OpenRA-RL: An Open Platform for AI Agents in Real-Time Strategy Games

Xiaochuang Yuan, Hui Xu, Yiyu Tian, Siqi Zhang, Ruiyue Wang, Kaiser Sun

April 2026

Abstract

Real-time strategy (RTS) games have driven landmark AI achievements, yet prior systems like AlphaStar and OpenAI Five relied on highly specialized architectures and training pipelines that do not generalize beyond their target domains. Meanwhile, LLM-based agents have emerged as a promising paradigm for general-purpose reasoning, and there is growing interest in applying them to RTS games. However, no existing RTS platform supports their distinct requirements: high-level action interfaces, asynchronous interaction, and tolerance for variable inference latencies—making current efforts difficult and ad-hoc. We introduce OpenRA-RL, an open-source platform enabling LLM-based agents to play the classic RTS game Red Alert. Our platform provides a Gymnasium-style Python API, integrates with the Model Context Protocol (MCP) to expose 50 game actions as tool calls, and implements an asynchronous architecture for agents operating slower than real-time. To validate the platform, we demonstrate a Qwen3 32B agent across five episodes, exposing an eight-dimensional reward vector that reveals specific strategic weaknesses: the agent achieves 0.58–0.80 scores on economic management but zero on combat execution, demonstrating that even frontier LLMs require substantial learning to master RTS games. By offering a unified testbed for evaluating LLM-based agents in a complex strategic environment, OpenRA-RL facilitates systematic research on long-horizon planning, strategic reasoning, and agent architectures.

1 Introduction

Real-time strategy (RTS) games have long served as a compelling testbed for artificial intelligence research, demanding capabilities that span rapid decision-making, long-horizon planning, resource management, and adversarial reasoning under partial observability. Landmark efforts in this domain include DeepMind's AlphaStar for StarCraft II Wang et al. [2021], OpenAI Five for Dota 2 Berner et al. [2019], and earlier work on StarCraft[Vinyals et al., 2017, Tian et al., 2017]. These systems achieved impressive performance, demonstrating that machine learning can master complex strategic environments.

However, these prior approaches relied on highly specialized architectures and training pipelines that do not generalize beyond their target games. AlphaStar, for instance, required a bespoke neural network architecture, imitation learning from human replays, and distributed reinforcement learning across thousands of TPUs Wang et al. [2021]. Such methods yield domain-specific experts but offer little reusable infrastructure for other tasks. Meanwhile, LLM-based agents have emerged as a promising paradigm for general-purpose reasoning, leveraging pretrained world knowledge, natural language reasoning, and high-level semantic actions to tackle diverse tasks, including web navigation, code generation, and tool use Yao et al. [2022].

The potential of LLM-based agents for RTS games has attracted growing research interest [Lee et al., 2018, Samvelyan et al., 2019, Han et al., 2021]. However, a critical infrastructure gap impedes progress: *no existing RTS platform is designed to support LLM-based agents.* Current platforms assume agents that operate at millisecond timescales with low-level action spaces, whereas LLM agents require high-level interfaces, asynchronous interaction patterns, and tolerance for variable inference latencies. This mismatch makes existing efforts to apply LLM agents to RTS games difficult and ad-hoc, hindering systematic research in this direction.

We address this gap with **OpenRA-RL**, an open-source platform that enables LLM agents to play the classic RTS game Red Alert. Our platform provides: (1) a Gymnasium-style Python API compatible with

standard RL tooling; (2) integration with the Model Context Protocol (MCP), exposing 50 game actions as tool calls for seamless LLM agent deployment; and (3) an asynchronous architecture that gracefully handles agents operating slower than real-time. By providing a unified testbed for evaluating LLM-based agents in a complex strategic environment, OpenRA-RL facilitates systematic research on long-horizon planning, strategic reasoning, and the comparative evaluation of agent architectures.

RTS games exercise capabilities with broad real-world relevance. The skills required—strategic planning under uncertainty, resource allocation, multi-objective optimization, and adaptive response to adversarial dynamics—closely mirror those demanded in domains such as supply chain management, autonomous logistics coordination, and multi-robot systems. Progress on RTS benchmarks may thus translate to consequential applications where agents must reason over extended time horizons in complex, dynamic environments.

To our knowledge, OpenRA-RL is the first platform specifically designed to support LLM-based agents in RTS games. Our contributions are twofold: (1) we provide the infrastructure necessary for systematic evaluation of LLM agents in a complex, real-time strategic environment; and (2) we enable direct comparison between LLM-based approaches and traditional methods, facilitating research into the strengths and limitations of general-purpose reasoning for strategic decision-making.

2 Architecture and Engineering Design

OpenRA-RL is built upon a modular architecture designed to accommodate diverse agent paradigms while maintaining real-time game execution. At its core, the platform consists of three interconnected layers: a modified OpenRA game engine written in C# that executes the game logic at approximately 25 ticks per second; a gRPC bridge that exposes game state and accepts agent commands; and a Python wrapper providing a standardized Gymnasium-style interface via FastAPI. This architecture decouples agent computation from game execution, allowing agents of varying speeds—from fast scripted bots to slower LLM-based reasoners—to interact with the same environment without disrupting game flow.

The platform supports three primary agent modalities. *Scripted bots* implement hand-crafted strategies through state machines, serving as baselines and demonstrations of the action space. *Reinforcement learning agents* consume the same gRPC bridge through a thin Gymnasium-compatible adapter, allowing standard RL training loops to drive the environment without any platform-specific scaffolding. Most notably, *LLM agents* interact through natural language reasoning, with the platform exposing 50 game actions as tools via the Model Context Protocol (MCP). This MCP integration enables seamless compatibility with frontier LLM systems, including Claude Desktop and other MCP-compatible clients. The platform supports both cloud-based LLM providers (OpenRouter, OpenAI) and local inference servers (Ollama, LM Studio), offering flexibility for both research experimentation and deployment.

