UC San Diego

Autonomous Trash Collection Robot with Vision-Guided Approach, LiDAR-Based Standoff Control, and Integrated Robotic Manipulation

Final Technical Report

David Carrillo
Darin Djapri
Lek Man
Ryan Chen
Marcus Greenan
Giovanni Clark
Harihar Kaushik
Lucas Johnson

Course: MAE 148

Project repository:
https://github.com/davidkariyo/UCSD-Winter-2026-MAE148-Team15-10_Final_Project

Contents

1 Abstract **1**

2 Introduction **1**

3 Related Work and Technical Context **2**

4 System Requirements and Design Goals **2**

4.1 Functional Requirements 3

4.2 Performance Goals 3

4.3 Constraints 3

4.4 Assumptions 3

5 Full System Architecture **4**

5.1 Data Flow 4

5.2 Functional Decomposition 5

6 Mechanical Design **5**

6.1 Chassis and Platform Considerations 6

6.2 Sensor Placement 6

6.3 Tradeoffs 6

7 Electrical and Hardware Integration **6**

8 Software Architecture **7**

8.1 Repository Structure 7

8.2 Execution Flow 8

8.3 Message Passing and Control Interfaces 8

8.4 Timing and Synchronization 8

8.5 Fault Points and Robustness 9

9 Perception and Sensing **9**

9.1 Vision Pipeline 9

9.2 Depth-Based Spatial Reasoning 10

9.3 LiDAR Processing 11

9.4 Noise, Uncertainty, and Failure Modes 12

10 Control System **13**

10.1 Steering Law 13

10.2 Longitudinal Control 13

10.3 Tuning Methodology 13

10.4 Stability and Robustness Discussion 14

11 Planning, Decision Logic, and Autonomy **14**

11.1 Behavioral Sequence 14

11.2 State Machine Interpretation 14

11.3 Algorithmic Tradeoffs 15

12 Implementation Details **15**

12.1 Important Scripts and Functions 15

12.2 Configuration and Parameters 16

12.3 Deployment Workflow 16

12.4 Reproducibility Concerns 17

13 Experimental Methodology **17**

13.1 Subsystem Validation 17

13.2 Controlled Variables 17

13.3 Likely Debugging Workflow 18

14 Results and Technical Analysis **18**

14.1 Directly Supported Outcomes 18

14.2 README-Reported Outcomes 18

14.3 Technical Interpretation 19

14.4 Gap Between Desired and Available Implementation 19

14.5 Root-Cause Analysis of Weaknesses 19

15 Engineering Tradeoffs **19**

15.1 Simplicity Versus Performance 19

15.2 Real-Time Performance Versus Algorithmic Sophistication 19

15.3 Robustness Versus Speed 19

15.4 Modularity Versus Integration Burden . . . 20
15.5 Development Time Versus Ideal Architecture . . . 20

16 Limitations **20**

17 Future Work **20**
17.1 Perception Improvements . . . 20
17.2 Control Improvements . . . 21
17.3 Manipulation Improvements . . . 21
17.4 Electrical and Systems Improvements . . . 21
17.5 Validation Improvements . . . 21

18 Conclusion **21**

A Appendix **22**
A.1 Parameter Tables . . . 22
A.2 Pseudocode . . . 22
A.3 Key Equations . . . 23
A.4 File and Module Summary . . . 23

1 Abstract

We developed an autonomous trash collection platform that combined a mobile robotic base, an OAK-D Lite vision sensor, an LD19 LiDAR, and a custom-mounted SO101 robotic arm. Our system used a YOLO-derived detector deployed to the OAK-D Myriad X accelerator, a ROS2 navigation node that published `/cmd_vel`, and a forward-cone LiDAR pipeline that estimated standoff distance from clustered range returns. Steering was driven by image-centroid error, while stopping was triggered by the nearest forward LiDAR centroid once the robot reached a manipulation-ready distance.

The project emphasized deployable systems engineering rather than algorithmic novelty. We reduced USB load by running inference on the OAK-D, limited sensor update rates to maintain stable timing on the Raspberry Pi host, and kept the controller simple enough to tune on real hardware. The resulting stack was coherent for a senior robotics project because each subsystem had a clear role: perception nominated a target, reactive control aligned the vehicle, LiDAR enforced stopping distance, and the arm handled pickup and deposit.

The implementation also exposed real limitations. The visible detector configuration was single-class, the integrated navigation logic remained reactive rather than map-based, and the delivered repository did not include every deployment artifact needed to reproduce the full end-to-end demo on a new machine. This report documents the implemented system, the sensing and control relations used in code, the engineering tradeoffs behind the design, and the most important limitations and next steps.

2 Introduction

Autonomous mobile manipulation forces perception, control, mechanical design, packaging, and embedded computing to work as one system. Even a task that sounds simple, such as picking up trash from the ground, requires the robot to detect a target in changing lighting, drive into a graspable pose, stop at the correct distance, and execute a reliable pickup sequence with limited compute, power, and time for calibration.

We addressed that problem in the MAE 148 final project by building an autonomous trash collection robot on a UCSD Robocar-style platform. Our goal was straightforward: detect a piece of litter, drive to it, stop at a standoff distance compatible with the arm reach envelope, pick it up, and place it into an onboard bin. The project artifacts show the major pieces of that system: detector deployment files, staged vision and LiDAR test scripts, an integrated ROS2 navigation node, and custom CAD for the manipulator and mounting hardware.

The core engineering challenge was not object detection alone. The harder problem was turning noisy, asynchronous sensor streams into behavior that remained predictable on constrained hardware. We therefore favored decisions that improved reliability in practice: on-device neural inference, a narrow forward LiDAR cone for stop sensing, fixed-rate control, and a simple proportional steering law that we could tune quickly on the real platform.

