encoder IRQs firing at 50–100 Hz during stepper acceleration ramps were stealing microseconds from sleep_us() calls, causing the stepper to miss steps and stall. The fix was to pause ALL timers and IRQs before any blocking stepper move via _pause_all_timers() and restart them after. This one change made the arm work reliably for the first time.

Class	Inherits	Adds
BaseMotor	—	PWM + direction (INA/INB)
EncodedMotor	BaseMotor	Quadrature encoder interrupt counting
SentientWheel	EncodedMotor	Velocity measurement at 100 Hz
RegulatedWheel	SentientWheel	PID velocity controller at 50 Hz
DiffDriveController	—	Two RegulatedWheels. WHEEL_SEP=0.477 m.

Table V. mobile_control class hierarchy (Pico firmware).

F. Kill switch

An optocoupler circuit isolates the 3.3 V Pi GPIO from the motor power domain. The Pi drives the LED through a current-limiting resistor; the transistor output controls the motor battery relay. This protects the Pi from motor back-EMF and lets software command an emergency stop without exposing the Pi to motor-side voltages.

IV. CODE FEATURE: STEPPER-POSITION STATE MACHINE

WonkyBot's central autonomy idea: instead of a software variable tracking what state the robot is in, the physical position of the arm stepper motor IS the state. If the arm is at home and the gripper is open, find a ball. If the arm is down, grab. If the arm is up with the gripper closed, deliver. The hardware cannot lie about where it is — a software variable can.

A. State definitions

steps_ver / steps_hor	State	Behavior
0 / 0	FIND_BALL	Drive forward, steer toward nearest ball
75000 / 0	GRAB	Arm down — close gripper, servos up, raise
0 / 31000	FIND_BUCKET	Navigate to hardcoded corner waypoint
moving=True	TRANSITION	Stop drive motors, wait for arm

B. Main control loop — sorting_master.py

The loop runs at 10 Hz. It reads the current arm position from /arm_state and routes to the correct behavior. Notice there is no 'current_state' variable — arm position is the only truth:

```
def _loop(self):
  # While arm is moving between positions,
  # stop the wheels and wait. Never drive
  # while the arm is mid-travel — the
  # geometry doesn't work out and you'll
  # hit things.
  if self.arm_moving:
    self._stop()
    return

  # Read real stepper counts published
  # by octo_pilot at 7 Hz. Tolerances
  # (±500 counts vertical, ±300 horiz)
  # absorb motor deceleration overshoot.
  at_home  = abs(self.steps_ver
             - VER_HOME) <= 500
  at_down  = abs(self.steps_ver
             - VER_DOWN) <= 500
  hor_open = abs(self.steps_hor
             - HOR_HOME) <= 300
  hor_clsd = abs(self.steps_hor
             - HOR_CLOSED) <= 300

  if at_home and hor_open:
    # Arm up, gripper open. Look for ball.
    self._grab_commanded = False
    self._phase_find_ball()

  elif at_down and hor_open:
    # Arm just reached ball level.
    # Close gripper AND raise arm in
    # one JSON command. octo_pilot does
    # vertical first (SEQUENTIAL_MOTION).
    if not self._grab_commanded:
      self._grab_commanded = True
      self._stop()
      self._send_arm(ver=VER_HOME,
                     hor=HOR_CLOSED)

  elif at_home and hor_clsd:
    # Arm up, ball gripped.
    # Navigate to matching bucket corner.
    self._release_commanded = False
    self._phase_find_bucket()

  else:
    # Unknown position — home everything.
    # Handles power-on and error recovery.
    self._send_arm(ver=VER_HOME,
                   hor=HOR_HOME)
```

Fig. — sorting_master.py _loop(). Arm position is the only state variable.