2.1 Asynchronous Agent-Environment Decoupling

A fundamental challenge in applying LLM-based agents to real-time strategy games is the mismatch between game speed and agent inference time: the game engine ticks at approximately 25 Hz, while a single LLM decision may take over 2 seconds. To decouple these two timescales, the platform implements a dual-channel architecture using .NET `System.Threading.Channels` with bounded, non-blocking semantics.

Observation channel (game → agent). Each tick, the game engine serializes the current world state—economy, military statistics, unit positions, visible enemies, and a spatial tensor—into a `GameObservation` protobuf message and writes it to a `BoundedChannel<GameObservation>` configured with capacity 1 and a `DropOldest` full-mode policy. Because the channel holds at most one observation, a new write silently replaces any unread previous state. The agent therefore always receives the most recent game snapshot, regardless of how many ticks elapsed during its thinking time. For a fast agent (∼40 ms per step), no observations are dropped; for a slow agent (∼2 s per step), approximately 50 intermediate ticks are discarded, but the agent still acts on up-to-date information.

Action channel (agent → game). When the agent completes a decision, it may issue multiple commands in a single batch—e.g., train a unit, move a squad, repair a building, and set a rally point. These commands

are written to a `BoundedChannel<AgentAction>` with capacity 16, providing sufficient buffer for command batches without blocking the agent. The game engine drains all pending commands at the start of each tick and translates them into internal orders. The asymmetric capacity reflects the distinct semantics of each direction: a missed observation is harmless (a fresher one will follow), whereas a dropped command means the agent's intended action is silently lost, directly affecting game outcomes.

Non-blocking guarantee. Both channels use `DropOldest` semantics, and all writes are non-blocking (`TryWrite`). The game thread never waits for the agent: if no action has arrived by the time a tick executes, the engine proceeds with a no-op. This ensures that game progression is entirely independent of agent latency, a critical property for fair benchmarking across agents with vastly different inference costs.

2.2 Multi-Session Architecture

Training and evaluating game-playing agents requires running many games concurrently. The initial design (v1) spawned a separate .NET process for each game, incurring repeated JIT compilation and mod data loading per instance. At 64 concurrent sessions, this consumed approximately 40 GB of memory and required 5–15 seconds per reset cycle, creating a severe bottleneck for large-scale experiments.

The current design (v2) hosts up to 64 concurrent game sessions within a single .NET process. The key insight is that game static data—unit statistics, building attributes, tech trees, map rules—is immutable after initialization. This `ModData` is loaded once and shared across all sessions without locks, eliminating approximately 35 GB of redundant memory. Each session maintains its own isolated game world (`World`, `OrderManager`, `BotBridge`), ensuring that state from one game cannot leak into another.

Game ticks are processed by a fixed-size worker pool with N threads equal to the CPU core count, backed by a `BlockingCollection<WorkItem>` queue. This pool is deliberately separate from the .NET ThreadPool used by gRPC request handlers—a design choice motivated by a critical failure observed during testing: when game tick tasks saturated the shared ThreadPool, gRPC handlers starved and could not accept new requests, causing 0 of 16 sessions to complete. Separating the two thread pools eliminates this contention entirely.

Each session is protected by a per-session `SemaphoreSlim(1,1)` that serializes `World` mutations, preventing concurrent ticks from corrupting game state while still allowing different sessions to tick in parallel. When the worker queue is full, the system returns a gRPC `RESOURCE_EXHAUSTED` status code, providing natural backpressure rather than unbounded queue growth.

Table 1: Multi-session architecture performance comparison.

Metric	**Legacy (v1)**	**Multi-Session (v2)**	**Improvement**
Reset latency	5–15 s	256 ms	∼40×
RSS (64 sessions)	∼40 GB	∼6 GB	∼7×
JIT compilations	64×	1×	64×
Active threads	∼200	∼20	∼10×
Aggregate ticks/sec	∼8K	∼15K	∼2×

2.3 Communication and Lifecycle Management

Communication between the Python environment server and the C# game engine is mediated by gRPC, defined in a shared `rl_bridge.proto` schema that compiles to both Python stubs and C# server code. The service exposes two communication patterns. For the real-time game loop, a bidirectional streaming RPC (`GameSession`) is used: the `RLBridgeService` implementation spawns two concurrent async tasks—an `ObsSender` that reads from the observation channel and writes to the gRPC output stream, and an `ActionReceiver` that reads from the gRPC input stream and writes to the action channel. Both tasks share a `CancellationToken`; when either the game ends or the agent disconnects, the token fires and both tasks exit, triggering resource cleanup. For discrete operations—creating a session, destroying a session, querying game state, or advancing a fixed number of ticks—the service provides unary RPCs (`CreateSession`,

`DestroySession`, `GetState`, `FastAdvance`) that accept a `session_id` parameter to route requests to the correct game instance within the multi-session process.