The implemented project objectives were:

- detect a target object with an onboard OAK-D Lite camera and a trained neural model,

- estimate bearing from image-space geometry,
- estimate stopping distance from LD19 LiDAR returns in the forward direction,
- publish mobile-base velocity commands in ROS2,
- stop at a reach-compatible standoff distance,
- package the SO101 arm, sensors, and collection bin on the vehicle.

This report stays close to the available code and project documentation. Where the repository omitted part of the final deployment, we state that gap directly rather than claiming behavior we cannot support from the delivered materials.

3 Related Work and Technical Context

This project sits at the intersection of four standard robotics areas: embedded vision, short-horizon reactive navigation, range-based stopping, and mobile manipulation.

A full research-grade solution would typically include semantic perception, state estimation, planning, trajectory optimization, and grasp planning. That architecture is powerful, but it is also integration-heavy and expensive in computation. For a compact student platform, a more practical design is a modular reactive stack: detect a target, center it in the image, estimate forward range, then stop and hand control to the arm. That is the approach we implemented.

Our perception pipeline followed a common embedded robotics pattern. A YOLO-family detector provides a strong speed-to-accuracy tradeoff, and our artifact set included `best.pt`, `best.onnx`, and an OpenVINO blob for OAK-D deployment. Compared with hand-tuned color thresholding, this approach generalizes better to variable texture and object shape. Compared with larger transformer-based models, it fits the bandwidth and accelerator limits of the Myriad X pipeline much better.

Our control strategy was intentionally classical. Rather than building a global planner or model-predictive controller, we used proportional steering based on image-centroid error and a fixed forward command. For short-range pursuit of a single target, that choice is easy to interpret, easy to tune, and light enough to run alongside the rest of the stack.

The LiDAR subsystem made the same kind of deliberate simplification. The LD19 can provide dense angular range data, but we only needed a reliable estimate of the nearest object in front of the vehicle. By filtering the scan to a forward cone and clustering the resulting points, we turned a general ranging sensor into a dedicated standoff estimator that matched the needs of the pickup task.

4 System Requirements and Design Goals

The delivered code, model artifacts, and README support the following requirements and design goals.

4.1 Functional Requirements

- Detect a target trash object using an onboard OAK-D Lite camera and trained neural model.
- Estimate object bearing from image-space geometry.
- Estimate stopping distance using LD19 LiDAR returns in the forward direction.
- Publish mobile-base velocity commands in ROS2.
- Stop at a reach-compatible standoff distance.
- Mechanically support an SO101 robotic arm and onboard collection bin.
- Support a downstream pick-and-place action after the approach phase.

4.2 Performance Goals

The project README reported approximately 85% trash detection accuracy, real-time inference, and successful target following. The source files further show the following operating targets:

- camera and control update rate near 10 Hz,
- LiDAR processing with minimal packet load per control cycle,
- stop distance of approximately 0.16 m,
- forward operating range filtered to roughly 0.15 m to 1.50 m.

4.3 Constraints

- limited USB bandwidth on the Raspberry Pi host,
- limited onboard compute, which motivated accelerator-side inference,
- power delivery limitations that affected arm and host stability,
- changing lighting conditions and target appearance,
- a semester-scale timeline for calibration, packaging, and integration.

4.4 Assumptions

Engineering Assumption A1. We treated the repository as a distilled project submission rather than the full deployment workspace. Some runtime packages described in the README are not present, so this report uses the checked-in scripts and artifacts as the authoritative implementation record.

Engineering Assumption A2. The mobile base consumed `/cmd_vel` through the UCSD Robocar stack or an equivalent bridge node, because the navigation script publishes `geometry_msgs/Twist` and does not directly command the motor drivers.

Engineering Assumption A3. The arm pickup sequence existed as a separate integrated subsystem, but not every launch or packaging artifact needed to reproduce that sequence was included in the delivered repository. We therefore discuss manipulation at the system level and keep software claims scoped to the files we can verify.

5 Full System Architecture

Our system architecture was a layered mobile-manipulation pipeline:

1. A trained single-class detector ran on the OAK-D Lite.
2. Neural detections were sent to the host with minimal image traffic.
3. The controller computed image-centroid error from the highest-confidence detection.
4. LiDAR packets were parsed over serial, filtered to a forward cone, clustered, and reduced to a nearest centroid distance.
5. A ROS2 node published `/cmd_vel` using fixed forward speed and proportional steering until the LiDAR stop threshold was reached.
6. After the vehicle stopped, the arm subsystem executed pickup and deposit in the full project workflow.

This architecture was deliberately reactive. We did not build a persistent world model, global map, or long-horizon planner for this task. Instead, the robot used the current best visual detection and the current nearest forward LiDAR cluster to decide immediate motion. That kept the system explainable, light enough for the hardware, and practical to debug.

5.1 Data Flow

The main software data path was:

$$\text{RGB stream} \rightarrow \text{ImageManip} \rightarrow \text{NeuralNetwork} \rightarrow \text{detection tensor}$$

$$\text{serial LiDAR bytes} \rightarrow \text{packet parse} \rightarrow \text{point cloud slice} \rightarrow \text{cluster centroid distance}$$

$$(\text{detection}, \text{distance}) \rightarrow \text{reactive controller} \rightarrow \text{/cmd_vel}$$

We optimized that path for freshness rather than buffering. In the integrated node, inference ran on the OAK-D, queue sizes were reduced to one, and LiDAR processing was limited to one packet per control loop. Those choices reduced stale data at the expense of redundancy, which was the right trade for short-range pursuit.

5.2 Functional Decomposition

The architecture decomposed into four main functions:

- **Detection:** identify candidate trash-like objects with confidence scores and bounding boxes.
- **Spatial gating:** estimate whether an object is within a manipulation-ready range.
- **Approach control:** align and drive the vehicle toward the target.
- **Manipulation:** grasp and deposit the object using the robotic arm.

