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 TurtleRabbit 2026 RoboCup SSL team, we adopted a year-long development cycle following the previous year's qualification, enabling a comprehensive set of improvements across both hardware and software subsystems.

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

On the hardware side, we developed a dedicated 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. Using simulation environments such as rSoccer and grSim, we investigate learning-based methods for cooperative behaviors including passing and ball retention, and explore attention mechanisms and exploration strategies for sparse-reward SSL 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

We have encountered hardware limitations in our robot platform, leading us to implement software-level improvements on existing electronics rather than replacing the hardware. This approach retained proven components while improving system performance through architectural refinements.

As a result, we rebuilt the control stack, most notably by introducing a new onboard software system that enables lower-latency and more deterministic control of the robot. These changes improve responsiveness, stability, and reliability under real-time conditions and are a key part of the current robot architecture.

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. While the full dual-mode kicker (chip and straight) remains under development, significant progress has been made on a high-voltage capacitor charging system as used by several leading SSL teams.

To ensure continuity in robot operation, we continued refining the existing **straight-kicker design** during development of the new system. This provides a reliable competition solution while allowing parallel validation and iteration of the new electronics platform.

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 our earlier iteration of our dribbler module, we used a conical shaped roller made from 95A Shore hardness TPU with a spiral surface pattern and an outer

layer of silicone tape. This roller was driven by a 5200KV brush-less motor, connected via two Nylon (PA66) gears with an approximate gear ratio of 7/15, all mounted in a PLA frame.

During testing, the ball was observed to bounce off the dribbler uncontrollably whilst the dribbler was active. Analysis indicated insufficient ball grip, low applied torque applied, and excessive roller speed as likely causes. To address these issues, we are exploring the addition of a damping module, adjusting the gear ratio, and alternative roller materials.

These changes are being incorporated into a new frame design, including a revised motor mount and an integrated housing for a small camera positioned above the dribbler. This camera will enable ball detection, allowing the robot to determine when to kick and dribble.

2.2.1 Dribbler Roller We are also investigating alternative fabrication methods for the dribbler mechanism, specifically cast molding. After reviewing available materials, DIY urethane was selected as a candidate, either for monolithic roller fabrication or as a surface coating.

In the qualification phase, a consistently positive outcome has not yet been achieved, primarily due to the high adhesion of urethane. While beneficial for ball grip, this property creates significant challenges during casting and demolding. Preliminary experiments indicate that applying a neutral release agent or lubricant as an internal mold lining can reduce surface adhesion and improve demolding.

2.2.2 Gear Train In the latest dribbler design,the roller often stalled upon ball impact because of insufficient torque to counter the ball's force. As a result, successful dribbling only occurred when the ball was slowed or gently fed into the roller.

To fix this, we added a third gear stage, creating a compound gear train that increased the overall gear ratio from 1:2 to 1:6. The higher reduction significantly increased the mechanical advantage and significantly increased the output torque at the roller.

2.2.3 Dribbler Frame - Motor Holder Reviewing the motor holder revealed that motor replacement was difficult because it was secured by screws requiring disassembly of the dribbler frame, and the bar connecting the sides of the dribbler frame was hard to install and remove.

As a solution, the motor holder clips onto the connecting bars, and the motor clips into the holder, eliminating inaccessible screws. The design ensures a tight fit while allowing easy assembly and disassembly for dribbler motor adjustments. In the design displayed in Figure 1, the connector rods clips into the motor holder, with a front-facing screw pressing down on the connector rods.

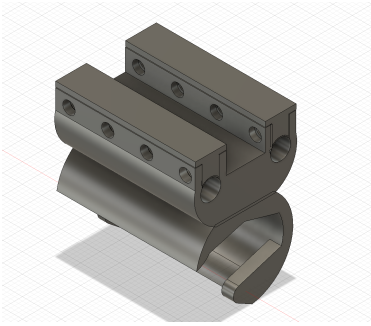

Fig. 1. Image of the new dribbler motor holder.

