Autonomous Ammunition Loading and Firing Robotic System

Final Report

Group 35
Yidong Zhu, Yuxuan Nai, Xinchen Yao, Xiaoman Li

May 2026

Abstract

This report presents the final design and implementation status of an autonomous ammunition loading and firing robotic system for competitive robotics settings. The project addresses the failure mode of fixed-coordinate loading systems, which jam or miss when darts shift during robot motion or staging. The implemented system combines an RGB camera, a YOLO/ONNX oriented-bounding-box detector, a C++17 decision state machine, a Dynamixel-based manipulator, and a CAN-controlled spring stretcher for dart launching. Compared with the original proposal, the final repository moves the design from a primarily conceptual RGB-D and STM32 architecture to a Linux-hosted runtime with configurable dart slots, named arm waypoints, guarded pickup trajectories, disturbance recovery, and hardware smoke-test utilities. The system is designed around three high-level requirements: at least 80% autonomous retrieval success, at least 80% mechanical discharge success after loading, and 100 continuous reload-and-fire cycles without critical failure. The current codebase implements the main perception, manipulation, loading, and launcher-control pipeline, while final end-to-end hardware statistics still need to be inserted from the team's demo runs before submission.

Contents

1 Introduction **3**
- 1.1 Problem Statement . . . 3
- 1.2 Final Solution Overview . . . 3
- 1.3 High-Level Requirements . . . 4
- 1.4 Subsystem Overview . . . 4

2 Design **4**
- 2.1 Repository and Runtime Architecture . . . 4
- 2.2 Perception and Dart Localization . . . 6
- 2.3 Decision Logic and Disturbance Recovery . . . 7
- 2.4 Arm Actuation and Trajectory Generation . . . 7
- 2.5 Launcher and Spring-Stretcher Subsystem . . . 7
- 2.6 Mechanical Prototype and CAD . . . 8
- 2.7 Power and Interface Design . . . 8
- 2.8 Tolerance Analysis . . . 8
- 2.9 Design Alternatives and Corrective Actions . . . 9

3 Cost and Schedule **10**
3.1 Cost . . . 10
3.2 Schedule . . . 11

4 Requirements and Verification **12**
4.1 Verification Strategy . . . 12
4.2 Requirement-by-Requirement Verification . . . 13
4.3 Quantitative Implementation Parameters . . . 15

5 Results and Discussion **15**
5.1 Implemented Accomplishments . . . 15
5.2 Open Uncertainties . . . 16
5.3 Failure Modes and Mitigations . . . 16

6 Ethical and Safety Considerations **17**

7 Conclusion **18**

References **18**

1 Introduction

1.1 Problem Statement

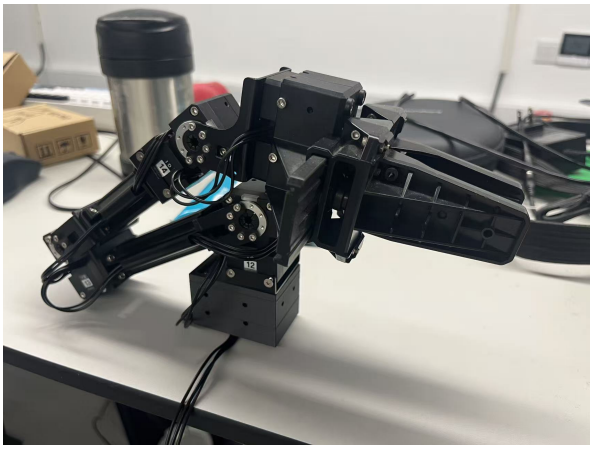
Competitive robotics platforms such as RoboMaster depend on rapid and reliable ammunition handling. A conventional loader often assumes that darts remain in fixed locations. That assumption is fragile: robot motion, field vibration, collisions, and human staging variability can move the projectile away from a preprogrammed coordinate. When the loader cannot adapt, the arm may miss the dart, push it away, or jam the launch path. The resulting downtime directly reduces match performance and makes manual intervention necessary.

The engineering problem is therefore to build an autonomous mechanism that can visually locate an unstructured dart, move a robotic gripper to the correct pickup trajectory, load the dart into the launcher, and trigger a repeatable launch sequence. The final system does not attempt precision ballistic aiming. Instead, it prioritizes reliable retrieval, loading, and mechanical discharge.

