Departments of Math and Computer Science

Final Report for
Applied Computing Research Labs

Building a Training Arena for Kubernetes Machine Learning Models

May 8, 2026

Team Members

Bob Gao
Omar Jimenez
Rui Zhang
Diya Gangwar (Fall Team Lead)
Cate Lewison (Spring Team Lead)

Advisor

Prof. Erin Talvitie

Liaison

Dr. David Morrison '08

Contents

Chapter 1

Introduction and Project Context

1.1 Sponsor Background

Applied Computing Research Labs (ACRL) is a tech consulting company dedicated to improving the reliability, monitoring, and cost-efficiency of client infrastructure. They specialize in scale as a service, focusing on resource optimization and advanced task scheduling for complex distributed systems.

This project advances ACRL's mission by building the core infrastructure needed to train machine learning models for Kubernetes management. By opening this up to both internal engineers and external users, this project provides a practical solution for reliability consulting and expands the capabilities and value of ACRL's existing tools, such as SimKube.

1.2 Problem Overview

Kubernetes is the open-source industry standard for container orchestration. It automates the deployment, scaling, and management of cloud-native applications across clusters of servers. While powerful, Kubernetes is complex and reliable operation is difficult. Cloud applications rely heav-

ily on Kubernetes, but small manual misconfigurations, such as incorrect CPU/memory requests or wrong replica counts, can lead to severe outages or significant resource waste. Testing fixes and evaluating configuration changes present a paradox for engineers: testing directly in live production carries a high risk of catastrophic downtime and lost revenue, whereas spinning up duplicate test clusters is resource-intensive and prohibitively expensive.

1.3 Project Motivation

Managing Kubernetes infrastructure introduces significant complexity and operational burden. To alleviate this, there is an opportunity to use machine learning agents that can independently diagnose issues and manage distributed infrastructure.

However, training these agents effectively requires datasets and extensive trial-and-error experimentation. Because production systems cannot safely be used for training and testing, companies currently lack a viable environment to evaluate how these models handle failure recovery before they are deployed. There is a need for a risk-free and low-cost environment where AI agents can learn to diagnose and fix clusters without significant consequences.

1.4 Project Vision

To address these challenges, we built the Kubernetes ML Model Training Arena (Sim-Arena), a risk-free, open-source simulated framework where ML agents can observe cluster states, take multiple actions to adjust configurations, and receive positive feedback when pods become healthy. Sim-Arena is built as a wrapper around ACRL's SimKube, a simulator that mimics the Kubernetes control plane to replay recorded traces of failing workloads inside a test cluster. To populate this environment, our team created our own custom traces to simulate specific and reproducible failure states for the models to resolve. Unlike general machine learning arenas, Sim-Arena is tailored to distributed systems, requiring strict guardrails in its action space to prevent over-allocation of resources.

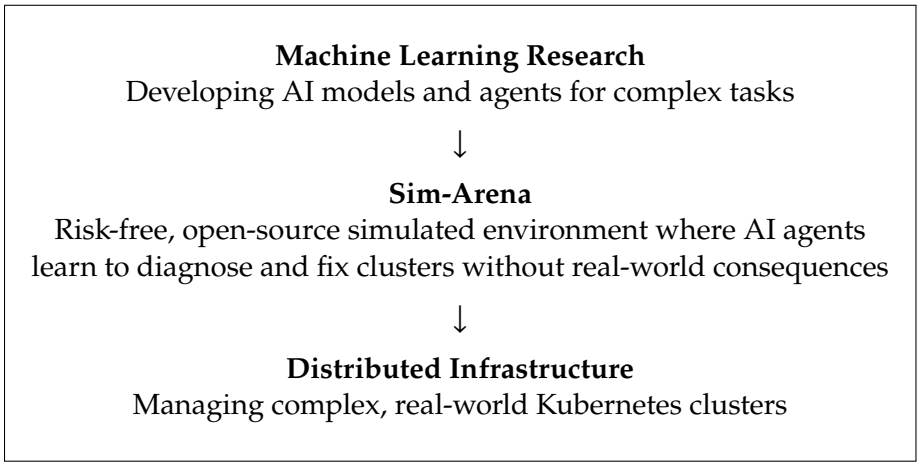

Figure 1.1 Project Vision: Sim-Arena acts as the missing link, bridging the gap between theoretical machine learning research and practical distributed infrastructure management.

1.5 Report Roadmap

This report details the design and implementation of the Sim-Arena training environment. It covers the core architecture, the minimum viable product (MVP) design, reinforcement learning and LLM integrations, and the distributed execution infrastructure built for scaling. The report concludes with a handoff guide and suggestions for future work.

Chapter 2

Original Goals, Deliverables, and Project Evolution

2.1 Initial Statement of Work & Technical Goals

The original clinic project problem statement tasked our team with using ACRL's SimKube to construct a training environment tailored for Kubernetes-specific machine learning models. We committed to delivering an open-source MVP (minimum viable product) implementation of a training arena, a demonstration showing model-agent interaction, evaluation protocols, and a reproducible testing framework. The core technical goals were to build a simulated training environment completely isolated from production risk, provide an interface where ML agents could query the cluster and apply defined configuration actions, and develop evaluation methods (such as binary or shaped rewards) to assess agent performance.

2.2 Project Evolution

Originally, the project focused heavily on traditional reinforcement learning. The project evolved to support an entire agent ecosystem, incorporating both RL pathways and LLM tracks. We significantly expanded the project by adding LLM benchmarking capabilities via the Model Context Protocol (MCP), allowing models like Gemini and Claude to query the cluster inter-

actively. We also added a standard Gymnasium API, which allows communication between RL algorithms and our environment. Additionally, we implemented distributed training infrastructure utilizing AWS EC2 workers and S3 to achieve parallelization and overcome the slow wall-clock time of single-machine training loops. In the process, we restricted the initial MVP action space to just seven discrete actions to ensure safety and prevent runaway scaling loops. As inputs for our simulations, we constructed custom traces with specific misconfiguration scenarios.

2.3 Final Deliverables & Outcomes

The final software suite includes the Sim-Arena environment wrapper, the S3-based job protocol for distributed AWS EC2 training, the Gymnasium API integration, and the MCP LLM benchmarking tools. We produced comprehensive documentation, including Developer Readmes, AWS Distributed Training guides, Gymnasium integration guides, and LLM Benchmarking instructions. The team successfully delivered benchmark results comparing standard RL baselines (like DQN) with prompt-based LLMs. We successfully achieved the end-to-end MVP architecture with an observe-action-reward loop with SimKube integration. Our code can be found on the Sim-Arena Github.

Project Aspect	**Initial Plan**	**Final Outcome**
Environment	Build an isolated MVP training arena.	Delivered functional SimArena wrapper running on a test cluster.
Agent Types	Focus on traditional Reinforcement Learning (RL).	Supported both RL and Language Models (Gemini, Claude) via MCP.
Infrastructure	Standard single-machine training loops.	Scaled to distributed training using AWS EC2 workers and S3.
Actions & Inputs	Allow agents to apply general configuration actions.	Safely restricted to 7 discrete actions using custom trace inputs.
Deliverables	A basic demo, evaluation methods, and testing framework.	Standard Gymnasium API, benchmark comparisons, and guides for testing.

Chapter 3

Background and Related Work

3.1 Kubernetes as a Distributed Systems Platform

Kubernetes (K8s) operates by managing applications across clusters of servers, automating the deployment and scaling of containers by detecting load bottlenecks and dynamically scheduling computation resources to meet user demand. Operating a cluster requires carefully balancing resource requests and limits. Incorrect configurations can lead to pending pods, out-of-memory (OOM) errors, and application lag.

3.2 SimKube

SimKube is an open-source tool developed by ACRL designed for replaying production data and troubleshooting incidents in an isolated environment. It mimics the Kubernetes control plane, allowing users to replay recorded traces of failing workloads inside a test cluster without risking live environments. Because it creates realistic, resource-efficient simulations of complex management scenarios, SimKube serves as the perfect underlying engine for our agent training arena.

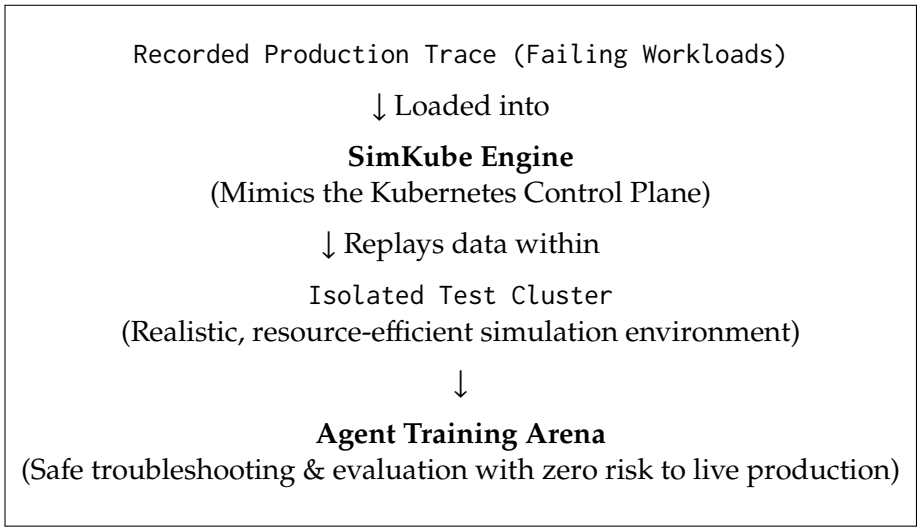

Figure 3.1 Overview of the SimKube engine. SimKube replays recorded production data and troubleshooting incidents within an isolated test cluster, creating a realistic, resource-efficient foundation for the agent training arena.

3.3 Machine Learning Agents for Systems Tasks

Machine learning allows systems to recognize patterns and make decisions from data without needing explicit programming. One specific approach is reinforcement learning (RL), where a software "agent" learns to make sequential decisions through trial and error. The agent interacts with an environment, choosing actions and receiving numerical rewards based on the outcome, ultimately learning a policy, the strategy or rulebook that an agent uses to make decisions, that optimizes system performance over time. Large Language Models (LLMs), which are advanced neural networks trained on massive amounts of text, can also work as agents. Instead of learning via repeated simulation, LLM agents can read unstructured context like system logs and use their pre-trained reasoning to diagnose and correct configuration issues directly.

Our environment supports two primary types of agents. Reinforcement learning agents interact with the environment iteratively over many episodes, observing the state and outputting discrete actions. Alternatively, LLM-based agents utilize zero-shot or few-shot reasoning. Through the MCP server, LLMs can observe the cluster directly, extracting pod logs or de-

scribing deployments before outputting a parsed JSON action schema. Diagnosing systems is difficult for both types of agents because the state space is vast and failures can be unclear.

Standardized benchmarks are critical in machine learning to objectively measure and compare the effectiveness of different models against an identical set of tasks. Other domains have successfully utilized platforms like SWE-bench to evaluate agents on real-world software engineering issues. Until now, there has not been a robust and risk-free simulation environment specifically for benchmarking ML models on Kubernetes configuration tasks. Sim-Arena provides this missing link by standardizing the action space, observation process, and evaluation metrics.

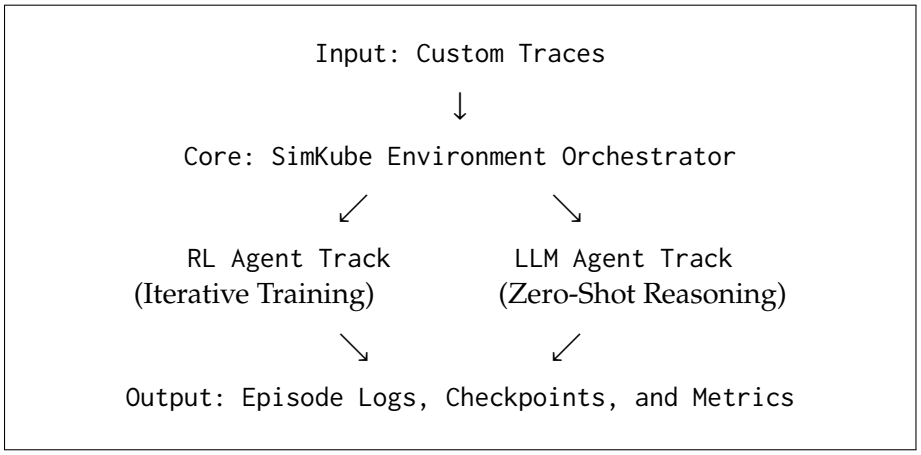

Figure 3.2 The unified Sim-Arena synthesis pipeline. Custom traces are fed into the SimKube orchestrator, which branches into dual paths to support both reinforcement learning and language model agent evaluations under a standardized protocol.

