TurtleRabbit 2026 SSL Team Description Paper

Darpan Gurung[1], Emma Tsang[1] ⋆, Austin Mckinder[1], Tharunimm Jamal[1], Lisa Graf[2], Adharshni Mohanbabu[1], Duc Minh Nguyen[1], Justin Shirzadian[1], Naomi Luyen[1], Rafael Subtil Schuch[1], Liam Vercueil[1], Srivatsan Kalyanasundaram[2], Tilman Rosenlicht[2], Tristan Trieu[1], Alex Pham[1], Kanaka Sai Jagarlamudi[1], Gizem Intepe[1], and Chng Wei Lau[1]

[1] Cognitive robotics lab, Western Sydney University, Penrith NSW 2751, Australia
[2] Neurorobotics lab, University of Freiburg, Germany
Corresponding author: Chng Wei Lau `c.lau@westernsydney.edu.au`
https://www.turtlerabbit.org

Abstract. This team description paper describes the design and development of the TurtleRabbit 2025–2026 RoboCup SSL team development cycle. We include further improvements on the chip kicker design, PCB design for both the straight and chip kicker to provide control and consistency. As more components are added into the robot, we have redesigned the middle chassis, allowing better spacing, and component placement. Dribblers are also modified for quicker access for replacement, and a camera has been added for enhancing ball detection since last year. Since 2024, we have identified issues with limitation on our Python firmware on our robots, we have migrated into a new C++ firmware. On the team software, to be more adapted to the environment, a new design has been implemented, with the integration of behavior tree decision making and Voronoi-based path planning. New reinforcement learning process has been recorded in this paper as well.

Keywords: RoboCup SSL · Chip Kicker · PCB Design · Camera Ball Detection · C++ Firmware · Behavior Tree · Voronoi Path Planning · Real-time Visualizer · Open-Source Hardware · Reinforcement Learning

1 Introduction

RoboCup Small Size League (SSL) presents a challenging and dynamic environment in which teams must integrate perception, control, communication, and decision-making into a cohesive system capable of operating in real time under strict physical and computational constraints. Success in SSL depends not only on mechanical performance, but also on the robustness and intelligence of the software that coordinates multiple agents in a highly adversarial setting.

For the RoboCup 2026, we adopted a year-long development cycle following the previous year's qualification, enabling a comprehensive set of improvements across both hardware and software. On the hardware side, we developed a dedicated

⋆ Authorship: Team Leads first then alphabetical order, academic lead last.

ball-tracking camera system, evaluated multiple dribbler configurations, and redesigned the robot frame to improve modularity and accommodate additional components. On the software side, the system architecture was restructured to improve isolation and reliability, the primary framework was migrated from Python to C++, a structured behavior tree–based decision-making framework was integrated, and the path-planning module was refined to guarantee obstacle clearance while increasing node density for improved maneuverability.

In parallel with classical robotics approaches, we expanded our research efforts in reinforcement learning for multi-agent coordination. We investigate reinforcement learning methods for cooperative behaviors including passing and ball retention, and explore attention mechanisms and exploration strategies for sparse-reward scenarios. This paper presents the TurtleRabbit 2026 system in detail, covering software architecture, behavior control, motion planning, networking, and reinforcement learning research, while also discussing current limitations and outlining future directions toward more robust, adaptive, and competitive autonomous robot soccer systems.

2 Hardware Design and Improvements

This section presents the hardware developments undertaken since our previous TDP, focusing on three key subsystems: the kicker, the dribbler, and ball detection. These areas were prioritised based on failure modes observed during testing and gameplay, where insufficient kick reliability, poor ball retention, and inconsistent ball sensing were identified as the primary mechanical limitations of the platform. For each subsystem, we describe the specific problems encountered, the design modifications explored, and the current state of implementation.

2.1 Kicker

Our kicker development followed two parallel paths: maintaining a **stable, competition-ready solution** while preparing a next-generation system for future use. Although the full dual-mode kicker (chip and straight) is still under development, significant progress has been made on a high-voltage capacitor charging system as used by several leading SSL teams.

Meanwhile, the existing **straight-kicker** continued to be refined to ensure a reliable competition solution while the new electronics platform is validated.

2.1.1 Straight-Kicker PCB For the current straight-kicker, we retained the simplicity and proven performance of the existing boost converter module, while improving compactness and protection. This resulted in a companion PCB with identical dimensions to those of the original boost circuit, allowing direct mounting via standoffs.

Due to space constraints, the micro-controller is connected using soldered wires and detachable JST connectors, enabling fast replacement and maintenance. The PCB significantly improves electrical safety through several protection features:

- A **fuse** for fault current protection
- An **inrush current limiter** to protect the battery
- A **TVS diode** across the capacitor to clamp voltage to 200 V
- **Bleed resistors** that discharge the capacitor from 200 V to safe levels in under one minute

The solenoid is switched directly by a MOSFET and controlled by a micro-controller through a gate driver, which enables faster switching and sharper peaks for more reliable and energy efficient driving of the MOSFET gate.

