RoboJackets 2026 Team Description Paper

Shourik Banerjee, Sanat Dhanyamraju, Aiden Lee, Aalind Tyagi, and Qingyan Lin

Georgia Institute of Technology
https://robojackets.org/

Abstract. This paper describes the improvements implemented by the Georgia Institute of Technology's RoboCup SSL team, RoboJackets, to compete in RoboCup 2026 in Incheon, South Korea. Our main focus this year was once again improving the robot's physical capabilities by upgrading our electrical system with new kicker and motor boards. Mechanical updates this year were limited and focused on planning the next design for our robots. Firmware updates this year focused heavily on developing a setup for easier hardware testing through simulating the software stack. Software development was focused on overhauling our communication system for better decision making and improving rules compliance regarding avoiding obstacles. As always, all of our designs and code are open-sourced. See the RoboCup SSL website for our links, or search for "RoboCup" on our GitHub page.

1 Mechanical

The mechanical updates for this year focused on planning a new design for the robot. The main issue with our robots last year was the poor movement exhibited from our fleet. One of our solutions to this problem is to redesign the wheels, in order to improve controlling the motion of our robot. One of the main features is increasing the number of contact points on the wheel to allow for more precise PID control. In addition to modifying the robot wheels, the team plans to redesign the top layer to allow more space for the electrical boards. More detail on these changes will be published in future Robojackets TDPs as these redesigns are completed.

1.1 Current Specifications

Our mechanical specifications for our robot this year are the same as last year's robots [4], found in the table below.

Table 1. 2026 Robot Specification

Dimension	∅ 180 mm × 146 mm
Total Weight	2.59 kg
Drive Motors	Maxon EC 45 Flat (651610)
Encoders	Maxon Encoder (673029)
Wheel Diameter	∅ 53 mm
Dribbler Motor	Maxon EC 16 (405812)
Dribbler Bar Diameter	15 mm
Dribbler Damping Material	Poron Microcellular Urethane
Kicker Charge	4000 μF @ 250 V
Kick Speed	6 m/s (max speed)
Chip Distance	approx. 3 m

2 Electrical

This year's electrical updates focus primarily on the redesign of the kicker board (v4.0) and the control board (v5.0). The kicker board now uses the LT3757 instead of the LT3751 to increase the charging frequency, and the control board replaces the Teensy 4.1 with a custom design based on the STM32H723VGT6 microcontroller. Table 2 summarizes the resulting changes to the electrical stack.

Table 2. Electrical stack changes from 2025 to 2026

Part	**2024**	**2025**
Radio	nRF24L01+	nRF24L01+
Base Station	RaspPi 4 + nRF24	RaspPi 4 + nRF24
MCU	Teensy 4.1	STM32
Control Board	v4.0	v5.0
Motorboard	v1.0	v1.0
Kicker Board	v3.3	v4.0
Battery	18.5V 90C 1600 mAh	18.5V 90C 1600 mAh

2.1 Kicker Board

The previous kicker board design was originally developed in 2019, which relies on high-voltage aluminum electrolytic capacitors [2]. As those capacitors age, electrolyte degradation will reduce capacitance, increase internal pressure, and pose safety risks from electrolyte leakage. These concerns of reliability and safety motivated a redesign of our kicker board to improve performance and renew electrolytic capacitors. The primary innovations of the new kicker board include a new LT3757-based charging architecture, a redesigned discharge circuit, and a reworked breakbeam interface.

Charge Circuit The kicker board v4.0 introduces a redesigned charging circuit based on an LT3757-controlled flyback architecture to replace the previous LT3751-based charger. The new circuit controls peak current through an external sense resistor and supports higher energy transfer per switching cycle.[3] The LT3757 allows the switching frequency to be externally adjusted, whereas the LT3751 does not. This adjustability enables future optimization for higher switching frequencies, allowing the capacitor to be charged faster while maintaining stable operation.