6 Mechanical Design

The mechanical design was driven by packaging and integration rather than arm kinematics alone. The project repository included STL assets for the SO101 arm links, motor holders, mounting plates, and additional structural parts for the sensor package. That file set shows that we designed and fabricated custom hardware rather than mounting only stock components.

Figure 1 summarizes the integration geometry. We mounted the custom SO101 arm on the vehicle centerline, packaged the sensor mast forward of the arm so the camera and LiDAR kept a clear view of the ground target, and reserved rear volume for the onboard trash bin. That geometry balanced reach, stability, visibility, and service access.

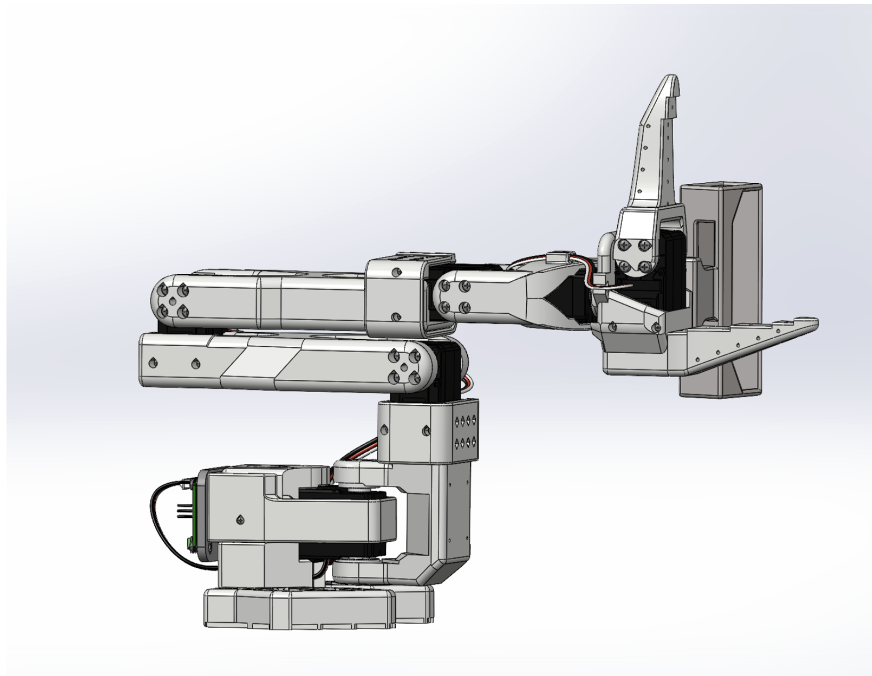

Figure 1: Custom SO101 arm and mounting geometry. The manipulator base is centered on the chassis, the forward sensor mast preserves target visibility, and the rear bin keeps the pickup-to-deposit path short.

6.1 Chassis and Platform Considerations

The README identified a Traxxas-based mobile platform with steering servo and traction drive-train. That was a practical starting point because RC chassis platforms offer robust locomotion, enough payload for sensors and compute, and a well-understood control interface. The challenge was that the base was originally designed for driving, not manipulation. Adding an arm and bin shifted the center of mass, consumed deck space, and introduced cable routing and power-distribution constraints.

To keep the platform usable, the layout had to:

- keep the arm close to the centerline to reduce yaw imbalance,
- place heavier electronics low in the chassis,
- preserve camera line of sight to low targets in front of the robot,
- leave enough access for batteries, converters, USB cabling, and servo maintenance.

6.2 Sensor Placement

The repository also included `Lidar+Cam.STL` and `RobotArmCAM.STL`, which indicates deliberate sensor packaging. That matters because our approach controller used image-space bearing while the stop logic used LiDAR range. If the camera axis and LiDAR forward axis drift apart mechanically, the robot can center the target visually while stopping relative to the wrong physical line. Sensor mounting geometry was therefore part of the control problem, not just a packaging detail.

6.3 Tradeoffs

The main mechanical tradeoffs were:

- **Reach versus stability:** a longer or more forward arm improves pickup envelope but increases pitching moment.
- **Compactness versus serviceability:** tight packaging reduces footprint but makes wiring and repairs harder.
- **Printability versus stiffness:** 3D-printed parts are fast to iterate but introduce compliance and tolerance stack-up.
- **Visibility versus protection:** exposed sensors improve line of sight but are more vulnerable to impact and cable strain.

7 Electrical and Hardware Integration

The main hardware stack consisted of a Traxxas mobile base, a Raspberry Pi 5, an OAK-D Lite camera, an LD19 LiDAR, an SO101 arm with STS3215 servos, a powered USB hub, a motor controller, and a DC/DC power-conversion stage. That selection matched the architecture: the Pi

handled ROS2 and serial I/O, the OAK-D handled inference, and the LiDAR supplied a dedicated stopping measurement.

Table 1: Primary hardware components used in the system.

Subsystem	Hardware
Mobile base	Traxxas chassis, traction motor, steering servo
Motor power/control	Flipsky FSESC 6.7 Pro, antispark switch, servo PDB
Compute	Raspberry Pi 5, 8 GB RAM
Vision sensor	OAK-D Lite camera
Range sensor	LDRobot LD19 LiDAR
Manipulator	SO101 arm, STS3215 servos, motor control board
Power conversion	12V to 12V + 5V DC/DC converter, LiPo 4S battery
I/O expansion	Powered USB 3.0 hub and peripheral cabling

Two integration issues dominated the hardware bring-up: USB bandwidth and power stability. Those are common failure modes on compact mobile robots, and we ran into both. Running inference directly on the OAK-D reduced USB traffic because the device only had to forward neural-network outputs rather than full image streams. We also reduced both the camera and LiDAR update rates to 10 Hz, which improved consistency on the host.