2.3 Ball Detection

The early robot versions used an IR sensor for ball detection, but it could not reliably detect the ball. This led us to adopt a small camera mounted above the dribbler for improved ball detection.

The newly designed camera holder (see Figure 2) uses clamps to securely attach the camera between the dribbler's two metal pipes, ensuring mechanical stability. Initial versions adjusted clamp size to fit the dribbler's pipes by referencing online models. Later iterations improved camera positioning by tilting it for better visibility of the ball and adding clamps and cube walls to prevent rotation. The final design was simplified by removing mesh structures and optimizing tolerances for a precise fit.

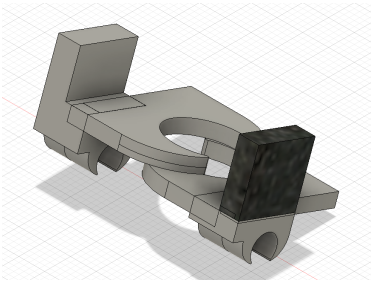

Fig. 2. Image of the holder for the ball detection camera.

3 Software Design and Implementation

Since the RoboCup Competition in Eindhoven 2024, we identified challenges in effectively diagnosing issues within our software stack. Over the past year,

we improved system observability, code readability, and modularity, including developing a lightweight real-time visualizer for plotting and providing immediate feedback during operation (see Figure 6). We also enhanced input validation and type safety to ensure data integrity and reduce runtime errors.

To improve maintainability, documentation has been reinforced across the team during development.

4 Robot - Onboard Software Upgrade

Since Eindhoven 2024 [11,5], our robots used an onboard Python-based system for low-level control. While this approach enabled rapid development, it caused performance issues primarily due to the Raspberry Pi 4's limited processing power. Using Python multiprocessing to control motors resulted in consistently high CPU utilization.

The high load generated excessive heat, causing the Raspberry Pi to thermally throttle during extended operation. This led to degraded performance and, in some cases, complete loss of communication, creating a major reliability concern.

An initial fix was adding a heat-sink to improve passive cooling, which provided short-term relief but failed to scale with growing computational demands.

Additionally, version mismatches and package incompatibilities arose as new boards and software updates were introduced, consuming significant time and resources to manage.

These challenges motivated the development of our current software architecture, designed for improved compatibility, performance, and maintainability.

The RobotFramework is divided into the following sections:

- A central state machine controller
- Networking communication
- Telemetry feedback and battery tracking
- Robot motion control
- Hardware PCB and Arduino interfacing

Some of those will be described in more detail in the following subsections.

4.1 Robot State Machine Controller

The robot operates in five defined states: STARTUP, IDLE, RUNNING, FAULT, and STOPPING.

These 5 states ensure that the robot:

- Initializes all systems safely before motion
- Executes commands and responds to sensory input
- Prevents unsafe actions
- Handles errors gracefully
- Shuts down in a controlled and predictable manner

Using a state machine structure allows us to tightly control which operations are active at any given time and greatly improves system safety and debugging clarity (see Figure 4).

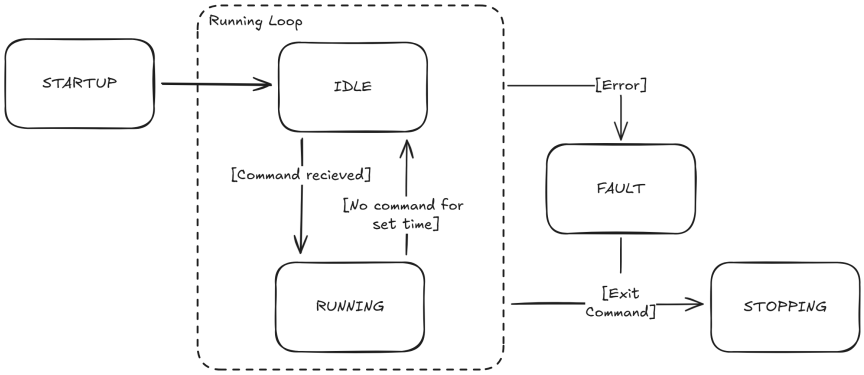

Fig. 3. RobotFramework state machine flowchart.