The environment lifecycle is managed as an explicit state machine with eight states: IDLE, LAUNCHING, LOADING, CONNECTING, STREAMING, PLAYING, GAME_OVER, and CLEANUP. On `reset()`, the system transitions from IDLE through process startup and map loading to CONNECTING, where the Python-side `BridgeClient` retries `GetState()` up to 120 times until the gRPC channel is ready. Once connected, the `GameSession` stream is established (STREAMING) and the game enters the PLAYING state, during which the agent repeatedly observes and acts while a replay file is recorded. Upon receiving a terminal signal (win, lose, or draw), the system transitions through GAME_OVER to CLEANUP, closing the gRPC stream and releasing session resources before returning to IDLE for the next episode. Two explicit error transitions handle failure: a TIMEOUT path (120 retries exhausted during connection) and a CONN_LOST path (gRPC stream interruption during play), both of which trigger an immediate abort and cleanup cycle rather than leaving resources in an indeterminate state. A health check endpoint independently verifies both daemon process liveness and gRPC channel connectivity, automatically restarting the game daemon if either check fails.

2.4 Replay and Observability

OpenRA-RL records every game session as a deterministic `.orarep` replay file. Each replay captures the complete sequence of orders and random seed, enabling exact tick-by-tick reproduction of the game via a `ReplayConnection` packet reader. Replay files also embed the Docker image version of the engine that produced them, ensuring playback fidelity even after engine upgrades. Viewing is handled through a browser-based noVNC interface running inside the Docker container (`openra-rl replay watch`), requiring no local game installation or graphics drivers—making replay analysis accessible from headless cloud training instances.

The platform further supports a local *arena* mode for side-by-side comparison of two replays, allowing researchers to visually contrast agent strategies and record qualitative preferences for use in reward modeling. Replays also integrate with the benchmarking pipeline: when submitting results to the OpenRA-Bench leaderboard, the corresponding `.orarep` file can be attached alongside JSON metrics, enabling independent verification of reported results.

2.5 Integration with OpenEnv

OpenRA-RL is built as a first-class environment for **OpenEnv** [Meta PyTorch Team, 2025], the emerging PyTorch-native standard for reinforcement learning environment creation and interoperability. OpenEnv defines a minimal, typed environment contract—reset, step, and structured observation/action spaces—together with a distribution and discovery layer on the Hugging Face Hub, so that an environment author can publish once and any compatible trainer can consume the result without environment-specific glue code. By conforming to this contract, OpenRA-RL inherits the OpenEnv ecosystem's guarantees of modularity, reproducibility, and interoperability at the environment layer.

Ecosystem compatibility. Because OpenRA-RL is an OpenEnv environment, it is immediately consumable by the PyTorch-native post-training stack—TRL [von Werra et al., 2020], torchforge, and Unsloth—without any environment-specific adapters on the trainer side. A researcher who wants to run GRPO, PPO, or any other policy-optimization algorithm against OpenRA-RL can do so by pointing their existing training script at the environment's OpenEnv identifier; no OpenRA-RL-specific scaffolding is required. This is a concrete demonstration of OpenEnv's stated goal of decoupling environment authorship from trainer implementation [Meta PyTorch Team, 2025].

Hub distribution. OpenRA-RL is published to the OpenEnv Hub on Hugging Face[1], where it can be pulled and instantiated in a few lines of Python. The Hub distribution packages the game server Docker image, the Python environment wrapper, and a versioned metadata manifest, so that an agent developed

[1] `https://huggingface.co/openenv`

against one published version of OpenRA-RL reproduces exactly when run by a third party against the same version. This reproducibility is a prerequisite for the kind of cross-paper comparisons that RTS-agent research has historically struggled with, and it is enabled entirely by the OpenEnv distribution layer rather than by any bespoke infrastructure on our side.

What OpenRA-RL contributes back. Most existing OpenEnv environments target narrow, short-horizon tasks: code execution, single-turn tool use, and small-scale games. OpenRA-RL extends OpenEnv's applicability to a *long-horizon, adversarial, real-time* regime with combinatorial action spaces and variable inference latencies. The asynchronous decoupling of Section 2.1, the multi-session architecture of Section 2.2, and the deterministic replay format of Section 2.4 are all general design patterns that other authors of complex OpenEnv environments may find reusable.

3 Demonstration

We have developed a Python normal AI bot that can beat OpenRA easy AI bot `https://github.com/huixu11/openra-rl-challenge`. The replay can be watched at `https://youtu.be/YaO-KyHiXfo`.

To exercise the platform end-to-end through the OpenEnv interface, we deploy Qwen3 32B served locally via Ollama as an LLM agent playing the Allied faction against the built-in Beginner AI on a 128×128 map. The agent receives structured game-state observations as tool responses and issues actions through the MCP tools described in Section 2, including a pre-game planning phase and a post-game reflection step whose extracted lessons are injected into subsequent episodes' system prompts. We run five episodes under two timing regimes: **Games 1–2 use a 30-minute time limit, while Games 3–5 use a 5-minute limit**, demonstrating that the platform supports variable episode lengths within a single experiment.

3.1 End-of-episode scorecard

Table 2 summarizes the five episodes. All five games ended without combat engagement (zero kills, zero casualties): the agent successfully establishes a base economy but does not translate that economy into offensive force before the time limit expires.

Game	Duration	Ticks	Assets	Bldgs	Army	Explored	Calls
Game 1†	30:23	1621	\$6,600	5	\$2,920	3.7%	62
Game 2†	30:15	1477	\$4,000	3	\$2,340	2.7%	81
Game 3	5:01	540	\$2,800	3	\$640	2.7%	18
Game 4	5:19	509	\$2,300	2	\$540	2.2%	19
Game 5	5:17	621	\$2,800	3	\$740	2.7%	21

Table 2: End-of-episode scorecard for five Qwen3 32B episodes against the Beginner AI. †Games 1–2 used a 30-minute time limit; Games 3–5 used a 5-minute limit. All five episodes terminated at the time limit with neither player eliminated and no combat engagement.

