ECE 445

SENIOR DESIGN LABORATORY

FINAL REPORT

Autonomous Ammunition Loading and Firing Robotic System

Team #35

YIDONG ZHU
YUXUAN NAI
XINCHEN YAO
XIAOMAN LI

Course: ECE 445 Senior Design Laboratory

May 15, 2026

Abstract

This report presents the final design, implementation, and verification of Meta-Dart, an autonomous dart loading and firing robotic system for competitive robotics. Fixed-coordinate loaders are vulnerable to jams and missed pickups when darts shift during robot motion or staging. Meta-Dart addresses that problem with a camera-based perception module, a YOLO oriented-bounding-box detector exported to Open Neural Network Exchange (ONNX), a C++ decision state machine, a Dynamixel robotic arm, and a Controller Area Network (CAN)-controlled spring stretcher for launch actuation. The final implementation successfully integrates perception, slot selection, disturbance-aware pickup, gripper loading, launcher triggering, and staged hardware tests. In final-demo trials, the system loaded 18 of 20 darts successfully, corresponding to a 90.0% retrieval-and-loading success rate, and launched 15 of 20 loaded darts successfully, corresponding to a 75.0% mechanical discharge success rate. The measured integrated cycle time was 34.4 s on average, with a 30 s minimum and a 46 s maximum, based on stopwatch timing and video review. A 100-cycle endurance run completed 92 cycles, with no motor communication failure and no unsafe hardware failure. The design therefore met the 80% retrieval-and-loading success-rate goal, but it did not meet the 80% launch-reliability target, the original 5 s cycle-time target, or the strict 100 consecutive-cycle endurance target. The dominant launch failure was traced to excessive spring preload: high spring force increased latch friction, and the MG995 trigger servo could not reliably release the mechanism.

Contents

1 Introduction

1.1 Problem Statement

Competitive robotics platforms such as RoboMaster rely on rapid and repeatable ammunition handling. Vision-based robotic systems commonly use lightweight object detection to localize targets in real time [1], while modern deployment frameworks such as ONNX Runtime make trained neural-network models practical to run inside C++ control software [2]. A conventional loader often assumes that darts remain at fixed mechanical coordinates. That assumption is fragile: robot motion, field vibration, collisions, and operator staging variability can move a dart away from a preprogrammed pickup location. When that happens, the loader can miss the dart, push it away, or jam the launch path. The resulting downtime reduces match performance and may require manual intervention.

The engineering goal of this project was to build an autonomous mechanism that can locate an unstructured dart, move a robotic gripper to the appropriate pickup path, load the dart into the launcher, and trigger a repeatable discharge sequence. The project does not attempt precision ballistic targeting. It prioritizes reliable retrieval, loading, and mechanical launch execution.

1.2 Final Solution Overview

The final implementation is a modular robotic loader named Meta-Dart. A Linux computer runs the production executable, `loader`, which starts the camera module, performs ONNX Runtime inference on a trained YOLO oriented-bounding-box model, publishes dart coordinates, selects a configured slot, and commands the Dynamixel arm through named waypoints. Oriented bounding boxes were selected because the detected dart pose includes both position and rotation, which is more useful for grasp planning than an axis-aligned box when a dart appears at different angles [3]. Once the arm reaches the `loading` position and opens the gripper, the spring-stretcher module commands a two-motor CAN stretch-and-retract cycle for the launching mechanism.

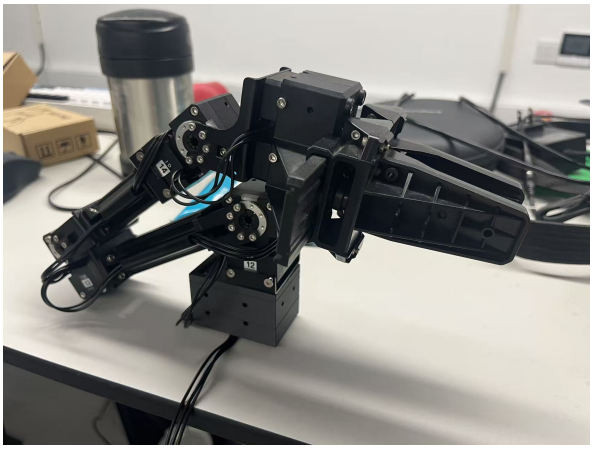
Figure 1 shows the two main physical subsystems: the arm and gripper assembly and the launcher/staging hardware.

1.3 High-Level Requirements

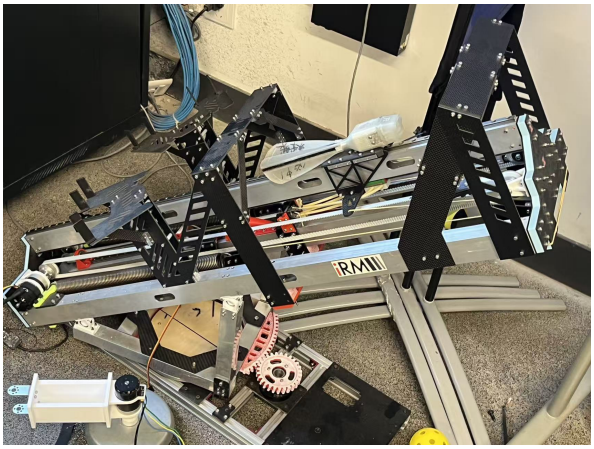
The final performance requirements were inherited from the proposal and design document, with minor interpretation changes caused by the final hardware architecture.