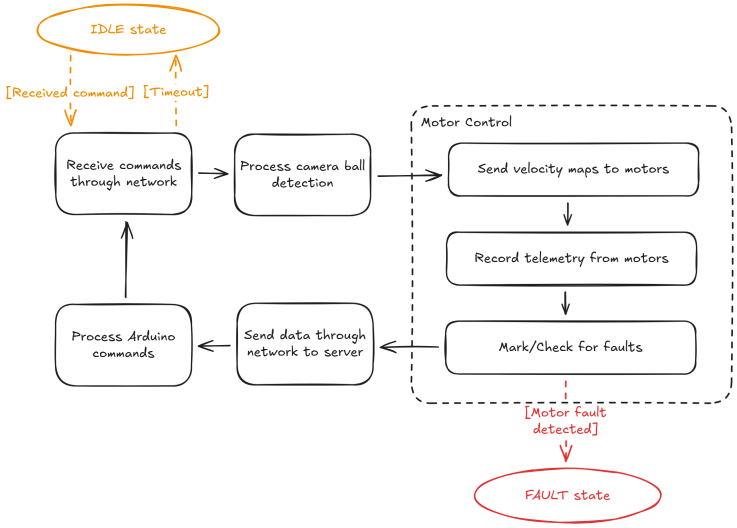

Fig. 4. RobotFramework RUNNING state process flowchart.

4.2 Robot Networking and Command Handling

Communication between the external server and robots uses **User Datagram Protocol (UDP)**, chosen for its low latency and simplicity despite some limitation, which makes it well suited for real-time control over a shared wireless network.

Since the past year [5], we observed a buffer in the Raspberry Pi 4 WiFi receiver causing received packets to be outdated because unicast UDP lacks a Time-To-Live (TTL). To address this, the RobotFramework was designed to prioritize the most recent command and discard old messages, ensuring the robot operates on the latest available packet.

Testing confirmed this approach delivers smooth, near real-time responsiveness in robot control.

4.3 Logging and Observability

During testing, we encountered issues with non-persistent console output, especially when debugging long trials or multiprocessing components. To address this, we implemented a logging module. This module allows us to separate files and processes for different log statements. Each entry is tagged with a timestamp (in milliseconds since initialization), a log level, and a module or subsystem identifier.

During debugging and performance evaluations, the logging system has improved our workflow in filtering and identifying the corresponding issues and locations, Hence, this has improved our overall efficiency in filtering and analysis.

5 Team Control System - Server-side Software

Since Eindhoven 2024 [5], we observed noticeable delays between the server and robots. To address this, reducing internal process halts, delays and failures has been our main goal of the new system design.(see Figure5).

5.1 Flow of System

In addition to Figure 5, SSL-vision packets, SSL-game controller packets and robot feedback messages are received by a dedicated process that performs comparison checks and storage. Once the checks are complete, data is forwarded to the runner, which updated the shared world model.

To avoid race conditions from concurrent updates, we added version control to the world model. Since the runner is the only writer to this world model, it serves as the single source of truth for all modules.

Tactical and behavioral decisions are made based on this model and converted into robot commands. These commands are then send by the dispatcher to individual robots.

5.1.1 Shutting Down and logging Process Since all processes run in isolated threads, we introduced a shutdown and logging process (see Logging and Observability section) to catch exceptions and logs before termination. This is crucial for troubleshooting failures and ensuring clean shutdowns.

5.2 Real-time Graphical Client (lightweight visualization)

To better display background compute strategies, we have developed a lightweight graphical client (see Figure 6) providing near real time visual feedback.