C. FIND_BALL — approach and grab trigger

When arm is home and gripper is open, the robot drives forward at 0.10 m/s and steers proportionally toward the ball's pixel centroid. Depth is read from the aligned depth image. The 5-frame confirmation window prevents a single noisy depth reading from triggering a grab:

```
# Constants at top of sorting_master.py:
# BALL_GRAB_DISTANCE_M = 0.2032  (8 in)
# CLOSE_FRAMES_REQUIRED = 5
# KP_ANGULAR = 0.004 rad/s per pixel

def _phase_find_ball(self):
  balls = [o for o in self.latest_objects
           if o.get('type') == 'BALL']
  if not balls:
    # Nothing visible — creep forward
    self._drive(SEARCH_FORWARD_SPEED, 0)
    self.close_count = 0
    return

  target  = self._pick_priority(balls)
  depth_m = target.get('depth_m', 0.0)
  cx      = target['cx']

  # Count consecutive close frames
  close = (depth_m > 0.0 and
           depth_m <= BALL_GRAB_DISTANCE_M)
  self.close_count = (
    self.close_count + 1 if close else 0)

  if self.close_count >=
CLOSE_FRAMES_REQUIRED:
    # Lock in color, lower the arm.
    # Next _loop() tick sees at_down=True
    # and triggers GRAB automatically.
    self.target_color = target['color']
    self.close_count = 0
    self._stop()
    self._send_arm(ver=VER_DOWN,
                   hor=HOR_HOME)
    return

  # Steer proportionally toward ball
  # while approaching
  angular = -KP_ANGULAR*(cx -
IMAGE_CENTRE_X)
  self._drive(APPROACH_SPEED, angular)
```

D. Vision classifier — all_color_detector.py

The shape classifier votes on four features to decide if a detected blob is a BALL or BUCKET. Each feature votes independently with a weight, and the final score determines the label. The key insight: a ball is round, square-ish in its bounding box, densely fills that box, and has no straight edges. A bucket fails all four. Ties go to BALL:

```
def classify_shape(cnt, gray):
  area  = cv.contourArea(cnt)
  bbox  = cv.boundingRect(cnt)
  circ  = _circularity(cnt)
  asp   = _aspect_ratio(bbox)
  ext   = _extent(cnt, bbox)

  # Skip line detection for large blobs.
  # At close range, a ball's curved top/
  # bottom arcs look horizontal to Hough
  # and would incorrectly vote 'bucket'.
  # SKIP_LINES_AREA = 10000 px²
  n_lines = (0 if area >= SKIP_LINES_AREA
             else _count_lines(cnt, gray))
  score = 0

  # -- Feature 1: Circularity (weight 2) --
  # Balls score > 0.55. Bucket edges and
  # corners drag circularity way down.
  # Neutral zone 0.35-0.55: motion blur
  # can slightly flatten a ball so we
  # don't penalise the marginal cases.
  if circ >= 0.55:   score += 2
  elif circ < 0.35:  score -= 2

  # -- Feature 2: Aspect ratio (weight 1)--
  # Balls fit in a roughly square bbox.
  # Buckets are often clearly wider or
  # taller depending on viewing angle.
  if abs(asp - 1.0) <= 0.35: score += 1
  elif asp >= 1.30 or asp <= 0.70:
                              score -= 1

  # -- Feature 3: Extent (weight 1) --
  # Balls fill their bounding box ≥60%.
  # Rectangular buckets with cut-out
  # openings score lower here.
  if ext >= 0.60:    score += 1
  elif ext < 0.50:   score -= 1

  # -- Feature 4: Straight lines (wt 1)--
  # Buckets have rectangular edges.
  # HoughLinesP finds them easily.
  # We only count near-horizontal (<20°)
  # or near-vertical (>70°) lines to
  # filter out ball arc tangents.
  if n_lines >= 7:  score -= 1
  else:             score += 1

  # Ties (score == 0) go to BALL.
  # We'd rather attempt a wrong grab
  # than miss a real ball.
  label = 'BALL' if score >= 0 else 'BUCKET'
  return label, score
```

Fig. — all_color_detector.py classify_shape(). Every vote is commented.

E. Arm command bridge — octo_pilot.py

octo_pilot.py bridges Pi and Pico at 75 Hz. Arm targets are tracked as counts and advanced toward the target by COUNTS_PER_TICK each tick. This gives smooth motion without blocking the ROS spin loop. Sequential mode (vertical first, then horizontal) ensures the arm is fully lowered before the gripper closes:

```
# COUNTS_PER_TICK = 500
# At 75 Hz: ~37,500 counts/sec
# Full 75,000-count vertical = ~2 seconds

def _update_arm(self):
  # ── Vertical stepper ──────────────────
  if not self._at_ver_target():
    rem  = self.target_ver - self.steps_ver
    move = min(COUNTS_PER_TICK, abs(rem))
    if rem > 0:
      self.step_dir = 1    # lower arm
      self.steps_ver += move
    else:
      self.step_dir = -1   # raise arm
      self.steps_ver -= move
  else:
    self.step_dir = 0      # hold

  # ── Horizontal stepper ────────────────
  # SEQUENTIAL_MOTION = True:
  # Don't move the gripper until the
  # vertical axis has fully settled.
  # This prevents the gripper from
  # closing while the arm is mid-air.
  if (SEQUENTIAL_MOTION and
      not self._at_ver_target()):
    self.step2_dir = 0  # wait
    return
  # ... (horizontal mirrors vertical)
```