3.4 Simulation Trace Design

Acquiring abundant dataset traces of broken environments is necessary for agents to learn how to diagnose failures. Our team created our own custom traces to replicate specific deployment issues. These handcrafted traces are loaded directly into the SimKube engine at the beginning of each run, acting as the reproducible starting point for every benchmarking episode and

allowing the agents to mutate the configurations and objectively observe if the resulting simulated state improves.

Chapter 4

System Overview

4.1 Definition of Sim-Arena

Sim-Arena is a framework for training and benchmarking machine learning agents that handle Kubernetes-like workload failures in a simulated setting. It fills the gap between a Kubernetes simulator and an agent by loading a workload trace, creating a repeatable simulation, showing the resulting cluster state as an observation, accepting a defined action from an agent, evaluating the result with a reward function, and saving the run as an experiment artifact. The system is based on ACRL's SimKube simulator. SimKube can replay Kubernetes traces in a controlled test environment. Sim-Arena uses SimKube as the execution base and adds the structure needed for machine learning research: agent interfaces, observation schemas, action definitions, reward computation, benchmark scenarios, logs, and support for distributed execution. In this way, Sim-Arena is not a replacement for SimKube but an experimentation arena built on top of it.

We can represent a single run of Sim-Arena as a controlled loop seen in Figure 4.1.

This continuous loop is the foundation of the architecture. Learning agents use it to iteratively optimize performance metrics, language-model agents use it to assess cluster states and deploy targeted fixes, and distributed workers duplicate it to run parallel simulations.

The framework was designed around two primary use cases:

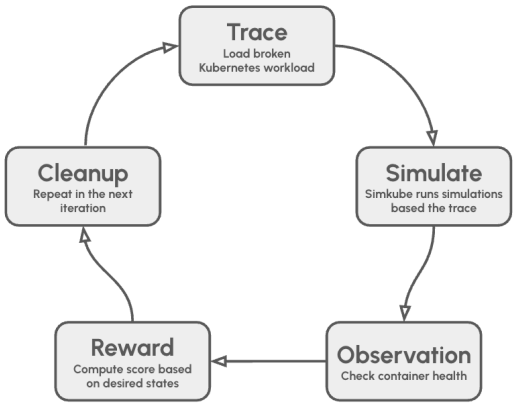

Figure 4.1 Sim-Arena loop for running simulations for agent training.

- **Safe Remediation Research:** Experimenting with CPU requests, memory limits, or replica counts on live clusters risks severe service disruption. Sim-Arena shifts this risk to a trace-driven environment where actions can be evaluated safely.
- **Repeatable Evaluation:** Measuring the effectiveness of an automated fix requires consistent initial conditions. By using fixed input traces and standardized evaluation metrics, Sim-Arena allows different policies and agents to be compared objectively.

The architecture adheres to four core design principles: production safety (restricting actions to prevent runaway resource scaling), reproducibility (relying on documented traces and persistent logs), agent independence (ensuring heuristic, RL, and LLM policies use the same interface), and extensibility.

4.2 End-to-End Workflow

Input Trace & Simulation Every experiment begins with an input trace representing a specific, reproducible Kubernetes workload history. Rather than operating on a live, fluctuating cluster, the environment layer feeds this trace into SimKube to generate a controlled simulation. The system then

pauses to allow pod scheduling, readiness, and pending states to naturally stabilize before any intervention occurs.

Observation & Action Once the cluster stabilizes, the system translates raw Kubernetes API outputs into a concise observation object focused on deployment health and resource metrics. The agent evaluates this state and selects a discrete operation, such as adjusting CPU requests or scaling replicas. Sim-Arena then applies this change as a controlled modification to the workload trace, explicitly keeping the action space within safe bounds.

Reward & Logging After the action is applied and the new cluster state stabilizes, the reward layer calculates a numerical score based on how efficiently the workload moved toward a healthy state (e.g., resolving pending pods without over-provisioning resources). Every step of this process (observations, actions, and rewards) is recorded in step-level logs, ensuring all experiments can be thoroughly audited and compared.

4.3 Major System Components

Because the specific implementation of these modules is covered in later chapters, this section provides a high-level summary of the architectural layers:

- **Runner Layer:** The central coordinator. It triggers observations, requests actions, applies trace mutations, and manages the lifecycle of each experimental episode.
- **Environment Layer:** The boundary between Sim-Arena and SimKube. It loads traces, manages the simulator lifecycle, and ensures idempotent cleanup to prevent resource contamination between runs.
- **Observation & Reward Layers:** These components extract raw simulated metrics into structured formats and evaluate them against scenario goals, separating infrastructure queries from the agent's internal logic.

- **Agent Layer:** The decision-making interface. It allows random baselines, heuristic policies, and complex models to interact with the cluster through identical contracts.
- **Protocol Layer:** Handles distributed execution, packaging jobs and syncing traces via S3 to scale throughput across multiple EC2 workers.

4.4 Architecture Diagram

Figure 4.2 illustrates the top-level data flow. The fundamental advantage of this architecture is the strict separation between the simulator and the experimental interface. SimKube owns the Kubernetes behavior, while Sim-Arena owns the execution loop.

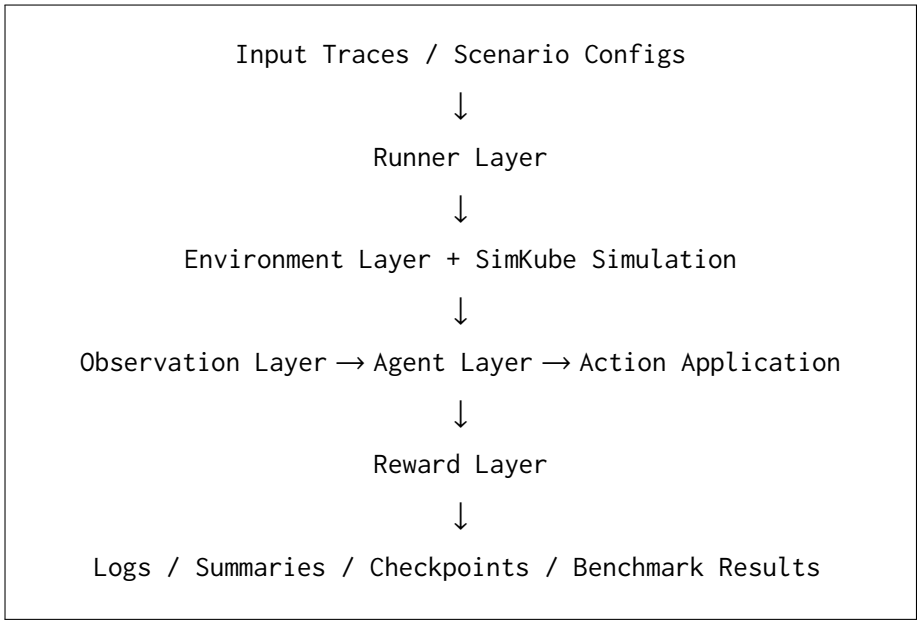

Figure 4.2 Top-level Sim-Arena architecture. The runner coordinates trace loading, simulation creation, observation, agent action, reward computation, and evaluation output.

The runner dictates the control flow. It orchestrates when to query the environment, when to poll the agent, and when to terminate the episode. Crucially, the agent only controls action selection; it cannot bypass the

safeguards, alter the reward logic, or manipulate the simulator directly. This guarantees a level playing field for evaluating different remediation strategies.

4.5 Chapter Summary

Sim-Arena serves as the experimentation layer that transforms SimKube-based replay into a structured performance evaluation environment. By grounding experiments in reproducible traces, extracting targeted metrics, and applying strictly bounded configuration changes, its modular architecture ensures different execution modes share the exact same core loop.

The following chapters detail the technical execution of these components, beginning with the initial minimum viable product design, moving through the core software implementation, and concluding with distributed execution and benchmarking outcomes.

Chapter 5

Designing the Minimum Viable Training Arena

5.1 MVP Objectives

The minimum viable product (MVP) for Sim-Arena aimed to answer a key question: can a machine learning agent interact with a simulated Kubernetes workload in a complete observe-act-reward loop? At the start of the project, the team did not intend to show that a complex reinforcement learning model could fix every Kubernetes failure. The main goal was to demonstrate that the basic training environment could function. For this reason, the MVP focused on a narrow but complete workflow. A run had to start from an input trace, create a SimKube simulation, wait for the simulated workload to appear, observe the state of the target deployment, allow an agent or policy to select a limited action, apply that action through a controlled trace change, calculate a reward, log the result, and clean up the simulation. Each of these steps must work together in one executable path.

The MVP also needed to show that Sim-Arena could interact with a real Kubernetes-like environment rather than just a mocked test object. Unit tests were useful for confirming individual functions, but the main value of the project relied on connecting to the actual simulation lifecycle. A successful MVP therefore required more than a Python interface. It required coordination among trace files, simulation resources, Kubernetes API observations, action application, and run artifacts. The most impor-

tant demonstration was a simple remediation scenario: a workload starts in an unhealthy or unsatisfied state, the arena observes that state, an action changes the configuration, and the resulting state is evaluated. Even if the first policy was straightforward, this demonstrated the infrastructure needed for future learning agents.

Several features were deliberately omitted from the MVP. This was not because they were unimportant, but because including them too early would have made the first working system much harder to validate. The MVP did not aim to expose the full Kubernetes state space. Kubernetes objects contain a lot of information: pod events, container statuses, resource requests and limits, node placement, scheduling conditions, logs, restart counts, image pull errors, service configuration, and many other details. Exposing all this information at once would have complicated the observation space and made it hard for early agents to use. Instead, the MVP focused on the minimum health indicators (such as container health) needed to assess whether a deployment was becoming ready. The MVP also did not allow arbitrary Kubernetes actions. A realistic operator might change various fields in a deployment specification, modify scheduling constraints, adjust autoscaling settings, or check logs before acting. For the MVP, this would have created an enormous and poorly defined action space. The initial arena instead used a small set of discrete actions related to CPU requests, memory requests, and replica counts. The MVP did not try to train a production-ready model. Early performance measures, if available, were seen as evidence that the training loop could run, not as evidence that the agent was ready for real cluster management. Strong model performance would require more complex scenarios, repeated trials, hyperparameter tuning, and broader evaluation. These tasks belonged to later project stages. Finally, the MVP did not include a fully automated distributed training platform. Distributed execution became an important addition, but the main focus was to ensure that one run in one environment was reliable. Scaling a fragile loop would only amplify failure cases, so the team first concentrated on the single-run path.

5.2 Problem Formulation

In the MVP, the agent sees a compact summary of the target workload. The key fields are the number of ready pods, the number of pending pods,

and the total number of pods tied to the deployment. These fields offer a basic view of the workload's health. A ready pod means Kubernetes sees the pod as ready to perform its role. A pending pod means the pod has been created but hasn't reached a running and ready state. Pending pods are particularly important for the MVP because they can show scheduling or resource-allocation issues. The total pod count helps put the other two values into context. You can think of the observation as a small state vector:

$$\texttt{observation} = \{\texttt{ready}, \texttt{pending}, \texttt{total}\}.$$

This observation does not completely explain why a workload is unhealthy. For instance, it may not identify issues like lack of CPU, insufficient memory, image pull failures, or node-level constraints. However, it provides enough information to outline a straightforward remediation problem: move the workload from an unhealthy state to one where the desired number of pods is ready and no pods are pending.

The MVP action space was intentionally small and clear. The agent could choose to do nothing, adjust CPU requests, adjust memory requests, or change the replica count. These actions represent a limited range of configuration changes a Kubernetes operator might make, but they are enough to illustrate the main idea of the arena. The no-op action allows the agent to keep the workload unchanged. This is important because not every situation should require a significant resource change. CPU adjustment actions enable the agent to increase or decrease the CPU request for the workload. Memory adjustment actions do the same for memory requests. Replica scaling actions let the agent increase or decrease the number of desired pod replicas. The available action space can be summarized as follows: These actions were viewed as limited operations rather than random edits. The limits were important because a learning agent might otherwise develop unwanted behavior, such as constantly increasing resources without regard for cost or situation.