The client visualizes the SSL field state. Blue and yellow markers represent the two teams with robot identifiers and orientation vectors. The ball position is

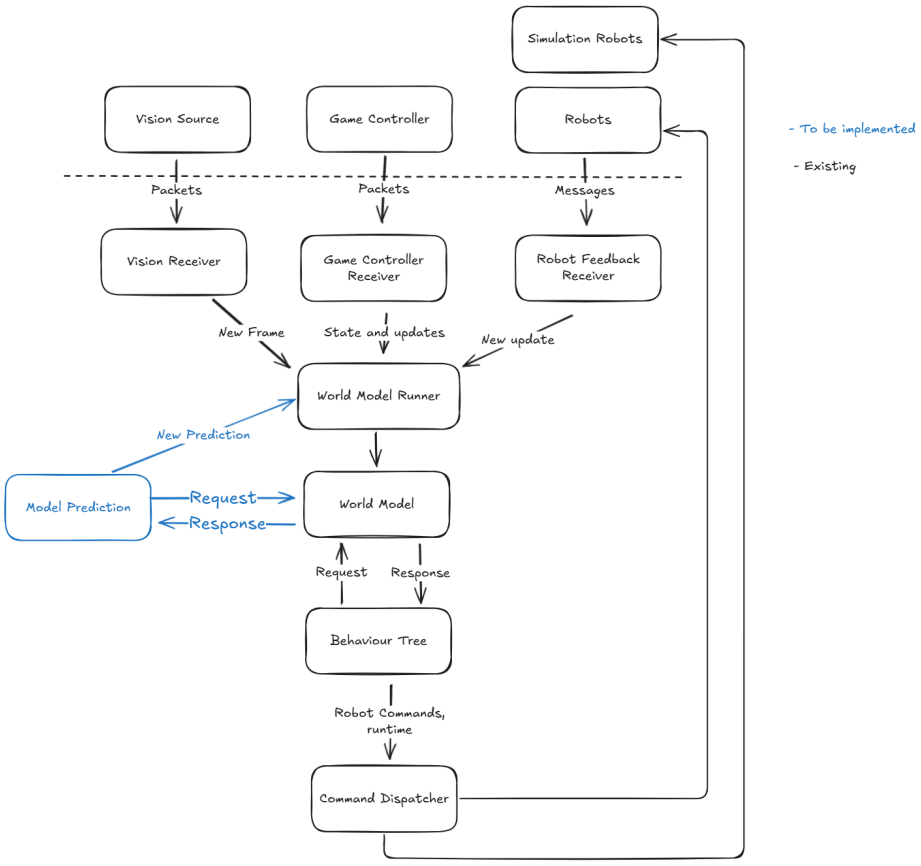

Fig. 5. Software Architecture as of 2026.

shown at the center, while the field layout and reference markings are rendered according to official SSL dimensions. The visualization updates in real time using UDP data streams.

The graphical interface is implemented using `Matplotlib` [4].

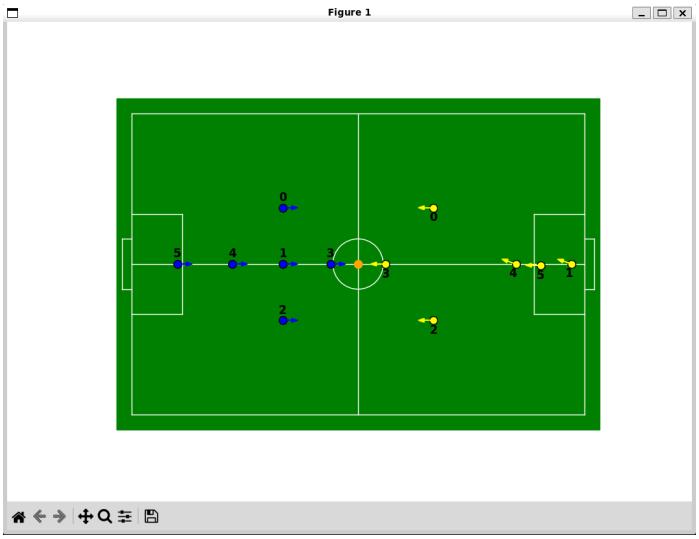

Fig. 6. Real-time graphical client.