V. SUCCESSES AND FAILURES

A. Month-by-month timeline

September 2025 — Team formed. Roles assigned: Lillian (software), Edward (electrical), Matthew (mechanical). Design settled on a lead-screw arm after servo-based grippers were ruled out as too weak. First chassis prototype assembled from 2020 extrusion. GitHub and Onshape set up.

October 2025 — Matthew got the first stepper spinning and printed the initial bracket. Several iterations were needed before it held the lead screw without flexing. Edward installed an RPLidar and got a first 2D map in RViz. Lillian connected the Raspberry Pi AI camera module and got color blob detection on screen. All three subsystems were alive independently for the first time.

November 2025 — Both steppers integrated into Pico firmware with PS controller joystick control. Drive system upgraded from direct-shaft to belt-and-pulley after shaft couplers slipped. IMU added. Lillian continued building YOLO training data but the model kept failing to detect red objects — a warning sign attributed to dataset size at the time.

December 2025 — Team paused camera work and focused on reliable driving and mapping. Semester ended with a robot that could drive by joystick and build rough maps. Nothing ran autonomously yet.

January 2026 — RTAB-Map installed as SLAM backend. After continued YOLO struggles, switched to classical HSV detection — immediately better red detection, nearly double the frame rate. Critical discovery: the competition arena has no walls for LiDAR to reflect off. RPLidar was useless in this environment. RealSense depth camera replaced it entirely, but this cost several weeks of Nav2 work that had to be re-done with the new sensor.

February 2026 — HSV detection labeling balls and buckets correctly on screen. Navigation still broken: robot built maps but refused to move when a goal was set. Root cause: clock skew between Pi 5 and SSH laptop caused ROS 2 node timing failures. Switching to CycloneDDS middleware and syncing NTP fixed it immediately. By late February the robot was navigating to RViz goal poses for the first time.

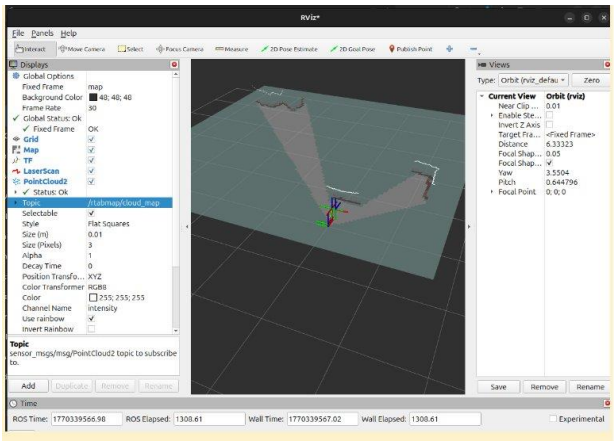

Fig. 12. RViz SLAM map — first successful Nav2 navigation run (Feb. 2026). Gray area is the mapped region; the robot is at the axes origin.

March 2026 — Nav2 working in the lab. But in competition testing, AMCL localization drifted badly in the open, featureless arena. Without walls or corners for the particle filter to lock onto, the robot would think it had delivered to the red bucket when it was actually at the yellow one. Decision made to replace Nav2 delivery with simple odometry-based waypoint navigation (see Section V-B). Nav2 is still used for the initial drive to the detection zone.