Figure 1: System architecture of OpenRA-RL. The platform consists of three interconnected layers: LLM agents interact through the Model Context Protocol (MCP) server, which communicates with a Python backend via FastAPI, and the Python backend connects to the C# game engine through a gRPC bridge. This modular design decouples agent computation from game execution, supporting diverse agent paradigms from scripted bots to LLM-based reasoners.

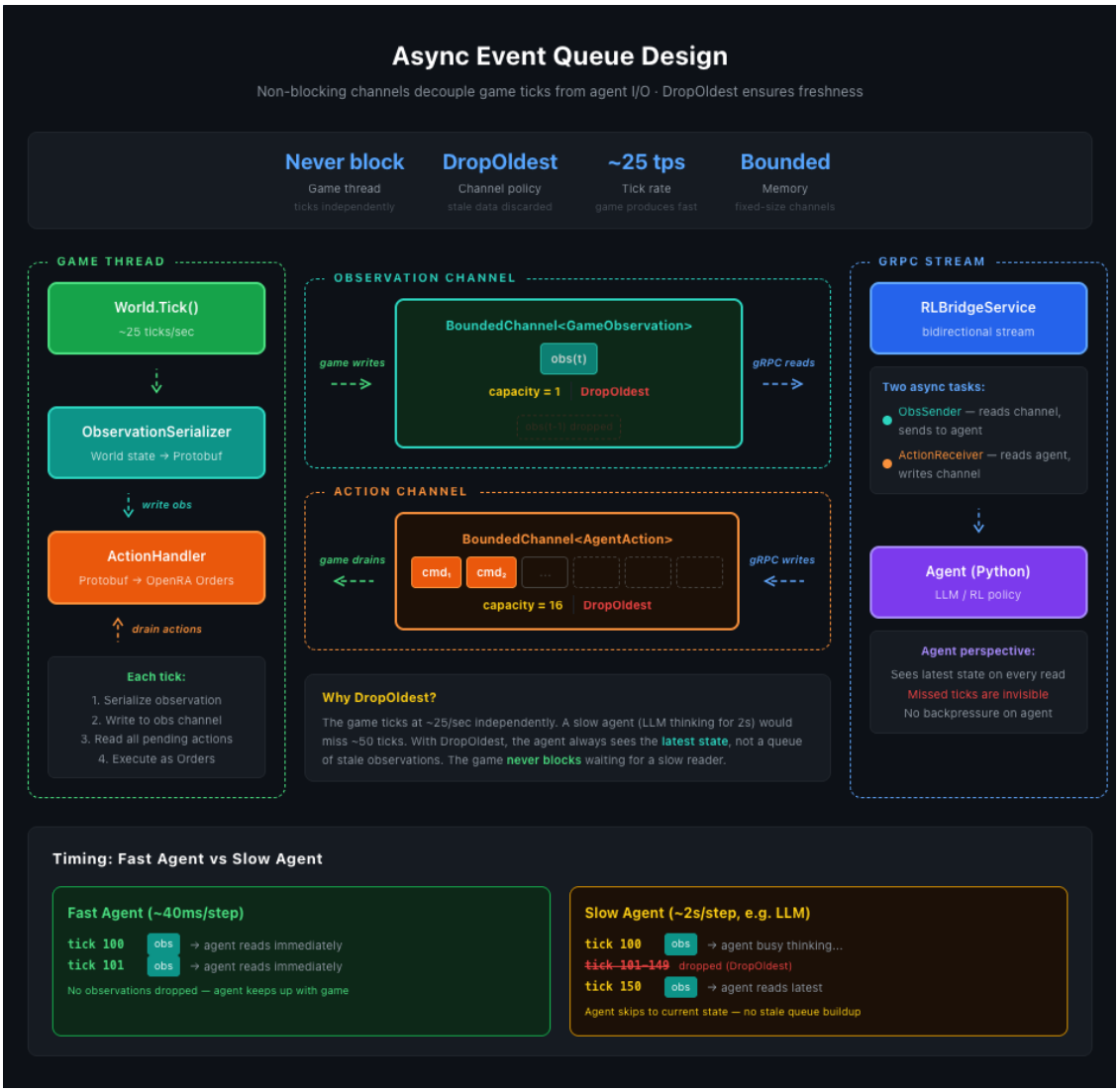

Figure 2: Asynchronous event queue design with bounded, non-blocking channels. The observation channel (capacity=1, DropOldest) ensures agents always receive the most recent game state, while the action channel (capacity=16) buffers agent commands. This asymmetric design decouples game ticks from agent inference time, allowing both fast scripted bots (∼40ms) and slow LLM agents (∼2s) to interact with the same 25Hz game engine.

Figure 3: Multi-session worker pool architecture (v2). Up to 64 concurrent game sessions run within a single .NET process, sharing immutable `ModData` to eliminate redundant memory usage. Game ticks are processed by a dedicated worker thread pool separate from gRPC request handlers, preventing resource contention. Each session is protected by a per-session semaphore for safe parallel execution.