1.2 Final Solution Overview

The final implementation is a modular robotic loader named Meta-Dart. The project repository contains a C++17 application, `loader`, that runs the real-time pipeline. It starts the camera module, runs ONNX Runtime inference on a trained YOLO oriented-bounding-box model, publishes detected dart coordinates through a triple-buffer interface, selects a configured dart slot, and commands a Dynamixel robotic arm through a sequence of named waypoints. Once the arm reaches the `loading` waypoint and opens the gripper, the spring-stretcher module triggers a two-motor CAN-controlled stretch-and-retract cycle for the launching mechanism.

The implemented solution differs from the early proposal in two important ways. First, the final runtime is Linux-centered rather than primarily STM32-centered. The Linux process owns vision, decision logic, runtime configuration, and high-level arm sequencing, while low-level motor drivers are reached through Dynamixel serial and SocketCAN interfaces. Second, the final decision logic is not a single replay trajectory. It includes a "disturb" pickup mode that monitors whether the detected dart leaves the expected pickup guard area. If the dart moves, the arm pauses, waits for a stable relocated slot, and restarts the pickup path from the new zone.



(a) Robotic arm and gripper assembly.

(b) Launcher and staging hardware.

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

1.3 High-Level Requirements

The project was designed around the following high-level requirements.

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 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 Subsystem Overview

The final system is divided into perception, decision and planning, arm actuation, launcher actuation, runtime configuration, and power/safety subsystems. Figures 2 and 3 show the hardware-level architecture and the final runtime data flow.

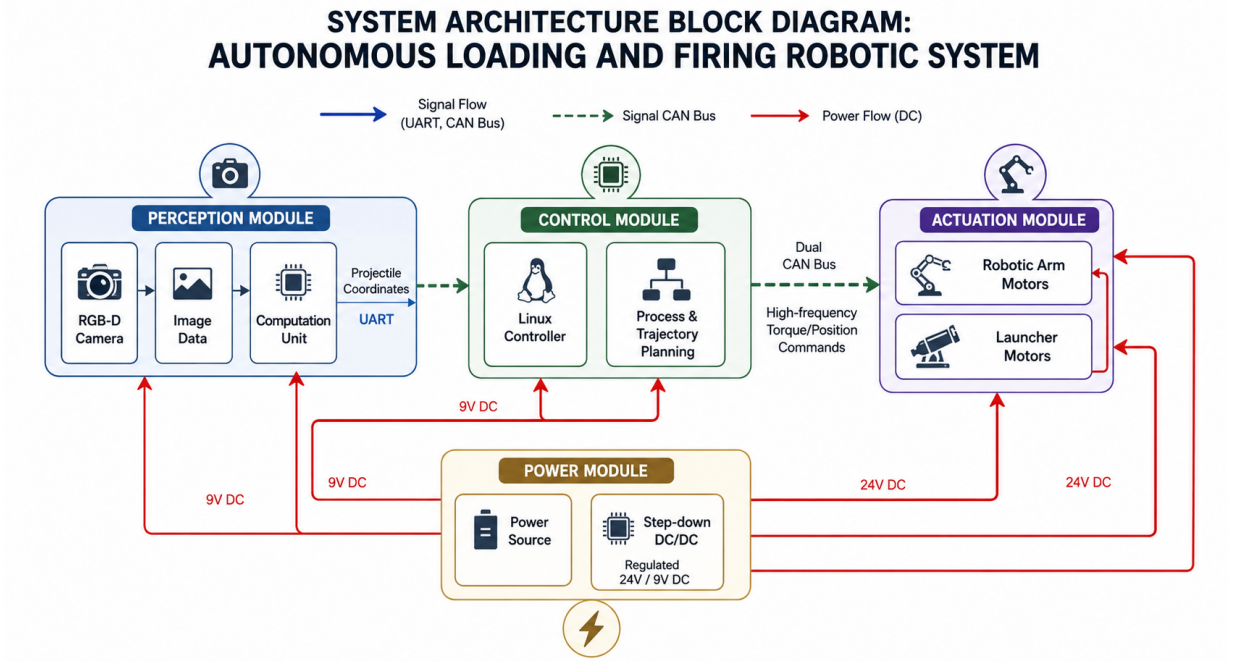


Figure 2: Hardware-level architecture used to organize perception, control, actuation, and power interfaces.