The simplest success measure is that the target deployment achieves a healthy state. In the MVP, this means the deployment has the target number of pods, all target pods are ready, and no pods are pending. This measure converts the operational goal into a checkable statement. A run fails when the workload does not reach the desired state within the allowed steps or episode structure. Failure can occur for various reasons. The selected action may not work, the action may be blocked by a safeguard, the simulation

Action Category	Representative Action	Purpose
No-op	Keep configuration unchanged	Allows the policy to avoid unnecessary changes
CPU adjustment	Increase or decrease CPU request	Tests whether CPU allocation affects workload readiness
Memory adjustment	Increase or decrease memory request	Tests whether memory allocation affects workload readiness
Replica scaling	Scale replicas up or down	Tests whether changing desired pod count helps satisfy the target state

Table 5.1 High-level MVP action categories. The action space was intentionally discrete so that early agents could be compared under the same rules.

may not produce the expected workload state, the history may indicate a problem that the current action choices cannot resolve, or the agent may choose an ineffective action or an unsuitable resource change. This definition of success and failure is intentionally straightforward. It does not yet address all real-world operational issues, such as latency, cost, long-term stability, or user-facing service quality. However, it suffices for the MVP because it gives the arena a clear binary target: the workload is either healthy according to the target condition or it is not.

The problem was limited to make the first training arena manageable. Kubernetes remediation is a broad and open-ended issue. Allowing the agent to see every cluster object and change every possible setting would make initial experiments very hard to debug. A poor outcome could stem from the agent, the reward function, the observation design, the simulator, or the action choices. By narrowing the MVP problem, the team reduced the number of variables that could fail simultaneously. The initial arena focused on one deployment, a small observation space, a limited set of actions, and a clear success condition. This setup allowed for independent validation of each layer before connecting them into one loop. The limitation also improved comparability. If all agents receive the same compact observation and choose from the same limited action set, their behavior can be compared more meaningfully. The goal was not to claim that this small problem represents all of Kubernetes operations but to build a stable base

for future teams to expand upon.

5.3 Simulation Design

The MVP simulation starts with an input trace. The trace defines the workload scenario that the simulator will replay. In Sim-Arena, this trace serves as the fixed starting point for the episode. Using traces instead of random live cluster changes allows the arena to repeat the same scenario multiple times. Creating a simulation from a trace serves two main purposes. First, it gives the agent a realistic Kubernetes-like environment to engage with. Second, it provides a reproducible initial condition for the experiment. This is important because the same agent action should be tested against the same initial problem when comparing agents or reward functions. The arena does not treat the trace as a static file only. Once an action is chosen, the trace can be modified in a controlled manner. The resulting modified trace becomes the next version of the scenario. This makes the action path clear: instead of silently changing a live cluster, the system logs how the workload configuration was altered.

Managing namespaces (groups of resources in a single physical cluster) and lifecycle (whether a pod is created, scheduled, executed, or terminated) was important in the MVP because simulated workloads need to be isolated from each other. A previous run should not impact the next. If old pods, deployments, or simulation resources remain in the cluster, later observations may reflect leftover conditions rather than the current experiment. The lifecycle of our MVP runs repeats after cleanup after each iteration, allowing for multi-episodes learning. The lifecycle of an MVP run can be summarized as: preflight $\rightarrow$ cleanup old state $\rightarrow$ create simulation $\rightarrow$ wait $\rightarrow$ observe $\rightarrow$ act $\rightarrow$ reward $\rightarrow$ cleanup $\rightarrow$ wait $\rightarrow$ observe $\rightarrow$ act $\rightarrow \ldots$. The namespace structure also matters because the namespace stated in the trace and the namespace where simulated objects appear may not always match. The arena thus needs a consistent way to know which namespace and deployment name should be observed. Otherwise, the observation layer may report that no pods exist while the simulation is running elsewhere. In the MVP, namespace handling was kept explicit rather than hidden. The user or runner specifies the relevant namespace and target deployment so that the observation layer can query the correct workload. This reduced confusion during debugging.

A simulation does not immediately provide stable observations. After the simulation is created, the system must wait for Kubernetes and SimKube to process the trace and reveal the simulated workload state. If the arena observes too early, it may incorrectly conclude that the deployment has no pods or that nothing has occurred. For this reason, the MVP includes a waiting period between creating the simulation and observing it. This waiting phase is a practical requirement rather than a detail of the learning algorithm. It allows the cluster enough time to create pods, attempt scheduling, and report readiness or pending status. The waiting step also exposes a key challenge in using real systems as learning environments: environmental steps are slow. Unlike a simple reinforcement learning environment where the next state is returned immediately, a Kubernetes-based environment may require several seconds before the results of an action are visible. This influenced later decisions regarding logging, batching, and distributed execution.

Cleanup is part of the experiment, not just an optional final step. If a simulation is not removed properly, the next run may start from a contaminated cluster state. This can lead to misleading observations and make results hard to reproduce. The MVP thus treats cleanup as a necessary part of the lifecycle. After a run finishes, the system tries to remove the simulation resources and any old workload artifacts linked with the experiment. Cleanup also needs to be safe to repeat. If a previous cleanup partially succeeded, another attempt should not cause the entire run to fail. Repeatability relies on this cleanup process. A reproducible arena must be able to run the same trace multiple times and yield results that reflect the scenario and agent, not leftover resources from a past experiment. This is particularly important for learning agents, where many episodes may be executed in sequence.

5.4 Reward Design

The simplest MVP reward is binary. The agent gets a positive reward if the target deployment reaches the desired healthy state and receives no positive reward otherwise. This reward is straightforward to define:

$$\texttt{reward} = 1 \quad \text{if the target condition is satisfied, and} \quad 0 \quad \text{otherwise.}$$

The benefit of a binary reward is interpretability. There is no ambiguity

about what the reward signifies. The agent either solved the scenario or did not. This made the binary reward useful for the first validation of the arena. The downside is sparsity. If the agent receives no feedback until the final condition is met, then many partially useful actions might appear the same as useless actions. This can slow reinforcement learning, especially when episodes take multiple steps.

Shaped rewards provide intermediate feedback before reaching the final success condition. For instance, the reward can increase when the number of ready pods goes up, decrease when the number of pending pods increases, or offer partial credit for getting closer to the target count. Unlike binary rewards, shaped rewards can be negative or decimal values, so we can manipulate them to provide more specific feedback to the model. The aim of reward shaping is to make learning easier. Instead of requiring the agent to uncover the entire solution from a sparse success signal, the arena can provide feedback on progress. In the MVP context, shaped rewards are especially valuable because environment steps take time. If each step takes significant wall-clock time, the agent benefits from receiving more informative feedback from each interaction. However, shaped rewards need careful design. A poorly designed reward can lead the agent to optimize a proxy instead of solving the real problem. For example, the agent might learn to maximize ready pod count without considering resource waste or the target replica count. For this reason, shaped rewards were seen as an extension of the basic binary reward rather than a replacement for the success condition.

A realistic Kubernetes remediation agent should not resolve every issue by allocating excessive resources. Increasing CPU, increasing memory, and scaling replicas may enhance readiness, but they also raise costs. A reward function that only values readiness may encourage wasteful behavior. Cost-aware reward variants address this problem by penalizing unnecessary resource usage or overly aggressive scaling. The goal is to reward the agent for achieving the healthy state efficiently, not just for reaching it at any expense. At the MVP stage, cost awareness was kept simple. The system required a demonstration that reward functions could incorporate both successes and penalties. Future work can refine these costs using more realistic measures, such as resource prices, utilization, latency, or service-level objectives. Thus, for training we primarily use a shaped reward that gives denser feedback:

$$r_{\text{shaped}} = \text{clip}_{[-1,1]}\left(-0.1|R - T| - 0.05P - 0.15\max(N - T, 0) - 0.08\max(T - N, 0)\right),$$

where the variables are defined as follows:

- R is the number of currently ready pods.
- T is the target replica count (the desired number of pods).
- P is the number of currently pending pods.
- N is the total number of current pods (the actual allocated replica count).

with the special success case

$$r_{\text{shaped}} = 1 \quad \text{when} \quad R = T,\ N = T,\ P = 0.$$

As such, the agent is motivated to both reach the objective of getting all target pods ready and avoid excessive resource allocation by eliminating excessive cpu, memories, and replica sets.

5.5 Chapter Summary

The MVP version of Sim-Arena was purposefully narrow. It concentrated on one key question: Can an agent interact with a trace-driven Kubernetes simulation through a complete observe-act-reward loop? To answer this question, the team limited the state space, action space, reward design, and simulation lifecycle to the smallest useful form.

This limitation was a strength of the MVP. It made the system testable, debuggable, and reproducible. The resulting arena could load traces, create simulations, observe deployment health, apply bounded actions, compute rewards, log outputs, and clean up afterward. These capabilities laid the groundwork for later work in implementation, reinforcement learning, LLM benchmarking, distributed execution, and evaluation described in the rest of the report.

Chapter 6

Core Implementation

6.1 Repository Organization

The primary software artifact developed during this project was `sim-arena`, a Python-based training and benchmarking framework. Rather than building a simulator from scratch, Sim-Arena operates as an experimentation layer built directly on top of Applied Computing Research Labs' (ACRL) existing infrastructure—specifically `simkube` (a Kubernetes simulation system for replaying workload traces) and `isengard` (a CLI tool for cluster automation).

By wrapping these foundational tools, the repository successfully isolates research logic from simulator internals. The `sim-arena` codebase is organized into modular directories:

- `runner/`: Training orchestration logic and execution safeguards.
- `env/`: Simulation lifecycle management and trace interaction.
- `observe/`: Cluster state extraction and reward computation.
- `agent/`: Agent implementations (heuristic, reinforcement learning, and LLM).
- `protocol/`, `benchmark/`, `ops/`, and `tests/`: Distributed job support, evaluation, operational tooling, and verification.

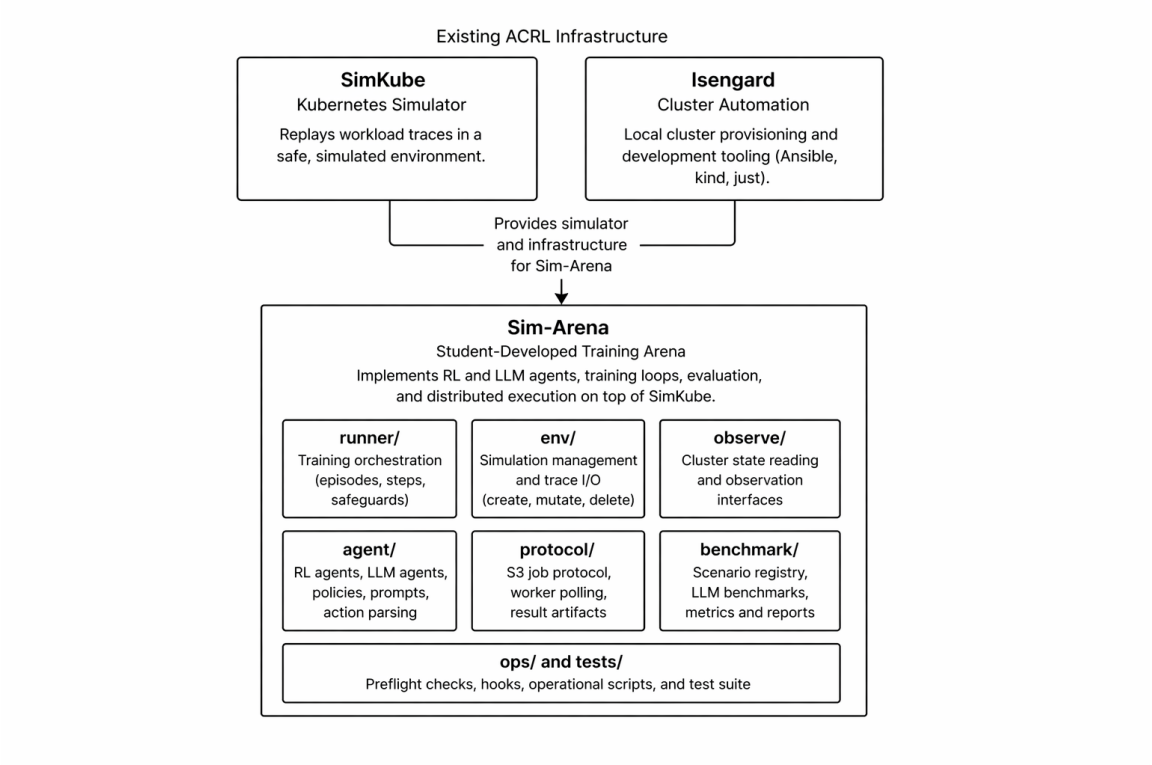

Figure 6.1 Repository context showing Sim-Arena as the primary student-developed layer built on top of existing ACRL infrastructure.

6.2 Runner Layer

Acting as the top-level controller, the runner coordinates the iterative learning loop. During each simulation step, it requests the current cluster state, queries the agent for an action, mutates the workload trace, calculates the reward, and logs the outcome.