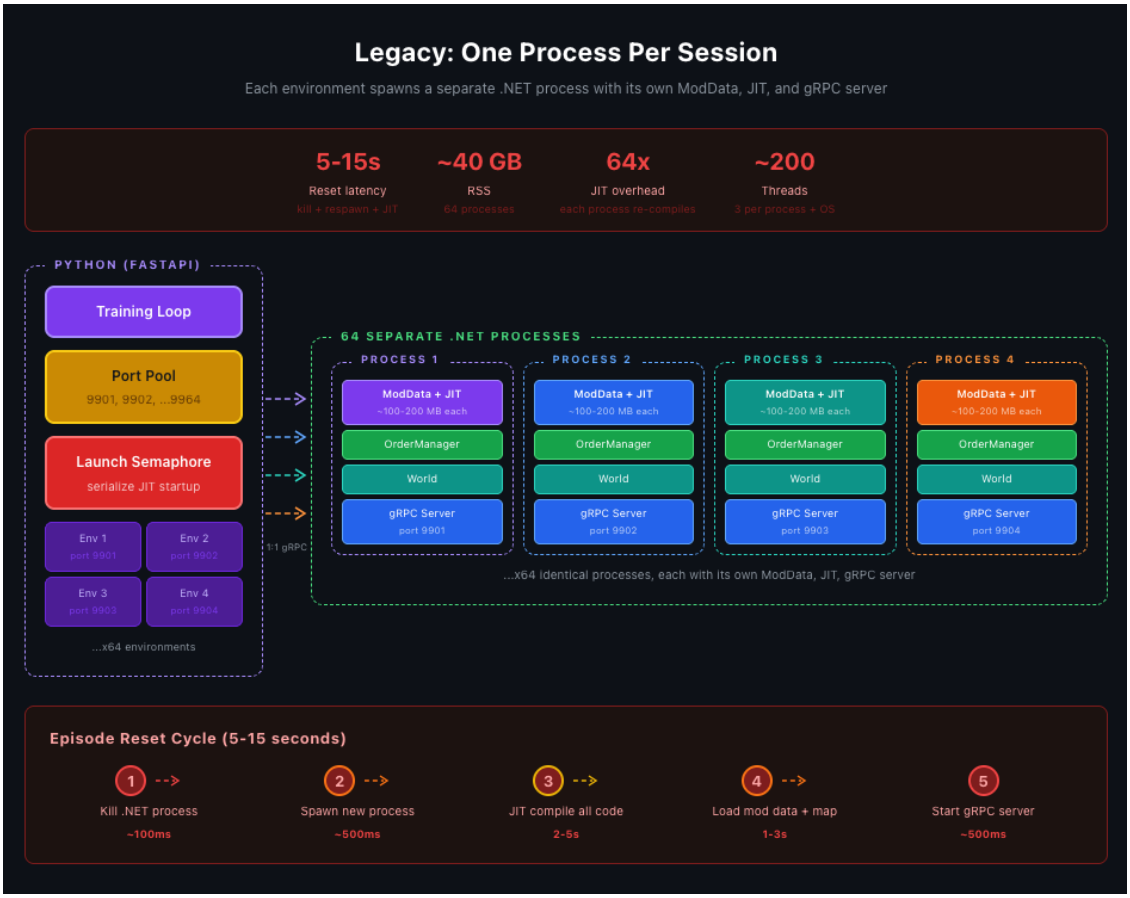

Figure 4: Legacy architecture (v1) comparison. The original design spawned 64 separate .NET processes, each with its own JIT compilation and `ModData` loading. This consumed ∼40GB RAM and required 5–15s reset latency, creating a severe bottleneck for large-scale experiments. The v2 architecture (Figure 3) provides ∼40× speedup and ∼7× memory reduction.

Figure 5: Game state machine lifecycle. The environment transitions through eight states from IDLE to CLEANUP. The PLAYING state maintains a bidirectional gRPC stream for real-time observation and action exchange. Two error paths (TIMEOUT and CONN_LOST) ensure clean resource cleanup on failure. The system also supports deterministic replay playback with the same state management.

3.2 Multi-dimensional reward profile

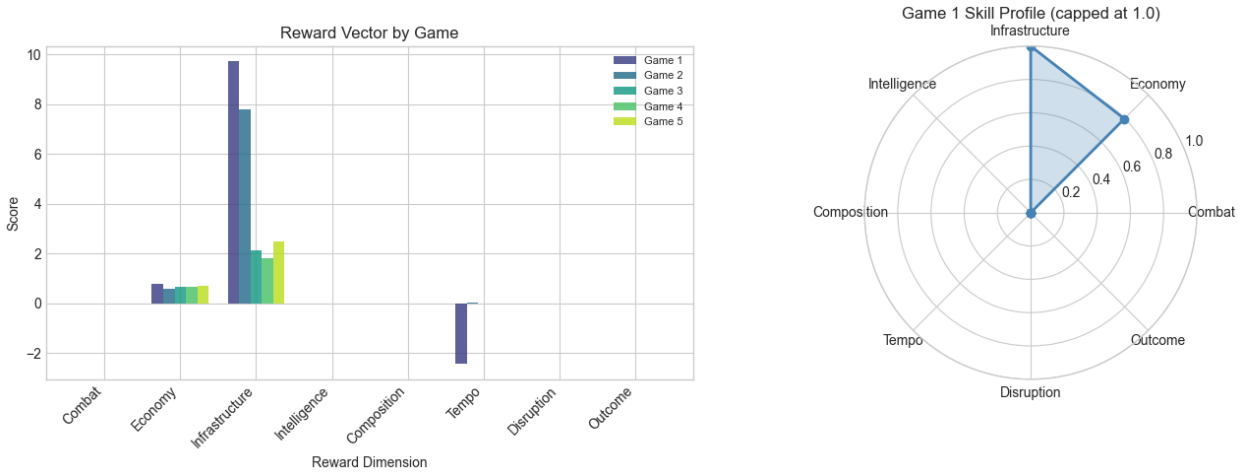

Figure 6: Eight-dimensional reward vector across the five episodes. Left: per-dimension scores for all five games. Right: radar plot of Game 1's skill profile (capped at 1.0). The agent registers non-trivial scores on *economy*, *infrastructure*, and *tempo*, but zero on *combat* and *disruption*, exactly the kind of multi-objective asymmetry that a single win/loss scalar would collapse into a single data point.