5.3 Server-side Networking

To better collaborate with the server side, a **Dispatcher module** was introduced. It continuously sends robot action commands for a defined duration. If a command exceeds its runtime, the dispatcher replaces it with a reset command, replacing the previous command. command handling process guarantees that robots receive constant updates each iterations despite delay that may be caused in decision making processes.

On each iteration, the Dispatcher: 1. Checks for any new commands and their runtimes 2. Sends the command to the corresponding robots 3. Reset expired commands

Commands are encoded as space-separated strings containing robot IDs, velocity targets, action flags (e.g., kick, dribble), and timestamps. On the robot, a dedicated **Network module** parses incoming packets and forwards commands to the relevant subsystems. To prevent buffer overflow and keep commands current, incoming commands immediately replace previous ones.

If the Dispatcher shuts down (due to error or intentionally), it must send a reset command to all robots to stop their actions.

5.4 Manual System Configurations

The manual configuration module retrieves all settings at the start of the team software, providing a quick insight into the user's current network configurations and speeding up troubleshooting.

After evaluating testing and development workflows, we recognized the need for a centralized configuration module. This module reads all network and user setup configurations, including:

- Whether the user uses grSim Vision or real SSL-Vision
- Whether the user wants to send commands to grSim and its location
- Communication IP addresses and ports for SSL software and robots
- Vision Robot Shell IDs and initial grSim Shell IDs
- IP and port configurations for both teams' robots

Manually updating this configuration file minimizes any internal code error, reducing troubleshooting time and isolating issues to OS, physical networking (e.g, physical Ethernet or WiFi), or firewall settings.

5.5 SSL-Software Information Handling in Team Software

As there are continuous incoming traffic from the SSL-Vision, SSL-Game-Controller, and our Robots in the same network, we have structured our software to

5.5.1 Behavior Tree One of the key challenges in our implementation is streamlining decision-making based on environmental conditions.

Optimizing our robots' decision-making is essential because it improves efficiency and maintaining competitiveness. While the basic behaviors are already implemented, we needed a system to coordinate them in an organized and efficient manner. While many computational models exist (i.e. finite-state machines and decision trees), we opted for behavior trees for their reactive, modular nature and their ability to promote behavior re-usability.

The behavior tree guides the robots on which action to perform next. Currently, we maintain two behavior trees: one for a typical striker and one for a goalie (see Figure 7). To ensure a clean and efficient implementation, we use `PyTrees`, a Python library that provides built-in logging, a shared memory, and a collection of abstract classes for blueprints. Leveraging this library saved significant development time and effort, allowing us to focus behavior design. Examples of elementary actions supported include:

- Going to the ball.
- Gathering robot positions.
- Performing dribble and kick actions.

Integrating this behavior tree into our existing framework required four steps:

1. Creating a high-level abstraction of the behavior tree.
2. Representing each node or behavior in the tree as a class inheriting PyTrees.
3. Overloading update functions so each node performs its designated task.
4. Returning a status (i.e. success, failure, or running) depending on task outcome.

Similarly, for the goalie, the process of implementing a behavior tree followed the same four step process.

The behavior tree integrates with our dispatcher to queue robot commands (see Figure 8). The `SendRobotCommand` node is responsible for interfacing with the dispatcher and returns a fail status if the queue is full else a success status is returned by the node and the command is sent to the robot to perform.

The behavior tree currently supports kicking and dribbling. In simulation, the robot autonomously chases the ball, dribbling the ball momentarily before performing a kick.

Future work aims to add behaviors allowing the robot to wait for the optimal shooting moment instead of shooting in a haphazard manner.

We identified issues with the goalie behavior tree: is not responsive to environmental changes, failing to react when the striker shoots to save the goal. Moreover, goalie behaviors never return a failure status, causing the tree to always signal success. This can lead to incorrect commands and unwanted robot behavior. Despite this, the striker behavior tree integrates well into our current architecture and improves task prioritization.