We are also extending this board with a digital potentiometer, replacing the analog potentiometer on the boost module. Together with a voltage divider connected to the micro-controller ADC, this enables digital control and monitoring of the charge voltage.

2.1.2 Dual-Solenoid Kicker System In parallel, we are developing a new unified kicker electronics platform supporting both straight and chip kicks from a single module. This system is based on a high-voltage capacitor charging architecture using the LT3751 capacitor charger IC and a flyback converter topology. This configuration allows fine-grained software control over both the target voltage and the charge time for consistent and repeatable kick strength.

The module controls two independent solenoid channels, one for the straight and one for the chip kicker, each driven by a dedicated gate driver. This ensures clean MOSFET switching while providing electrical isolation between the high-voltage domain and the micro-controller. Control is handled by the ESP32-C6 micro-controller, selected for its compact size, improved performance, and compatibility with our existing workflow. It remains programmable using the Arduino IDE, allowing us to reuse most of our existing firmware with minimal modification.

A voltage divider connected to the ESP32-C6 ADC enables real-time capacitor voltage measurement, supporting closed-loop kick power regulation and fault detection (e.g over-voltage or slow charging). Communication between the ESP32-C6 and the robot's onboard computer is handled via USB serial, providing a simple and reliable interface for low-level actuator control. This choice is also driven by hardware constraints imposed by our moteus Pi3bat boards, which occupy all available GPIO pins on the Raspberry Pi.

2.2 Dribbler

In previous iterations, the dribbler used a conical roller made from 95A Shore hardness TPU with a spiral surface pattern and an outer layer of silicone tape. This was driven by a 5200 KV brushless motor connected via two Nylon (PA66) gears with an approximate ratio of 7/15, all mounted in a PLA frame. During testing, the ball was observed to bounce off the roller uncontrollably while the dribbler was active. Analysis indicated three likely causes: insufficient output torque, which prevented the roller from maintaining rotation under impact; excessive roller speed, which reduced contact time with the ball; and low surface friction between the roller and ball.

2.2.1 Gear Train Redesign In the previous configuration, the roller frequently stalled upon ball impact because the available torque was insufficient to counter the ball's force. Successful dribbling only occurred when the ball was slowed or gently fed into the roller. To address this, we added a third gear stage, creating a compound gear train that changed the overall ratio from approximately 1:2 to 1:6. This approximately tripled the output torque at the roller while reducing roller speed, directly targeting both the stalling and ball rejection failures. The motor operates at approximately 10,000 RPM under load, with the new gear train producing a minimum roller speed of approximately 1,000 RPM at the output stage. As part of the revised frame design, the motor holder was also updated to a clip-based mounting system, enabling rapid motor replacement without full dribbler disassembly.

2.2.2 Roller Material Investigation We investigated cast-molded urethane as an alternative to the original TPU roller. Cast molding was selected as the fabrication method because, once a correct mold is established, rollers can be rapidly manufactured with consistent results — an advantage over 3D-printed TPU, which we lacked the facilities to produce reliably. We selected Smooth-On VytaFlex 60, a 60A Shore hardness urethane, for its high friction coefficient and favourable mechanical properties for ball interaction. The 60A hardness was chosen as a balance between sufficient grip and structural durability under repeated ball impact. Silicone-based approaches were also considered but ruled out due to silicone's susceptibility to wear and surface degradation, which would undermine the durability gains of moving away from the original silicone tape wrap.

Early prototypes encountered demolding challenges due to urethane's high adhesion to our 3D-printed molds. While this adhesion is beneficial for ball grip, it creates significant challenges during casting. Preliminary testing with wax as a release agent, chosen for its availability and ease of application, showed improved demolding, though further testing with alternative release agents is ongoing. Additional optimisation of the urethane composition and mold surface texture is also required.

2.2.3 Current Limitations and Future Work Quantitative measurement of output torque and ball retention performance is underway but not yet complete. Additionally, integration of a damping mechanism to reduce ball bounce on initial contact is under investigation.

2.3 Ball Detection and Camera

The early robot used an IR sensor for ball detection, but its unreliability led us to replace it with a small camera mounted above the dribbler.

The new camera holder (see Figure 1) clamps the camera securely between the dribbler's two metal pipes, ensuring stability. Later iterations improved ball visibility by tilting the camera and added clamps and cube walls to prevent

rotation. The final design was iterated over multiple until one fit out physical layout.