Figure 6 shows the multi-dimensional reward vector OpenRA-RL computes for each episode. Decomposing performance this way exposes an asymmetric agent: strong on economic and infrastructural sub-skills, absent on combat and disruption. This is the practical motivation for exposing reward as a vector rather than a scalar: it identifies which strategic dimensions need improvement rather than only that the game was lost.

3.3 Build-order timelines

Figure 7 shows what the agent built and when in each episode. Across all five games, the agent constructs a Power Plant, Barracks, and (in some episodes) a Refinery; unit production, however, lags significantly, and no offensive units reach the field before the time limit expires in any episode.

Figure 7: Build-order timelines for the five episodes, reconstructed from each game's deterministic `.orarep` replay file (see Section 2.4). The x-axes are *not* shared: Games 1–2 span ∼1500 ticks (30 minutes), Games 3–5 span ∼600 ticks (5 minutes).

3.4 Tool-call distribution

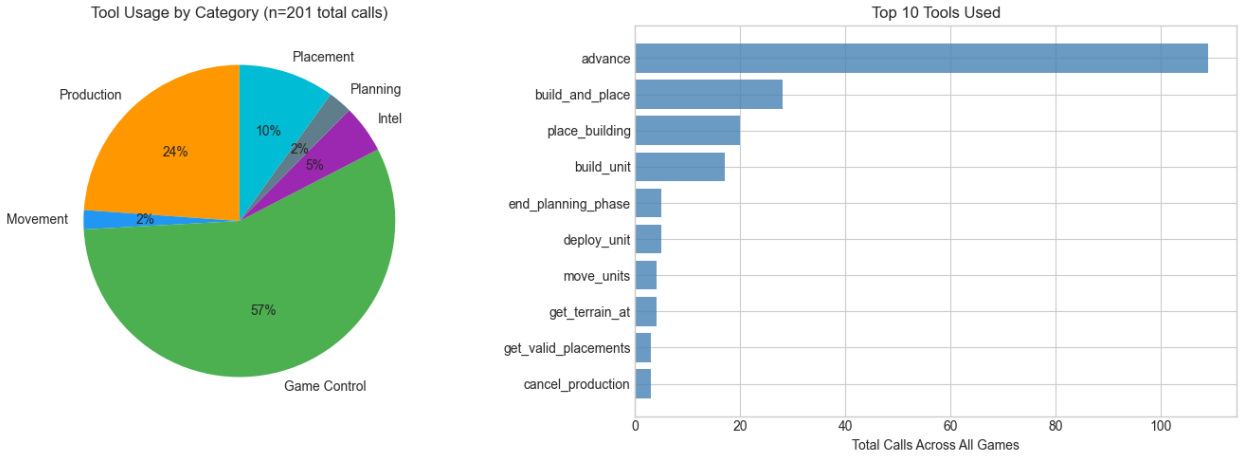

Figure 8: Distribution of MCP tool calls across all five episodes (n=201 total calls). Left: by category. Right: top ten individual tools. The dominance of `advance` (∼57% of all calls) reflects the asynchronous architecture from Section 2.1: a slow LLM reasoner explicitly compresses idle game time by skipping ticks, decoupling its ∼2-second decision latency from the engine's 25 Hz tick rate.

Figure 8 shows how the agent allocates its 201 total tool calls across the available actions. The most frequent call by a wide margin is `advance`, which time-skips the game forward by a specified number of ticks, this is the primary mechanism by which an LLM agent operates at multi-second decision latencies can interact with a 25 Hz game engine without leaving the game idle.

3.5 Macro-management trends

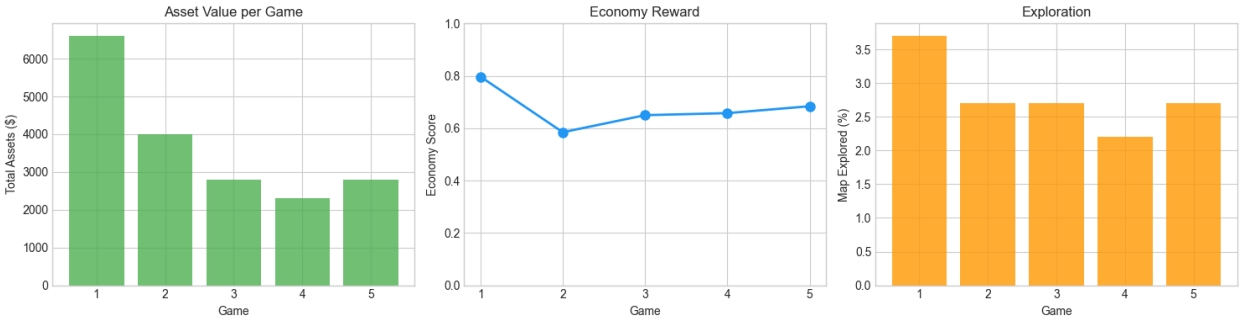

Figure 9: Macro-management indicators across the five episodes. Left: total asset value at end-of-game. Centre: per-tick-normalized economy reward score. Right: percentage of map explored. The drop in raw asset value from Game 1 ($6,600) to Games 3–5 (∼$2,500) is an artifact of the shortened time limit, *not* agent regression; the per-tick-normalized economy score remains within 0.58–0.80 across both regimes.

Figure 9 tracks economic indicators across episodes. Because Games 1–2 ran for 30 minutes and Games 3–5 for only 5 minutes, the raw end-of-game asset totals are not directly comparable across the two regimes. The per-tick-normalized *economy reward* provides a fairer comparison and stays in the 0.58–0.80 range for all five episodes, indicating that the agent's macro-management quality is consistent regardless of horizon length, even though absolute outputs scale with available time.