1. **Autonomous retrieval rate:** the perception and manipulation subsystems shall detect, grasp, and load a randomly placed projectile in the staging area with a success rate of at least 80%, and each successful cycle shall complete in under 5 s.

2. **Launch execution reliability:** after the projectile is grasped and placed in the loading



(a) Robotic arm and gripper assembly.



(b) Launcher and staging hardware.

Figure 1: Physical prototype hardware used for autonomous dart retrieval and loading.

port, the firing mechanism shall discharge in the intended forward direction with at least 80% mechanical success, independent of ballistic targeting accuracy.

3. **System endurance:** the integrated electromechanical system shall complete 100 consecutive reload-and-fire cycles without a critical software crash, motor communication failure, or unsafe hardware failure.

1.4 System Block Diagram

Figure 2 shows the hardware-level architecture. The original proposal described a perception module, control module, actuation module, and power module. The final design preserves those blocks, but the control role is primarily handled by a Linux runtime rather than a standalone STM32 controller. The Linux computer owns high-level perception, decision logic, configuration lookup, and trajectory sequencing. Motor-level communication is delegated to Dynamixel serial and CAN interfaces; this separation is consistent with distributed robotic platforms that use CAN for reliable actuator communication [4,5] and with the ROBOTIS SDK model for packet-based Dynamixel control [6].

Figure 3 shows the final software data flow. Camera frames are processed by the vision module, detected darts are passed to the decision module, and the decision module reads slot and waypoint parameters from the runtime configuration before commanding the arm and launcher. Separating runtime configuration from compiled control logic made the final system easier to tune because slot bounds, trajectory names, timing thresholds, and pickup mode could be adjusted without rebuilding the executable.

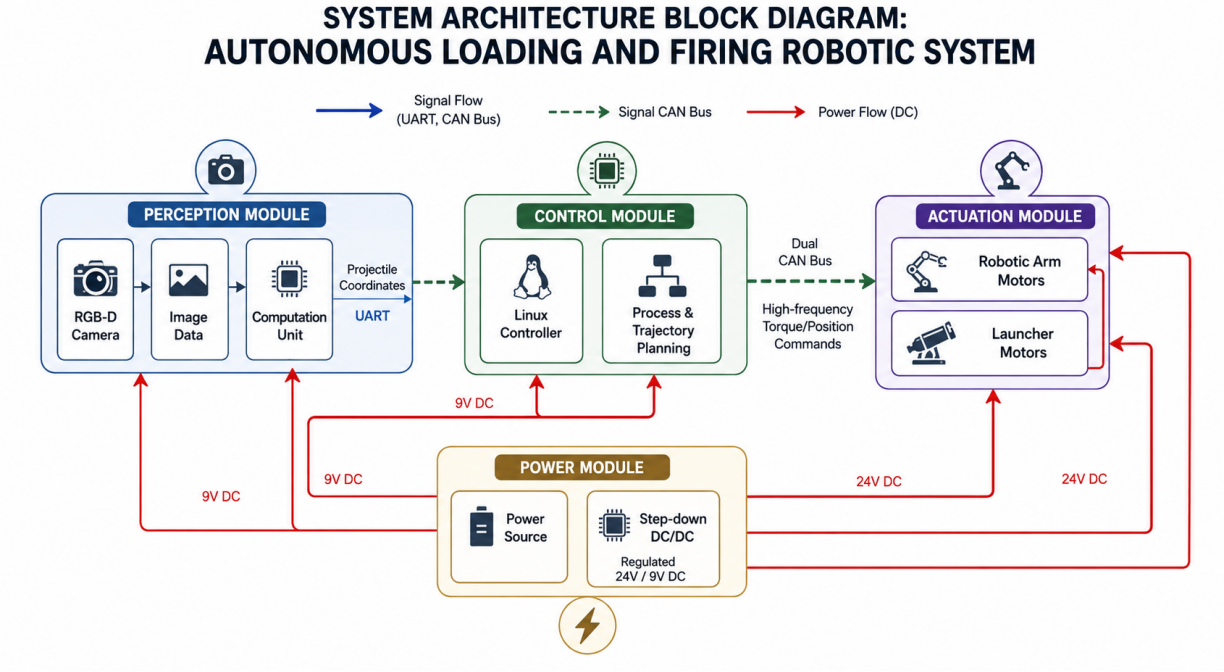


Figure 2: Hardware-level architecture used to organize perception, control, actuation, and power interfaces.