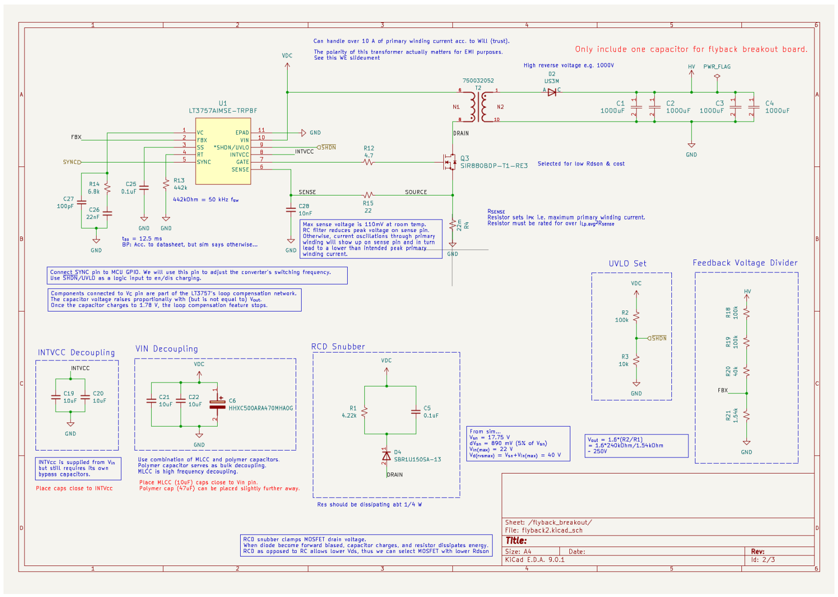


Fig. 1. Schematic of the charge circuit.

Discharge Circuit The discharge stage replaces the IXYS IXGK400N30A3 IGBT and IXYS IX4427N gate driver with the Infineon IKB40N65ES5 IGBT and Infineon 2EDB9259Y dual-channel IGBT gate driver. While the new IGBT provides a lower voltage rating, it has a more compact package and is more power-efficient. The dedicated gate driver supplies a clean, regulated gate-drive rail with stronger and more consistent gate charge and discharge behavior. Together, these changes improve turn-on repeatability and address cases in the previous design where the discharge event occasionally failed to actuate.

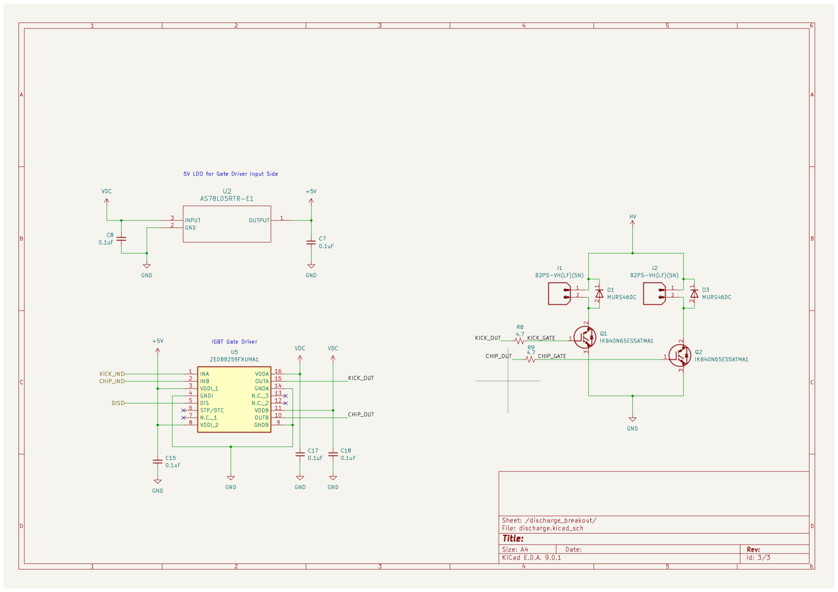


Fig. 2. Schematic of the discharge circuit.

Breakbeam The breakbeam system was redesigned into a single board with wired red laser emitter and receiver. Direct mounting of both the emitter and receiver to the kicker assembly improves mechanical stability, reduces misalignment, and eliminates inter-board connectors, resulting in a more robust and easier-to-debug system.