3.6 Sample agent reasoning trace

To make the agent's decision loop concrete, Listing 1 reproduces the first ten turns of Game 5 verbatim from the structured run log. The trace illustrates the three phases of an episode: an initial intelligence and planning phase (`get_faction_briefing`, `end_planning_phase`), construction of the early-game economy (`deploy_unit`, `build_and_place`), and explicit time-skipping via `advance` to bridge the gap between the agent's multi-second decision latency and the 25 Hz game tick rate.

Listing 1: First 10 turns of Game 5 (Qwen3 32B), as recorded in the run log. Tool calls are issued by the agent; arrows show the game-state delta returned by the gRPC bridge.

```
Turn  1 | get_faction_briefing()
Turn  2 | end_planning_phase(strategy="Deploy MCV, build Power
       Plant (300 ticks), then Barracks/War Factory (500/750),
       train E1 + 1tnk, attack AI base at (64,64).")
       -> tick=2,   cash=$5000, units=1, buildings=0
Turn  3 | deploy_unit(unit_id=120)
       -> tick=4,   cash=$5000, units=1, buildings=0
Turn  4 | build_and_place(building_type="apwr")
Turn  5 | advance(ticks=100)
       -> tick=55,  cash=$5000, units=0, buildings=1
Turn  6 | build_and_place(building_type="apwr")
Turn  7 | build_and_place(building_type="powr")
       -> tick=57,  cash=$5000, units=0, buildings=1
Turn  8 | advance(ticks=60)
       -> tick=108, cash=$4923, units=0, buildings=1
Turn  9 | advance(ticks=130)
       -> tick=159, cash=$4838, units=0, buildings=1
Turn 10 | advance(ticks=80)
       -> tick=210, cash=$4753, units=0, buildings=1
```

3.7 API usage

Listing 2 shows the minimal Python code needed to instantiate an OpenRA-RL environment, reset it into a scenario, step it with a structured action, and close it to finalize the replay. The surface follows the standard OpenEnv `reset`/`step`/`close` contract, so the same handle is consumed by scripted-bot baselines, the LLM-agent demonstration above, and downstream RL trainers like TRL [von Werra et al., 2020].

Listing 2: Minimal end-to-end example of driving an OpenRA-RL environment from Python through the standard OpenEnv contract.

```
from openra_env.config import load_config
from openra_env.server.openra_environment import OpenRAEnvironment
from openra_env.models import ActionType, CommandModel, OpenRAAction

# 1. Configure and instantiate the environment.
config = load_config(game={
    "grpc_port": 8000,
    "map_name": "tank-duel-basic",
    "headless": True,
})
env = OpenRAEnvironment(config=config)

# 2. Reset into a scenario; obs is a structured observation
#    (economy, military, unit/building lists, 9-channel spatial map).
obs = env.reset(seed=0)

# 3. Issue a structured action. OpenRAAction wraps one or more
#    CommandModel entries drawn from 21 ActionType values
#    (MOVE, ATTACK, BUILD, TRAIN, DEPLOY, ...).
action = OpenRAAction(commands=[
    CommandModel(action=ActionType.BUILD, item_type="powr"),
])
obs = env.step(action)
```

```
# 4. Close the environment; this finalizes the .orarep replay file.
env.close()
```

The same environment object is exposed to LLM agents through the 50 MCP tools described in Section 2, which wrap the `ActionType` enum in higher-level tool calls such as `build_and_place` and `advance`. Scripted bots and RL trainers call `env.step` directly with `OpenRAAction` objects; LLM agents issue MCP tool calls that the bridge translates into the same `OpenRAAction` shape before forwarding to `env.step`. Both paths share the observation and reward pipeline, which is what makes the same environment usable by both an in-the-loop LLM and a TRL-based GRPO trainer without rewiring.

3.8 What this demonstrates

The five-episode baseline does not produce a winning agent, and deliberately so: its purpose is to exercise the platform across every major design surface described in Section 2 and to characterize OpenRA-RL as a research testbed rather than to announce state-of-the-art results. Read through that lens, the demonstration validates five properties of the environment that are relevant both to the OpenEnv ecosystem and to downstream RTS research.

The environment is strategically deep. Five episodes of a frontier LLM (Qwen3 32B) playing the simplest built-in opponent produced zero combat engagement, five draws, and no units ever reaching the enemy base. This is not a failure of the platform—it is evidence that *Red Alert*, even against a Beginner AI, requires real strategic reasoning (resource management, build ordering, army composition, attack timing) that is not yet trivially captured by prompt-driven LLM agents. The gap between current LLM agents and even a tutorial-level opponent is exactly the kind of headroom a long-horizon RL research testbed needs.

The multi-dimensional reward vector localizes weakness. A scalar win/loss metric would collapse all five episodes into a single data point ("draw"). The eight-dimensional reward vector exposed through OpenRA-RL's OpenEnv interface instead reveals a precise failure mode: the agent achieves 0.58–0.80 on *economy* and *infrastructure* but zero on *combat* and *disruption* (Figure 6). This decomposition gives downstream researchers a concrete target for reward shaping, curriculum design, and credit assignment without needing to reverse-engineer what the agent is doing wrong.

The asynchronous architecture is load-bearing, not decorative. 57% of the agent's 201 tool calls are `advance` calls that explicitly time-skip the game engine (Figure 8). Without the observation-drop / action-bounded channel design of Section 2.1, an LLM agent operating at ∼2-second decision latencies could not meaningfully play a 25 Hz real-time strategy game at all. The asynchronous decoupling is what makes an LLM agent a first-class citizen of the environment rather than a post-hoc workaround, and it generalizes to any slow reasoner an OpenEnv user might plug in.