Power delivery was a second major constraint. High-current arm motions can induce brownouts, resets, or noise on shared rails, and the README documented exactly that kind of behavior. The project operated, but the power architecture remained one of the main limits on robustness. A production-grade revision would separate logic and actuation domains more cleanly and leave more margin for transient current draw.

8 Software Architecture

The software was organized around staged testing and then integration.

8.1 Repository Structure

Table 2: Repository-level software and design artifacts.

Path	Role in the project
`README.md`	Project overview, hardware list, accomplishments, and demonstration links.
`Driving_Test_Files/sensingtest.py`	Standalone OAK-D perception and depth script for visualizing detections and estimating object coordinates.
`Driving_Test_Files/lidar_test.py`	Standalone LD19 LiDAR parsing and live forward-cluster visualization tool.
`Driving_Test_Files/trash_navigator.py`	ROS2 node integrating on-device inference, LiDAR standoff estimation, and `/cmd_vel` publishing.
`Driving_Test_Files/best.pt`	Trained PyTorch model artifact.

Path	Role in the project
Driving_Test_Files/last.pt	Alternate or later PyTorch checkpoint.
Driving_Test_Files/best.onnx	Exported ONNX model used for deployment conversion.
Driving_Test_Files/best_openvino_2022.1_6shave.blob	OpenVINO/Myriad deployment artifact for OAK-D inference.
SO101_STL_files/*.stl	Custom manipulator parts and mounts.
Misc_STL/*.STL	Additional sensor and camera mounting geometry.

8.2 Execution Flow

The project evolved through three layers:

1. **Sensor validation:** `lidar_test.py` validated packet parsing, angular filtering, clustering, and centroid range extraction.
2. **Perception validation:** `sensingtest.py` validated OAK-D inference and optional depth-based spatial estimation.
3. **Integrated control:** `trash_navigator.py` fused visual detection and LiDAR distance to drive the robot under ROS2.

That staged workflow was one of the stronger engineering choices in the project. Testing subsystems independently reduced ambiguity during integration and made it easier to isolate issues in sensing, timing, and control.

8.3 Message Passing and Control Interfaces

The visible runtime interfaces were:

- OAK-D output queues for neural inference results,
- serial communication with the LD19 LiDAR,
- a ROS2 publisher on `/cmd_vel`,
- lower-level motor and servo nodes outside the delivered repository.

8.4 Timing and Synchronization

The integrated navigation loop ran on a 0.1 s timer, matching the 10 Hz camera rate. Queue sizes were reduced to one so the controller always acted on the freshest detection available. LiDAR processing was similarly restricted to one packet per cycle in the integrated node. Those choices reduced bandwidth and CPU load, but they also meant the controller had little temporal smoothing if a frame or packet was missed.

8.5 Fault Points and Robustness

The source exposed several predictable failure points:

- If no detections were present, the integrated node stopped updating pursuit commands and did not perform an active search.
- OAK-D startup depended on device state and a valid blob path.
- The integrated script referenced a different blob filename than the one stored in the repository, which is a reproducibility risk.
- The LiDAR stop logic used the nearest forward object, not an identity-matched object associated with the camera detection.
- The delivered scripts did not implement a full task executive for repeated pickup attempts or multi-object collection.

9 Perception and Sensing

We used two sensing modes in the project: a depth-assisted perception script for validation and a lighter detector-plus-LiDAR path for integrated navigation.

9.1 Vision Pipeline

The vision pipeline ran a YOLO-derived detector on the OAK-D Lite. We parsed the raw output tensor directly rather than relying on a higher-level convenience wrapper. For a given prediction row,

$$r = [c_x, c_y, w, h, s_1, s_2, \ldots, s_{n_c}],$$

where c_x and c_y are the box center coordinates, w and h are width and height, and s_k is the class confidence for class k. We select the maximum-scoring class,

$$k^\star = \arg\max_k s_k, \qquad \hat{s} = s_{k^\star},$$

and accept the detection only if $\hat{s} \geq \tau_c$, where $\tau_c = 0.5$. We then apply non-maximum suppression with IoU threshold $\tau_{\text{IoU}} = 0.4$.

In the delivered scripts, the class list is plain text: `CLASSES = ["Raisin Box"]`. That is important context. The visible detector configuration targeted a single surrogate object class rather than a broad taxonomy of trash. That is a common and reasonable intermediate step in robotics validation because it lets the team stabilize the pipeline before expanding the dataset and class coverage.

Figure 2 shows the type of detection output used during the approach phase. We centered the highest-confidence detection in the image and used the centroid offset to generate steering commands.