To support rapid debugging without launching full multi-episode training sessions, the runner can execute in a single-step mode. This isolates issues stemming from cluster readiness or SimKube timing from the learning algorithms themselves.

Additionally, the layer enforces strict configuration safeguards. It blocks actions that exceed defined CPU, memory, or replica limits to prevent models from learning degenerate scaling strategies rather than solving the under-

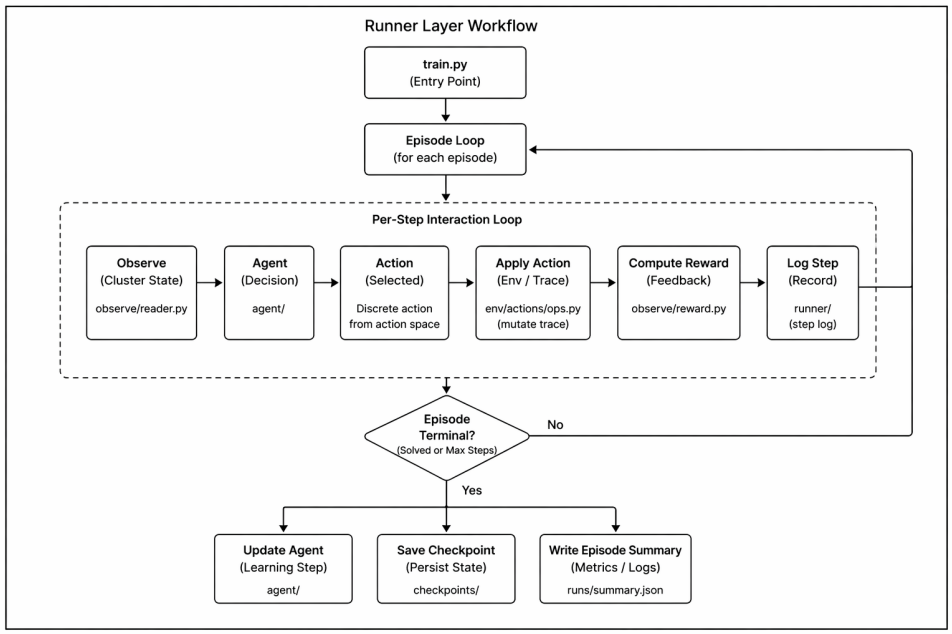

Figure 6.2 Runner-layer workflow coordinating environment interaction, agent actions, reward computation, and training outputs.

lying scheduling problem.

6.3 Environment Layer

This layer serves as the direct interface between Sim-Arena and SimKube. At the start of an episode, it loads a structured workload trace, creates a simulation, and allows it to run until pod scheduling and cluster behaviors stabilize.

Since traces are preserved, scenarios remain reproducible. The environment layer also handles cleanup routines. It safely deletes simulations once a step is complete to prevent any contamination across runs and invalidate future results.

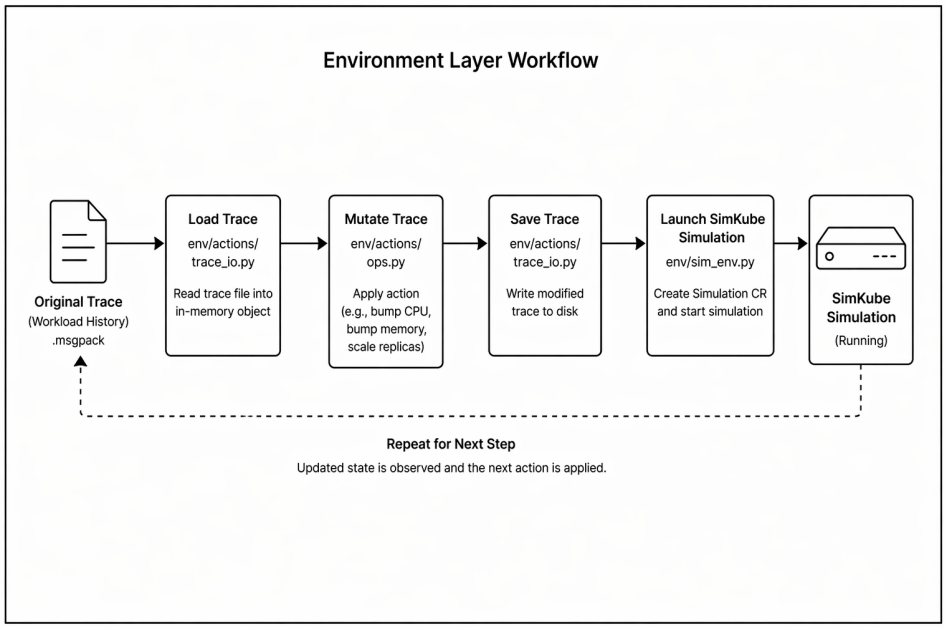

Figure 6.3 Environment-layer workflow for loading traces, applying actions, and launching simulations through SimKube.

6.4 Observation Layer

Instead of exposing the full complexity of Kubernetes APIs, the observation layer extracts a summary of deployment health, primarily focusing on performance metrics like ready, pending, and total pods.

This successfully separates infrastructure queries from learning logic, ensuring agents can process state consistently. It also includes handling to return safe default values during simulation startup delays, making long-running experiments significantly more stable.

6.5 Reward Layer

The reward layer translates the extracted observations into numerical feedback to define successful behavior. While simple binary rewards flag when a deployment reaches a healthy state, they are often too sparse for efficient

learning.

Consequently, the system supports shaped, cost-aware rewards. These functions offer partial credit for incremental improvements while actively penalizing wasteful resource over-provisioning, aligning the agent's learned behavior with realistic operator goals.

6.6 Hooks and Preflight Checks

Reliable experimentation requires robust infrastructure management alongside the training code. Before any simulation launches, the preflight layer verifies Kubernetes API connectivity, namespace availability, and SimKube CRD installation to catch errors early in the pipeline.

Cleanup hooks are also integrated into the lifecycle to automatically purge stale artifacts from previous runs or crashes, reducing manual intervention and ensuring a clean state for reproducibility.

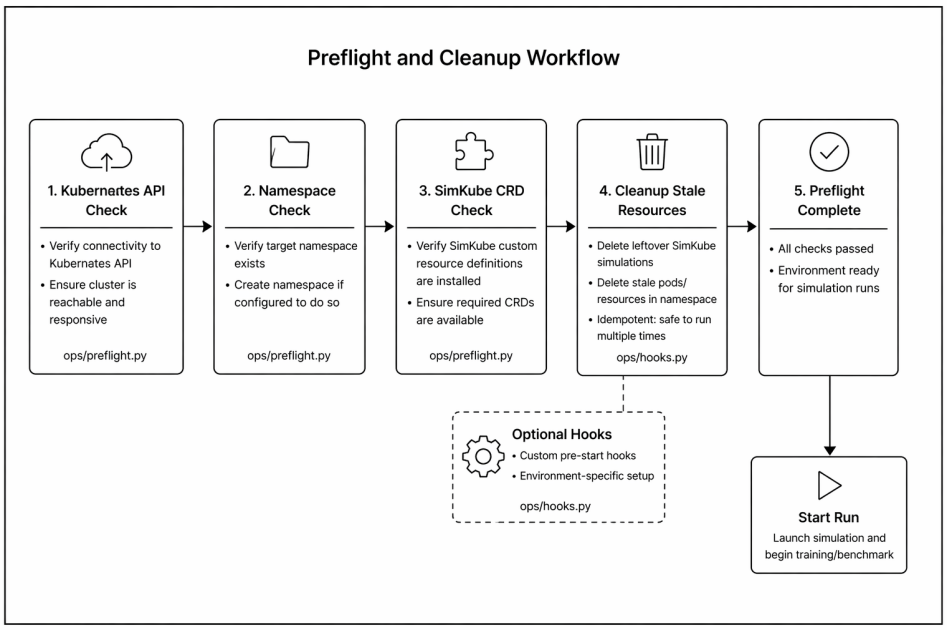

Figure 6.4 Preflight and cleanup workflow used before launching training or benchmark simulations.

6.7 Logging and Run Artifacts

Because learning runs span long horizons, the system generates persistent step-level logs, high-level metric summaries, and intermediate model checkpoints.

These artifacts are vital for tracking agent behavior, resuming interrupted runs, and isolating whether failures originated in the agent logic or the SimKube environment during scale-out distributed experiments.

6.8 Chapter Summary

Sim-Arena's modular architecture effectively separates machine learning orchestration from underlying Kubernetes simulation. By wrapping ACRL's existing systems with dedicated runner, environment, observation, and reward layers, the system provides a stable, reproducible foundation for evaluating distributed systems management policies across RL, LLM, and distributed worker setups.

Chapter 7

Reinforcement Learning Agents

7.1 Purpose and Control Loop

The reinforcement learning path treats Sim-Arena as a sequential decision environment rather than a one-shot configuration generator. At the high level, one step begins with a trace, uses SimKube to materialize that trace as a simulated Kubernetes deployment, observes the resulting cluster state, asks an agent for one bounded remediation action, edits the trace, computes a reward, and then uses the edited trace as the starting point for the next state. This design matters because the learned policy is not trained against a hand-written transition function. It is trained against the same simulator-backed lifecycle that other parts of the project use, so delays, failed scheduling behavior, cleanup issues, and trace mutation all remain visible to the learning loop.

The low-level implementation divides those responsibilities across the runner files. `runner/one_step.py` is the transition boundary: it runs pre-start hooks, copies the trace into the node data directory when needed, creates the simulation custom resource, waits for the SimKube driver pod and deployment, waits for the remaining simulation window, calls observation and resource readers, asks the agent or fixed policy for an action, validates and applies the trace edit, computes reward, writes JSONL and summary logs, and deletes the simulation. `runner/multi_step.py` links

these transitions into an episode by passing each step's output trace into the next step. `runner/train.py` repeats episodes, samples traces, manages checkpoint folders, redirects logs for long runs, and optionally switches to Gymnasium mode. The main RL algorithm evaluated in this track is the Deep Q-Network (DQN) [6] agent introduced later in this chapter, and `runner/train.py` is the script that exposes its training hyperparameters. The agent never patches the live cluster directly; it proposes an action index, and the runner translates that index into a validated trace edit.

7.2 Agent Interface and Baselines

All RL agents are reached through the facade in `agent/agent.py`. The facade exposes a small interface – `act`, `update`, `save`, `load`, `reset`, and optional visualization methods – so the runner does not need to know whether the underlying policy is random, bandit-style, neural, or LLM-backed. `AgentType.RANDOM` is implemented by reusing the epsilon-greedy agent with $\epsilon = 1.0$, giving a uniform random lower bound. `AgentType.EPSILON_GREEDY` uses `agent/eps_greedy.py`, which keeps per-action counts, incremental average rewards, and reward history; it ignores the state vector, so it is best understood as a state-free bandit baseline. `AgentType.DQN` uses `agent/dqn.py`, consumes the five-dimensional state, and performs replay-buffer Q-learning.

Agent	Primary file	Role in the project
Base	`agent/agent.py`	Sanity-check lower bound; samples all actions uniformly.
Epsilon-greedy	`agent/eps_greedy.py`	Lightweight adaptive baseline; estimates average reward per action.
DQN	`agent/dqn.py`	Main RL agent; learns Q-values from state-action transitions.

Table 7.1 RL agent implementations exposed through the shared facade.

7.3 State and Action Contracts

The action contract is intentionally small so experiments remain interpretable and comparable across RL and LLM agents. Both `runner/one_`

`step.py` and `env/simkube_gymenv.py` map seven discrete indices to trace edits: no-op, CPU increase, memory increase, replica scale-up, CPU decrease, memory decrease, and replica scale-down. Before applying a trace edit, `runner/safeguards.py` validates the requested change against hard bounds: CPU stays between 50m and 16,000m, memory between 64Mi and 32Gi, and replicas between 1 and 100. If an edit would leave those bounds, the trace is saved unchanged and the step record marks the action as blocked, giving reward functions a chance to penalize unsafe behavior without terminating the run.

For the DQN path, the runner produces a normalized five-number summary after it calls `observe()` and `current_requests()`. The raw observation contributes ready, pending, and total pod counts, while the deployment resource reader contributes CPU, memory, and replica information. In the default `base` state space, CPU is divided by 4000 millicores, memory is divided by 4096 MiB, pending pods are divided by 5, target distance is divided by 5, and replicas are divided by 8 then clipped at 1.0. The target distance feature is computed as target replicas minus total observed pods, so positive values mean the deployment is below target and negative values mean it has too many pods. The alternate `scale` state space keeps CPU and memory normalization the same but uses wider normalizers for pending pods, target distance, and replicas, making it more appropriate for larger replica-count experiments.