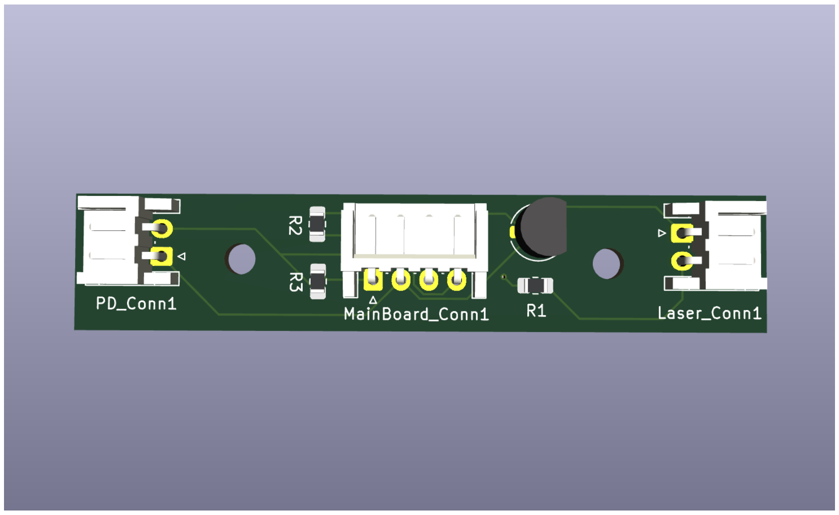


Fig. 3. Topview of breakbeam v2.0.

2.2 Control Board

Control Board v5.0 replaces the Teensy 4.1 with a custom STM32H723VGT6-based design [6], adds a USB-C interface, dedicated hardware reset buttons, and tighter integration with the motor board to improve robustness and manufacturability.

USB-C Interface A key addition in v5.0 is the integration of a USB-C interface. USB-C now serves three roles: it provides regulated 3.3 V and 5 V power, supports firmware programming, and enables direct debugging access. This allows the Control Board to operate independently of the Motorboard during development, eliminating the prior dependency on system power and reducing reliance on JTAG for routine programming tasks. To safely manage multiple power sources, the board incorporates a TPS2117DRLR power multiplexer. This device automatically selects between the motor board and USB-C power, with the motor board supply given priority to reflect the intended production configuration. If Motorboard power is absent, the system seamlessly switches to USB-C, allowing uninterrupted standalone operation during bench testing.

Design Refinement This redesign also addresses practical limitations discovered during field testing of Control Board v4.0. Mechanical tolerances were not adequately considered during the original IMU placement, resulting in component crowding and assembly difficulties. In v5.0, the IMU was repositioned with appropriate clearance to improve manufacturability. In the previous revision, debugging was challenging because of the lack of accessible hardware reset options is also informed the redesign. The new board includes dedicated reset buttons for both the microcontroller and the motor board interface, along with dedicated debugging lines to simplify firmware development, signal probing, and board bring-up. Overall, Control Board v5.0 allows for a robust foundation for future development while maintaining compatibility with the system's established interfaces.

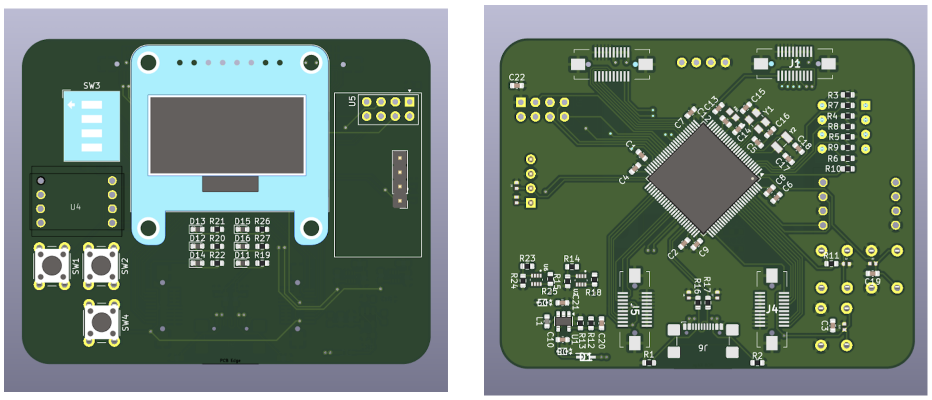


Fig. 4. Top view of control board v5.0 front and back.

3 Firmware

3.1 Handheld Controller

In the last Team Description Paper, a method of testing robot subsystems remotely was presented. This allowed the use of the existing communications stack to prompt the robot to enter into one of several debugging modes, upon which it would perform a series of pre-programmed actions[4]. While this approach was functional, running any sort of test required the full software stack to be set up, which is a time-intensive exercise. Consequently regular unit testing was not performed, as manually checking connections was often faster than setting up the infrastructure to test.