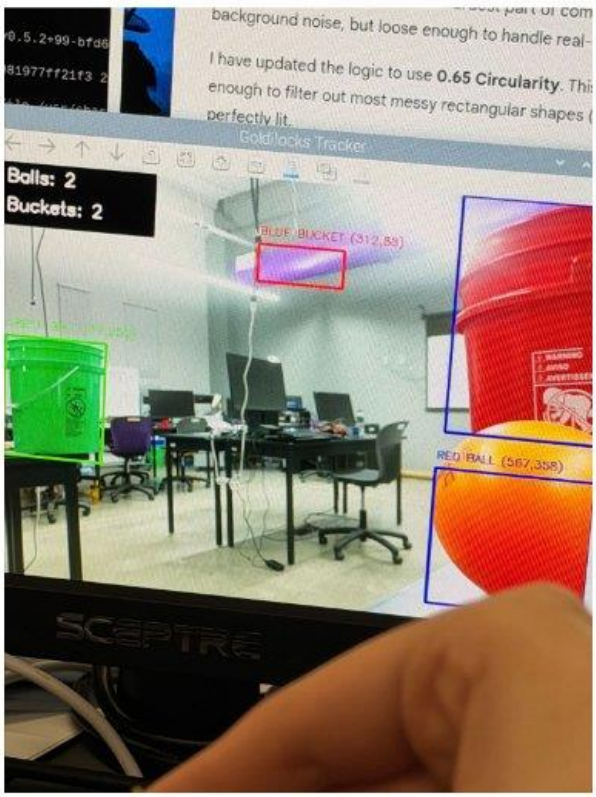

Fig. 13. HSV detection output with four-feature classifier — 'BLUE BUCKET' and 'RED BALL' correctly labeled in real-time.

April 2026 — Most productive and most stressful month. Stepper stall bug traced to PID timers and encoder

IRQs stealing microseconds from sleep_us() calls during acceleration ramps — pausing all timers before any blocking move fixed it. Two servo motors added for wrist angle control. Serial protocol audited: Pico expected 4 values but was receiving 2, silently dropping all arm commands. Full autonomy stack integrated in the final week.

B. The Nav2 decision

Four months of work went into Nav2 — and it worked in the lab. The robot could map a room and navigate to goal poses in RViz with reliable path planning. However, AMCL localization in a wide open arena with no distinctive features proved unreliable for competition. Without fixed landmarks, the particle filter's position estimate drifts over time. The robot could be at the wrong bucket and have no way to know.

The replacement is a simple proportional odometry controller inside sorting_master.py. After picking up a ball, the robot drives from its current odometry position to a hardcoded corner coordinate using basic trigonometry — no map, no AMCL, no particle filter. This is deterministic and predictable. The tradeoff is that odometry accumulates drift too, so the corner coordinates need to be recalibrated for each arena setup. But in competition conditions it is far more reliable than AMCL in an open space.

Nav2 is still used for one thing: autonomy_master.py uses it to drive the robot to the initial detection zone at the start of a run. That single navigation goal is less sensitive to drift because the robot starts in a known position, and Nav2's path-following is smoother than a simple bang-bang controller for that initial approach.

C. What we would do differently

Test the competition environment before committing to a sensing approach. We assumed LiDAR would work in the arena for two months before discovering it was useless in an open space. That assumption cost six weeks of Nav2 bring-up that had to be redone.

The YOLO work consumed most of Lillian's first semester. Classical HSV detection was always the right tool — four known colors under controlled lighting. YOLO is overkill for this problem and its red detection was poor due to training data imbalance. The sign was there in October; we should have pivoted sooner.

Add a serial protocol handshake at startup. The 4-value vs 2-value mismatch went undetected for weeks because the robot could still drive — only the arm was silently broken. A two-line startup exchange that verifies protocol version on both ends would catch this class of bug on the first test.

Keep the firmware in one file. The confusion between two versions of main.py — one that expected PICKUP/RELEASE commands, one that didn't — was entirely self-inflicted. One canonical firmware file per robot, enforced by the git repo, would have prevented weeks of confusion.

REFERENCES

[1] J. Quigley et al., "ROS: an open-source Robot Operating System," ICRA Workshop on Open Source Software, 2009.

[2] T. Moore and D. Stouch, "A generalized extended Kalman filter implementation for the Robot Operating System," Proc. 13th Int. Conf. Intelligent Autonomous Systems, 2014.

[3] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," arXiv:1804.02767, 2018.

[4] R. C. Gonzalez and R. E. Woods, Digital Image Processing, 4th ed. Pearson, 2018.

[5] Intel RealSense SDK 2.0 Documentation. [Online]. Available: https://dev.intelrealsense.com

[6] M. Macenski et al., "Nav2: The Next-Generation Navigation System for Robots," IROS 2020.

[7] Raspberry Pi Foundation. Pico 2 Datasheet. [Online]. Available: https://datasheets.raspberrypi.com

[8] W. Garage, "twist_mux: A ROS Velocity Multiplexer," ROS Wiki. [Online]. Available: http://wiki.ros.org/twist_mux