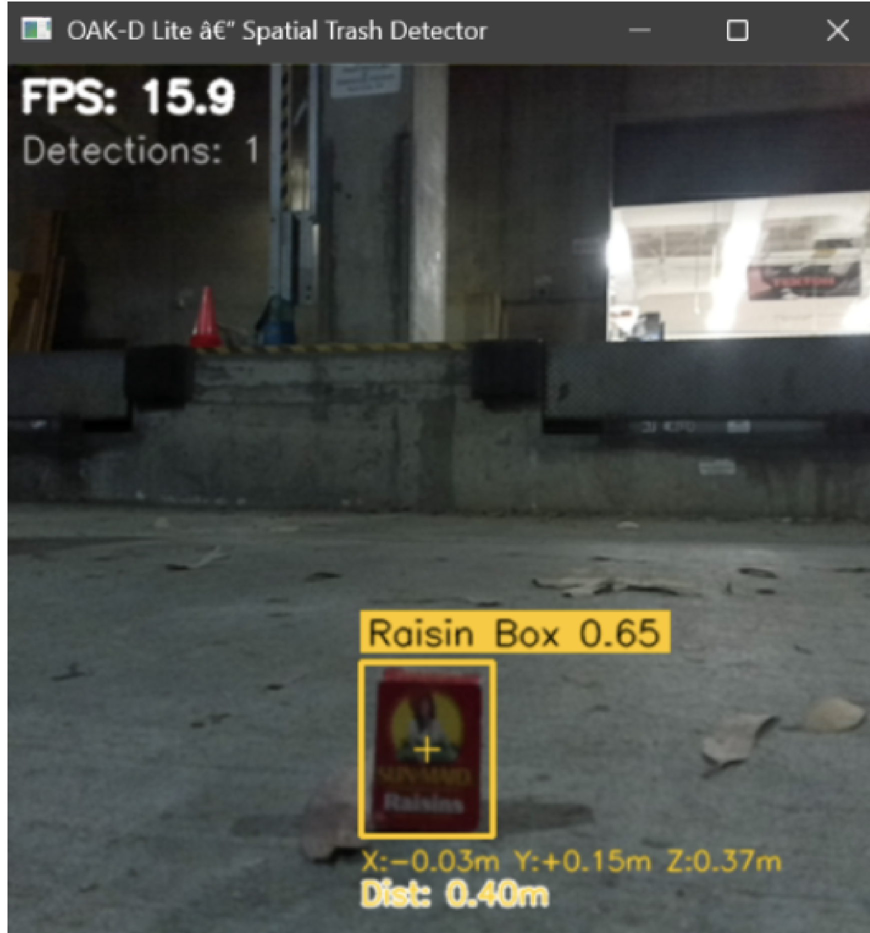

Figure 2: OAK-D Lite detection output used during the approach phase. The detector returns a target bounding box and centroid, and our controller uses the centroid offset from the image centerline to generate steering corrections.

9.2 Depth-Based Spatial Reasoning

The standalone sensing script also computed approximate object coordinates from stereo depth. Given a depth ROI around the box center, the script takes the median valid depth Z and applies a pinhole back-projection:

$$X = \frac{(u - c_x)Z}{f_x}, \qquad Y = \frac{(v - c_y)Z}{f_y},$$

where (u, v) is the ROI center in pixels, (c_x, c_y) are the camera principal point coordinates, and f_x, f_y are focal lengths in pixel units. It then computes Euclidean range as

$$d = \sqrt{X^2 + Y^2 + Z^2}.$$

That method is standard and effective for approximate spatial reasoning, but it depends on valid depth, decent calibration, and consistent alignment between the RGB and depth streams. In our project, it served mainly as a validation and debugging tool; the integrated navigation node relied on camera bearing plus LiDAR stopping distance rather than full depth-based closed-loop control.

9.3 LiDAR Processing

The LiDAR pipeline was deliberately geometric. Each LD19 packet contains 12 range samples spanning an angular interval between a start angle θ_s and end angle θ_e. For sample index $i \in \{0, \dots, 11\}$, we interpolate angle as

$$\theta_i = \theta_s + \frac{i}{11}(\theta_e - \theta_s),$$

with wraparound handling for the $0^\circ/360^\circ$ boundary. Distances outside the accepted interval

$$d_i \notin [d_{\min}, d_{\max}] = [0.15,\ 1.50]\ \mathrm{m}$$

are rejected, and only points inside a forward cone of approximately $\pm 20^\circ$ are retained. Polar measurements are converted to Cartesian coordinates:

$$x_i = d_i \cos\theta_i, \qquad y_i = d_i \sin\theta_i.$$

Adjacent points are clustered using a Euclidean threshold:

$$\|p_i - p_{i-1}\|_2 \leq d_c,$$

where $d_c = 0.08\,\mathrm{m}$. For each surviving cluster C, the centroid is

$$\bar{x} = \frac{1}{|C|}\sum_{p \in C} x, \qquad \bar{y} = \frac{1}{|C|}\sum_{p \in C} y,$$

and the representative distance is

$$d_C = \sqrt{\bar{x}^2 + \bar{y}^2}.$$

The smallest centroid distance becomes the stop measurement.

Figure 3 shows the forward-cone scan geometry that supported this logic. We used the nearest clustered centroid in front of the vehicle as the standoff estimate rather than trying to reconstruct the full environment.