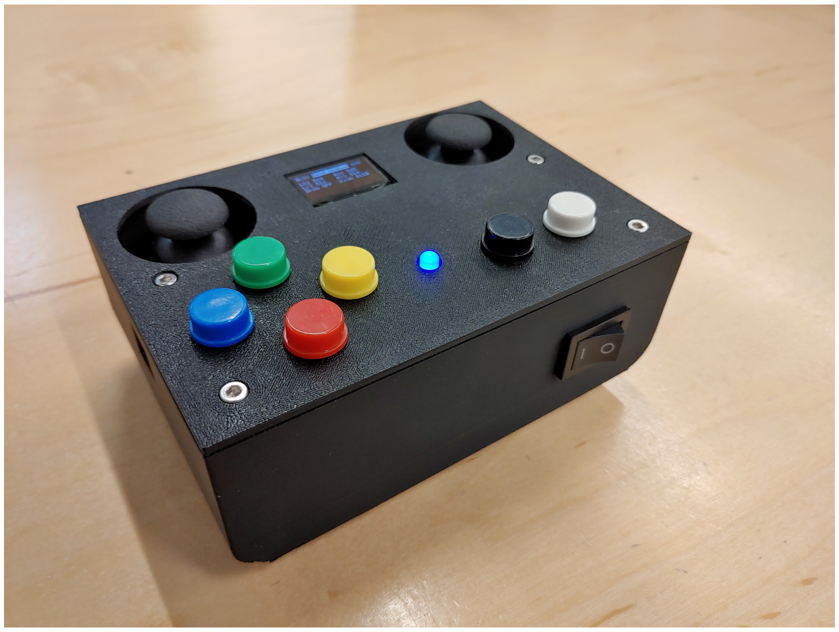


Fig. 5. A handheld controller unit for testing robot functionality.

Hardware Overview The found solution was to create a handheld device that replicated the necessary parts of the software stack to appear as a genuine base station to a robot. The device, henceforth referred to as the "controller", features an OLED screen and a set of generic controls (two joysticks and several push-buttons) not unlike those used for entertainment. The controller communicates with robots using an internal nRF24RL01+ radio module.

Software Internals In order to facilitate easy development of future testing modes, a co-operative operating system was created in order to abstract hardware details by exposing a set of standard APIs through which a test runner can interact with the underlying hardware. Under this system, various tests are packaged into "modules" and registered with the kernel at compile-time, after which they are available to the scheduler. Task switching is performed through voluntary yields by the currently running task. It is possible for a yielding task to request a specific task to scheduled next, a feature that is utilized by the Menu task.

An Example Task: Drive Mode The Drive Mode task allows the operator to use the joysticks and buttons to prompt a robot to move, kick, or chip while monitoring live telemetry from the robot such as battery status, connection integrity, and kicker unit health. This task was used extensively at the Salvador

2025 competition to quickly perform full function tests of robots before connecting the software stack, a process that used to involve flashing every robot's firmware multiple times.

3.2 Shoot-Through Prevention

At the Salvador 2025 competition, we experienced instances of robot internal temperatures rising well beyond expected levels, often to the point where metal components were uncomfortably hot to the touch. After preliminary testing, the new motor drivers were found to have a current draw of up 5A per motor, compared to around 2A with the previous iteration. This was later found to be caused by shoot-through, or when both the high and low-side MOSFETs of a half-bridge are enabled at the same time, creating a direct current path from power to ground (see Figure 6).

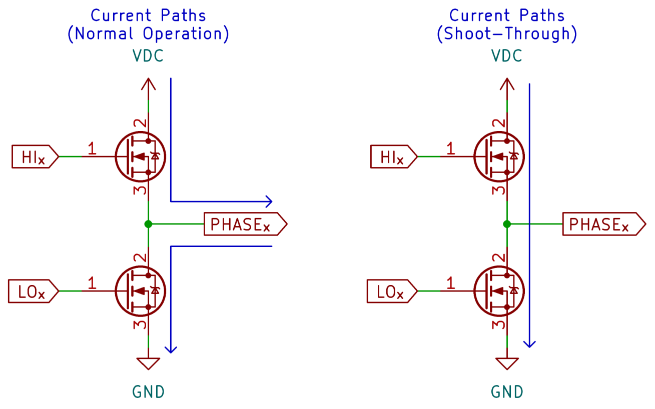


Fig. 6. Illustration of current paths through a half-brige during normal operation and during shoot-through.