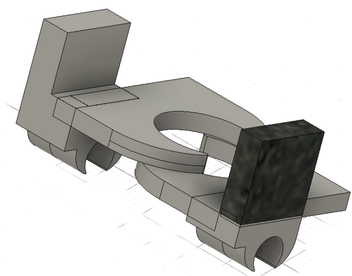


Fig. 1. Image of the holder for the ball detection camera.

3 Software Design and Implementation

Following RoboCup Eindhoven 2024, the TurtleRabbit V0 software [14] exposed several limitations in system reliability and observability. In particular, the team lacked visibility into process execution, failure states, and inter-module communication. Debugging during extended runs relied heavily on terminal output, making it difficult to trace failures across processes. In addition, tightly coupled components reduced system robustness, where failures in one module could propagate across the system.

To address these issues, the 2026 system [15] redesign focuses on **modularity, process isolation, and observability**, allowing the team to better understand, debug, and control system behaviour during both simulation and real-world operation.

3.1 System Architecture

The system was redesigned around process-level decomposition using Python multiprocessing, where each major subsystem operates as an independent process. These subsystems include vision handling, world model updates, decision-making, and command dispatch.

The system follows a structured lifecycle:

- Setup – configuration loading and resource allocation
- Initialization – process startup and inter-process communication setup
- Running – continuous perception, decision, and control loop
- Shutdown – controlled termination with log preservation

This design isolates failures within individual processes, preventing error propagation across the system and improving fault tolerance and debugging

clarity. It also enables clearer reasoning about system behaviour by separating concerns across modules.

Figure 2 illustrates the overall system architecture and data flow between modules.

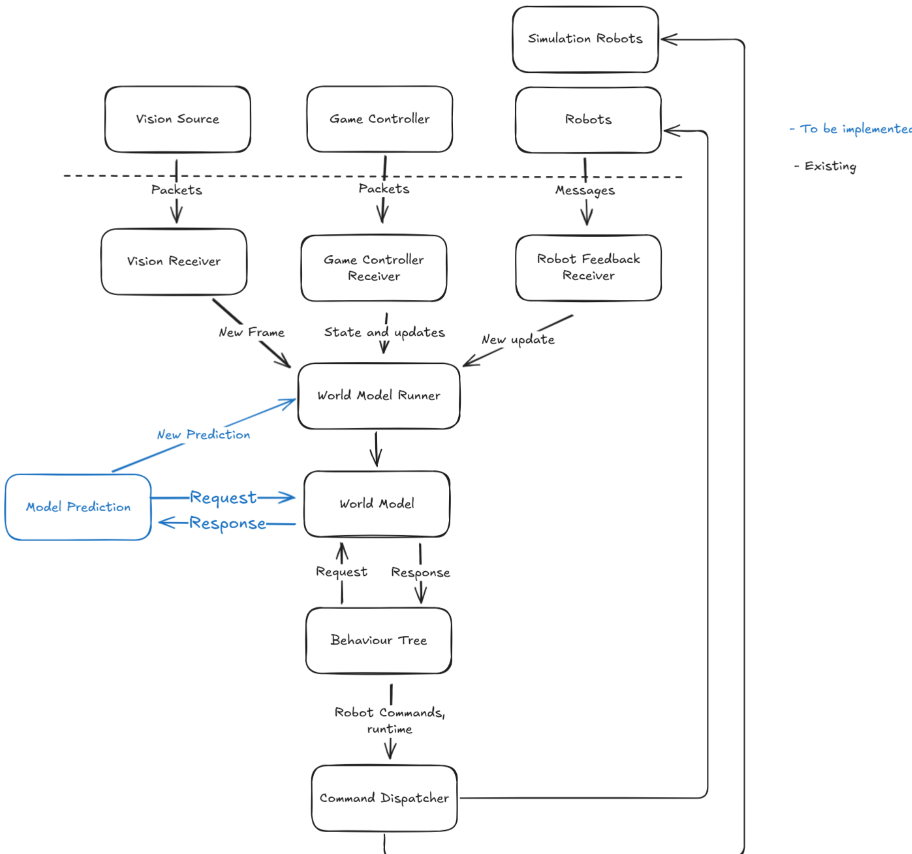


Fig. 2. Software architecture and inter-process data flow for the 2026 system.

3.2 System Performance and Latency Analysis

To evaluate the behaviour of the redesigned system, we profiled the end-to-end processing pipeline across key stages, including vision processing, world model updates, decision-making, and command dispatch.

The system achieves an average end-to-end latency of approximately 8.5 ms from vision input to command dispatch. Computational stages such as world model updates and decision ticks consistently execute within sub-millisecond timeframes, while queue delays contribute the largest portion of overall latency.

Table 1. Latency distribution across system processing stages (measured over 365 cycles).