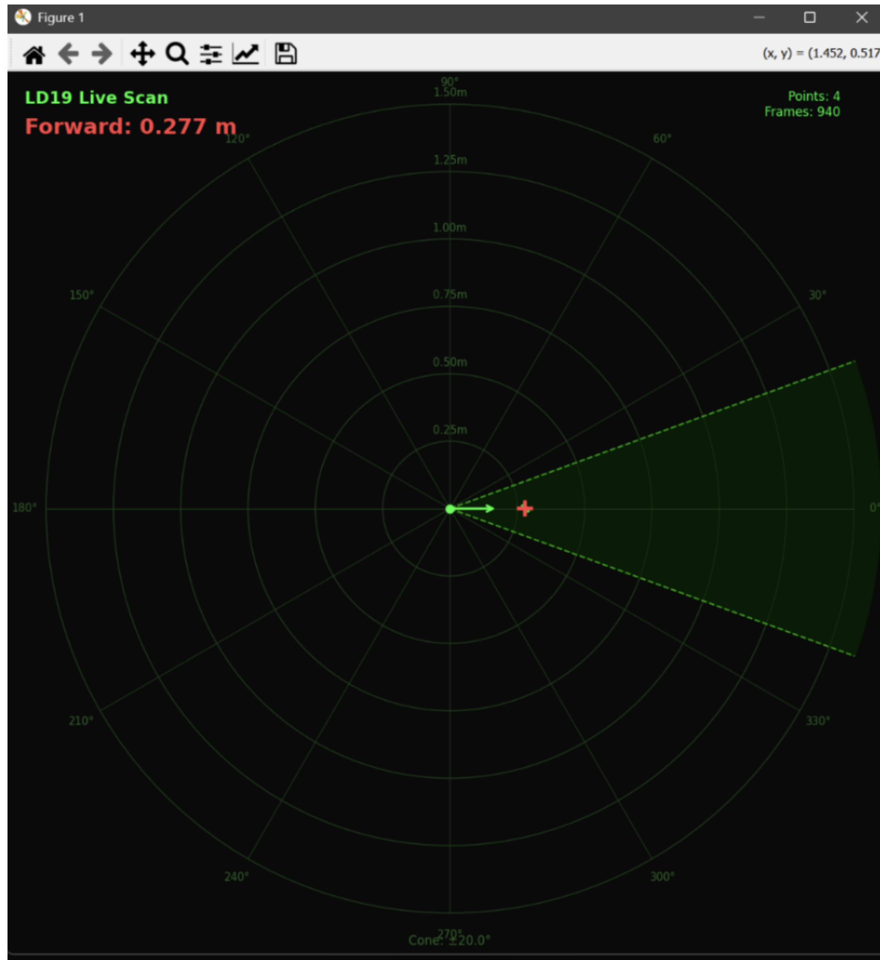

Figure 3: Forward-cone LiDAR processing used for stopping. LD19 returns are filtered to a narrow frontal sector, clustered in Cartesian space, and reduced to the nearest centroid distance, which triggers the stop command once the target enters the manipulation-ready region.

9.4 Noise, Uncertainty, and Failure Modes

The sensing stack was practical but not immune to uncertainty:

- **Vision uncertainty:** lighting changes, shadows, and domain mismatch can suppress detections.
- **Depth uncertainty:** low-texture or reflective surfaces degrade stereo quality and bias the ROI depth estimate.
- **LiDAR uncertainty:** thin objects, oblique surfaces, and fragmented returns can shift cluster centroids.
- **Cross-sensor ambiguity:** the nearest LiDAR object in the forward cone is not guaranteed to be the visually detected target.

10 Control System

The integrated controller had one narrow objective: drive the vehicle toward the detected target while keeping it centered in the image, then stop at a fixed LiDAR distance compatible with the arm reach envelope.

10.1 Steering Law

Let the image width be $N = 416$ pixels. For a selected detection box with center coordinate u, the normalized lateral error is

$$e_x = \frac{u - N/2}{N/2}.$$

This yields $e_x \in [-1, 1]$ when the box center lies within the image bounds.

The control law implemented in `trash_navigator.py` is

$$v = v_{\max}, \qquad \omega = -K_p e_x,$$

where $v_{\max} = 0.2$ is the fixed forward command and $K_p = 1.0$ is the proportional steering gain. The negative sign makes the vehicle steer back toward the target when the image centroid drifts off center.

This is a simple visual-servoing policy. It is computationally cheap, easy to interpret, and appropriate for low-speed pursuit. Its main limits are the absence of integral action, no damping term, and no speed scheduling when the steering error grows.

10.2 Longitudinal Control

The longitudinal policy is a constant-speed advance terminated by a binary stop condition:

$$v = \begin{cases} v_{\max}, & d_{\text{LiDAR}} > d_{\text{stop}}, \\ 0, & d_{\text{LiDAR}} \le d_{\text{stop}}, \end{cases}$$

with

$$d_{\text{stop}} = 0.16\,\text{m}.$$

That is a threshold-based standoff controller rather than a continuous distance controller. It kept the implementation simple, but it also means the final stop point depends on drivetrain lag, surface traction, and control-loop latency.

10.3 Tuning Methodology

The repository did not include a formal tuning log, but the scripts and README make the likely workflow clear:

- tune confidence and NMS thresholds until detections are stable,
- adjust the proportional gain until the vehicle centers the target without large visible oscillation,

- cap the forward speed low enough to keep pursuit controllable,
- tune forward-cone width, clustering threshold, and stop distance from repeated approach tests.

10.4 Stability and Robustness Discussion

At low speed and small image error, the controller should remain locally stable because a centered target generates little steering command while the robot continues forward. For larger errors, behavior depends on the drivetrain and field of view. Because the forward speed is constant rather than reduced with curvature, the vehicle can arc or overshoot if the target drifts near the edge of the frame.

A more refined controller could use a distance- or curvature-dependent speed law such as

$$v = v_{\max} \max(0, 1 - \alpha|e_x|),$$

but we did not need that extra complexity to demonstrate the core approach behavior.

11 Planning, Decision Logic, and Autonomy

The autonomy logic was best described as event-driven reactive behavior rather than classical deliberative planning.

11.1 Behavioral Sequence

The implemented sequence was:

1. Initialize the OAK-D pipeline and LiDAR serial interface.
2. Wait for a detection.
3. If no detection is present, hold rather than execute an active search maneuver.
4. If a detection is present, select the highest-confidence box.
5. Steer proportionally to center the target in the image.
6. Continuously evaluate the nearest forward LiDAR cluster.
7. Stop when the nearest cluster centroid is within 0.16 m.

11.2 State Machine Interpretation

Even though the delivered source did not define a formal finite state machine, the integrated behavior can be interpreted as four implicit states:

- **Initialize**

- **Search/Hold**
- **Approach**
- **Stop**

This interpretation also makes the missing autonomy features clear:

- a transition from stop into arm execution,
- confirmation of successful grasp,
- deposit and reset behavior,
- re-acquisition of additional targets,
- recovery after detection loss or failed pickup.

11.3 Algorithmic Tradeoffs

The reactive architecture had three major advantages: it was easy to reason about, light enough for the platform, and compatible with staged subsystem testing. The cost was limited context awareness and reduced robustness in cluttered or ambiguous scenes.

12 Implementation Details

This section connects the main scripts to their system-level roles.