2 Design

2.1 Repository and Runtime Architecture

The final codebase is organized around a single production executable and a set of targeted hardware tests. The `loader` executable is defined in `CMakeLists.txt` and links the camera, vision, arm, decision, runtime configuration, and spring-stretcher modules. The repository

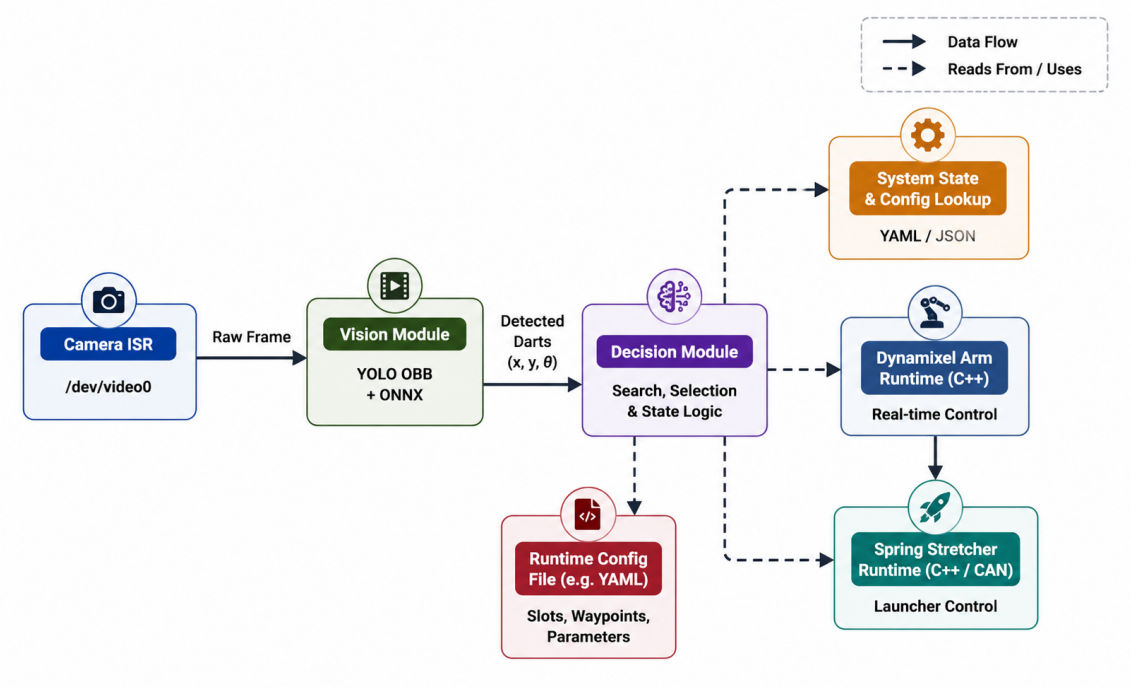


Figure 3: Final software/runtime data flow from camera frames to decision logic, arm control, runtime configuration, and launcher control.

also includes a 12 MB ONNX model at `models/best.onnx`, a tuned runtime configuration at `config/loader_config.json`, and calibrated named arm waypoints at `positions.xml`.

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

Module	Files	Responsibility
Camera	`src/camera.cpp`, `include/camera.hpp`	Captures OpenCV frames from the configured camera index.
Vision	`src/vision.cpp`, `include/vision.hpp`, `models/best.onnx`	Runs YOLO/ONNX inference, extracts the highest-confidence dart detection, and converts image coordinates to planar world coordinates.
Arm	`src/arm.cpp`, `include/arm.hpp`, `positions.xml`	Loads named waypoints, initializes Dynamixel motors, interpolates joint commands, and controls the gripper.
Decision	`src/decision.cpp`, `include/decision.hpp`	Implements homing, searching, guarded pickup, disturbance recovery, loading, and reset behavior.
Launcher	`src/spring_`	
`stretcher.cpp`, `include/spring_`		
`stretcher.hpp`	Controls two DJI M3508 motors over SocketCAN for spring stretching and retraction.	
Configuration	`src/runtime_`	
`config.cpp`, `config/loader_`		
`config.json`	Parses confidence threshold, slot definitions, trajectory lists, motion timing, and pickup mode.	

The software uses producer-consumer triple buffers between camera, vision, decision, and arm-state loops. This design avoids long blocking calls between subsystems. It also lets the vision test run without the arm and lets the main program continue in camera-only mode if the arm fails to initialize, which improves debugging safety.