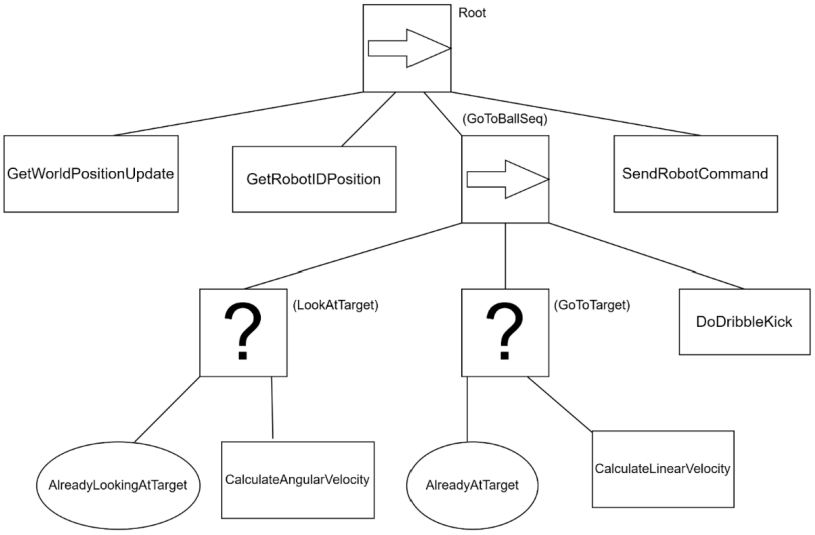

Fig. 7. Current behaviour tree implementation for the goalie showing elementary behaviors and subtrees (GoToBallSeq, LookAtTarget, and GoToTarget).

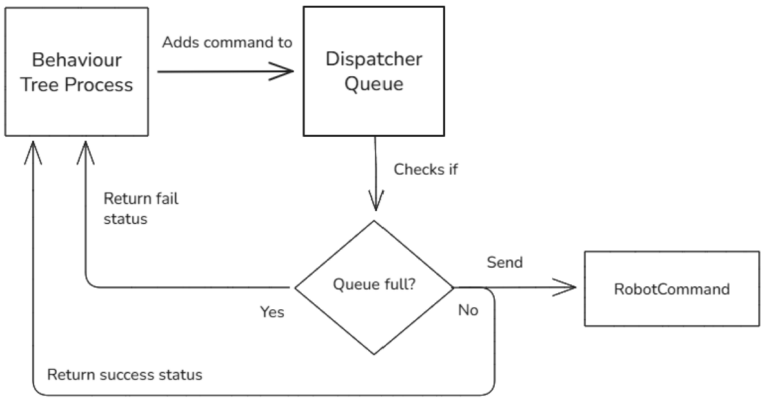

Fig. 8. A diagram showing how the behavior tree process interacts with the dispatcher queue

5.6 Voronoi Path Planner

Building on last year's Voronoi Path Planner [5], we further improved obstacle avoidance by generating additional nodes with a defined clearance around the obstacles. This ensures the robot can navigate safely around them (see Figure 9).

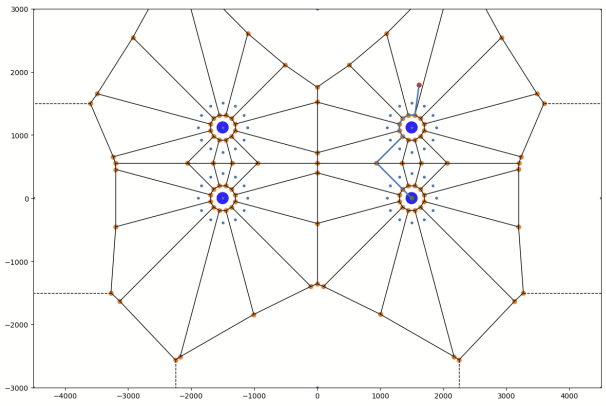

Fig. 9. Voronoi graph generation with additional nodes around obstacles.

5.6.1 Future Plans and Improvements Though recently added, the behavior tree has great potential to improve responsiveness and decision-making, which is essential for being competitive. Further improvements and fixes will optimize its efficiency.

A major issue is that the goalie behavior tree always returns success signals; the cause is currently unknown and under investigation.