Stage	**Min (ms)**	**Avg (ms)**	**P95 (ms)**	**Max (ms)**
Vision Assembly	2.04	2.35	2.74	2.98
Vision Queue Wait	1.03	2.20	3.00	4.80
World Model Update	0.62	0.86	1.16	1.87
Decision Tick	0.59	0.88	1.19	2.34
Dispatch Queue Wait	1.56	2.17	2.77	5.91
Total (Avg)	–	**8.46**	–	–

The relatively small gap between average and 95th percentile (P95) latencies indicates stable system performance under continuous operation. The primary internal bottleneck is inter-process communication, particularly serialization within the WorldModelManager, which contributes approximately 0.6–1.9 ms per frame. This reflects a trade-off introduced by process isolation, where improved modularity and fault tolerance come at the cost of additional communication overhead.

During real-robot operation, a known 50 ms command throttling mechanism is applied at the dispatch stage to ensure stable communication with hardware. As a result, this external constraint dominates the effective response latency of the physical system, rather than internal processing delays.

These results indicate that the redesigned architecture provides consistent and predictable performance, with sufficiently low latency for real-time robot control.

3.3 Onboard Robot Framework

The onboard robot system was redesigned to address performance and reliability limitations observed in the previous Python-based implementation. The original system, running on a Raspberry Pi 4, relied on Python multiprocessing to handle motor control, UDP communication, and serial communication with an Arduino. In practice, this approach consistently saturated all four Cortex-A72 cores at close to 100% utilization.

This behaviour is largely due to the limitations of Python multiprocessing on resource-constrained systems. Each process introduces additional memory overhead (approximately 20 MB per worker), while inter-process communication requires frequent serialization, both of which significantly reduce effective parallel performance [3] [1]. As a result, the system exhibited reduced responsiveness and instability during extended operation.

In addition, the system relied on the *moteus* motor controller, which is actively developed and frequently updated. These updates introduced changes to motor initialization and calibration behaviour without explicit notification, leading to inconsistencies during system startup and deployment. The lack of deterministic initialization further complicated debugging and reduced system reliability.

To address these issues, the team transitioned to a new onboard architecture implemented in C++ [16], reducing overhead and enabling more deterministic and predictable execution.

The RobotFramework consists of several key components:

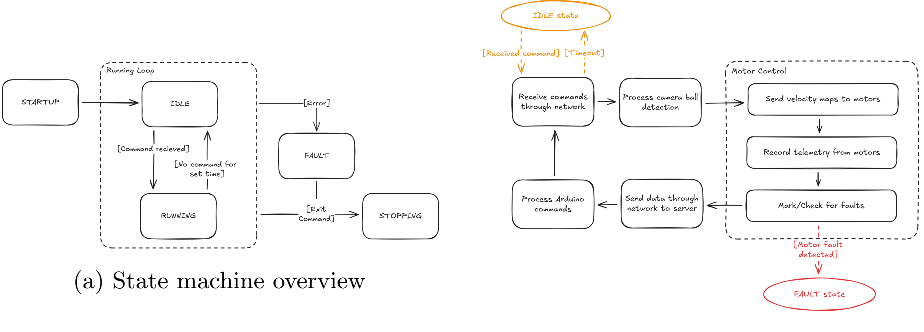
- A central state machine controller
- Networking and command handling
- Telemetry feedback and battery tracking
- Robot motion control
- Hardware interfacing with onboard electronics

3.3.1 State Machine Controller The onboard system is structured around a finite state machine consisting of five states: *STARTUP*, *IDLE*, *RUNNING*, *FAULT*, and *STOPPING* (see Figure 3).

This structure enforces controlled transitions between operational phases, ensuring that initialization, execution, and shutdown occur in a predictable and safe manner. In particular, it prevents unsafe actions during incomplete initialization, enables controlled handling of runtime faults, and ensures consistent system behaviour across deployments.

The *RUNNING* state encapsulates the full control loop, including command reception, motion execution, telemetry feedback, and fault detection (see Figure 3). This separation isolates operational logic from initialization and error-handling routines, improving system reliability and debuggability.

This transition significantly improved runtime stability and reduced unpredictable behaviour during extended operation.



(a) State machine overview

(b) RUNNING state process

Fig. 3. RobotFramework control structure showing (a) the high-level state machine and (b) the RUNNING state execution pipeline.

4 Team Control and Decision-Making Framework

The overall system pipeline consists of perception (vision), state estimation (world model), decision-making (behavior tree), motion planning (Voronoi planner), and execution (dispatcher and robot onboard control).

4.1 Behavior Tree

A key challenge in our system is coordinating decision-making under rapidly changing environmental conditions. In our previous implementation [14], decision logic was implemented as hardcoded conditional sequences without formal structure. This approach had several limitations: adding new behaviours required modifying existing logic (risking regressions), failure handling was not explicitly managed, and behaviours were tightly coupled, making them difficult to test in isolation.