This state contract is intentionally practical rather than complete. It gives the DQN enough information to learn basic resource and scaling corrections, and it keeps logs and checkpoints easy to inspect. At the same time, it does not encode detailed event messages, pod restart reasons, container logs, node placement, or rollout conditions. Future work can add richer state spaces, but those additions should be versioned by name because changing feature order or normalization changes the meaning of old checkpoints.

7.4 DQN Learning Mechanics

The DQN implementation is intentionally small enough for future developers to modify, but it still contains the core machinery expected from a usable Q-learning baseline. `agent/dqn.py` defines `QNetwork` as a PyTorch multilayer perceptron with hidden layers of 24 and 48 units, ReLU activations, and one

Feature	Source	Default normalization
CPU request	Deployment resources	Millicores divided by 4000.
Memory request	Deployment resources	MiB divided by 4096.
Pending pods	Pod observation	Pending count divided by 5.
Target distance	Target and total pods	target − total, divided by 5.
Replica count	Deployment resources	Replicas divided by 8 and clipped at 1.0.

Table 7.2 Five-dimensional DQN state used by the default base state space.

Index	Action type	Trace edit
0	`noop`	Leave trace unchanged.
1	`bump_cpu_small`	Add 500m CPU request.
2	`bump_mem_small`	Add 256Mi memory request.
3	`scale_up_replicas`	Add one replica.
4	`reduce_cpu_small`	Remove 500m CPU request.
5	`reduce_mem_small`	Remove 256Mi memory request.
6	`scale_down_replicas`	Remove one replica.

Table 7.3 Shared seven-action remediation space used by RL training and LLM benchmarking.

output Q-value per action. `DQNAgent` owns the online network, a separate target network, an RMSprop optimizer, a bounded replay buffer, reward history, loss history, and episode-return history. Its default hyperparameters are exposed by `runner/train.py`: learning rate 0.001, discount factor 0.97, replay buffer size 2000, batch size 32, epsilon decay from 1.0 to 0.1 over 1000 steps, and target-network refresh every 50 environment steps.

Action selection and learning are separated cleanly. During `act`, the agent first computes the current epsilon from the linear decay schedule. With probability epsilon it explores by sampling a random action; otherwise it converts the state to a float tensor on the configured device, adds a batch dimension if necessary, runs the online network without gradients, and returns the action with the largest predicted Q-value. During `update`, the agent records the scalar reward, adds it to the current episode return, closes the episode return if the transition is terminal, converts the state, action, next state, reward, and terminal flag to tensors, and pushes that transition into replay memory. It then increments the global step counter, performs a training step once the buffer contains at least one full batch, and refreshes

the target network whenever the configured update frequency divides the step count.

The private training method performs the low-level Q-learning update. It samples a minibatch uniformly from replay memory, concatenates the stored tensors into batch tensors, gathers the online network prediction for the actions actually taken, and computes a target value from the observed reward plus discounted target-network value for the next state. Terminal transitions do not bootstrap from the next state, so their future-value term is zero. The loss is mean squared error between the gathered online Q-values and the computed targets, and RMSprop applies the gradient update. Checkpoints store both networks, optimizer state, total steps, reward and loss histories, episode histories, current partial episode reward, and core hyperparameters. This full checkpoint is important because simulator-backed training can run for a long time and should be resumable after interruption.

7.5 Training Paths and Artifacts

There are two execution paths. The legacy path calls `run_episode()` in `runner/multi_step.py`; each state calls `one_step()`, then the next observed state completes the previous transition and triggers `agent.update`. This delayed update is important: the reward for an action is only known after the next SimKube replay. For DQN, the runner can also perform extra gradient updates per environment step through `updates_per_step`. Episodes terminate when `ready == target`, `total == target`, and `pending == 0`; they can also stop at max steps or below a minimum return threshold.

The Gymnasium path is implemented in `env/simkube_gymenv.py` and selected by `-gym` in `runner/train.py`. Gymnasium is a maintained standard API for reinforcement learning environments from the Farama Foundation, and this path was added so Sim-Arena could interoperate with standard RL tooling without rewriting the underlying simulator-backed environment logic[2]. The wrapper exposes `spaces.Discrete(7)` actions and a five-dimensional `spaces.Box` observation, then wraps the same simulation lifecycle inside `reset()` and `step()`. Both paths write useful artifacts: step logs under `runs/`, checkpoint folders under `checkpoints/`, latest and interval model files, optional visualizations, and early performance measures.

The training script can sample from a directory of traces, resume from a checkpoint, continue in the loaded checkpoint folder, or use transfer mode to load weights while resetting history.

7.6 Design Considerations

The main engineering lesson is that RL speed is dominated by environment latency, not neural-network computation. A single transition includes trace copying into the kind node data directory, simulation custom-resource creation, driver-pod synchronization, deployment polling, the remaining simulation wait, Kubernetes observation, trace mutation, reward calculation, logging, and cleanup. For a project manager with Kubernetes experience, this means the highest-leverage scaling work is likely around simulation throughput, cleanup reliability, and distributed scheduling rather than GPU tuning.

The main research limitation is partial observability. The DQN gets a compact and stable state schema, which is good for early experiments, but it cannot see the event and log evidence available to the LLM benchmark. Future work should add richer state spaces carefully, under explicit names such as `-state-space events-v1`, and should document checkpoint compatibility when feature order or normalization changes. The safest extension path is to keep the seven-action contract as the comparable baseline, add richer observations incrementally, and evaluate each change on the same scenario set before widening the action space.

Chapter 8

Distributed Simulation and Scaling Infrastructure

8.1 Why Scaling Was Necessary

The initial Sim-Arena workflow was built to validate the end-to-end training loop on a single machine. This local setup was effective for early development, debugging, and proof-of-concept testing, but it became a bottleneck once the project expanded beyond one-step validation and small manual experiments. Each run required creating a simulation, waiting for the simulated Kubernetes state to stabilize, observing the cluster, applying an action, and repeating the process across multiple steps or episodes. Because each simulation step could take tens of seconds, even modest experiment batches accumulated significant wall-clock time.

As the project matured, we needed to evaluate more than whether a single run worked. We wanted to compare traces, reward modes, and agent configurations, and later support repeated training runs rather than isolated demonstrations. Running these jobs sequentially on one machine limited throughput and slowed iteration. In practice, the main bottlenecks were not only the agent update process, but also simulation startup overhead, Kubernetes stabilization delays, repeated artifact handling, and the manual cleanup required between runs. Under these conditions, distributed execution became necessary for the project to scale beyond a local prototype.

AWS was the natural platform for this scaling effort because our distributed design already depended on cloud-based infrastructure. In particular, we used Amazon EC2 to provision worker machines that could run simulations in parallel, and Amazon S3 to store traces, manifests, logs, checkpoints, and other job artifacts shared across runs. Using AWS let us move from a single local prototype to a practical multi-worker setup without introducing an entirely separate infrastructure stack. This made it easier to coordinate experiments, persist results, and support the larger volume of runs needed for distributed evaluation and training.

8.2 S3-Based Job Protocol

To coordinate distributed runs, we adopted an S3-based job protocol. In this design, each training run is represented by a manifest file containing the information needed to execute the job, such as the trace location, training parameters, reward configuration, and optional checkpoint references.

A dispatcher submits jobs by writing manifests to a designated S3 location. Workers then poll that location for pending jobs, claim work, run the specified training command, and upload structured outputs when finished. These outputs include status metadata, logs, and model checkpoints, allowing operators to inspect results without directly logging into each worker machine.

This design also supports checkpoint handoff between runs. One job can produce model weights that a later job uses as input, making it possible to continue training across separate executions. We chose S3 because it was already part of the AWS-based workflow, provided durable storage for both metadata and larger artifacts, and avoided the need to deploy and maintain additional queueing infrastructure. The main drawback is that polling-based coordination is less efficient than a dedicated event-driven scheduler, but for the scope of this project the simplicity of the S3 model was a worthwhile tradeoff.

8.3 Worker Architecture

Each worker follows a simple polling loop. It checks the S3 job location for a manifest, downloads the selected manifest and any required inputs, executes the corresponding training job, and uploads its outputs back to S3. In our implementation, the worker is responsible for retrieving traces, loading prior checkpoints when continuation is requested, running the training command, and packaging the result artifacts in a standardized format.

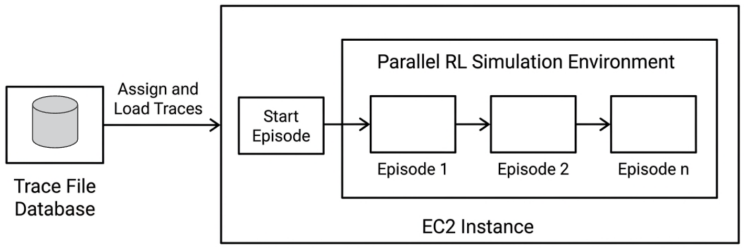

Figure 8.1 Parallel episode execution within a worker instance using shared trace inputs.

This structure keeps worker behavior explicit and modular. Because the logic is concentrated in a small number of scripts, the system is easier to debug than a more opaque distributed framework. The worker design also improves failure visibility. If a job fails, the worker can still upload partial logs or structured failure metadata rather than failing silently. That makes it easier to diagnose what went wrong and rerun jobs when necessary.

8.4 EC2-Based Execution

We paired the S3 job protocol with EC2-based execution so that multiple workers could be launched on demand. EC2 was a natural choice because it integrated directly with the AWS-based storage layer and gave us control over machine setup, networking, and lifecycle management. Rather than

configuring every instance from scratch, we used an AMI-based workflow so that dependencies and baseline environment configuration could be prepared in advance. This reduced setup time, improved consistency across workers, and lowered the risk of environment drift.

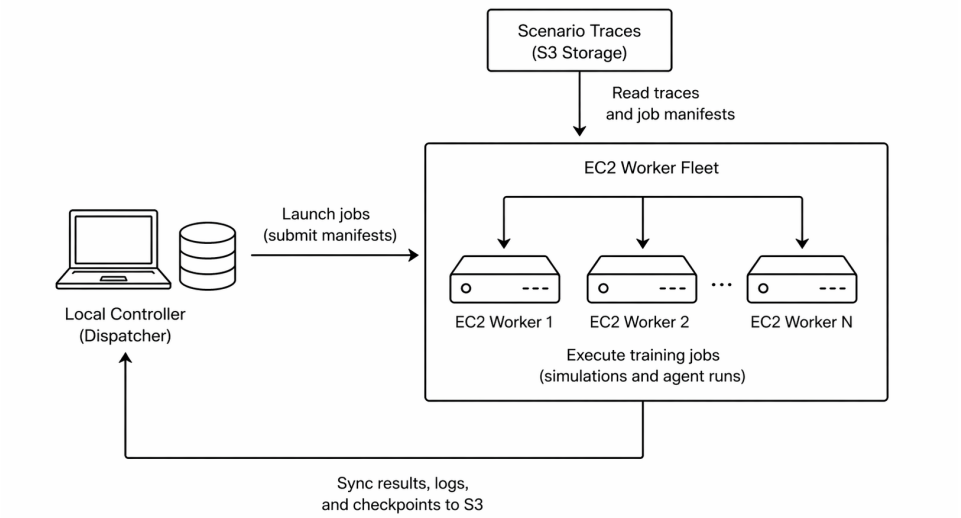

Figure 8.2 EC2-based execution workflow for running Sim-Arena jobs across multiple worker instances. Each worker reads traces and job manifests, executes training runs, and syncs result artifacts back to S3.

Operationally, the workflow was straightforward. An operator prepared traces and job manifests, launched one or more EC2 instances, started the worker polling loop on each instance, monitored outputs through S3, and then shut down the instances when the experiment run was complete. This made horizontal scaling relatively simple: additional throughput could be achieved primarily by adding more worker instances.

8.5 Benefits and Tradeoff

The distributed design provided three main benefits. First, it improved scalability by allowing multiple jobs to run in parallel instead of competing for one machine. Second, it improved reproducibility because each run was defined by an explicit manifest and produced persistent output artifacts.

Third, it preserved clarity: job state was visible through ordinary storage paths, and worker behavior was easier to reason about than a more complex orchestration stack.

At the same time, the design has clear limitations. Polling-based coordination is less efficient than event-driven scheduling, worker management and monitoring still involve manual steps, and reproducibility depends on careful environment setup across instances. In other words, the system scales effectively for the needs of this project, but it is not yet a fully automated distributed training platform.