Currently, the tree lacks subtrees. Introducing subtrees could improve efficiency. For example, making behaviors such as `RobotLookAtBall` and `RobotGoToTarget` selectors or fallback nodes with children that check preconditions and execute action.

Future plans include adding strategic behaviors such as passing and positioning, providing robots with more flexibile in-game options beyond dribbling and shooting.

5.7 Reinforcement Learning

The Small Size League (SSL) provides an excellent environment for Reinforcement Learning (RL), advancing Multi Agent RL (MARL) research while advancing autonomous coordination and decision-making in robotic teams.

We focused the keep-away task [10], where three keepers try to maintain ball possession against two attackers. Using the rSoccer simulator [6], keepers were trained while attackers executed a hard-coded strategy (see Figure 10).

Our goal is to develop solutions that generalize beyond the keepaway task to more complex scenarios, such as a full SSL game. This results in a sparse-reward environment, making learning more challenging and placing greater importance on effective exploration.

Priour work [5], demonstrated that in a sparse-reward SSL environment, Soft Actor-Critic (SAC) [3] outperforms other reinforcement learning algorithms

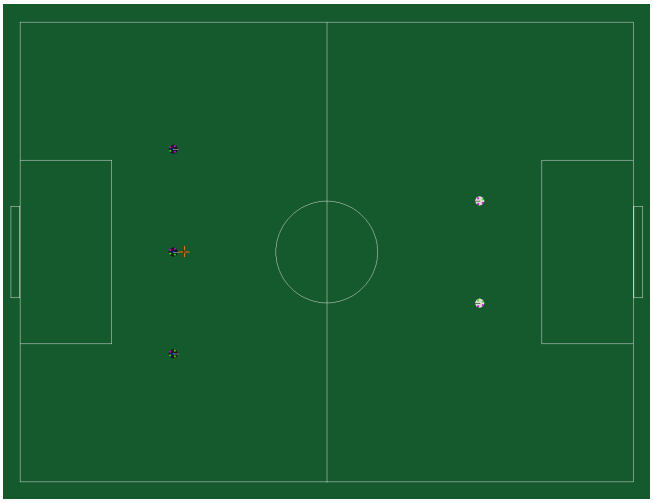

Fig. 10. Image of a 3 vs 2 keepaway task environment in the rSoccer simulator: Keepers (black) vs attackers (white). The ball is marked with an orange cross for better visibility.

such as Proximal Policy Optimization (PPO) [9]. This performance advantage is primarily attributed to SAC's maximum-entropy objective, which encourages more effective exploration. We selected the multi-agent extension of SAC, Multi-Agent Soft Actor-Critic (MASAC) [12] for our experiments.

Inspired by the work of Belinchon and Kedziora [2] on the Team-Attention Actor-Critic (TAAC), we added attention mechanisms into SAC's actor and critic networks. This allows each agent to focus on the most relevant observations, such as nearby teammates, opponents, or the ball, while reducing the impact of less informative inputs. We call this Team Attention Soft Actor-Critic (TASAC).

In MARL, credit assignment is a well known challenge, as it requires determining each agent's contribution to the overall outcome and assigning rewards accordingly. Attention mechanisms help alleviate this issue by enabling the critic to selectively weight information from different agents and state features, resulting in clearer learning signals.

We compare TASAC to MASAC (see Figure 11) in the keepaway task, where keepers aim to maximise the episode length before the attackers steal the ball. MASAC generally achieves longer episode lengths by better exploiting the environment, for example by blocking the attackers' movement or maneuvering the ball into less accessible areas of the field, compared to TASAC. In contrast, TASAC demonstrates substantially higher cooperation, as measured by the number of passes exchanged.