To address these issues, we adopted behavior trees [2] as the primary decision-making framework. Compared to finite-state machines and decision trees, behavior trees provide a reactive and modular structure. Each tick re-evaluates conditions from the root, allowing the robot to respond dynamically to changes in the environment without requiring explicit re-check logic within individual actions.

This decision layer operates on top of the shared world model described in Section 3.1 and interfaces directly with the dispatcher, forming a bridge between perception and execution.

4.1.1 Implementation and Integration The behavior tree is implemented using `PyTrees` [12], which provides logging, a shared blackboard, and reusable node abstractions. We currently maintain two behavior trees: one for a striker and one for a goalie (see Figure 4).

Each behavior is implemented as a node class with an overridden update function, returning a status of *success*, *failure*, or *running*. This enables modular composition of behaviours into reusable subtrees.

Typical actions include moving to the ball, gathering world state information, and executing dribble and kick behaviours.

The behavior tree integrates directly with the system dispatcher (see Figure 5). The `SendRobotCommand` node enqueues commands to the dispatcher and returns a failure status if the queue is full, introducing natural backpressure into the control pipeline. This allows higher-level decision logic to react to system constraints rather than blindly issuing commands, aligning decision-making with system-level execution limits.

4.1.2 System Behaviour In simulation, the striker behavior tree enables the robot to autonomously track the ball, perform short dribbling actions, and execute a kick. Continuous re-evaluation of conditions allows the robot to adjust its actions in response to environmental changes, resulting in more responsive and adaptive behaviour compared to the previous implementation.

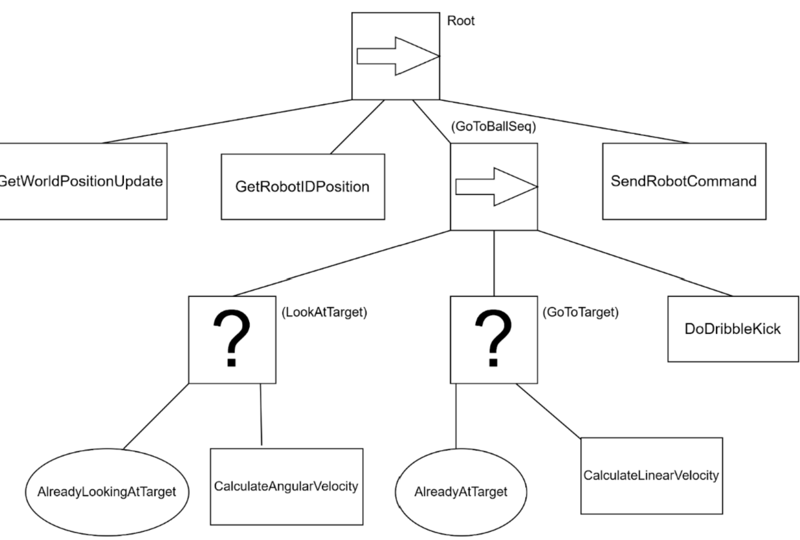


Fig. 4. Behavior tree implementation for the goalie, showing modular subtrees such as GoToBallSeq, LookAtTarget, and GoToTarget.

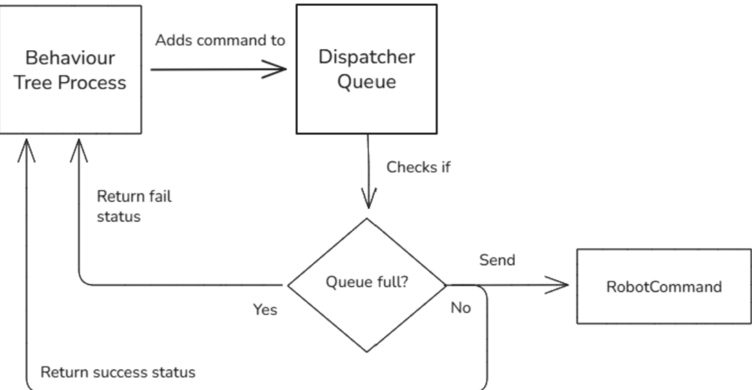


Fig. 5. Interaction between the behavior tree and dispatcher queue. Commands are enqueued and validated before being sent to the robot.

4.1.3 Known Limitations The current implementation still has limitations. The goalie behavior tree does not consistently return failure statuses, causing fallback behaviours to never activate. This reduces responsiveness in defensive scenarios, such as reacting to incoming shots. Additionally, some behaviours lack explicit precondition checks (e.g., verifying ball proximity before kicking), which can lead to inefficient or incorrect actions.

These issues highlight areas for improvement, including better failure propagation and the introduction of more structured conditional checks within subtrees. Future work will also focus on extending the behavior set to include more strategic actions such as passing and positioning.