8.6 Unfinished Features

Some distributed components remain partial or planned rather than fully implemented. The repository documentation describes a centralized training server design, but that server is not currently present as a completed implementation in the active codebase. Similarly, experience-collection paths for more advanced distributed workflows are discussed in the design materials but are not yet operational end to end.

The current system does support distributed training jobs through S3 manifests and EC2 workers, but it does not yet include the full distributed architecture originally envisioned. The most important next steps are to improve retry and synchronization behavior, formalize job-state transitions, strengthen monitoring, and complete the unfinished distributed runner paths.

Even in its current form, however, the distributed infrastructure represents a meaningful extension of Sim-Arena. It transformed the project from a local proof of concept into a system capable of parallel experimentation, while still preserving the transparency and reproducibility that motivated the training arena in the first place.

Chapter 9

LLM Benchmarking and MCP Integration

9.1 Purpose and Shared Contract

The LLM benchmark was added because Kubernetes remediation often depends on semantic evidence that the RL state vector intentionally omits. A human operator would normally inspect pod phases, deployment status, warning events, and logs before deciding whether to reduce a resource request, scale replicas, or wait. The LLM path models that workflow by giving the model read-only observability tools through the Model Context Protocol (MCP), while still requiring it to return exactly one of the seven standard action indices. It is therefore not a separate simulator or a direct Kubernetes automation layer; it is another agent plugged into the same trace, action, reward, and episode contract used by RL.

This shared contract is what makes comparisons meaningful. `benchmark/run.py` reports human-readable action names, but `runner/one_step.py` still defines the real action semantics and `runner/safeguards.py` still validates the resulting trace edit. The benchmark also passes the selected reward name into `run_episode()`, so shaped and cost-aware rewards are reused. For future developers, the rule of thumb is simple: improve prompting, tools, providers, or metrics without creating a second action space unless the whole benchmark contract is intentionally revised.

9.2 Scenario Registry and Benchmark Entry Point

Scenarios are declared in `benchmark/scenarios/index.json` and loaded by `benchmark/scenarios/__init__.py`. Each scenario records a name, trace path, target replica count, problem type, failure mode, severity, initial resource settings, and a short description. The current registry includes healthy baselines, replica-scaling cases, and resource-contention cases such as over-requested CPU, over-requested memory, and combined CPU-memory pressure. This keeps evaluation content separate from benchmark code: a new case usually requires adding a trace and one JSON entry, not changing the runner.

`benchmark/run.py` is the command-line entry point. It loads API keys from the environment, selects a provider and model, filters scenarios with `-scenario` or `-filter-type`, configures namespace, deployment, duration, max steps, reward, seed, and output directory, then creates an `AgentType.LLM` instance. For each scenario, it injects the scenario name into the LLM agent and calls the same `run_episode()` function used elsewhere in Sim-Arena. The `-list-scenarios` flag is especially useful for project managers because it previews the benchmark matrix without launching any simulations.

9.3 MCP Server, Client, and Tool Surface

MCP is used to separate model-provider logic from Kubernetes inspection logic. `sim_mcp/server.py` runs a FastMCP server over stdio and registers four read-only tools. Each tool delegates to a small module in `sim_mcp/tools/`; those modules create Kubernetes clients through `sim_mcp/tools/_k8s.py`, which first tries in-cluster configuration and then falls back to the local kubeconfig. This keeps the tool layer portable: the same code can run inside a cluster or from a developer machine with kubeconfig access.

The client side is deliberately lightweight. `sim_mcp/client.py` starts the MCP server as a subprocess, initializes a session, asks the server for its tool schema, converts that schema to Anthropic-compatible tool definitions, and exposes a synchronous `call_tool(name, arguments)` method through `MCPClientSync`. The benchmark runner creates one synchronous MCP client and passes it into the LLM agent. From there, the provider runs a model-specific tool loop: the model requests a tool call, the provider sends that

request to the MCP client, the MCP server queries Kubernetes, and the provider feeds the tool result back to the model. Only after the model stops requesting tools does the provider parse a final JSON action index. This design keeps Kubernetes observability reusable across providers while keeping Gemini-specific and Anthropic-specific message handling out of the MCP server.

Tool	Implementation	Evidence returned
`get_pods`	`sim_mcp/tools/pods.py`	Pod phase, node, conditions, restart counts, container state.
`describe_deployment`	`sim_mcp/tools/deployments.py`	Desired/ready replicas, resources, images, deployment conditions.
`get_events`	`sim_mcp/tools/events.py`	Recent events, filtered by deployment, with warnings sorted first.
`get_pod_logs`	`sim_mcp/tools/logs.py`	Recent container logs, with container auto-resolution when possible.

Table 9.1 Read-only MCP tools exposed to language-model agents.

9.4 Prompting, Providers, and Parsing

`agent/llm_agent.py` is the provider-agnostic LLM agent. Its `act()` method builds the prompt, calls the provider, records tool calls, latency, rounds, and reasoning, then returns only an integer action index to the runner. Prompt construction lives in `agent/prompt_builder.py`. The system prompt intentionally defines two phases: investigate with exactly four tools, then return final JSON with `action_index` and one-sentence `reasoning`. It describes remediation choices as numeric indices rather than function names, because function-calling models can otherwise confuse actions with callable tools.

Provider-specific loops live in `agent/providers/`. `make_provider()` maps CLI names to default models: Gemini uses `gemini-2.5-flash-lite`, and Anthropic uses `claude-sonnet-4-20250514` unless overridden. `GeminiProvider` converts MCP schemas into Gemini function declarations, runs with temperature 0, retries transient 429 and 503 errors, and rejects invalid tool calls

with a corrective function response. `AnthropicProvider` passes the MCP tool schema directly to the Messages API and follows the `tool_use` stop-reason loop. Final model text is parsed by `agent/action_parser.py`, which tries full JSON, then a JSON block, then a bare integer; if all parsing fails, the benchmark logs the problem and falls back to action 0.

9.5 Metrics and Output Artifacts

Benchmark metrics are implemented in `benchmark/metrics.py`. Step records include scenario name, step index, raw observation, action index, action label, reward, tool-call list, tool-call count, latency, target status, and model reasoning. Episode summaries compute solved status, steps to solve, total reward, total tool calls, average tool calls per step, total and average latency, action distribution, tool distribution, and elapsed time. Run-level aggregation reports solve rate, average reward, average steps, average steps to solve, average tool calls, average latency, and per-problem-type statistics.

Each benchmark run writes a timestamped directory under `benchmark/results/` unless `–results-dir` is supplied. The run stores `command.txt`, `benchmark.log` when terminal logging is disabled, `report.json` for programmatic analysis, and `report.md` for quick review. This artifact structure is important because LLM runs are less deterministic than RL runs: model output format, API latency, quota behavior, and tool-use choices can vary, so the benchmark needs enough recorded evidence to explain a success or failure after the run.

9.6 Design Considerations

The strength of the LLM track is also its risk: the model sees richer evidence, but its behavior depends on external APIs, provider-specific tool semantics, prompt following, and output formatting. The current code mitigates common failures with strict prompts, bounded tool rounds, invalid-tool rejection in the Gemini provider, no-op fallback parsing, and detailed per-step metadata. These are engineering guardrails, not correctness guarantees.

Future work should keep the LLM path as one agent inside the shared arena. New tools should remain read-only unless the action contract is deliberately

redesigned; new providers should implement `LLMProvider.run_step()` and return `StepResult`; and new benchmark fields should be added through `benchmark/metrics.py` so reports stay comparable. Useful next steps include provider timeouts, clearer failure categories in `report.json`, mocked transcript tests that avoid paid API calls, and optional local-model providers for runs where reproducibility or cost matters more than frontier-model performance.

Chapter 10

Evaluation and Results

10.1 Evaluation Methodology

The evaluation for Sim-Arena focused on two kinds of success: engineering success and research feasibility. Because this project was an open-ended systems research project, the goal was not to prove that one agent could solve all Kubernetes remediation problems. Instead, the goal was to show that the arena could support reproducible agent interaction, testing, benchmarking, logging, and scaling. This matches the original project goal of building a simulated training environment where Kubernetes-specific machine learning models can query, interact with, manipulate, and be evaluated inside a safe environment.

We evaluated the system across several layers. At the core implementation level, we checked whether traces could be loaded, simulations could be created and deleted, observations could be collected, actions could mutate traces, rewards could be computed, and run artifacts could be saved. At the training level, we examined whether reinforcement learning agents could run across episodes and produce checkpoints and learning outputs. At the infrastructure level, we evaluated whether jobs could be dispatched to workers and whether distributed execution reduced the limitations of single-machine experimentation. At the benchmarking level, we examined whether LLM agents could run through the same scenario structure and action space as RL agents.

The evaluation was intentionally cautious. The current results should be interpreted as feasibility and systems validation rather than large-scale statistical proof. The project did not include exhaustive hyperparameter sweeps, production cluster deployment, or broad testing across the full range of Kubernetes workloads. Instead, the results demonstrate that the core arena is functional, extensible, and suitable for future research.

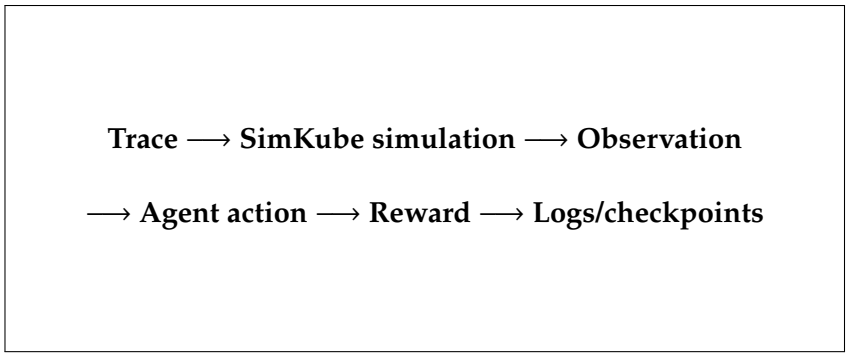

Figure 10.1 Sim-Arena pipeline layout.

10.2 System Validation Results

The first evaluation goal was to validate that the minimum viable training loop worked end to end. A successful step required the system to load a trace, create a SimKube simulation, wait for cluster behavior to emerge, observe workload health, apply a bounded action, compute reward, log the result, and clean up simulation resources.

The MVP validation showed that the main software components were implemented and connected. The environment module supported simulation creation, waiting, and deletion. The observation module returned structured pod health information such as ready, pending, and total pod counts. The action layer supported trace modifications such as increasing CPU, increasing memory, and scaling replicas. The runner layer connected these components into a repeatable single-step workflow. The MVP status report also records completed modules for environment handling, observations, rewards, trace actions, hooks, preflight checks, and the minimal agent loop.

This validation was important because later features depended on the same

Component	Validation Goal	Result
Environment layer	Create, wait for, and delete SimKube simulations	Passed
Observation layer	Return ready, pending, and total pod counts	Passed
Action layer	Modify trace CPU, memory, or replica settings	Passed
Reward layer	Compute reward from observed workload state	Passed
Runner layer	Execute one complete observe-act-reward step	Passed
Hooks and preflight	Check cluster readiness and clean stale state	Passed

Table 10.1 MVP validation summary for the core Sim-Arena training loop.

core loop. Table 10.1 summarizes the validation results for each core Sim-Arena component, all of which passed the MVP validation goals. Reinforcement learning, LLM benchmarking, and distributed workers all rely on the ability to run one reliable simulation step. For that reason, the MVP result should be viewed as the foundation of the rest of the project.

10.3 Testing Strategy

Testing was divided into unit tests, integration tests, protocol tests, and manual end-to-end validation. Unit tests were used for fast validation of pure logic, including trace I/O, action operations, reward calculation, safeguards, and policy behavior. Integration tests focused on the runner workflow, especially the order of operations in a complete step. Protocol tests validated the S3 job schema and worker dispatch behavior without requiring live AWS credentials. Manual tests were still necessary because Kubernetes and SimKube behavior cannot be fully represented by mocked tests.

The test strategy reflected the layered design of the system. Table 10.2 summarizes the main areas of test coverage and the purpose of each testing layer.The team tested isolated components where possible, then tested how those components were wired together. This was especially important be-

cause many failures in Sim-Arena were not caused by individual functions, but by interactions between traces, namespaces, simulations, observations, and cleanup.