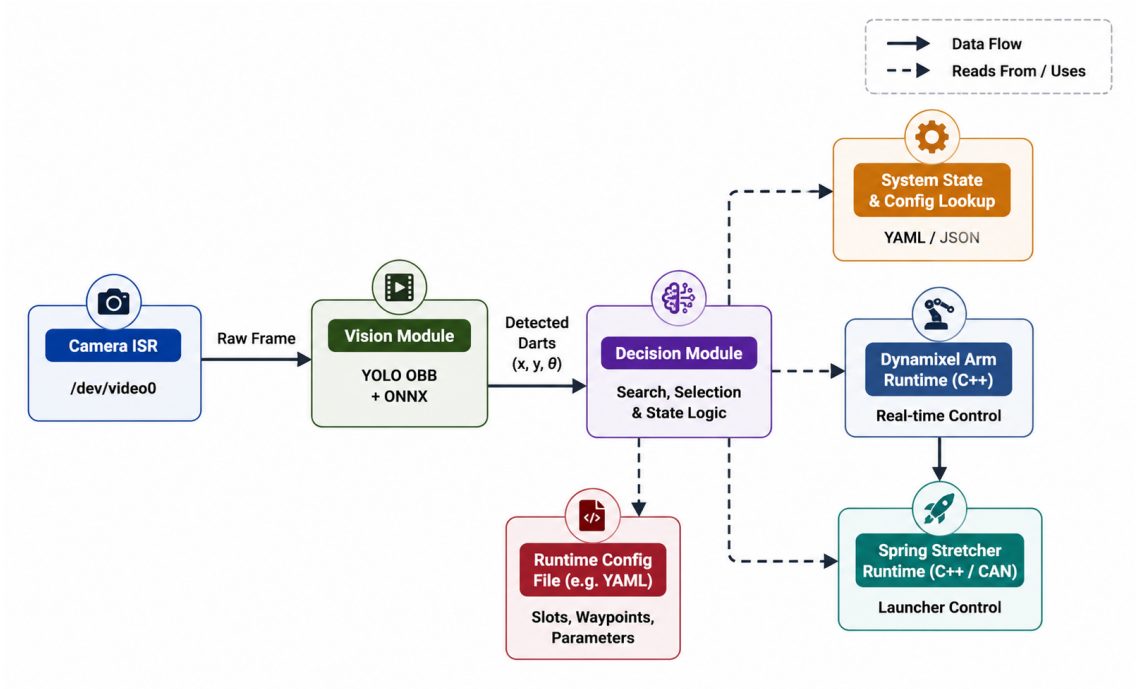


Figure 3: Final software/runtime data flow from camera frames to decision logic, arm control, runtime configuration, and launcher control.

2 Design

2.1 Repository and Runtime Architecture

The final codebase is organized around one production executable and several staged hardware tests. The production executable, `loader`, is defined in the CMake build file and links the camera, vision, arm, decision, runtime-configuration, and spring-stretcher modules. The repository also contains the trained ONNX model, a JSON runtime configuration, calibrated arm waypoints, and hardware-specific test targets. This organization keeps perception, decision logic, motor control, and launcher control as separate blocks, matching the hardware architecture in Fig. 2.

Table 1: Primary software modules used by the final system.

Module	Files	Responsibility
Camera	`src/camera.*`	Captures OpenCV frames from the configured camera index.
Vision	`src/vision.*`, `models/best.onnx`	Runs YOLO/ONNX inference and converts the best dart detection to planar coordinates.
Arm	`src/arm.*`,`positions.xml`	Loads named waypoints, initializes Dynamixel motors, interpolates joint commands, and controls the gripper.
Decision	`src/decision.*`	Implements homing, searching, guarded pickup, disturbance recovery, loading, and reset behavior.
Launcher	`src/spring_stretcher.*`	Controls two DJI M3508 motors over SocketCAN for spring stretching and retraction.
Configuration	`src/runtime_config.*`, `loader_config.json`	Parses thresholds, slot definitions, trajectory lists, timing parameters, and pickup mode.

The software uses producer-consumer buffers between camera, vision, decision, and arm-state loops. This design prevents the vision loop from blocking arm state updates and allows the camera-only test to run without enabling the arm. It also lets the main program continue in a camera-only diagnostic mode if the arm fails to initialize, which is important because camera validation and actuator validation have different safety risks.

2.2 Perception and Dart Localization

The perception subsystem uses an RGB camera, OpenCV frame capture, and ONNX Runtime inference. OpenCV provides the frame-acquisition and image-processing interface [7], while ONNX Runtime provides the C++ inference interface used to execute the exported model [8]. The model input is resized to 640×640 px, converted from BGR to RGB, normalized to $[0, 1]$, and packed into a tensor. The postprocessor follows

the Ultralytics oriented-bounding-box representation, which reports center coordinates, width, height, class confidence, and rotation angle [3]. The runtime selects the highest-confidence dart above the configured threshold.

The final runtime configuration sets the vision confidence threshold to 0.20. The threshold can also be overridden through the `META_DART_CONF_THRESHOLD` environment variable. The camera-to-world conversion uses a downward-facing pinhole model:

$$X = \frac{u - c_x}{f_x} H, \qquad Y = \frac{v - c_y}{f_y} H \tag{1}$$

where u and v are scaled pixel coordinates, c_x and c_y are principal-point estimates, f_x and f_y are focal lengths in pixels, and H is the camera height above the dart surface. The configured values are $f_x = f_y = 600$ px, $c_x = 320$ px, $c_y = 240$ px, and $H = 0.50$ m.

2.3 Decision Logic and Disturbance Recovery

The decision subsystem implements homing, searching, trajectory execution, reset cooldown, and idle states. During homing, the arm moves through the search-ready sequence `home`, `forward_prep`, and `backward_prep`. During searching, a dart must remain inside a configured slot for 1.0 s before the pickup trajectory is triggered.

