NexusPrime – Team Description 2026

Enyan Liu[1,2], Tengwei Bi[1,2], Jonathan Loo[1,2⋆], and Yan Sun[1,2]

[1] School of Electronic Engineering and Computer Science (EECS), Queen Mary University of London, UK
[2] Joint Student Innovation Centre (JSIC), Queen Mary University of London, UK
j.loo@qmul.ac.uk

Abstract. The NexusPrime team, founded in October 2025, aims to bridge the gap between modern multi-agent reinforcement learning (MARL) techniques and the complex RoboCup 2D Soccer Simulation (RCSS) environment. We first introduce a novel cloud-native RCSS cluster management system, rcss_cluster [12], which transforms the rcsserver into an elastically scalable service for massive parallel data collection. Second, to bridge the C++ environment with Python-based RL libraries, we leverage the open-source SoccerSimulationProxy (SSP) developed by the Cyrus2D team, which is a gRPC-based, language-agnostic framework that seamlessly integrates with the helios-base code [1]. Building upon this, we designed match_composer [12], which orchestrates the match for certain training scenarios by simple configurations and integrates and hosts the SSP as player processes, providing the "action-observation" loop interaction through the gRPC interface, and a distributed RL training framework based on Ray. Currently, our submitted agent is based on an exploratory training phase using Stable-Baselines3 (SB3) deeply integrated with the SSP. Although massive distributed training via Ray has not yet officially commenced, our work provides a scalable, modern foundation for applying data-driven RL techniques to RoboCup 2D, and we anticipate significant performance improvements once large-scale training is deployed.

1 Introduction

1.1 Research Background

The RoboCup Soccer Simulation 2D (RCSS) league, established in 1997, is one of the oldest and most classic environments for multi-agent systems [2]. It aims to foster research on how multiple autonomous agents cooperate in complex, dynamic, and adversarial environments. With its partially observable, non-stationary, and continuous state-action spaces, RCSS 2D naturally represents a prototypical Multi-Agent Reinforcement Learning (MARL) problem [13].

Historically, the most competitive teams in the league (e.g., Helios [3], Cyrus [4], and YuShan [5]) have accumulated profound domain knowledge, relying heavily on handcrafted rule-based systems and finite-state machines. This engineering-driven paradigm has proven highly effective due to its stability and interpretability. In recent years,

⋆ Corresponding author: Jonathan Loo (j.loo@qmul.ac.uk)

while Reinforcement Learning (RL) has begun to emerge within the community, it remains a niche approach [6]. Most current RL applications in RCSS 2D focus on modular replacement—such as optimizing specific skills like passing, intercepting, or shooting—while still being deeply embedded within traditional heuristic frameworks. Despite RCSS 2D being an ideal testbed for MARL, there is a conspicuous absence of mature, modern, and scalable MARL solutions in the community.

1.2 Motivation and Challenges

The NexusPrime team, originating from an undergraduate graduation project initiated in October 2025, conducted a systematic diagnosis to understand why modern MARL has failed to take root in this community. We identified that the primary bottlenecks are not the RL algorithms themselves, but the severe limitations of the legacy simulation infrastructure. Specifically, applying modern RL to RCSS 2D faces the following critical challenges:

Archaic Simulation Infrastructure: The official rcssserver, operates on a single-process, single-room paradigm [2]. It lacks native support for programmatic orchestration, lifecycle management, or remote control. Modern RL requires the ability to spawn, reset, and tear down thousands of parallel episodes efficiently, a task the original server is entirely unequipped to handle.

Outdated Network Design: The server relies on a legacy UDP protocol with a 1:1 socket model. Crucially, the server always binds to random ports to respond to clients after the initial connection. This design severely interferes with modern Network Address Translation (NAT) mechanisms, resulting in poor connectivity. It is fundamentally incompatible with modern decoupled client-server architectures. Furthermore, the connectionless nature of UDP increases the maintenance cost for clients, lacks multiplexing capabilities, and provides poor observability during large-scale training.