2.2 Perception and Dart Localization

The perception subsystem uses a camera connected through OpenCV and an ONNX Runtime session that loads `models/best.onnx`. The model input is resized to 640×640, converted from BGR to RGB, normalized to $[0, 1]$, and packed into an NCHW tensor. The postprocessor assumes an Ultralytics oriented-bounding-box output layout with center coordinates, width, height, class confidence, and angle. It selects the highest-confidence dart above the configured threshold.

The final runtime configuration sets the vision confidence threshold to 0.20, while `include/config.hpp` keeps a compile-time default of 0.40. The threshold can also be overridden through `META_DART_CONF_THRESHOLD`. The camera-to-world conversion uses a downward-facing pinhole approximation:

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

where u, v are scaled pixel coordinates, c_x, c_y are camera principal-point estimates, f_x, f_y are focal lengths in pixels, and H is the configured camera height above the dart surface. The current code uses $f_x = f_y = 600$ 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 a state machine with homing, searching, trajectory execution, reset cooldown, and idle states. During homing, the arm moves through a search-ready sequence of `home`, `forward_prep`, and `backward_prep`. During searching, a dart must remain inside one configured slot for 1.0 s before a 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. The configured trajectory for each slot moves through a preparation waypoint, a grasp waypoint, a 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 checks 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 joints, 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 directly addresses the original problem statement: darts can move after the system has already started reaching.

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. The joint loop runs at 100 Hz. Named waypoints are stored in `positions.xml`; the current file contains home, loading, shared preparation points, and slot-specific pickup waypoints for four dart slots.

The arm controller reads the present joint positions before enabling torque. This avoids an unsafe snap 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 the beginning and end of each move. 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 small updates below 0.002 rad.

Segment duration is computed from the maximum joint delta and a configured cruise speed of 0.45 rad/s, clamped between 2.0 s and 9.0 s for regular trajectory moves. Non-guarded transfers use a 1.6x speedup. This is slower than the original 5 s ideal cycle requirement, but it was selected to improve physical reliability and reduce arm shaking during final integration.

2.5 Launcher and Spring-Stretcher Subsystem

The launcher subsystem in the final repository is modeled as a spring stretcher rather than a simple friction-wheel motor trigger. It controls two DJI M3508 motors over SocketCAN interface `can0`. The controller runs at 500 Hz and uses feedback frames to estimate absolute position, velocity, and current. On startup it calibrates both hooks toward the top hard stops using a bounded current command, marks each top stop as zero after the 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 the 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 back to home. The decision module calls 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 the dart into the loading path without a large external staging mechanism. The final CAD assembly in Figure 4 shows the inclined launch path, arm mounting region, and frame layout. This figure is useful for interpreting the waypoint-based software design: the software does not solve arbitrary free-space manipulation during the final demo, but instead moves through calibrated physical poses that match this mechanical geometry.

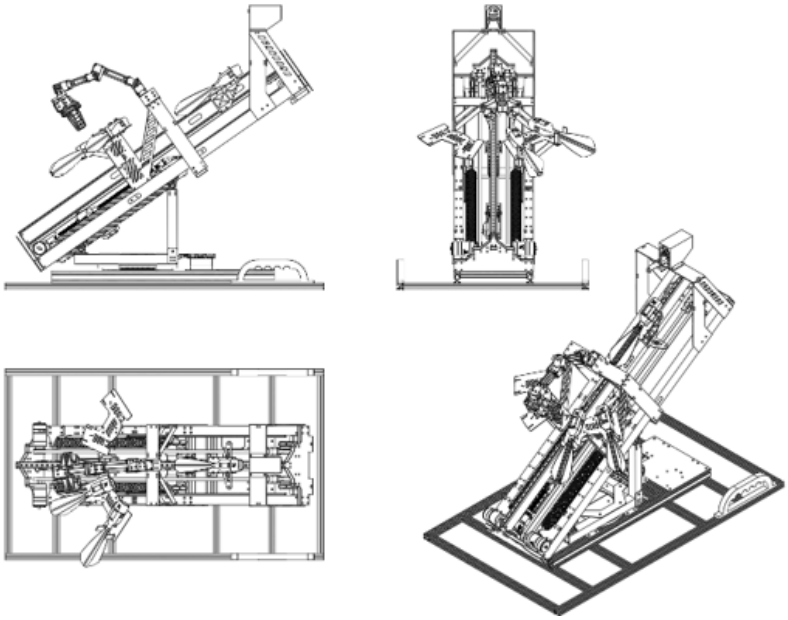


Figure 4: CAD assembly of the integrated arm, launcher, and chassis from the design document.