Area	Representative Test Coverage	Purpose
Trace and actions	Trace loading, saving, CPU/memory bumps, replica scaling	Correctness
Observation and reward	Pod count extraction, reward modes, missing deployment behavior	Correctness
Runner	One-step execution, logging, cleanup, idempotency	Integration
Safeguards	Resource bounds and blocked actions	Safety
Protocol	Job manifests, result schemas, worker logic with mocks	Distribution
Agents and policies	Random, greedy, DQN, and policy behavior	Stability

Table 10.2 Testing strategy used to validate Sim-Arena across implementation layers.

The most important limitation is that mocked integration tests do not prove live cluster behavior. They validate software wiring and expected control flow, but live SimKube runs are still required to validate the full environment. For this reason, the testing strategy combined automated tests with manual smoke tests on real cluster setups.

10.4 RL Training Performance

Reinforcement learning evaluation focused on whether agents could run through repeated episodes, produce checkpoints, and show interpretable training behavior on representative Sim-Arena remediation tasks. In these tasks, the agent interacted with recorded Kubernetes traces and learned to apply actions such as adjusting CPU or memory requests or scaling replicas in order to improve pod health and move the system toward a target state. The primary goal was not to claim a production-ready remediation policy,

but to verify that Sim-Arena could support the kind of training loop required for future research.

Training runs showed that the system could execute multi-step and multi-episode experiments, save intermediate outputs, and produce learning artifacts. These experiments were run on trace-based scenarios representing Kubernetes resource and scaling problems, rather than on only a single one-step demonstration. The Sim-Arena supports DQN, epsilon-greedy, random, and LLM agents, along with checkpointing and learning outputs. Figure 10.2 shows preliminary RL training performance on the restricted two-action task, where the DQN agent generally outperformed the random baseline over training episodes.

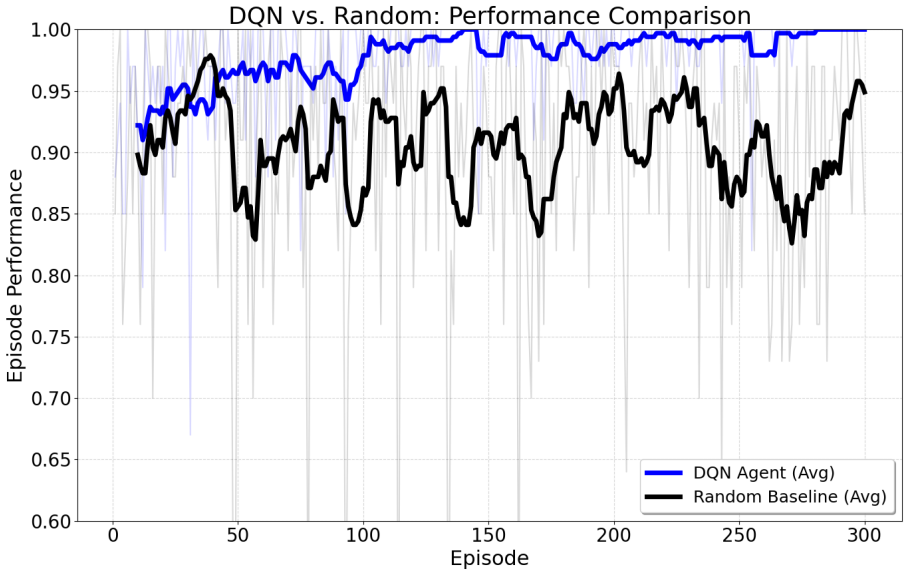

Figure 10.2 Preliminary reinforcement learning results on a restricted two-action task, where the agent could only choose between noop and increase replicas, compared against a random baseline over training episodes.

The current RL results support a modest conclusion: the training pipeline works, produces artifacts, and can be used for future learning experiments. They do not yet prove broad generalization across unseen Kubernetes workloads. Additional experiments with more traces, repeated seeds, and stronger baselines would be needed before making stronger claims.

10.5 Distributed Worker Performance

Distributed evaluation focused on whether Sim-Arena could move beyond single-machine execution. The S3-based protocol allowed jobs to be represented as manifests, processed by EC2 workers, and returned as result files, logs, and checkpoints.

The main result is that the distributed design successfully separated job submission from job execution. This made it possible to launch workers independently, assign them jobs, and inspect outputs through shared storage. This is an important engineering outcome because local training was too slow for repeated experimentation.

The distributed system also revealed operational constraints. EC2 setup, S3 credentials, worker environment consistency, bucket names, and trace paths all affected whether jobs completed successfully. In addition, we encountered an unresolved bug in the worker-side simulation path, which is documented elsewhere in the report. As a result, distributed execution should be interpreted as a successful prototype and scaling direction, not a fully managed training platform. The framework is in place to support distributed system runs, but additional code and testing is required to ensure that each individual container runs a robust simulation

10.6 LLM Benchmark Results

LLM benchmarking evaluated whether language-model agents could use the same scenario structure, action space, and reward logic as reinforcement learning agents. This was important because one of the project goals was to support comparable evaluation across different types of Kubernetes agents.

The benchmark harness supports scenario execution, MCP-based tool calls, action parsing, and result reporting. We completed LLM benchmarking with Gemini and Claude models, where agents can query Kubernetes state through MCP tools such as pod state, events, deployment descriptions, and logs before selecting an action. This allowed LLM agents to operate inside the same environment loop rather than being evaluated separately.

The current LLM results should be framed as benchmark feasibility rather than final model ranking. The important result is that LLM agents can be

placed inside the same evaluation loop as RL agents. Stronger claims about which model performs best would require a larger scenario set, repeated runs, and careful control over model versions and prompting.

10.7 Failure Cases and Limitations

Failure cases were an important part of the evaluation. In a systems research project, failed runs are not just noise; they reveal where the platform is fragile and where future engineering work is needed.

The main failure categories were environment failures, simulation failures, invalid or blocked actions, distributed worker issues, and model-level failures. Environment failures included namespace mismatches, missing deployments, stale simulations, or cluster startup problems. Simulation failures included driver delays or traces that did not produce the expected workload state. Distributed failures included credential problems, bucket mismatches, or incomplete worker outputs. LLM-specific failures included malformed responses or action selections that did not match the expected action format.

These limitations shape the interpretation of the results. Sim-Arena should not yet be viewed as a production-grade Kubernetes operator. It is a research platform for controlled experimentation. The evidence supports the claim that the arena works as a reusable evaluation framework, but not the claim that trained agents are ready to manage live production clusters.

10.8 Key Findings

The evaluation produced several key findings.

First, the core Sim-Arena loop is functional. The system can connect traces, simulations, observations, actions, rewards, and logs into a repeatable workflow.

Second, automated tests were essential but not sufficient. Unit and integration tests validated much of the software logic, but live Kubernetes and SimKube testing remained necessary because test stubs could not fully

capture the breadth of real cluster behavior.

Third, reinforcement learning experiments are feasible but slow. The training loop can run and produce checkpoints, but meaningful learning experiments require careful reward design, sufficient compute, and repeated runs.

Fourth, distributed execution provides a practical path to higher throughput. The EC2 and S3 workflow allowed jobs to be dispatched and processed outside a single local machine, although operational reliability remains an area for improvement.

Fifth, LLM benchmarking became a valuable extension because it allowed tool-using models to be evaluated under the same action and reward structure as RL agents.

Overall, the evaluation supports the central project claim: Sim-Arena is a working training and benchmarking arena for Kubernetes-focused machine learning agents. Its strongest contribution is not a final optimized agent, but the reusable infrastructure needed to study such agents systematically.

Chapter 11

Challenges, Pitfalls, and Dead Ends

11.1 Technical Challenges

A major challenge throughout the project was that Sim-Arena was not built on a simple mock environment. It interacted with real Kubernetes tooling, SimKube simulation resources, driver pods, namespaces, traces, and cluster state. This meant that failures were often caused by the surrounding systems rather than the learning code itself. A run could fail because a simulation resource was created but the expected pods never appeared, because the driver was delayed, because a namespace was wrong, or because stale resources from a previous run affected the next experiment.

SimKube integration introduced its own learning curve. Sim-Arena depended on SimKube to replay traces and create simulated workload behavior, but Sim-Arena did not control every part of that process. One important lesson was that creating a Simulation custom resource did not guarantee that the expected workload had appeared. We often had to verify driver health, trace paths, and the virtual namespace where simulated pods were expected to run.

Trace and namespace handling was another recurring source of confusion. A trace might describe a workload in the `default` namespace, while SimKube would expose the simulated pods in a namespace such as `cluster`.

When this mapping was misunderstood, observations could return zero ready pods, zero pending pods, and zero total pods, making the issue look like an agent failure even though the real problem was environmental.

Debugging distributed behavior was even harder. Once jobs were moved to EC2 workers and S3, failures had to be diagnosed across local commands, S3 objects, worker logs, SSH sessions, Kubernetes state, and result files. A successful EC2 launch did not necessarily mean that the worker could run a healthy simulation. This made logging, preflight checks, and explicit cleanup essential parts of the system rather than optional conveniences.

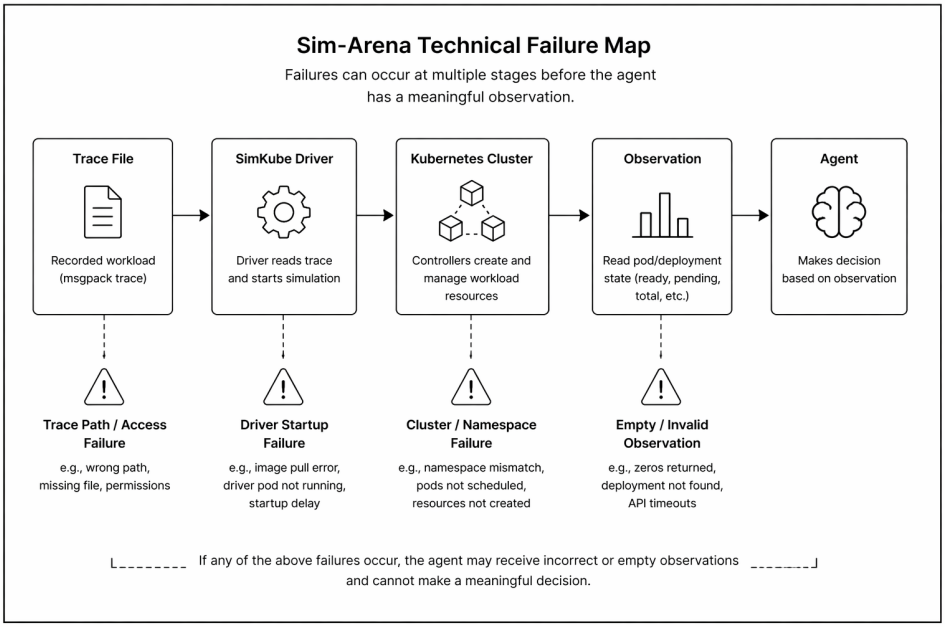

Figure 11.1 Simplified failure map showing how environment issues can occur before the agent receives a meaningful observation.

11.2 Training Challenges

Training was slowed by long feedback loops. Each step required creating or updating a simulation, waiting for the simulated workload to stabilize, observing cluster state, applying an action, computing reward, logging results, and cleaning up. Even when the code was correct, this made experimen-

tation slow. Small changes to reward functions, hyperparameters, or trace selection could require long reruns before their effects became visible.

Reward design was one of the most difficult modeling problems. Simple binary rewards were easy to interpret but often too sparse for efficient learning. Shaped rewards provided more feedback but introduced the risk of encouraging behavior that improved the metric without truly solving the underlying workload issue. Cost-aware rewards added another layer of difficulty because the agent needed to improve workload health without simply increasing resources indefinitely.

The state representation was intentionally constrained. Early versions of the environment focused on compact signals such as ready pods, pending pods, and total pods. This kept the problem tractable and made the system easier to debug, but it also limited what the agent could learn. Richer observations may improve future performance, but they would also increase complexity and require corresponding changes to the agent, reward logic, and tests.

Many failures were also hard to interpret. A bad reward curve did not always mean the agent was learning poorly. It could mean that the deployment never appeared, the observation returned default values, the action was blocked by safeguards, or the simulation did not reach a meaningful state. This reinforced the importance of separating environment validation from model evaluation.

11.3 Scaling Challenges

Scaling the system introduced a different class of problems. EC2 workers required machine images, security groups, SSH access, dependencies, Kubernetes configuration, S3 access, and consistent environment variables. A setup that worked locally did not automatically work on a worker instance, especially when trace paths, credentials, or cluster configuration differed.

AWS credential and bucket management also required care. Workers needed access to job manifests, trace inputs, checkpoints, and result outputs. SimKube itself sometimes needed credentials in order to read traces from S3. Using the wrong bucket, region, account, or credential source could cause failures that were difficult to distinguish from simulation errors.