The Language Barrier and Performance Trade-offs: The most mature codebases in the community (e.g., Helios-base [3], Cyrus2D [4]) are implemented in C++. Integrating modern Python-based deep learning frameworks with these C++ bases incurs extremely high development and communication costs. Conversely, pure Python reimplementations (e.g., Pyrus2D [7]) often suffer from poor execution performance, struggling to meet the strict 100ms real-time decision constraint of the simulator.

1.3 Project Objectives and Contributions

To bridge the gap between modern MARL algorithms and the RCSS environment, the NexusPrime team aims to reconstruct the training infrastructure from the ground up. Our main contributions are summarized as follows:

Scalable Cloud-Native Simulation Infrastructure: We designed rcss_cluster [12], a robust simulation cluster management system developed in Rust. By leveraging Kubernetes and Google's Agones [8], we wrapped the legacy rcssserver into an elastically scalable, cloud-native microservice, enabling massive parallelization for data collection.

Cross-Language match Control Bridge: To solve the language barrier and address the complexity of coordinating multi-process matches, we designed match_composer [12],

a specialized Sidecar service. It adopts a declarative configuration to automatically orchestrate the entire match lifecycle, effectively replacing manual scripts with a robust, API-driven control plane.

Distributed RL Training Framework: We built a distributed RL training pipeline based on Ray [9]. Instead of relying on complex hierarchical architectures, we define the action space using six fundamental atomic actions (catch, turn, dash, kick, move, and tackle). We validated the framework's adaptability by conducting exploratory training using Stable-Baselines3 (SB3) [10] and Proximal Policy Optimization (PPO) [11] under the Centralized Training with Decentralized Execution (CTDE) paradigm.

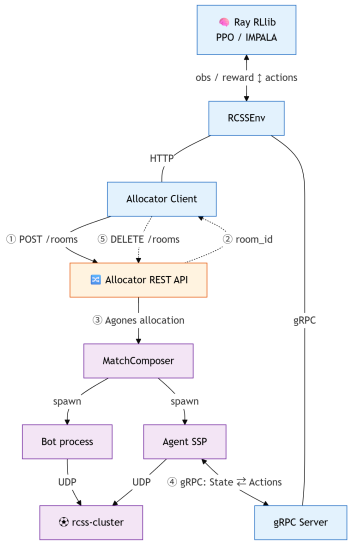

Fig. 1. Architecture of the NexusPrime MARL Solution

2 Scalable Simulation Infrastructure – rcss_cluster

The most technically demanding and architecturally novel contribution so far is the design and implementation of rcss_cluster, a cloud-native simulation cluster management system for RCSS 2D [12].

The rcss_cluster is implemented entirely in Rust and is ready for deployment. The choice of Rust is because it provides the performance guarantees needed for real-time communication, multi-process management, and protocol translation, while offering modern abstractions for asynchronous execution, concurrency, error handling, and low-level control.

2.1 Process Management Layer (`process` crate)

The most fundamental problem is how to control the rcssserver process. The `process` crate addresses this by wrapping the rcssserver binary in a managed lifecycle abstraction.

The `process` crate encapsulates the process management logic and creates an OfflineCoach client for RCSS room environment management operations. Outer structs provide capabilities such as structured configuration APIs, log management, and graceful shutdown.

This layer solves the raw operational problem: rather than manually starting rcssserver, configuring ports, and relying on successful initialization, the `process` crate provides a deterministic, programmatic interface for spawning and controlling simulation instances.

2.2 Protocol Translation Layer (`common` crate and `server` crate)

The second fundamental problem is the rcssserver UDP protocol. The `common` crate contains a full command encoding/decoding system that parses and generates the text-based UDP protocol used by rcssserver for both trainer and player commands.

Building on this, the `server` crate exposes rcssserver through modern HTTP and WebSocket APIs, including HTTP endpoints for trainer commands, WebSocket endpoints for player connections, and interfaces for server-state tracking.