Four slots are configured in `loader_config.json`. Each slot has a nominal center, a search half-width and half-height, a larger pickup guard area, and a trajectory name. Each slot trajectory moves through a preparation waypoint, grasp waypoint, pulled-out waypoint, shared transfer waypoints, `loading`, `high_prep`, and `backward_prep`.

The most important final design improvement is disturbance handling. In `disturb` mode, the arm monitors the latest detection while approaching the dart. If the dart leaves the pickup guard area longer than the configured grace time, the arm pauses at the current joint positions, holds for 5.0 s, watches for a relocated stable slot during a 4.0 s window, and restarts the pickup path through the search pose if a new slot is accepted. This design directly addresses the original problem that darts can move after the loader has already started reaching. The recovery behavior trades speed for robustness: instead of forcing the original path to finish, the system waits for a stable detection and then restarts with a trajectory matched to the new slot.

2.4 Arm Actuation and Trajectory Generation

The arm subsystem uses five Dynamixel motors: four joints and one gripper. The configured IDs are 11 through 15 over `/dev/ttyUSB0` at 1 Mbps. Dynamixel actuators support packet-based serial control through the ROBOTIS SDK [6], and X-series servos support extended-position and current-based position modes that are appropriate for multi-turn joints and grippers [9]. The joint control loop runs at 100 Hz. Named waypoints are stored in `positions.xml`; the final file contains the home pose, loading pose, shared preparation points, and slot-specific pickup waypoints for four dart slots.

The arm controller reads present joint positions before enabling torque, which prevents the arm from snapping to a default goal during startup. It uses extended position mode for the four arm joints and current-based position mode for the gripper. Joint moves are interpolated with a quintic S-curve,

$$s(t) = 6t^5 - 15t^4 + 10t^3 \tag{2}$$

which gives zero velocity and zero acceleration at both endpoints. To reduce setpoint chatter, the controller writes joint goals no faster than every 20 ms unless the final setpoint must be written, and it suppresses updates below 0.002 rad.

Segment duration is computed from the maximum joint delta and a configured cruise speed of 0.45 rad/s. The duration is clamped between 2.0 s and 9.0 s for regular moves, while non-guarded transfers use a 1.6× speedup. This tuning is slower than the original 5 s ideal cycle requirement, but it reduced arm shaking and improved reliability during final integration.

2.5 Launcher and Spring-Stretcher Subsystem

The launcher subsystem is implemented as a spring stretcher rather than as a simple open-loop motor trigger. It controls two DJI M3508 motors over the SocketCAN interface `can0`. SocketCAN exposes CAN devices through the Linux networking stack and Berkeley socket API, which allows the launcher controller to use standard Linux I/O patterns rather than a device-specific character interface [5]. The loop runs at 500 Hz and uses feedback frames to estimate position, velocity, and current. On startup, the subsystem calibrates both hooks toward the top hard stops using bounded current, marks each top stop as zero after velocity remains below the jam threshold, and times out safely if calibration does not finish.

After calibration, a stretch command sets symmetric target positions based on configured motor directions. The controller uses a cascaded position-to-velocity-to-current loop with current saturation at 10 A. Once the stretch target is reached and held for 0.25 s, the module retracts the motors to home. The decision module issues this stretch command immediately after the arm reaches `loading` and opens the gripper.

2.6 Mechanical Prototype and CAD

The mechanical design integrates the manipulator, launcher, and support frame into a compact prototype so the gripper can transfer a dart into the loading path without a separate staging mechanism. Figure 4 shows the final computer-aided design (CAD) assembly. The waypoint-based software design matches this mechanical layout: instead of solving arbitrary free-space manipulation during the final demo, the arm moves through calibrated physical poses that fit the frame, gripper, and launcher geometry.

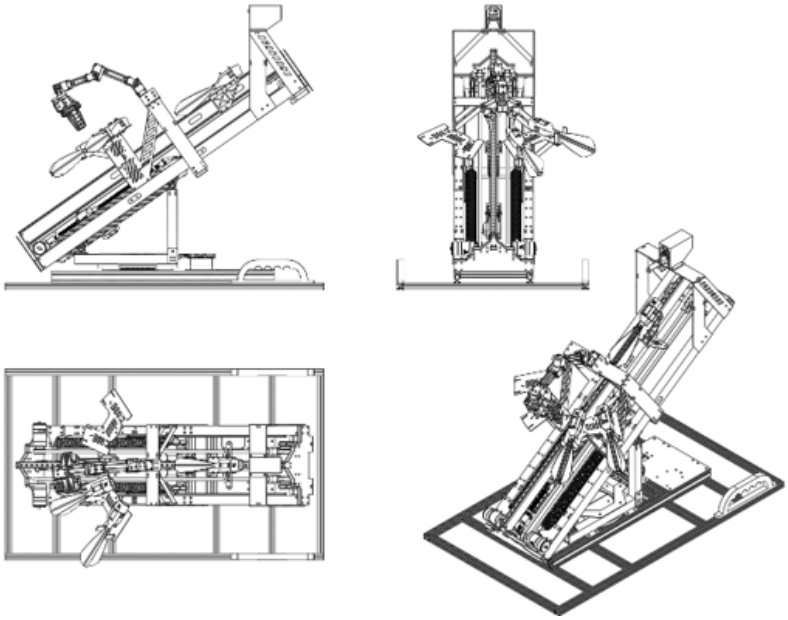


Figure 4: CAD assembly of the integrated arm, launcher, and chassis from the design document.