4.2 Voronoi Path Planner

Building on the Voronoi-based path planner developed in previous work [7], we improved obstacle avoidance by introducing additional sampling nodes around obstacles with a defined clearance distance.

For each detected obstacle, a set of additional nodes is generated at a fixed radial offset (e.g., 150 mm) from the obstacle boundary. These nodes expand the Voronoi graph locally, increasing path options in constrained environments and ensuring that generated paths maintain a safe clearance from obstacles (see Figure 6). The number of nodes n per obstacle is chosen to balance path resolution and computational cost.

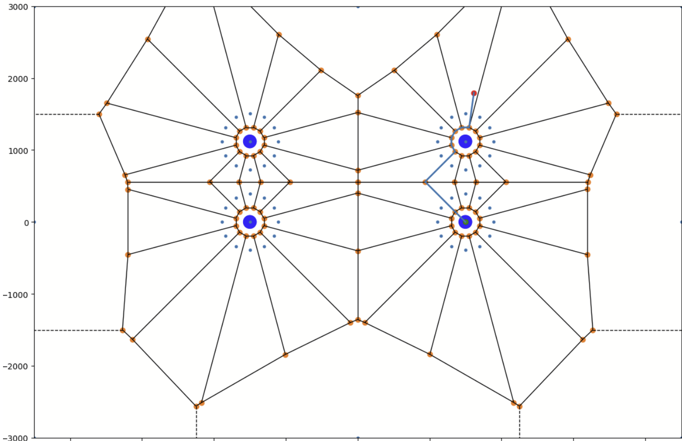


Fig. 6. Voronoi graph generation with additional nodes around obstacles on field (9m x 6m) to enforce clearance.

This planner operates as part of the motion layer and is invoked by the behavior tree when navigation is required. By separating high-level decision-making from path generation, the system maintains modularity while ensuring that movement decisions remain feasible under dynamic constraints.

The addition of clearance-based nodes improves path safety and reduces the likelihood of trajectories intersecting with obstacles, particularly in dense environments.

4.3 Future Plans and Improvements

The recently added behavior tree shows promise for improving responsiveness and decision-making. Current issues include the goalie behavior tree always returning success signals, and the lack of subtrees, which could improve efficiency, by structuring actions like `RobotLookAtBall` and `RobotGoToTarget` with precondition checks.

Future plans include adding strategic behaviors, such as passing and positioning, to give more flexible in-game options beyond dribbling and shooting.

4.4 Reinforcement Learning

The Small Size League (SSL) provides an excellent environment for Reinforcement Learning (RL), particularly for multi-agent coordination tasks that are difficult to fully capture using hand-crafted decision logic. In this work, we investigate RL as a potential extension to our existing decision-making framework, with the goal of improving coordination and adaptability in complex game scenarios.

We focus on the keep-away task [11], where three keeper agents attempt to maintain ball possession against two attackers. Using the rSoccer simulator [8], keepers are trained while attackers execute a fixed strategy (see Figure 7). This task represents a sparse-reward, multi-agent coordination problem and serves as a controlled environment for evaluating learning efficiency and cooperation.

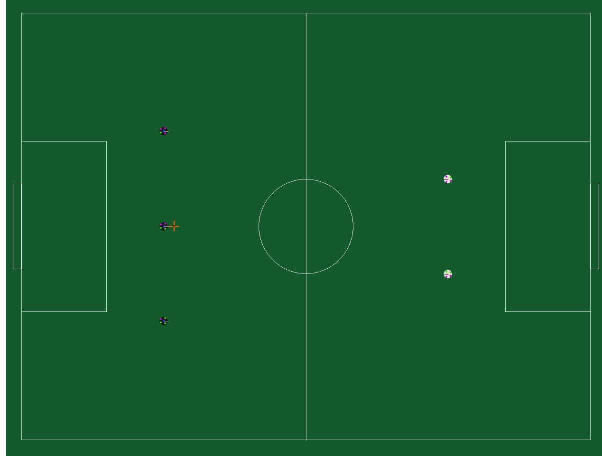


Fig. 7. 3 vs 2 keepaway task in the rSoccer simulator.

Our long-term objective is to develop RL-based policies that can complement or partially replace structured decision-making components, such as behavior trees, particularly for higher-level coordination tasks including passing and positioning.

4.4.1 Algorithm Selection

Based on prior work [7], we selected Soft Actor-Critic (SAC) [6] for its entropy-driven exploration, and its multi-agent extension MASAC [17] for cooperative learning.

We further extend MASAC by incorporating an attention mechanism inspired by prior work [5], resulting in Team Attention Soft Actor-Critic (TASAC). This enables agents to prioritise relevant observations (e.g., ball, teammates, opponents), reduce noise from irrelevant inputs, and improve credit assignment in multi-agent settings.