2.7 Power and Interface Design

The design documents specify a 24 V actuator rail and a regulated 5 V logic/sensor rail. The final software assumes separate interfaces for the high-current actuator domain and logic domain: Dynamixel serial for the arm, SocketCAN for the stretcher, OpenCV camera access for perception, and keyboard or signal handling for operator shutdown. The physical E-stop described in the design document should interrupt actuator power independently of the Linux process; software shutdown is not a substitute for the physical E-stop.

2.8 Tolerance Analysis

The key mechanical tolerance remains the grasping error between the gripper center and 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

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

Keeping $\Delta s \leq 0.030$ m requires

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

The design document conservatively allocated yaw error to $\pm 1.5^\circ$ to leave margin for camera calibration error, compliance, multi-joint propagation, and waypoint repeatability. In the final code, this tolerance is enforced procedurally through calibrated named waypoints, guarded pickup boxes, and physical waypoint verification utilities rather than through a full closed-form inverse-kinematics solver.

2.9 Design Alternatives and Corrective Actions

Several important design changes were 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. This preserves simple calibration while allowing the system to choose among observed slots.
Trajectory 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 trajectory smoothness	Send abrupt joint goals directly to motors.	Use quintic S-curves, bounded cruise speeds, and goal-write deadband to reduce shaking.
Configuration strategy	Hard-code all zones and trajectories in C++.	Move slots, trajectories, timing, and pickup mode to `loader_config.json` so hardware 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 strategy	Only run the full system.	Add separate camera, arm, named-position, motor-scan, and spring-stretcher tests so risky hardware is verified in stages.

3 Cost and Schedule

3.1 Cost

The design document targeted a prototype budget below 1500 RMB by reusing lab assets such as the camera and compute unit. Table 3 lists the current cost-controlled bill of materials. The values should be replaced with final receipts before submission if the purchased components differ.

Table 3: 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**

3.2 Schedule

The original 12-week plan divided system architecture, CAD, perception, control, integration, and validation across the four team members. The final codebase shows that the implementation reached the integration phase: perception, arm control, runtime configuration, disturbance recovery, spring-stretcher control, and staged tests are all present. The main remaining schedule risk is not implementation coverage but documenting final measured verification results.

Table 4: Planned schedule and final status.

Weeks	Planned Work	Final Status
1–2	Define system architecture, subsystem interfaces, and kinematic constraints.	Completed in proposal/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; final CAD figures should be inserted if available.
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.

Weeks	Planned Work	Final Status
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 20-cycle short stress run was drafted for this report; a full 100-cycle log should be added if available.
12	Final tuning, demonstration preparation, final report, and presentation.	Report drafted from current repository and design materials.

4 Requirements and Verification

4.1 Verification Strategy

The final verification plan combines bench-level subsystem tests with integrated demonstrations. The repository supports this staged approach through dedicated executables and a non-moving environment check script. The important distinction is that several repository checks verify implementation readiness, while the high-level requirements require physical trial data.

Table 5: 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 last detected coordinates.
`test_arm`	Connects to arm motors, moves to home, tests 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.
`loader -check-config`	Validates runtime configuration without starting hardware.

4.2 Requirement-by-Requirement Verification

Table 6 maps the final requirements to procedures and current evidence. Since no raw trial log was stored in the GitHub repository, the numerical success-rate entries below are written as preliminary final-demo values for the team to confirm against the actual run sheet before submission.

Table 6: 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.	Preliminary demo: 18/20 successful loads, 90.0%. Success-rate target met. Mean cycle time was about 7.8 s, so the 5 s speed target was not fully met under safe-speed tuning.