Cross-episode reflection works, but is not enough. After each game, the agent generates a reflection over the episode trace and extracts a small set of lessons, which are injected into the next episode's system prompt. Episode 2's reflection identifies a specific build-order mistake (*"prioritizing a War Factory first (t=15) over the Power Plant"*) and produces a concrete corrective rule (*"Build Power Plant first (t=50–80) to accelerate resource generation"*); by Episode 4, the agent's pre-game plan explicitly opens with a Power Plant. In-context learning via prompt injection is sufficient to correct build order but does not close the combat gap—suggesting OpenRA-RL is exactly the kind of environment in which the step from in-context adaptation to weight-updating RL training should measurably matter.

The environment is ready for the OpenEnv ecosystem. Because the demonstration exercises only the standard OpenEnv environment interface, everything shown in this section—planning, acting, rewarding, reflecting, replaying—is immediately available to any OpenEnv-compatible trainer. A researcher who wants to run GRPO or PPO on OpenRA-RL through TRL, torchforge, or Unsloth can do so by pointing their

existing training script at the environment on the OpenEnv Hub, with no environment-side changes required. The baseline in this section thus doubles as a ready-to-use starting point for future training runs.

4 Conclusion

We have presented OpenRA-RL, the first platform specifically designed to support LLM-based agents in real-time strategy games. Unlike prior RTS AI systems that relied on specialized architectures limited to a single game, OpenRA-RL provides general-purpose infrastructure built on the OpenEnv standard, enabling systematic research on long-horizon planning and strategic reasoning with diverse agent paradigms.

The platform addresses a critical infrastructure gap through three key contributions. First, a modular three-layer architecture decouples agent computation from game execution via a gRPC bridge, Gymnasium-style Python API, and Model Context Protocol integration that exposes 50 game actions as tool calls compatible with frontier LLM systems. Second, an asynchronous dual-channel design gracefully handles agents operating slower than real-time, with bounded observation and action buffers that prevent game progression from blocking on agent latency. This is essential for evaluating LLM agents with multi-second inference times. Third, a multi-session architecture hosts 64 concurrent game sessions in a single process, reducing reset latency from 5–15 seconds to 256ms and memory consumption from approximately 40GB to approximately 6GB, removing a critical bottleneck for large-scale training and evaluation.

Our demonstration with a Qwen3 32B agent validates both the platform's technical capabilities and its utility as a research testbed. The agent's performance (achieving 0.58–0.80 scores on economic management but zero on combat execution across five episodes) reveals that even frontier LLMs require substantial learning to master RTS games, confirming strategic depth that is neither trivially solved by prompt engineering nor reducible to short-horizon reasoning. The eight-dimensional reward vector exposed through the OpenEnv interface localizes specific weaknesses that a scalar win/loss metric would collapse, enabling targeted research on credit assignment and curriculum design.

As a first-class OpenEnv environment distributed via the Hugging Face Hub, OpenRA-RL is immediately consumable by PyTorch-native training frameworks including TRL, torchforge, and Unsloth without environment-specific adapters. This standards-based approach ensures that advances in agent architectures, training algorithms, and evaluation methodologies developed using OpenRA-RL generalize across the OpenEnv ecosystem. We release the platform as open-source software and invite the research community to build upon this foundation for advancing strategic reasoning in AI agents.

References

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

Lei Han, Jiechao Xiong, Peng Sun, Xinghai Sun, Meng Fang, Qingwei Guo, Qiaobo Chen, Tengfei Shi, Hongsheng Yu, Xipeng Wu, and Zhengyou Zhang. TStarBot-X: An Open-Sourced and Comprehensive Study for Efficient League Training in StarCraft II Full Game, April 2021. URL `http://arxiv.org/abs/2011.13729`. arXiv:2011.13729 [cs].

Dennis Lee, Haoran Tang, Jeffrey Zhang, Huazhe Xu, Trevor Darrell, and Pieter Abbeel. Modular Architecture for StarCraft II with Deep Reinforcement Learning. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 14(1):187–193, September 2018. ISSN 2334-0924, 2326-909X. doi: 10.1609/aiide.v14i1.13033. URL `https://ojs.aaai.org/index.php/AIIDE/article/view/13033`.

Meta PyTorch Team. OpenEnv: A Standard for RL Environment Creation and Interoperability, 2025. URL `https://github.com/meta-pytorch/OpenEnv`.

Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philip H. S. Torr, Jakob Foerster, and Shimon Whiteson. The StarCraft

Multi-Agent Challenge, December 2019. URL http://arxiv.org/abs/1902.04043. arXiv:1902.04043 [cs].

Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C. Lawrence Zitnick. ELF: An Extensive, Lightweight and Flexible Research Platform for Real-time Strategy Games, November 2017. URL http://arxiv.org/abs/1707.01067. arXiv:1707.01067 [cs].

Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. StarCraft II: A New Challenge for Reinforcement Learning, August 2017. URL http://arxiv.org/abs/1708.04782. arXiv:1708.04782 [cs].

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. Trl: Transformer reinforcement learning. https://github.com/huggingface/trl, 2020.

Xiangjun Wang, Junxiao Song, Penghui Qi, Peng Peng, Zhenkun Tang, Wei Zhang, Weimin Li, Xiongjun Pi, Jujie He, Chao Gao, Haitao Long, and Quan Yuan. SCC: an efficient deep reinforcement learning agent mastering the game of StarCraft II. In *Proceedings of the 38th International Conference on Machine Learning*, pages 10905–10915. PMLR, July 2021. URL https://proceedings.mlr.press/v139/wang21v.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.