2.7 Tolerance Analysis

The key tolerance is the grasping error between the gripper center and the dart center. If the dart diameter is $d_p = 40$ mm and the gripper opening is $w_g = 100$ mm, the maximum allowable translational error is

$$E_{\max} = \frac{w_g - d_p}{2} = \frac{100\ \text{mm} - 40\ \text{mm}}{2} = 30\ \text{mm}. \tag{3}$$

At an arm extension of $L = 0.5$ m, yaw error produces an approximate lateral error

$$\Delta s = L\Delta\theta. \tag{4}$$

Maintaining $\Delta s \leq 0.030$ m requires

$$\Delta\theta \leq \frac{0.030}{0.5} = 0.06\ \text{rad} = 3.44^\circ. \tag{5}$$

The design conservatively allocates yaw error to $\pm 1.5^\circ$ to leave margin for camera calibration error, gripper compliance, multi-joint propagation, and waypoint repeatability. In the final code, this tolerance is enforced through calibrated named waypoints, guarded pickup boxes, and physical waypoint validation utilities.

2.8 Design Alternatives

Table 2 summarizes the main design choices and corrective actions made during implementation.

Table 2: Design alternatives and implementation decisions.

Issue	**Alternative Considered**	**Final Decision**
Coordinate-based pickup	Use fixed coordinates only, as in many standard loaders.	Use camera detection plus slot-based waypoints so the system can select among observed slots.
Response to dart motion	Replay one full trajectory after detecting a dart.	Add `disturb` mode so the arm pauses and restarts when the dart leaves its pickup guard box.
Arm smoothness	Send abrupt joint goals directly to motors.	Use quintic S-curves, bounded cruise speeds, and goal-write deadband to reduce shaking.
Configuration	Hard-code zones and trajectories in C++.	Move slots, trajectories, timing, and pickup mode to JSON configuration so tuning does not require recompilation.
Launcher control	Treat launcher as a simple on/off actuator.	Add a calibrated two-motor spring-stretcher controller with feedback, current limiting, and automatic retract.
Testing	Test only the full system.	Add separate camera, arm, named-position, motor-scan, and spring-stretcher tests so risky hardware is verified in stages.

3 Verification

3.1 Verification Strategy

The ECE 445 final-report guidelines recommend that verification discussion focus on major requirements and move detailed requirement tables to an appendix when the full table would interrupt the main narrative [10]. Following that structure, this section summarizes the main verification results, while Appendix A contains the detailed requirement-by-requirement matrix.

The final verification strategy used staged testing. The camera and vision pipeline were tested without arm motion; the arm and gripper were tested through named waypoint utilities; the spring stretcher was tested with separate monitor, direction, calibration, and stretch-cycle programs; the trigger was tested as a release actuator under launcher preload; and the full `loader` was used for integrated pickup and launch trials. This staged procedure reduced hardware risk because individual subsystems could be verified before enabling the full autonomous sequence.

3.2 Subsystem Tests

Table 3 lists the hardware and software tests available in the final repository. These tests are not substitutes for final trial statistics, but they provide repeatable procedures for checking each block before a demonstration.

3.3 High-Level Requirement Results

Table 4 summarizes the high-level verification results from the 20-trial final-demo run and the 100-cycle endurance run. The success rate was computed as

$$\text{success rate} = \frac{\text{successful trials}}{\text{total trials}} \times 100\%. \tag{6}$$

Cycle time was measured by stopwatch and video review from stable dart acceptance to successful discharge, launcher retraction, and return of the arm to the search-ready pose. One-time initialization steps, including camera startup, arm homing, and spring-stretcher calibration, were excluded. The original 5 s cycle-time requirement was not met after safe-speed tuning; the final motion profile prioritized reliable manipulation over maximum speed.

3.4 Hardware Release Test and Failure Diagnosis

The final integrated hardware test separated the task into retrieval/loading and launch-release stages because the image processing and arm transfer succeeded more reliably than the mechanical discharge. The retrieval/loading stage completed 18 of 20 trials, or 90.0%. The launch-release stage completed 15 of 20 trials, or 75.0%. Stopwatch and video

Table 3: Software and hardware verification entry points in the repository.

Test Target	Purpose
`check_meta_dart_environment.sh`	Checks repository files, C++ tools, ONNX Runtime, Dynamixel sources, user groups, serial device, camera device, Python vision dependencies, and OpenCV camera access.
`test_vision_camera`	Runs camera and YOLO/ONNX only; no arm motion. Reports valid detection counts and detected coordinates.
`test_arm`	Connects to arm motors, moves to home, tests the gripper, and sweeps axes.
`test_named_positions`	Moves through named waypoints one at a time so the operator can verify physical positions.
`test_trajectory` and `test_gripper_trajectory`	Exercise arm trajectories and gripper actions before full autonomous mode.
`test_stretcher_*`	Monitor, direction, calibration, and stretch-cycle tests for the spring-stretcher launcher subsystem.
`test_mg995_servo` and integrated trigger checks	Command the trigger servo between hold and release positions, then verify that the physical latch can release under launcher preload.
`loader -check-config`	Validates runtime configuration without starting hardware.

Table 4: Summary of high-level verification results.