12.1 Important Scripts and Functions

`sensingtest.py` implemented:

- `parse_yolov8` for manual detector decoding and non-maximum suppression,
- `get_spatial_coords` for depth ROI extraction and pinhole projection,
- `draw_detections` for annotated visual debugging.

`lidar_test.py` implemented:

- serial packet synchronization using the LD19 packet header,
- angle interpolation and point generation,
- adjacency clustering in Cartesian space,
- live radar-style visualization for forward-cone debugging.

`trash_navigator.py` implemented:

- DepthAI pipeline creation with accelerator-side inference,
- serial LiDAR reading with low packet load,
- ROS2 node construction and timer-driven control,
- proportional steering and LiDAR-based stop logic,
- limited runtime recovery when the OAK-D queue throws an exception.

12.2 Configuration and Parameters

The implementation used hard-coded parameters rather than an external configuration system. That simplified setup on the robot but reduced portability and reproducibility.

Table 3: Key parameters visible in the source.

Parameter	Value	Role
`IMG_SIZE`	416	Input resolution for detector pipeline
`CONF_THRES`	0.5	Detection confidence threshold
`IOU_THRES`	0.4	NMS overlap threshold
`STOP_DISTANCE_M`	0.16 m	Vehicle stop distance
`FORWARD_CONE`	20°	LiDAR angular gating
`MIN_RANGE`	0.15 m	LiDAR lower range filter
`MAX_RANGE`	1.50 m	LiDAR upper range filter
`CLUSTER_DIST`	0.08 m	LiDAR adjacency threshold
`MIN_CLUSTER_SIZE`	2	Minimum retained LiDAR cluster size
`KP`	1.0	Proportional steering gain
`MAX_THROTTLE`	0.2	Constant forward command

12.3 Deployment Workflow

The artifact set supports the following deployment sequence:

1. train the detector and save a PyTorch checkpoint,
2. export the model to ONNX,
3. convert the ONNX model to an OpenVINO blob for the OAK-D,
4. validate detection and depth-assisted localization in a standalone script,
5. validate LiDAR parsing and forward-range clustering,
6. run the integrated ROS2 navigation node on the robot.

12.4 Reproducibility Concerns

The repository captured the project well enough for technical review, but not well enough for one-step reproduction on a new machine. The main gaps were:

- missing ROS2 workspace and package manifests,
- host-specific blob paths and serial device names,
- missing launch files, calibration files, and dependency lock information,
- incomplete packaging of the full arm-control deployment path.

13 Experimental Methodology

The repository did not include a formal benchmark suite, but the script organization reveals a credible validation path.

13.1 Subsystem Validation

A reasonable test sequence for this system was:

1. validate OAK-D inference and bounding-box quality in static scenes,
2. validate depth-based coordinate estimates against known distances,
3. validate LiDAR packet decoding and centroid distance on isolated objects,
4. validate integrated pursuit behavior with a single target in a controlled space,
5. validate stop distance against the arm reach envelope,
6. validate pickup and deposit separately before full integration.

13.2 Controlled Variables

The most important controlled variables were:

- object type and appearance,
- ambient lighting and shadows,
- initial target angle and range,
- floor surface and wheel traction,
- battery state of charge,
- sensor frame rate and USB port assignment.

13.3 Likely Debugging Workflow

The project followed a familiar mechatronics loop:

- isolate the failing subsystem,
- reduce operating rate to regain stability,
- add visualization to inspect sensor behavior,
- adjust thresholds and gains empirically,
- retest under increasingly integrated conditions.

14 Results and Technical Analysis

The project evidence is a mix of direct implementation facts and README-reported outcomes.

14.1 Directly Supported Outcomes

The following outcomes are supported directly by the delivered files:

- a working OAK-D perception script exists for object detection and depth-assisted spatial estimation,
- a working LiDAR processing script exists for live forward-object ranging,
- an integrated ROS2 node exists that publishes approach commands and stops on a LiDAR threshold,
- trained model and deployment artifacts were produced,
- custom CAD hardware was designed for the manipulator and mounts.

14.2 README-Reported Outcomes

The README further reported:

- approximately 85% trash detection accuracy,
- successful OAK-D plus YOLO integration,
- successful object following using centroid tracking,
- successful stopping before contact,
- arm pickup and deposit demonstrations,
- operation inside a modified UCSD Robocar Docker environment.

Those claims are plausible and consistent with the artifact set, but they are not all independently reproducible from the repository alone.

14.3 Technical Interpretation

The most important result of the project was not a new perception algorithm. It was that we found a workable systems balance among compute, bandwidth, sensing scope, and control simplicity. Running inference on the OAK-D, minimizing queue depth, and reducing LiDAR data to a task-specific stopping estimate kept the system within the hardware envelope. That is often what separates a working robot from an overbuilt prototype that never stabilizes.

14.4 Gap Between Desired and Available Implementation

The intended system behavior was full autonomous trash collection, including approach, pickup, and deposit. The available code clearly demonstrates the approach and stopping subsystems. The remaining gap is that the delivered repository does not package the full end-to-end runtime in a way that a new user could reproduce without additional files, configuration, and calibration work.

14.5 Root-Cause Analysis of Weaknesses

The main weaknesses align with the project constraints:

- **USB bottlenecks** forced lower sensing rates and tighter queue management.
- **Power limitations** constrained integrated arm-plus-vehicle operation.
- **Calibration burden** affected manipulation reliability and sensor alignment.
- **Repository incompleteness** limited portability and external reproducibility.

15 Engineering Tradeoffs

15.1 Simplicity Versus Performance

The proportional controller and hard stop threshold are far simpler than a full visual-servoing or MPC pipeline. That simplicity made the system easier to tune and more likely to work on schedule, at the cost of smoother final approach behavior and richer failure recovery.

15.2 Real-Time Performance Versus Algorithmic Sophistication

Accelerator-side inference and low queue depth traded raw perception richness for reliable embedded timing. That was the correct trade on this platform.