This translation layer is the key innovation that makes rcssserver accessible to any HTTP-capable client without requiring low-level UDP protocol handling.

2.3 Kubernetes Integration (`service` crate)

A single rcssserver instance can host only one match at a time. For RL, however, training efficiency scales almost linearly with the number of parallel environments: modern RL algorithms such as PPO routinely collect experience from hundreds or thousands of simultaneous rollouts [11]. This creates a fundamental tension: the simulator was designed for one match, but training demands many.

The `service` crate resolves this tension by providing two modes: a StandaloneService for local single-instance development and debugging, and an AgonesService that integrates with Agones on Kubernetes [8]. The former preserves a low-barrier entry point for algorithm prototyping; the latter transforms rcssserver into an elastically scalable, cloud-native service.

Agones is an open-source Kubernetes controller from Google, originally designed for multiplayer game server orchestration [8]. It provides exactly the primitives that large-scale RL training requires, including ready-to-use replica pools, allocation APIs, and lifecycle hooks.

Up to this point, the training framework can request a new simulation room through a single REST call and obtain the game server within seconds. When the training episode completes, the pod is automatically recycled and returned to the ready pool.

This on-demand, stateless interaction model enables the transition from training on a single match to training across an arbitrarily large cluster, which is the infrastructural prerequisite for the data-hungry MARL training that follows.

3 Match Composer – Declarative Match Orchestration

The previous section described how `rcss_cluster` transforms the legacy rcssserver into an elastically scalable cloud service. However, provisioning a simulation room is only half the battle. A more nuanced, yet equally critical challenge lies in what happens *inside* that room: orchestrating a complete match. In the traditional RCSS workflow, launching a single game requires manually starting the server, sequentially spawning up to 22 player processes with the correct team name, uniform number, goalie flag, and server address for each, and then monitoring all of them throughout the match. For Agent-type players that rely on the SoccerSimulationProxy (SSP) [1], additional gRPC endpoint configuration is needed. In a research context where hundreds of matches must be launched, torn down, and reconfigured per hour, this manual process constitutes a severe operational bottleneck.

This observation motivates `match_composer` [12], the second major component of our infrastructure. Rather than scripting ad-hoc match setups, we ask: *what if the entire specification of a match—teams, players, policies, initial states, and stopping conditions—could be captured in a single declarative document, and the system would handle the rest?* This shift from imperative scripting to declarative orchestration is the central design thesis of the Match Composer.

3.1 Sidecar Deployment Model

The Match Composer is deployed as a Sidecar container within the same Kubernetes Pod as the rcssserver instance managed by `rcss_cluster`. This co-location is rooted in two observations. First, player processes must communicate with the rcssserver over proxied UDP on `localhost`, so network locality is essential. Second, the Sidecar pattern cleanly separates concerns: the main container hosts the simulation engine and the protocol translation layer described in Section 2, while the Sidecar focuses exclusively on match-level orchestration. Neither component needs to know the internal implementation of the other; they interact only through well-defined interfaces.

At startup, the Match Composer retrieves the `GameServer` object's metadata annotations via the Agones SDK. This configuration is then parsed, validated, and transformed into an executable match plan. The entire bootstrap is thus *zero-touch*: the training framework allocates a GameServer from the Agones pool, the Match Composer reads its intent from the annotations, and a fully configured match begins automatically—no human intervention, no shell scripts, no race conditions.

3.2 Declarative Configuration with Heterogeneous Policies

The top-level configuration schema captures the complete specification of a match: the server endpoint, two opposing teams, referee settings, stopping conditions, and global initial states such as ball placement. Each player entry declares its uniform number, goalie status, and a *policy* that determines its runtime behavior. A player is either a `Bot` (a traditional rule-based agent backed by a precompiled binary such as helios-base [3]) or an `Agent` (a proxy-based player whose decisions are delegated to an external RL controller via gRPC, using the SSP [1]).