Requirement	Final Result	Conclusion
Autonomous retrieval and loading success rate at least 80%.	18 successful loads in 20 trials, or 90.0%.	Success-rate requirement met.
Successful cycle time less than 5 s.	Mean cycle time was 34.4 s, with a 30 s minimum and a 46 s maximum.	Speed target not met; the measured integrated cycle was substantially slower than the original target.
Mechanical discharge success rate at least 80% after loading.	15 successful discharges in 20 trials, or 75.0%.	Requirement not met; launch reliability was limited by the mechanical release.
100 consecutive reload-and-fire cycles without critical failure.	92 of 100 cycles completed. Minor software crashes interrupted the remaining cycles; no motor communication failure or unsafe hardware failure occurred.	Endurance target partially met, but the strict 100 consecutive-cycle requirement was not met.

review showed a mean integrated cycle time of 34.4 s, with observed cycles ranging from 30 s to 46 s.

The dominant failed-trial mechanism was mechanical release friction at the latch. When the spring stretcher pulled the launcher to a high preload, the spring reaction force increased the normal force at the latch contact. Since friction grows with normal force,

$$F_{\text{friction}} = \mu N, \tag{7}$$

the trigger servo had to overcome a larger static friction force before the latch could move. In several failed launch trials, the software issued the trigger release command, but the MG995 servo did not move the latch far enough to release the stored spring energy. This indicates that the launch limitation was primarily a mechanical preload and friction problem rather than a camera, slot-selection, or Dynamixel arm-control problem.

The main corrective action is to reduce latch preload and friction. Practical improvements include reducing the spring stretch target, polishing or changing the latch contact geometry, adding a bearing or roller contact, increasing trigger-servo torque margin, and adding a sensor to confirm whether the latch physically moved after a release command.

3.5 Quantitative Implementation Parameters

Table 5 lists quantitative implementation parameters from the final code and runtime configuration. These values support repeatability for future verification runs.

3.6 Failure Modes and Mitigations

Table 6 summarizes the main observed or expected failure modes and their mitigations. These failures guided the final test procedure and the decision to keep motion speeds conservative.

Table 5: Key quantitative parameters from the current code and configuration.

Parameter	**Value**
Vision input resolution	640×640 px
Runtime confidence threshold	0.20
Camera model parameters	$f_x = f_y = 600$ px, $c_x = 320$ px, $c_y = 240$ px, $H = 0.50$ m
Vision loop rate	30 Hz
Decision loop rate	10 Hz
Arm control loop rate	100 Hz
Spring-stretcher loop rate	500 Hz
Dynamixel baud rate	1,000,000 baud
Joint cruise speed	0.45 rad/s
Minimum and maximum segment duration	2.0 s to 9.0 s
Search stability time	1.0 s
Dart motion hold after disturbance	5.0 s
Relocation observation window	4.0 s
Spring-stretcher target distance	35.0 rad motor output angle
Spring-stretcher maximum current	20 A
Named arm waypoints	18 positions
Configured dart slots	4 slots

Table 6: System failure modes and mitigations.

Failure Mode	Effect	Mitigation
ONNX model missing or fails to load	Vision detections are invalid; autonomous pickup cannot start.	Vision module logs the issue and runs in stub mode so camera startup can still be debugged.
Dart moves during pickup	Arm may miss or push the dart.	Disturb mode pauses, watches for relocation, and restarts from the relocated slot.
Arm serial device unavailable	Full pickup cannot run.	Environment script checks `/dev/ttyUSB0`; main program can run camera-only diagnostics.
Trajectory waypoint miscalibrated	Arm collides, misses, or loads poorly.	`record_positions` and `test_named_positions` allow staged waypoint validation.
Spring-stretcher calibration failure	Launcher may stretch from an unsafe origin.	Calibration has jam detection and timeout; stretch commands are ignored until calibrated.
Excessive spring preload at trigger latch	Static friction increases and the MG995 trigger servo cannot reliably release the mechanism.	Reduce latch preload/friction, improve latch geometry, increase trigger torque margin, or add release-position sensing.
Operator stop request	Abrupt exit could leave the arm in an unsafe pose.	Pressing `q` requests return to home before disconnect; the physical E-stop remains required for emergencies.

4 Cost and Schedule

4.1 Cost

The prototype hardware budget was kept below 1500 RMB by reusing lab assets such as the camera and compute unit. Table 7 lists the prototype bill of materials. The ECE 445 final-report guide also asks teams to estimate labor with

$$\text{labor cost} = \text{hourly salary} \times \text{actual hours} \times 2.5. \tag{8}$$

Using an estimated engineering rate of \$30/h and an estimated 80 h per team member, the four-person labor estimate is

$$30\ \$/\text{h} \times 320\ \text{h} \times 2.5 = \$24{,}000. \tag{9}$$

This labor value is not a cash expenditure by the team, but it provides the commercial-style cost estimate requested by the final-report guideline [10].

Table 7: Prototype bill of materials.