15.3 Robustness Versus Speed

A fixed low throttle slowed the task but improved controllability and reduced overshoot near the target, which matters more than cycle time during the approach-to-grasp phase.

15.4 Modularity Versus Integration Burden

Separating perception, navigation, and manipulation made debugging easier, but it also increased the amount of cross-module calibration and packaging work needed to deliver a completely reproducible system.

15.5 Development Time Versus Ideal Architecture

A more ambitious system would add target tracking, better state management, power monitoring, and structured logging. We stopped at the boundary where the architecture remained understandable and workable on the real hardware.

16 Limitations

The main limitations of the delivered system were:

- the visible detector configuration was single-class,
- the integrated node did not include an active search behavior after detection loss,
- the LiDAR stop condition did not identity-match the nearest range return to the camera detection,
- pathing remained purely reactive with no obstacle avoidance or global planning,
- power delivery remained a major reliability constraint during arm operation,
- portability was reduced by host-specific paths and missing deployment artifacts,
- the repository evidence was qualitative rather than statistically rigorous.

17 Future Work

17.1 Perception Improvements

- expand the detector to multiple trash classes with a broader outdoor dataset,
- log false positives and false negatives under controlled lighting changes,
- add lightweight temporal filtering to reduce one-frame detection loss,
- improve camera-to-LiDAR calibration and object association.

17.2 Control Improvements

- add speed scheduling as a function of steering error and range,
- replace the hard stop threshold with a continuous near-target distance controller,
- add target reacquisition behavior when detections disappear,
- incorporate odometry to stabilize approach trajectories.

17.3 Manipulation Improvements

- package the arm controller with the main repository and document calibration procedures,
- estimate object pose in the arm frame rather than relying on standoff-only positioning,
- add grasp verification using current, encoder, or simple contact feedback,
- parameterize pickup trajectories for different object sizes and shapes.

17.4 Electrical and Systems Improvements

- redesign the power system with cleaner rail isolation and more current margin,
- add health monitoring for voltage sag and device disconnects,
- formalize launch procedures and device mapping for repeatable deployment,
- add synchronized logging for detections, distances, and velocity commands.

17.5 Validation Improvements

- define quantitative metrics such as stop-distance error, pickup success rate, and cycle time,
- run repeated trials across lighting, surface, and target-placement conditions,
- compare the current reactive controller against at least one improved baseline.

18 Conclusion

This project demonstrates a credible autonomous trash collection system built under real hardware constraints. We combined on-device neural inference, geometric LiDAR processing, ROS2 vehicle control, and custom mechanical integration into a platform that could detect a target, drive toward it, stop at a useful distance, and support arm-based collection. The main engineering achievement was not novelty in any single subsystem; it was the fact that the sensing, compute, power, and packaging choices were reduced to something that could operate together on a real robot.

The remaining gaps are also instructive. The system still depended on careful calibration, a constrained target class, and incomplete packaging of the final deployment stack. Those are normal boundaries for a semester robotics build, and they define the most important next steps for a more

field-ready version. As delivered, the project shows solid embedded robotics engineering and a clear understanding of how to trade architectural ambition for a system that can actually run.

A Appendix

A.1 Parameter Tables

Table 3 summarizes the principal sensing and control parameters exposed in the source. For convenience, the main runtime values are repeated below.

Table 4: Condensed operational parameters for implementation and tuning.

Parameter	**Nominal Value**	**Interpretation**
Input image size	416×416	Detector operating resolution
Camera frame rate	10 Hz	OAK-D runtime in integrated node
LiDAR packet load	1 packet/loop	Minimal serial traffic in integrated node
Forward cone	$\pm 20°$	Acceptable stop-sensing region
Stop threshold	0.16 m	Manipulation-ready standoff target
Steering gain K_p	1.0	Image-centroid proportional gain
Throttle command	0.2	Constant longitudinal drive command
Cluster threshold	0.08 m	LiDAR point adjacency rule
Minimum cluster size	2 points	LiDAR cluster retention threshold

A.2 Pseudocode

The integrated autonomy behavior can be summarized as:

```
Initialize OAK-D device and load OpenVINO blob
Initialize LiDAR serial port
Create ROS2 publisher for /cmd_vel

Loop at 10 Hz:
    Read nearest forward LiDAR cluster distance
    If distance <= stop_threshold:
        publish zero Twist
        continue
```

```
Read latest neural network output
If no detection:
    hold current state
    continue

Select highest-confidence detection
Compute normalized image-center error
Set linear velocity = constant throttle
Set angular velocity = -Kp * image_error
Publish /cmd_vel
```

A.3 Key Equations

For clarity, the main equations used by the repository are collected here:

$$e_x = \frac{u - N/2}{N/2}$$

$$\omega = -K_p e_x$$

$$\theta_i = \theta_s + \frac{i}{11}(\theta_e - \theta_s)$$

$$x_i = d_i \cos\theta_i, \qquad y_i = d_i \sin\theta_i$$

$$d_C = \sqrt{\bar{x}^2 + \bar{y}^2}$$

$$X = \frac{(u - c_x)Z}{f_x}, \qquad Y = \frac{(v - c_y)Z}{f_y}, \qquad d = \sqrt{X^2 + Y^2 + Z^2}$$

A.4 File and Module Summary

Artifact	Engineering significance
`trash_navigator.py`	Integrated autonomy file; combines on-device inference, ROS2 command generation, and LiDAR stop logic.
`sensingtest.py`	Demonstrates depth-assisted object localization and perception debugging.
`lidar_test.py`	Validates LiDAR parsing, clustering, and forward-range estimation.
Model artifacts	Capture the deployment path from training to OAK-D inference.
STL assets	Show that the project included custom hardware integration and not just software integration.