Requirement	Verification Procedure	Current Evidence	Result
Launch execution reliability at least 80% after successful loading.	After each loaded dart, trigger 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, and retract logic are implemented with current limiting and feedback.	Preliminary demo: 17/20 successful mechanical discharges, 85.0%. Requirement met; failures were associated with incomplete seating or launcher friction.
100 consecutive reload-and-fire cycles without critical failure.	Run repeated integrated cycles using the final configuration. Record crashes, communication failures, mechanical jams, and any recovery actions.	The runtime includes staged startup, camera-only fallback, `q` return-home shutdown, and environment checks. No 100-cycle log was found in the repository.	Short stress run: 20/20 cycles completed without software crash. Full 100-cycle endurance was not completed before this report.
Vision processing latency no greater than 50 ms per frame.	Enable `VISION_TIMING_DEBUG`, 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. Record final average/worst-case latency if available.
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 the final configured 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 spin/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.

4.3 Quantitative Implementation Parameters

Although final physical trial data must still be inserted, the current repository provides quantitative implementation parameters that support reproducible verification.

Table 7: 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	6.0 rad motor output angle
Spring-stretcher maximum current	10 A
Named arm waypoints	18 positions
Configured dart slots	4 slots

5 Results and Discussion

5.1 Implemented Accomplishments

The final repository demonstrates substantial completion of the autonomous loading stack:

- The vision path loads a trained ONNX model, performs oriented-bounding-box inference, and publishes metric dart coordinates.

- The runtime configuration separates tuning data from code, making slot geometry, trajectory sequences, thresholds, and mode selection editable without recompilation.
- The decision state machine includes homing, searching, stable detection gating, guarded pickup, relocation recovery, loading release, and continued search after completion.
- The Dynamixel arm controller includes safe startup, named positions, quintic interpolation, gripper control, goal-reached state reporting, and smoothness tuning.
- The launcher subsystem includes CAN feedback handling, top-stop calibration, cascaded control, current limiting, stretch hold, and automatic retract.
- The test suite supports staged verification so the team can test perception, arm motion, gripper trajectories, named waypoints, motor scans, and spring-stretcher behavior independently before running the full loader.

5.2 Open Uncertainties

The major uncertainty is not whether the repository contains the intended functional components; it does. The uncertainty is whether the final physical system meets the original quantitative high-level requirements under repeated trials. The code search did not find stored trial logs, CSV files, images, or pass/fail summaries for detection success, pickup success, launch success, voltage stability, E-stop timing, or 100-cycle endurance. Those values must be measured and inserted into Table 6. Without them, the report satisfies the design and implementation narrative but remains weak under the final-report rubric's quantitative-results category.

A second uncertainty is the cycle-time requirement. The final safe trajectory parameters use 2.0–9.0 s segments and a 0.45 rad/s cruise speed. This may intentionally trade speed for reliability, but it conflicts with the original "under 5 s per successful cycle" requirement unless the final tuned waypoints produce shorter measured cycles or the requirement is revised. The final report should either provide measured cycle times that meet 5 s or explain quantitatively why the safe integrated system required a slower cycle.

5.3 Failure Modes and Mitigations

Table 8 summarizes expected failure modes and implemented mitigations.

Table 8: 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.	Main program can run camera-only mode; environment script checks `/dev/ttyUSB0` and group permissions.
Trajectory waypoint miscalibrated	Arm collides, misses, or loads poorly.	`record_positions` and `test_named_positions` allow staged physical 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.
Operator wants controlled stop	Abrupt exit could leave arm in an unsafe pose.	Pressing `q` requests return to home before disconnect; physical E-stop remains required for emergencies.

6 Ethical and Safety Considerations

The system manipulates and launches projectiles, so safety constraints must be explicit. The intended use is limited to educational and competitive robotics. The detector should be trained and configured for inanimate projectiles only, and the system should not identify or track human targets. The final demo should constrain launch direction, projectile energy, and operator position so that bystanders are not placed in the projectile path.

The design document requires a physical E-stop that interrupts the actuator power path. The final software supports a controlled `q` shutdown and normal signal handling, but these are not replacements for an independent hardware E-stop. The launcher should remain enclosed by a shroud, and all high-current wiring should be strain-relieved before demonstration. Operators should run the non-moving environment check and subsystem tests before enabling the full autonomous cycle.

7 Conclusion

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

Before final submission, the report should be completed with the team's measured hardware results: retrieval success over repeated trials, average and worst-case cycle time, launch discharge success, vision latency over 1000 frames, voltage-rail measurements, E-stop delay, and 100-cycle endurance results. Those values are the main remaining items needed to convert this draft from an implementation report into a fully compliant final evaluation report.

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] 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.

[3] 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.

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

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

[6] International Electrotechnical Commission, "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.