To tackle sparse rewards, we explored bonus exploration methods. Inspired by imitation learning, we combined a partial expert policy, which is a hardcoded passing behavior, with the model's predicted policy during training. This "Expert Weaving" (EW) maintains proximity to the originally sampled actions while

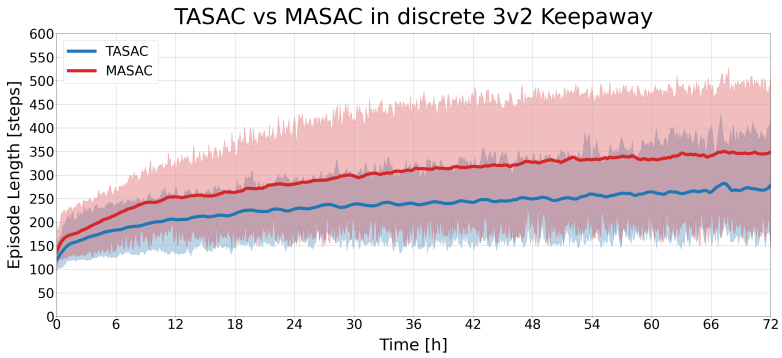

Fig. 11. TASAC vs MASAC: Mean evaluation episode length and 90% confidence interval in the discrete 3 vs 2 keepaway task over 128 different seeds.

injecting external guidance. EW successfully encourages the desired passing behavior and significantly increases the number of passes performed between agents.

Additionally, we investigated Random Distribution Distillation (RDD) [1], which improves overall performance with only a modest increase in computational cost.Table 1 compares TASAC and MASAC using these different exploration methods.

			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 1. Episode length with added Expert Weaving (EW) and Random Distribution Distillation (RDD) in the discrete 3 vs 2 keepaway task. *All* shows the mean and standard deviation over all data points, and *Best* the mean and standard deviation of the highest value from each of the 128 runs. *Pass Occurrence* indicates the percentage of runs in which a pass occurs during the highest-value evaluation.

Future work willo extend this approach beyond the keepaway task to the full SSL game, addressing greater complexity and coordination challenges. We also plan to bridge the sim-to-real gap to deploy the learned policies on physical robots for real-world RoboCup SSL competitions.

6 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 cerated 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, alignmed 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 [7,8] 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, robustness, and reproducibility.

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.

7 Future Improvements and Development

We look forward to reinforce robots' fundamental behaviors, with behavior tree and Voronoi Path Planner.

7.1 Current Motion Control and Limitations

While working with SSL-Vision, we have noticed gaps in between the real world distance versus the SSL-vision's interpretation. As a result, it has lead to our approach of current motion control threshold zones since 2024 [11] has been challenging. As we noticed during the competition in Eindhoven, there was a possibility that the robot or ball in the vision being disappeared, we currently have adapted an object last known position approach, but as the ball gets covered or disappeared out of range, this approach is still lacking.

Hence, further investigation and improvements will be made on this area.

7.2 Prediction Model and Trajectory Planning

During the making of this qualification video, we have identified the incapability to provide an estimated time or velocity for the ball's travel, resulting in our

goalie's somewhat delayed or too quick response time, this will also be taken into consideration for our next model.

Hence, We aim to focus on improving our existing Ball Trajectory Linear Regression Model [11]. Currently, the existing model is only applied on our Goalie Robot Behavior, however, we are in the works to ensure that the new model can be used at any point within the field, for both the ball and the robots.

8 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. 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
2. Garrido-Lestache, H., Kedziora, J.: Enhancing multi-agent collaboration with attention-based actor-critic policies (2025), https://arxiv.org/abs/2507.22782
3. 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
4. Hunter, J.D.: Matplotlib: A 2d graphics environment. Computing in Science & Engineering **9**(3), 90–95 (2007). https://doi.org/10.1109/MCSE.2007.55
5. Linh, T.: TurtleRabbit 2025 SSL Team Description Paper (Jan 2025)
6. 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)
7. Monajjemi, V., Koochakzadeh, A., Ghidary, S.S.: grsim - robocup small size robot soccer simulator. In: RoboCup (2011)
8. 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
9. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms (2017), https://arxiv.org/abs/1707.06347

10. Stone, P., Sutton, R.S., Kuhlmann, G.: Reinforcement learning for robocup soccer keepaway. Adaptive Behavior **13**(3), 165–188 (2005)
11. 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
12. 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)