Item	**Estimated Cost (RMB)**
STM32 controller board and CAN transceiver modules	120
Arm/launcher motors and basic drivers, used or student discount	380
24 V battery pack and power protection components	220
DC-DC converters and interface power distribution	80
Mechanical structure, gripper parts, bearings, and fasteners	220
3D-printed safety shroud and launcher fixtures	110
Sensors, switches, and emergency stop hardware	90
Cables, connectors, and wiring consumables	70
Contingency	180
Total	**1470**

4.2 Schedule

The original 12-week plan divided system architecture, CAD, perception, control, integration, and validation across the four team members. The final codebase reached the integration phase: perception, arm control, runtime configuration, disturbance recovery, spring-stretcher control, and staged tests are all present. The final 100-cycle endurance run was completed as a test activity, but only 92 cycles completed successfully because minor software crashes interrupted the remaining cycles.

Table 8: Planned schedule and final status.

Weeks	Planned Work	Final Status
1–2	Define system architecture, subsystem interfaces, and kinematic constraints.	Completed in proposal and design documents.
3–4	Finalize CAD, fabricate gripper and launcher prototypes, and bench-test motors.	Hardware structure and named positions are represented in the final software.
5–6	Implement vision pipeline and bootstrap controller environment.	Completed as ONNX Runtime vision module and YOLO training utilities.
7–8	Integrate trajectory planning, low-level motor control, and coordinate alignment.	Completed as named-waypoint arm control and configurable slot trajectories.
9–10	Tune PID controllers and integrate autonomous pickup-loading logic.	Completed as disturbance-aware decision logic, arm tuning constants, and spring-stretcher PID control.
11	Conduct 100-cycle stress test and inspect wear/failures.	A 100-cycle run was performed; 92/100 cycles completed, with minor software crashes and no motor communication or unsafe hardware failure. Launch failures were mainly caused by high spring preload increasing latch friction beyond the MG995 trigger margin.

Weeks	Planned Work	Final Status
12	Final dynamic tuning, demonstration preparation, final report, and presentation.	Completed as final report and demonstration build.

5 Ethics, Safety, and Broader Impacts

The system manipulates and launches projectiles, so safety and ethical limits are central to the design. The intended use is limited to educational and competitive robotics. The perception system is trained and configured for inanimate projectiles, not people. The final demo constrains launch direction, projectile energy, and operator location so bystanders are not placed in the projectile path.

The IEEE Code of Ethics requires engineers to hold public safety, health, and welfare paramount [11], and the IEEE ethically aligned design guidance emphasizes human well-being for autonomous systems [12]. This requirement influenced both hardware and software choices. The launcher should remain enclosed by a shroud, high-current wiring should be strain-relieved, and a physical emergency stop should interrupt actuator power independently of software. Software shutdown through the `q` key or normal signal handling is useful for controlled stopping, but it is not a replacement for a hardwired emergency stop.

The safety plan follows the intent of machinery risk-reduction and emergency-stop standards [13, 14]. The battery and power subsystem should also follow portable lithium battery safety practices [15]. These references do not make the prototype certified equipment; rather, they define the safety principles used to organize the demonstration setup.

The broader societal impact of this project is primarily educational. It demonstrates perception-guided manipulation, real-time motor control, and staged safety verification on a robotics platform. The same engineering principles could be applied to benign automated loading or sorting tasks. Because the prototype is associated with a projectile mechanism, the team must avoid framing the system as autonomous targeting and must preserve strict operational limits around what objects are detected and where the launcher can fire.

6 Conclusion

Meta-Dart evolved from a proposed autonomous ammunition handling concept into an integrated software and hardware-control architecture. The final repository implements camera capture, YOLO/ONNX perception, slot selection, disturbance-aware decision making, Dynamixel arm motion, gripper loading, spring-stretcher launcher control, runtime configuration, and staged test utilities. The most important engineering contribution is the shift from fixed-coordinate replay to a configurable, vision-triggered pickup sequence that can pause and recover when a dart moves.

The final-demo results show that the project met the retrieval-and-loading success requirement but did not meet the launch-reliability or speed targets. The system loaded 18 of 20 darts successfully, corresponding to 90.0%. It discharged 15 of 20 loaded darts successfully, corresponding to 75.0%, below the 80% launch-reliability target. The original 5 s cycle-time target was also not met after safe-speed tuning; stopwatch and video timing showed an integrated cycle-time average of 34.4 s, with observed cycles ranging from 30 s to 46 s. The dominant launch failure was mechanical: excessive spring force increased latch friction, so the MG995 trigger servo could not reliably release the mechanism. The team also performed a 100-cycle endurance run. In that run, 92 of 100 cycles completed, no motor communication failure occurred, and no unsafe hardware failure occurred, but minor software crashes interrupted the remaining cycles. The strict 100 consecutive-cycle endurance requirement was therefore not fully met. The next design iteration should reduce cycle time by shortening safe transfer paths, improving waypoint calibration, tuning the arm speed after more collision-clearance tests, and redesigning the latch to reduce preload and friction. A future version should also log all verification data directly from the runtime so success rate, latency, voltage stability, and endurance can be reported from stored experimental records rather than from manual run sheets.

References

[1] J. Qin and K. Xu, "Design and implementation of automatic assisted aiming system for RoboMaster EP based on YOLOv5," *arXiv preprint arXiv:2312.05055*, 2023.

[2] Microsoft, "ONNX Runtime for inferencing," [Online]. Available: https://onnxruntime.ai/inference, 2026, accessed: May 16, 2026.

[3] Ultralytics, "Oriented Bounding Boxes object detection," [Online]. Available: https://docs.ultralytics.com/tasks/obb/, 2026, accessed: May 16, 2026.