A solution to this issue was the insertion of 0.5-1 µs of "dead time" between disabling one MOSFET and enabling the other. While simple in theory due to hardware support from the STSPIN series motor driver we utilized, this proved to be a major challenge due to lack of Rust support for this feature. Initial testing was performed by setting the control register to arbitrary values and observing the resultant behavior while proper Rust support was implemented. It is currently planned to contribute the team's improvements to the open-source driver once verification testing concludes.

Testing with dead-time of 1 µs resulted in a significant drop in current draw from 5A to 2.5A per motor with little loss in torque.

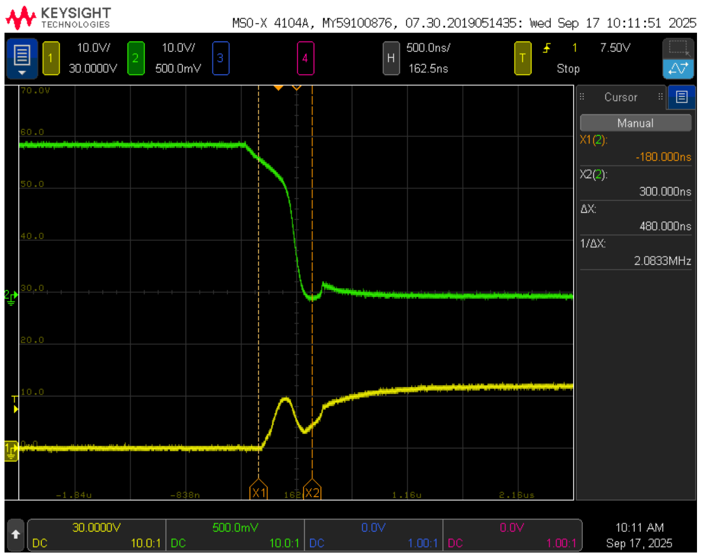


Fig. 7. An instance of an observed shoot-through event. The yellow and green signals are gates of the high and low-side MOSFETs of a half bridge.

4 Software

4.1 Coordinators

The communication system used in Salvador relied on communication between pairs of robots: each position created request/response structs that are tracked with IDs; this communication scheme meant that if a robot wished to perform an action, the requests are sent and accepted based on an individual robot's world view and state. While this is useful for situations where only 2 robots that need to send (such as passing), a few main concerns arise with more complex strategies:

1. **Fragmented decision making:** With the existing architecture, if robots wish to perform a complex strategy with one another, such as robots forming a wall in front of the goal, it is not clear which robot would initiate such a request. Additionally, getting all robots on the same page is a lengthy process: what if one robot wants them in a wall, while another does not?
2. **Race conditions:** Robots may have a different view of the world state at any given point. When decisions are locally made and communicated, this discrepancy can cause problems when decision making.
3. **Separation of concerns:** Logic for multi robot coordination should not exist within a singular robot's position code. Position code should exclusively relate to making the plans of the robot it is controlling.

These issues resulted in the development of a new communication paradigm: coordinators. The main role of the coordinator is to assess the current state of a particular position and control which robots are within a specific coordinator group; there is one coordinator for each position, resulting in an entire suite of coordinators working at once. When a robot wishes to enter a specific position, it sends a request to that specific coordinator along with associated data. The coordinator then uses its own logic to "accept" or "reject" a robot from entering a specific position and performing the corresponding actions. If a robot is permitted to enter a coordinator's group, it will then subscribe to that coordinator's topic to determine the state of the coordinator's group. By subscribing to the coordinator's topic, the robot gains an understanding of its world state, the game state, and the state of other robots in the same coordinator group.

The logic for accepting or declining a robot from joining a specific coordinator group exists in a client for that coordinator. The position will use the client's API to join the group, and receive a callback when the group is joined and subscription results come in.

Coordinator Example: Kicker Picker The Kicker Picker decides what robot should kick in the case of kickoff, penalty, and free kicks. Currently, the closest robot to the ball is selected. A robot sends a request containing a `uint8` specifying the robot ID and a `bool` specifying if the robot wants to join or leave the group. The response is a `bool` which is always true. The topic associated with the Kicker Picker broadcasts the selected kicker using the robot ID.