This distinction at the schema level is critical: it directly reflects the mixed-mode training scenarios common in MARL research. For instance, one may train a single RL agent alongside ten scripted teammates against a full team of rule-based opponents, or gradually increase the number of RL-controlled players as training progresses. By making this heterogeneity a first-class citizen of the configuration, the Match Composer supports arbitrary combinations without any code changes.

3.3 Lifecycle Management and the HTTP Control Plane

Each spawned player process is wrapped in a monitored handle that captures structured logs and exposes a reactive status channel. If any player process exits unexpectedly, the error propagates immediately to the team state, ensuring that the training framework is promptly notified of failures and can recycle the episode without silent hangs.

The Match Composer exposes its capabilities through a lightweight Restful API, offering endpoints.

3.4 Integration with the Training Loop

From the training framework's perspective, the interaction model is simple: allocate a GameServer, which triggers the Match Composer to automatically bootstrap a match. The `/status` endpoint returns the gRPC connection information for all Agent-type players, enabling the training framework to establish the observation-action loop via the SSP. When an episode concludes, `/restart` recycles the environment for the next rollout.

This design effectively transforms the complex, multi-process, multi-protocol task of running an RCSS match into a stateless, API-driven service. Each GameServer Pod becomes a self-contained simulation unit that can be allocated, configured, used, and recycled—the fundamental primitive that makes large-scale parallel RL training feasible.

4 Exploratory RL with Stable-Baselines3 – rcss-rl-sb3

Before committing to the full distributed training pipeline described in Section 5, we conducted an earlier, exploratory integration of RL into the RCSS environment using Stable-Baselines3 (SB3) [10]. This prototype was implemented in our rcss-rl-sb3 codebase [12] and serves as our current submitted team for this stage. The primary purpose of this effort was not to produce a state-of-the-art team, but rather to answer a more fundamental question: *can a neural network learn to make tactically meaningful decisions within the helios-base framework, and if so, how should RL be embedded into an existing rule-based codebase without destabilising its proven strengths?* The results of this exploration informed nearly every architectural choice in the subsequent distributed framework.

4.1 Deep Embedding: RL as a Strategic Selector, Not a Motor Controller

The most consequential design decision in rcss-rl-sb3 is the level of abstraction at which RL operates. Unlike the approach in Section 5, where the action space consists of raw atomic actions (dash, kick, turn, etc.), this earlier system deliberately operates at a *higher* level: the RL agent does not control low-level motor commands at all. Instead, it acts as a **strategic selector** that determines *which mode* of the helios-base `HeliosChainAction` planner to activate at each decision point.

This design is rooted in a pragmatic observation: helios-base's `HeliosChainAction` [3] already encapsulates decades of domain-engineered skill chains—dribble planners, pass evaluators, shoot calculators, cross generators—each of which converts a high-level intent into a sequence of low-level simulator commands. Discarding this accumulated expertise in favour of raw RL control would be wasteful and, at this stage of the project, impractical. The question is therefore not "RL *versus* rules," but rather "can RL learn to *orchestrate* rules better than the default heuristic?"

Concretely, the action space is a 7-class discrete set: `Aggressive Dribble` (enabling long and short dribble planners), `Safe Dribble` (simple dribble with short dribble), `Killer Pass` (through pass and lead pass), `Safe Pass` (direct pass and simple pass), `Cross`, `Shoot`, and `Auto` (all planners enabled, letting helios-base's own field evaluator choose). Each action corresponds to a specific boolean configuration of the `HeliosChainAction` parameters, which is then passed to the C++ engine for execution. The RL agent thus chooses *what* to do; helios-base determines *how* to do it.