4.4.2 Experimental Evaluation

We evaluate both MASAC and TASAC in the discrete 3 vs 2 keepaway task over 128 independent training runs.

Results (Figure 8 and Table 2) show that MASAC achieves longer episode durations, indicating stronger ball retention, while TASAC produces significantly more passes, demonstrating improved cooperative behaviour. However, both methods exhibit high variance across runs, indicating instability in training. In particular, TASAC trades stability for improved coordination.

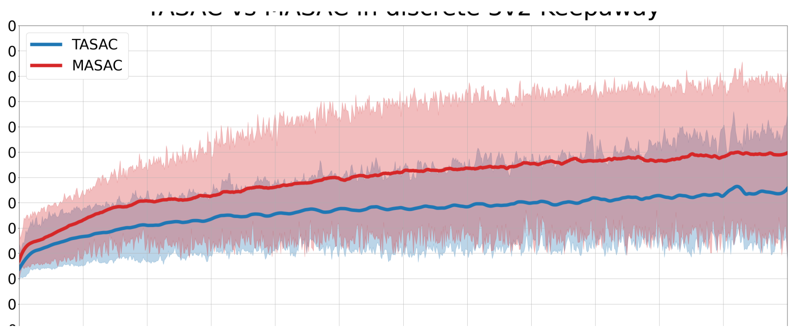


Fig. 8. TASAC vs MASAC performance comparison.

To address sparse rewards, we explored additional exploration strategies. Expert Weaving (EW) combines a partial expert policy (hardcoded passing behaviour) with the learned policy during training, encouraging cooperative behaviour. Random Distribution Distillation (RDD) [4] further improves performance with modest computational overhead. Table 2 summarises these results.

Table 2 highlights the trade-off between stability and cooperation. MASAC achieves higher episode lengths, indicating stronger control performance, while TASAC consistently increases pass occurrence, demonstrating improved cooperative behaviour. The addition of Expert Weaving (EW) significantly boosts passing behaviour across both methods, whereas Random Distribution Distillation (RDD) improves peak performance but with higher variance.

			All	Best	Pass Occurrence
3 vs 2 Discrete	TASAC	—	232 ±23	419 ±327	60.9% (78)
		+ EW	**246 ±18**	422 ±62	**99.2% (127)**
		+ RDD	244 ±21	**457 ±322**	67.9% (87)
	MASAC	—	294 ±33	577 ±318	53.1% (68)
		+ EW	**311 ±36**	567 ±210	**90.6% (116)**
		+ RDD	298 ±29	**597 ±336**	48.4% (62)

Table 2. Comparison of TASAC and MASAC with exploration strategies (EW, RDD) in the 3 vs 2 keepaway task. MASAC achieves higher episode lengths (better retention), while TASAC increases pass occurrence (better cooperation). EW improves passing across both methods, and RDD improves peak performance with higher variance. *All*: mean across runs; *Best*: highest-performing episodes.

4.4.3 Role in System and Future Work At present, these RL methods are evaluated in simulation and are not yet integrated into the main robot control pipeline. The current system relies on behavior trees for deterministic and interpretable decision-making during gameplay.

We are actively investigating how reinforcement learning can be incorporated into the decision-making process alongside existing methods. In particular, RL is being explored as a complementary layer for higher-level coordination, where hand-crafted behaviours become difficult to scale.

Future work will focus on:

- integrating RL-based policies with the existing behavior tree framework,
- extending evaluation to full SSL gameplay scenarios, and
- addressing sim-to-real transfer for deployment on physical robots.

5 Education and Research Integration

Alongside our main Sydney team focused on hardware development and the core team controller, we run a research-oriented subgroup in Freiburg, Germany, specializing in advanced methods like reinforcement learning. To support this, we created a RoboCup SSL lab course https://nr.informatik.uni-freiburg.de/teaching/ws202526/robotic-soccer-lab.

The course introduces students to SSL-style robotic soccer in simulation, aligned with the team's overall system architecture. Twelve students form four teams, each improving a shared baseline code base that offers basic robot control and ball following in the grSim [9,10] simulator, encouraging iterative development.

During the semester, teams complete four two-week assignments targeting key SSL subsystems: motion control and goalkeeper behavior, goal-directed kicking, obstacle avoidance, and coordinated passing. Assignments include technical presentations and live demonstrations in simulation emphasizing clear technical reasoning, and robustness. The course concludes with a final competition consisting of a simplified soccer game executed entirely in grSim.

The lab course onboards students, tests algorithms, and identifies candidates for further research projects. Engaged students are encouraged to pursue further research projects and theses involving physical SSL robots in Freiburg. Successful integration of the RL methods in the real world would also enhance collaboration between the Freiburg research subgroup and the main team in Sydney.