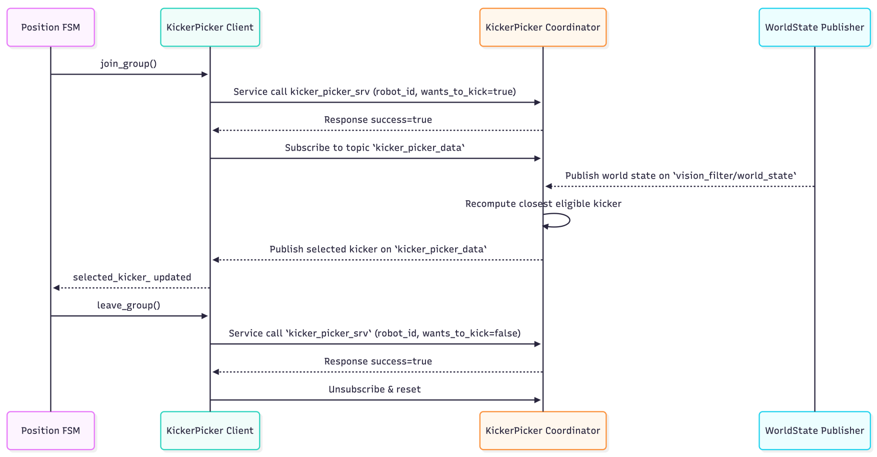

Position FSM
KickerPicker Client
KickerPicker Coordinator
WorldState Publisher
join_group()
Service call kicker_picker_srv (robot_id, wants_to_kick=true)
Response success=true
Subscribe to topic ‘kicker_picker_data‘
Publish world state on ‘vision_filter/world_state‘
Recompute closest eligible kicker
Publish selected kicker on ‘kicker_picker_data‘
selected_kicker_ updated
leave_group()
Service call ‘kicker_picker_srv‘ (robot_id, wants_to_kick=false)
Response success=true
Unsubscribe & reset
Position FSM
KickerPicker Client
KickerPicker Coordinator
WorldState Publisher

Fig. 8. Sequence diagram depicting `KickerPicker` Coordinator flow.

4.2 Obstacle Handling

Obstacle detection plays a key role in our robot's ability to navigate correctly and safely across the field. However, previous versions of obstacle detection led to a wide variety of issues:

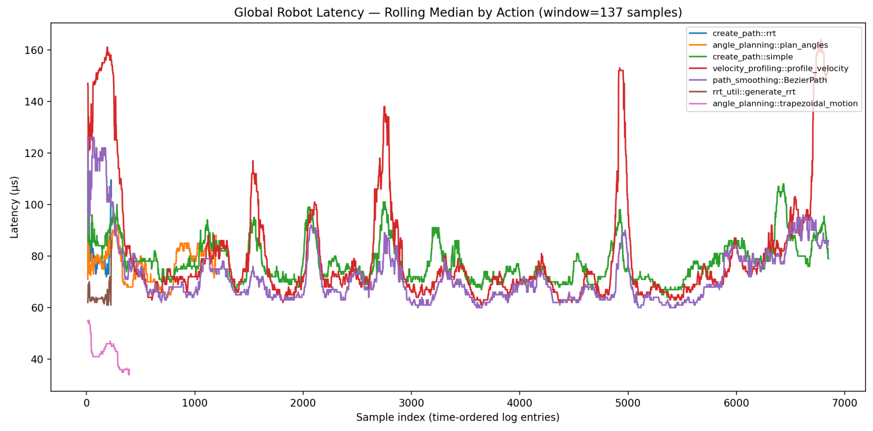
1. **Separation of Obstacles and Constraints:** Previous code did not clearly differentiate between field & static constraints vs. robot-specific constraints, making it difficult to reason about what the planner is avoiding.
2. **Lack of Extensibility:** Obstacles were constructed primitively as raw shapes in many places, so adding new behaviors required adjusting many different files in code.
3. **Poor Avoidance of Moving Obstacles:** Moving obstacles (such as other robots) were primarily modeled as static circles, making it difficult to avoid opponent robots.