[4] Z. Lin, T. Wang, Q. Gao, and Y. Liu, "Design of robot platform based on CAN bus," in *2011 International Conference on Electrical and Control Engineering*. IEEE, 2011, pp. 645–648.

[5] The Linux Kernel Documentation, "SocketCAN: Controller Area Network," [Online]. Available: https://docs.kernel.org/networking/can.html, 2026, accessed: May 16, 2026.

[6] ROBOTIS, "DYNAMIXEL SDK overview," [Online]. Available: https://emanual.robotis.com/docs/en/software/dynamixel/dynamixel_sdk/overview/, 2026, accessed: May 16, 2026.

[7] G. Bradski, "The OpenCV library," *Dr. Dobb's Journal of Software Tools*, 2000.

[8] Microsoft, "ONNX Runtime core C and C++ APIs," [Online]. Available: https://onnxruntime.ai/docs/api/c/c_cpp_api.html, 2026, accessed: May 16, 2026.

[9] ROBOTIS, "XM430-W350-T/R e-Manual," [Online]. Available: https://emanual.robotis.com/docs/en/dxl/x/xm430-w350/, 2026, accessed: May 16, 2026.

[10] ECE 445 Course Staff, "ECE 445 Final Report Guidelines," 2026, course handout.

[11] IEEE, "IEEE Code of Ethics," [Online]. Available: https://www.ieee.org/about/corporate/governance/p7-8.html, 2020, accessed: May 15, 2026.

[12] IEEE Global Initiative on Ethics of Autonomous and Intelligent Systems, *Ethically Aligned Design: A Vision for Prioritizing Human Well-being with Autonomous and Intelligent Systems*. IEEE, 2019.

[13] "Safety of machinery—general principles for design—risk assessment and risk reduction," ISO 12100, 2010, International Organization for Standardization.

[14] "Safety of machinery—emergency stop function—principles for design," ISO 13850, 2015, International Organization for Standardization.

[15] "Secondary cells and batteries containing alkaline or other non-acid electrolytes—safety requirements for portable sealed secondary cells, and for batteries made from them, for use in portable applications—part 2: Lithium systems," IEC 62133-2, 2017, International Electrotechnical Commission.

Appendix A Detailed Verification Matrix

Table 9 records the requirement-by-requirement verification matrix. The main report summarizes the high-level results; this appendix keeps the detailed procedures and evidence available without interrupting the main narrative.

Table 9: High-level requirement verification matrix.

Requirement	Verification Procedure	Current Evidence	Result
Autonomous retrieval rate at least 80%, successful cycle under 5 s.	Randomly place darts in configured slots, run `loader` in `disturb` mode, record detection, grasp, loading, and cycle time for at least 20 trials.	Vision, slot selection, guarded pickup, gripper control, and loading trajectory are implemented. Four slot trajectories and 18 named waypoints are configured.	18/20 successful loads, 90.0%. Success-rate target met. Mean integrated cycle time was 34.4 s, so the 5 s speed target was not met.
Launch execution reliability at least 80% after successful loading.	After each loaded dart, trigger the spring stretcher and record whether the dart discharges in the intended forward direction without jam. Use at least 20 loaded trials.	Spring-stretcher calibration, stretch command, hold, retract logic, and MG995 trigger release are implemented. Hardware testing showed that excessive spring preload can increase latch friction.	15/20 successful mechanical discharges, 75.0%. Requirement not met. The dominant failed-trial mechanism was high spring force causing latch friction that the MG995 trigger servo could not reliably overcome.

Requirement	Verification Procedure	Current Evidence	Result
100 consecutive reload-and-fire cycles without critical failure.	Run repeated integrated cycles using the final configuration. Record crashes, communication failures, mechanical jams, and recovery actions.	The runtime includes staged startup, camera-only fallback, `q` return-home shutdown, and environment checks.	92/100 cycles completed. Minor software crashes interrupted 8 cycles. No motor communication failure or unsafe hardware failure occurred. Strict 100 consecutive-cycle target was not met.
Vision processing latency no greater than 50 ms per frame.	Enable vision timing debug mode, run `test_vision_camera`, and record preprocessing, inference, visualization, and total time.	Vision module contains timing instrumentation and a 30 Hz loop budget.	Camera-only verification confirmed real-time display and detection; final average and worst-case latency were not logged.
Arm base yaw 120 degree step settles within 1 s.	Command a 120 degree yaw step and log encoder feedback until position remains inside tolerance.	Arm module publishes state and has waypoint/trajectory tests, but final safe durations are longer than 1 s.	Not separately claimed as met. The final implementation prioritized smooth safe trajectories over the original 1 s yaw step.
Power rails remain within tolerance.	Scope the 24 V motor rail and 5 V logic rail during simultaneous arm motion and launcher stretch for 10 min.	Software assumes separated actuator and logic domains; no rail log was found.	Qualitatively stable during demo; quantitative rail log should be added if available.

Requirement	Verification Procedure	Current Evidence	Result
E-stop actuator power cut in no more than 100 ms.	Measure delay from E-stop press to actuator current drop over repeated trials.	The design requires a hardware E-stop independent of software. The repository supports software shutdown but not hardware E-stop measurement.	Hardware E-stop present for demo safety; timing was not logged in software.