6 Future Improvements and Development

While working with SSL-vision, we observed discrepancies between the real world distances versus the SSL-vision's measurements. Since 2024 [13], we have used motion control threshold zones to address this. During the competition in Eindhoven, we noted that the robot or ball could disappear from vision. We currently use a last known position approach, but it remains limited when the robots or the ball are out of range or not recognized by SSL-vision.

During the making of the qualification video, we observed that our system could not reliably estimated the ball's travel time or velocity, affecting the goalie's response timing. This insight will guide improvements in our next model. We aim to enhance our Ball Trajectory Linear Regression Model [13]. Currently applied only for the goalie, the upgraded model will be extended for use anywhere on the field, for both the ball and the robots.

7 Conclusion

To conclude, the above are the adjustments and new development progress we have made throughout the past year.

Acknowledgments We received partial support from the Office of the NSW Chief Scientist & Engineer in 2024. We also received partial support from Westpac Banking Corporation to travel to RoboCup 2024 in Eindhoven, NL.

Declaration The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Anonymous: Analysis of python multiprocessing overhead on resource-constrained systems. arXiv preprint arXiv:2601.10582 (2026), https://arxiv.org/abs/2601.10582, reports 20 MB memory overhead per worker process on Raspberry Pi 4
2. Colledanchise, M., Ögren, P.: Behavior Trees in Robotics and AI: An Introduction. CRC Press (2018). https://doi.org/10.1201/9780429489105
3. Cornell ECE5990: Python multiprocessing performance on raspberry pi. https://courses.ece.cornell.edu/ece5990/ECE5990_Fall15_FinalProjects/Sharma_Mody_Digit_Recognition_Project/Python_multi.html (2015), observed 3× speedup on 4 cores due to multiprocessing overhead

4. Fang, Z., Yang, K., Tao, J., Lyu, J., Li, L., Shen, L., Li, X.: Exploration by random distribution distillation (2025), https://arxiv.org/abs/2505.11044
5. Garrido-Lestache, H., Kedziora, J.: Enhancing multi-agent collaboration with attention-based actor-critic policies (2025), https://arxiv.org/abs/2507.22782
6. Haarnoja, T., Zhou, A., Abbeel, P., Levine, S.: Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In: Dy, J., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 80, pp. 1861–1870. PMLR (10–15 Jul 2018), https://proceedings.mlr.press/v80/haarnoja18b.html
7. Linh, T.: TurtleRabbit 2025 SSL Team Description Paper (Jan 2025)
8. Martins, F.B., Machado, M.G., Bassani, H.F., Braga, P.H.M., Barros, E.S.: rsoccer: A framework for studying reinforcement learning in small and very small size robot soccer. In: Alami, R., Biswas, J., Cakmak, M., Obst, O. (eds.) RoboCup 2021: Robot World Cup XXIV. pp. 165–176. Springer International Publishing, Cham (2022)
9. Monajjemi, V., Koochakzadeh, A., Ghidary, S.S.: grsim - robocup small size robot soccer simulator. In: RoboCup (2011)
10. Rahimi, M.M., Segre, J., Monajjemi, V., Koochakzadeh, A., MohaimenianPour, S., Ommer, N., Kimura, A.K., Feltracco, J., Sato, K., Ahsani, A.: Grsim. https://github.com/RoboCup-SSL/grSim/ (2021), gitHub repository
11. Stone, P., Sutton, R.S., Kuhlmann, G.: Reinforcement learning for robocup soccer keepaway. Adaptive Behavior **13**(3), 165–188 (2005)
12. Stonier, D.: Pytrees. https://py-trees.readthedocs.io/ (2024)
13. Trinh, L., Anzuman, A., Batkhuu, E., Chan, D., Graf, L., Gurung, D., Jamal, T., Namgyal, J., Ng, J., Tsang, W.L., Wang, X.R., Yilmaz, E., Obst, O.: TurtleRabbit 2024 SSL Team Description Paper (Feb 2024). https://doi.org/10.48550/arXiv.2402.08205
14. Western Sydney University TurtleRabbit Team: Turtlerabbit team control system (2024 post-eindhoven). https://github.com/WSU-TurtleRabbit/team-control/tree/2024-post-eindhoven (2024), accessed: 2026-04-05
15. Western Sydney University TurtleRabbit Team: Turtlerabbit teamcontrol system 2025. https://github.com/WSU-TurtleRabbit/2025-TeamControl (2025), accessed: 2026-04-05
16. Western Sydney University TurtleRabbit Team: Turtlerabbit robotframework. https://github.com/WSU-TurtleRabbit/RobotFramework (2026), accessed: 2026-04-05
17. Xiao, S., HUANG, Z., ZHANG, G., et al.: Deep reinforcement learning algorithm of multi-agent based on sac. Acta Electronica Sinica **49**(9), 1675–1681 (2021)