4.2 Hybrid Decision Pipeline via the SSP

The integration is achieved by deeply embedding the RL loop into the SoccerSimulationProxy (SSP) [1] gRPC pipeline. Each of the 11 player processes runs a modified SSP that connects to the rcssserver as a standard helios-base client. On every simulation cycle, the SSP extracts a 124-dimensional state vector from its internal `WorldModel` and forwards it via a TCP socket to a Python-side Gymnasium environment (`SocketSoccerEnv`). The Python environment passes this observation to an SB3-trained PPO policy [11], receives an action index, and sends it back through the same socket. The SSP then translates this index into a `HeliosChainAction` parameter configuration and invokes the C++ planner.

A subtle but important aspect of this pipeline is the **conditional execution model**: RL decisions are only executed when the player is *kickable* (i.e., within ball-control distance). When the player is off the ball, the system transparently falls back to `HeliosBasicMove`, the native helios-base positioning algorithm. This means that the team's overall formation, off-ball movement, and defensive structure remain entirely governed by the battle-tested heuristics, while the RL agent intervenes only at the critical moment of on-ball decision-making. The observation is still sent to the RL server during non-kickable cycles to maintain cycle synchronisation, but the returned action is discarded. This design ensures that the RL agent's learning signal is clean—it only receives credit or blame for decisions it actually executed—while the team retains a stable tactical baseline.

4.3 Preliminary Results

To evaluate whether the RL-embedded framework produces coherent team behavior, we conducted a series of full 11-vs-11 matches against the unmodified helios-base [3]. The training used PPO [11] with a simple goal-based reward signal (+1 for scoring, −1 for conceding), augmented with shaping terms for ball progression and stall detection to mitigate reward sparsity.

Over a series of evaluation matches, the RL-augmented team achieved an average score ratio of approximately $1 : 2.41$, as shown in Fig. 2. It is important to note that this preliminary result was obtained under strict practical constraints: all training and debugging were conducted on CPU-only servers, and the entire development cycle was completed within less than five months. While the RL team does not yet match the pure helios-base, this result is encouraging for several reasons. First, it demonstrates that the RL agent *does not catastrophically degrade* the underlying helios-base performance—a non-trivial achievement given that a poorly trained policy could easily produce worse results than a random baseline. Second, the score trend over training episodes shows a gradual improvement (Fig. 2), indicating that the PPO agent is indeed learning meaningful strategic preferences rather than converging to a degenerate "always Auto" policy. Third, and most importantly, the experiment validates the entire SSP-based integration pipeline—from gRPC state extraction through TCP-socket RL communication to C++ action execution—confirming that modern RL algorithms can be practically embedded into the helios-base decision loop.

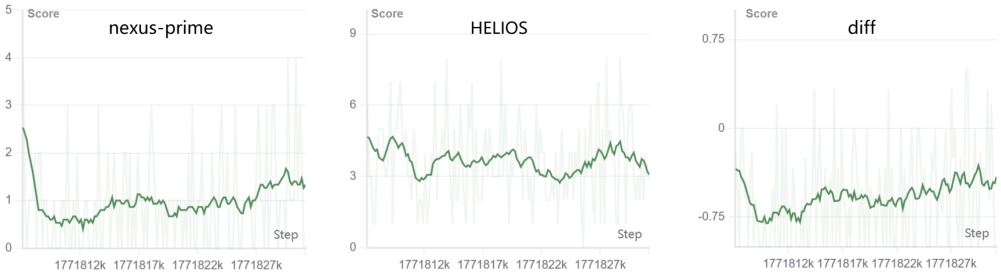

Fig. 2. Score trend of the RL-augmented team (rcss-rl-sb3) against unmodified helios over training episodes.