To resolve these shortcomings, a new obstacle handling system was written. In code, each `Obstacle` is made up of a core shape and a padding shape: the core shape is the actual physical obstacle footprint, and the padding shape is a safety margin around the core shape. The `ObstacleSet` serves as a contained for `Obstacle` objects, with collision checks and visualization tools using the `ShapeSet` object. All planners have now migrated to using the obstacles from `ObstacleSet` to check for potential collisions and navigate around them. This has greatly improved our code quality and allows the planners to rely on the same interface for obstacle management.

Moving obstacles are treated as static obstacles built using stadium shapes that have variable, direction-biased padding based on the current velocity of the obstacles. Each planning cycle rebuilds this obstacle set using information from the latest world state, effectively making the padding of the obstacle dynamic as the obstacle moves. Planners are used to avoid the padding zones, effectively preventing the robot from moving to a location where another moving obstacle is likely to occupy next. The logic for escaping obstacles was also updated. Once the current obstacle set is built, if the robot is inside the padding of an obstacle, it immediately finds a nearby unblocked point and plans a short path to that point.

4.3 Latency Benchmarking

As our codebase continues to grow with multiple real-time processes, it becomes useful to understand if any one particular process ends up being a bottleneck towards achieving a particular action. In the fall, we developed `rj_benchmarking`, a latency logging pipeline used to public latency samples to a ROS 2 registry node, which then writes the latency data to a CSV for each robot on shutdown.



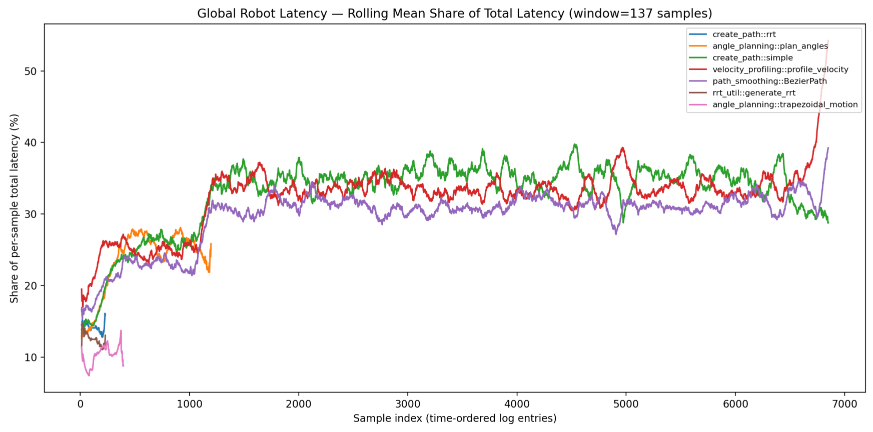


Fig. 9. Measured global robot latency using `rj_benchmarking`. **Top:** rolling median latency per action over time. **Bottom:** rolling mean share of per-sample total latency.

4.4 Switching to Colcon

This season, we fully migrated to using colcon as our build system. This enabled us to build and compile our code in line with what is recommended in the ROS 2 developer guide [5]. Previously, code was built using a top-level makefile that wrapped CMake and Ninja. The switch to colcon has offered the ability to perform targeted builds, the ability to build packages in parallel with improved caching, and a standardized package layout. Furthermore, the use of colcon has made it easier for new members to onboard, as many experienced in ROS 2 workspace management can easily adjust to the package structure now used in our stack. The Docker container developed last season continues to be the primary environment used by team members and was successfully changed to accommodate the switch to colcon.

5 Open Source

RoboJackets continues to open source all aspects of development. Links to software, electrical, and mechanical materials can be found on the RoboCup SSL [1], or by searching for "RoboCup" on our GitHub page.

References

1. Open source contributions. https://ssl.robocup.org/open-source-contributions/
2. Almargo, et al.: Robojackets 2019 team description paper (2019)
3. Analog Devices, Inc.: LT3757 High Voltage Flyback Controller Datasheet. Analog Devices (2023), available at: https://www.analog.com/media/en/technical-documentation/data-sheets/lt3757-3757a.pdf
4. Kota, et al.: Robojackets 2025 team description paper (2025)
5. Open Robotics: Using colcon to build packages, available at: https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries/Colcon-Tutorial.html
6. STMicroelectronics: STM32H723VE Datasheet. STMicroelectronics (2024), available at: https://www.st.com/resource/en/datasheet/stm32h723ve.pdf