4.4 Lessons for the Distributed Framework

The rcss-rl-sb3 experiment yielded three critical lessons that directly shaped the design of rcss-rl-ray [12] (Section 5). First, the single-process, single-match training loop of SB3 is prohibitively slow: each match takes approximately 10 minutes in real time, and PPO requires hundreds of episodes to converge, resulting in training runs spanning days. This motivated the cloud-native parallel infrastructure (Sections 2–3) and the

Ray-based distributed framework. Second, the high-level action space, while stable, is inherently limited by the expressiveness of helios-base's own planners; the agent can only select *among* pre-engineered strategies, not discover novel ones. This informed the decision to adopt atomic actions in rcss-rl-ray [12], trading short-term stability for long-term learning potential. Third, the conditional execution model—where only kickable cycles trigger RL steps—revealed a critical training instability. Because ball possession is unevenly distributed across players, agents with rare ball contact accumulate rewards over thousands of idle cycles, causing per-player loss magnitudes to diverge by factors of 10^3 to 10^4. This pathological imbalance was the decisive motivation for the shift in rcss-rl-ray [12] (Section 5) to step *every* simulation cycle for *every* agent, regardless of possession. While stable massive training on the Ray framework is still pending, these critical lessons have been fully integrated into its architectural design.

In parallel, several works have explored end-to-end RL in RoboCup-like settings, where policies map observations directly to parameterized actions with minimal manual decomposition [14, 15]. This paradigm can enable the emergence of fine-grained coordinated behaviors that are difficult to encode with hand-crafted modules. Prior evidence, including FRA-UNIted and related studies, demonstrates feasibility in simplified scenarios or isolated tasks such as dribbling and goal-directed movement [14, 15].

5 Distributed RL Training Framework – rcss-rl-ray [12]

Sections 2 and 3 established a cloud-native simulation cluster and a declarative match orchestrator. Together, they provide an on-demand, API-driven stream of simulation environments. For training, we introduce `rcss-rl-ray` [12], a Python-based distributed training framework that bridges the gap between the Rust-based infrastructure and the Python-native RL ecosystem, completing the full pipeline from environment provisioning to gradient updates.

5.1 Design Philosophy: Atomic Actions over Hierarchical Skills

A critical design choice—and a deliberate departure from most existing RL attempts in the RCSS community—is our definition of the action space. The prevailing approach treats RL as a modular replacement within a traditional framework: neural networks are trained to make high-level decisions (e.g., "pass to player 7" or "dribble left"), while the underlying execution still relies on hand-crafted skill chains. This creates a hybrid system in which the RL agent's learning is fundamentally constrained by the expressiveness and biases of the pre-engineered skill library.

We take the opposite stance. Our action space is defined at the level of six fundamental atomic actions directly supported by the rcssserver protocol: `catch`, `dash`, `kick`, `move`, `tackle`, and `turn`. Each action type carries its own continuous parameters (e.g., power and direction for `dash` and `kick`, position coordinates for `move`). This results in a hybrid action space: a discrete choice among six action types, combined with a shared continuous parameter vector that is linearly mapped to the specific constraint range of the selected action. This design is motivated by a simple but important principle: if the

goal is to validate that modern RL can learn emergent soccer behaviors from scratch, the action space must be as unconstrained as the simulator permits. Any pre-imposed skill abstraction would conflate the RL algorithm's learning capability with the quality of human-designed heuristics.

5.2 RCSSEnv: A MultiAgentEnv as the Integration Hub

The core of the framework is `RCSSEnv`, a Gymnasium-compatible `MultiAgentEnv` that implements the standard `reset()`/`step()`/`close()` interface expected by Ray RLlib [9]. This design choice is deliberate: by conforming to the standard `MultiAgentEnv` contract, we gain immediate access to the entire RLlib algorithm library—PPO [11], IMPALA, and any future algorithms—without writing custom training loops.

The environment's lifecycle reveals how the previously described infrastructure components come together. On `reset()`, the environment requests a simulation room from the cluster via a REST allocator client, which triggers the Agones allocation pipeline (Section 2) and the Match Composer's automatic bootstrap (Section 3). The environment simultaneously hosts a gRPC server that receives protobuf-encoded `State` messages from the SSP sidecars running inside the allocated Pod. On each `step()`, the environment encodes the RL agent's actions into protobuf `PlayerAction` messages, sends them back through the gRPC channel, and waits for the next round of states. When an episode ends, the room is released and a fresh one is allocated, ensuring stateless recycling.

This architecture means that `RCSSEnv` itself contains no simulation logic; it is purely a communication and translation layer. The actual physics runs in the remote rcssserver Pod, while the SSP handles the low-level protocol bridging between the server's UDP interface and the gRPC channel. This separation is what enables true distributed training: the environments and the simulator Pods can run on entirely different machines, connected only via gRPC and REST.

6 Conclusion

This paper presented the NexusPrime team's end-to-end infrastructure for applying modern MARL to the RoboCup 2D Soccer Simulation. We introduced `rcss_cluster` [12], a Rust-based cloud-native system that transforms the legacy rcssserver into an elastically scalable training environment via Kubernetes and Agones; `match_composer` [12], a declarative sidecar that automates match orchestration with heterogeneous player policies; and `rcss-rl-ray` [12], a Ray-based distributed training framework operating on atomic actions under the CTDE paradigm. Our exploratory experiments with SB3—which constitute our currently submitted agent—validated the feasibility of embedding RL into the helios-base decision loop and revealed critical insights that guided the design of the distributed pipeline. While the current system has not yet surpassed established rule-based teams, we confidently anticipate that transitioning to massive distributed training via our now-completed Ray framework will yield substantial performance gains. The infrastructure provides a scalable, open foundation upon which increasingly capable MARL algorithms can be developed, and we believe it represents a meaningful step toward modernising the RCSS 2D research ecosystem.

References

1. Zare, N., Sayareh, A., Sadraii, A., Firouzkouhi, A., & Soares, A. (2024). Cross language soccer framework: An open source framework for the RoboCup 2D soccer simulation. *arXiv preprint arXiv:2406.05621*.
2. Noda, I., Matsubara, H., Hiraki, K., & Frank, I. (1998). Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence*, *12*(2-3), 233–250.
3. Akiyama, H., & Nakashima, T. (2014). Helios base: An open source package for the RoboCup soccer 2D simulation. In *RoboCup 2013: Robot World Cup XVII* (pp. 528–535). Springer.
4. Zare, N., Sayareh, A., Sarvmaili, M., Amini, O., Soares, A., & Matwin, S. (2022). CYRUS soccer simulation 2D team description paper 2021. In *RoboCup 2021 Symposium and Competitions*.
5. Cheng, Z., Zhang, F., Guang, B., & Wang, L. (2021). YuShan2021 team description paper for RoboCup2021. In *RoboCup 2021 Symposium and Competitions*.
6. Gabel, T., & Riedmiller, M. (2011). On progress in RoboCup: The simulation league showcase. In *RoboCup 2010: Robot Soccer World Cup XIV* (pp. 36–47). Springer.
7. Zare, N., Sayareh, A., Amini, O., Sarvmaili, M., Firouzkouhi, A., Matwin, S., & Soares, A. (2023). Pyrus Base: An open source Python framework for the RoboCup 2D soccer simulation. *arXiv preprint arXiv:2307.16875*.
8. Google Cloud Platform. (2024). *Agones* [Computer software]. GitHub. `https://github.com/googleforgames/agones`
9. Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., & Stoica, A. (2018). Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (pp. 561–577).
10. Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021). Stable-Baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, *22*(268), 1–8.
11. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.
12. jonaloo19. (2024). *nexus-prime* [Computer software]. GitHub. `https://github.com/jonaloo19/nexus-prime`
13. Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*. MIT Press.
14. Gabel, T., Eren, B., Eren, Y., Sommer, F., & Godehardt, E. (2023). FRA-UNIted — Team Description 2023. *RoboCup 2023 Team Description Paper*.
15. Hausknecht, M., & Stone, P. (2015). Deep reinforcement learning in parameterized action space. *arXiv preprint arXiv:1511.04143*. `https://doi.org/10.48550/arXiv.1511.04143`