The distributed system was operationally fragile because it depended on

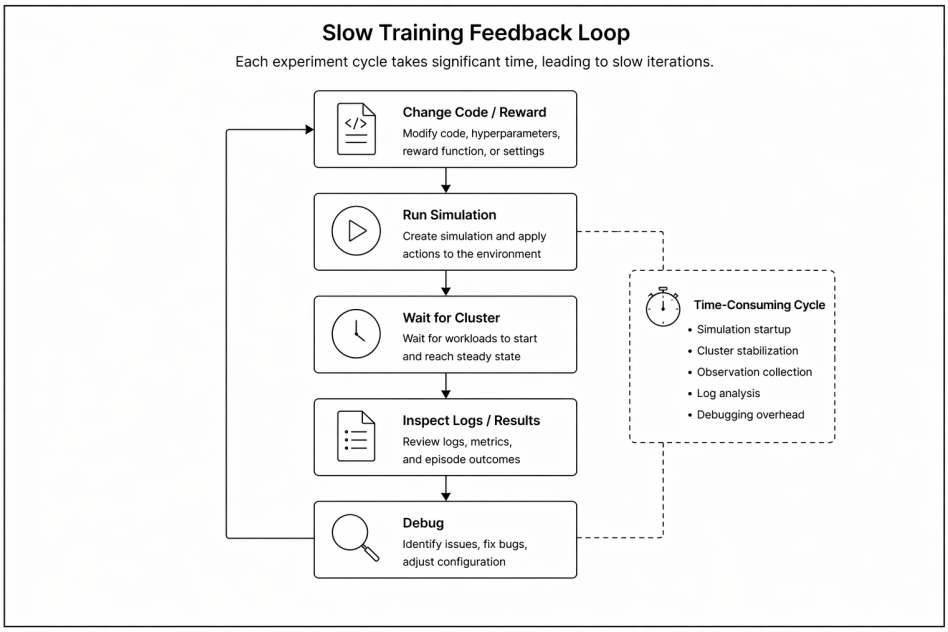

Figure 11.2 Slow experimentation loop caused by simulation runtime, cluster stabilization, and repeated debugging.

several moving parts at once. Workers could become stale, simulations could be left behind, long-running processes could continue after failed runs, and clusters could degrade over time. This made cleanup scripts, health checks, and runbooks central to the workflow.

Coordination through S3 kept the design simple, but it also introduced overhead. S3 worked well as a durable and inspectable coordination layer, but it was not a full scheduler. Polling, job claiming, retries, and result inspection required operator discipline. The design was appropriate for the project stage, but future teams may want stronger scheduling, monitoring, and retry mechanisms.

11.4 Design Tradeoffs

Several design choices were made deliberately to keep the project achievable. The state space was kept small so that early experiments could be debugged and interpreted. A larger state representation may eventually be

necessary, but it would have made the first working loop harder to build.

The action space was also kept discrete. Instead of allowing arbitrary Kubernetes edits, the system used bounded actions such as increasing CPU, increasing memory, scaling replicas, or doing nothing. This made actions easier to test, easier to compare across agents, and easier to constrain with safeguards.

The team also chose S3-based coordination instead of a more complex scheduler. This avoided the need to deploy a separate queueing service or custom control plane. The tradeoff was that the system had weaker scheduling semantics and more manual operational steps. For a research prototype, however, transparency and simplicity were more valuable than production-grade orchestration.

Some extensions were postponed because they would have expanded the scope before the core system was stable. More advanced experience collection, a central dashboard, richer observations, and broader scenario libraries are all valuable future directions, but they depend on a reliable base training and evaluation workflow.

Taken together, these decisions define the project's main design tradeoffs. A small state representation was chosen over a richer but harder-to-debug observation space. A discrete seven-action interface was chosen over arbitrary Kubernetes edits so that policies could be compared and constrained safely. S3-based coordination was chosen over a stronger scheduler because it was easier to inspect and deploy during the project timeline. Advanced features such as broader scenario libraries, richer diagnostics, and more complex distributed coordination were deferred in favor of first making the one-step training and evaluation loop reliable. Each tradeoff reduced capability in one dimension, but each one also made the initial system easier to implement, test, and hand off.

11.5 Dead Ends and Abandoned Ideas

Not every idea improved the system. Some early designs introduced too much indirection between the runner and the action logic. These abstractions sounded useful in theory, but they made stack traces harder to follow and slowed debugging. The implementation became clearer when the run-

ner called trace mutation operations more directly.

Some planned features were also left as design directions rather than completed systems. A central training dashboard, for example, would have improved monitoring, but it was not essential to validating the training loop. Similarly, more advanced distributed experience collection was deferred because the basic distributed training path was a higher priority.

The main lesson is that future teams should avoid scaling or adding interfaces before the one-step path is reliable. A stable single simulation step is the foundation for training, benchmarking, and distributed execution. Without that foundation, larger systems only multiply the number of places where failures can hide.

11.6 Project Management Lessons

The team made the most progress when the project was scoped around a clear end-to-end path: load a trace, create a simulation, observe state, apply an action, compute reward, and log the result. This concrete loop gave the team a way to test each layer and avoid getting lost in open-ended research possibilities.

Progress was slowed by environment drift, long-running experiments, cloud setup, and ambiguous failures. Documentation helped reduce these issues. Runbooks, setup notes, architecture diagrams, and explicit command examples became part of the deliverable because they made the system easier for others to reproduce.

The project scope changed significantly over the year. The original focus was a minimum viable training arena. As the system matured, the scope expanded to include LLM benchmarking and distributed execution. At the same time, some ideas were narrowed or postponed, including production-style automation and more advanced distributed coordination.

In a future clinic project, we would define infrastructure readiness earlier, collect performance baselines sooner, and require every major feature to pass a one-step test on the target environment before scaling it. We would also centralize environment configuration earlier to avoid confusion around bucket names, namespaces, credentials, and machine-specific setup.

Chapter 12

Handoff and Reproducibility Guide

12.1 Who This Chapter Is For

This section details how anyone involved in ACRL or this project should view the documentation. From ACRL engineers to interns, new hires, and future clinic teams.

For the ACRL engineering team responsible for maintaining the simulation infrastructure and distributed training pipelines. It outlines the mechanics of the S3-based communication protocol where job manifests are dispatched to EC2 worker nodes for parallel execution. It also provides the necessary architectural context for integrating new Large Language Models (LLMs) into the `sim_mcp` benchmarking suite by subclassing the `LLMProvider` abstract base class.

For new developers onboarding onto the project, this section serves as a technical quick-start guide. It covers initializing the Python virtual environment via `setup_env.sh` and running single-step debug experiments (e.g., executing a single `bump_cpu` policy via `runner/one_step.py`) to understand how SimKube modifies trace files and interacts with the `virtual-default` namespace.

For the next cohort of Harvey Mudd College Clinic students, they will need

to understand the underlying Gymnasium wrapper (`env/simkube_gymenv.py`) to train new Reinforcement Learning (RL) agents. Importance should also be given to the `protocol/` directory, which they will need to maintain or expand for distributed training scenarios.

12.2 Finding Important Parts of the Repository

The repository strictly separates orchestration, observation, and infrastructure. The top-level directories include:

- `runner/`: Orchestrates the simulation steps, including `train.py` (the multi-episode RL loop) and `safeguards.py` (which validates resource limits).
- `env/`: Contains the environment wrapper (`sim_env.py`) and trace mutation operations (`actions/ops.py`).
- `observe/`: Handles state extraction from the cluster via `reader.py` and computes continuous or binary fitness via `reward.py`.
- `ops/`: Contains cluster preflight health checks and EC2 fleet management hooks (`ec2_workers.py`).
- `demo/`: Houses 100 generated execution traces (`.msgpack` and `.json`) used for training and evaluation.

The primary entry point for reinforcement learning is `runner/train.py`, which orchestrates the multi-episode training loop and handles checkpointing. The logic for the agents themselves, including the Deep Q-Network (DQN) and Epsilon-Greedy implementations, resides in the `agent/` directory.

LLM evaluation is driven by `benchmark/run.py`. This script pulls from the JSON scenario registry located at `benchmark/scenarios/index.json`. The connection between the LLM agents and the Kubernetes cluster is the MCP server, located at `sim_mcp/server.py`, which exposes real-time observability tools to the models. All logic pertaining to the multi-worker distributed architecture is housed in the `protocol/` directory. The most critical files here are `dispatch.py` for submitting jobs, `worker.py` for the EC2 polling loop, and `sync_server.py` for coordinating federated averaging across nodes.

A complete setup guide can be found in the repositories `README.md` Extensive internal system specifications are maintained in the `docs/` folder. Future operators should consult `WORKER_PROTOCOL.md` for the exact S3 JSON schema layout, and the `EC2_MULTI_WORKER_RUNBOOK.md` for commands to launch and terminate fleets from the custom SimArena AMI.

12.3 How to Run the Core System

To begin, clone the repository to your local machine. Source the environment using source `setup_env.sh` to activate the virtual environment and set the `PYTHONPATH`. Finally, install all necessary dependencies by running `pip install -r requirements.txt`.

Operators must ensure no ghost SimKube simulations remain from previous crashes. The recommended health check is running `kubectl get pods -A | grep kwok` to ensure the simulated control plane is healthy, followed by `kubectl delete simulations.simkube.io -all -n cluster` to purge stale resources.

Before running any experiments, it is important to execute `make preflight`. This verifies cluster health and ensures that SimKube CRDs are properly installed. Skipping this step often results in "ghost simulations," where lingering resources from previous runs interfere with new deployments.

To execute a single-step simulation for debugging or manual testing, use the following command:
`python runner/one_step.py -trace demo/traces/trace-0001.msgpack -ns cluster`.

Full reinforcement learning training is triggered via the main runner. Execute python `runner/train.py -agent dqn` to begin training. The system will automatically save `.pt` model checkpoints and learning curve logs to the `checkpoints/` directory at standard intervals. To initiate a full DQN training loop with auto-checkpointing, execute:

```
nohup python runner/train.py -trace demo/traces/trace-0001.msgpack
-ns cluster -deploy web -target 3 -agent dqn -episodes 50 &
```

To evaluate the default language model (`gemini-2.5-flash-lite`) against all failing trace scenarios, execute:

```
python benchmark/run.py --provider gemini --ns cluster
```

To test Anthropic's Claude 3.5 Sonnet, pass `-provider anthropic`. The harness records token latency, tool calls, and success rates, outputting a highly detailed `report.json` and `report.md`. Both of which (and a copy of the command) can be found in `benchmark/results/<timestamp>_<provider>_<model>/`

To utilize EC2, first export your `JOBS_BUCKET`. Submit a job manifest specifying the federation group:

```
python protocol/dispatch.py submit --bucket "$JOBS_BUCKET" \
--trace s3://diya-simarena-traces/demo/trace-mem-slight.msgpack \
--agent dqn --federation-group "$GROUP" --federation-size 2
```

Keep `python protocol/sync_server.py` running locally to act as the FedAvg coordinator, and trigger `python protocol/worker.py -run-once` on the EC2 instances.

12.4 Dependencies and Environment Assumptions

The environment's critical runtime dependencies include `torch` (for the Deep Q-Network), `gymnasium` (for the standard RL wrapper API), `kubernetes` (for control plane communication), `boto3` (for S3 and EC2 orchestration), and `mcp` (for the Model Context Protocol server). It also relies on `msgpack` for trace deserialization.

The system relies on a locally provisioned `kind` cluster with KWOK installed. SimKube dynamically creates pods in a virtual namespace directly derived from the trace; for example, a trace in the `default` namespace creates pods in `virtual-default`.

SimKube controllers must be active and accessible. Specifically, the system creates `simulations.simkube.io` Custom Resources. The preflight checks require the `sk-ctrl` processes to actively respond to trace generation requests within the specified duration window.

Scaling requires an S3 bucket configured for the standard job layout (`jobs/pending/`, `jobs/in_progress/`, `results/`). Workers must securely access this bucket either via attached EC2 IAM roles or by having

`AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` injected as environment variables.

For LLM evaluation, operators must copy `.env.example` to `.env`. Depending on the target provider, they must supply valid credentials to `GEMINI_API_KEY` (utilizing the `google-genai` SDK) or `ANTHROPIC_API_KEY`.

12.5 Known Issues and Caveats

The `runner/dist_run.py` script is currently a stub; it lacks an end-to-end implementation for `job_type=experience_collection`. Furthermore, the `TRAINING_SERVER_README.md` file acts as a design specification for a Flask-based telemetry dashboard, which is not yet built.

To prevent explosive resource consumption during random exploration, `runner/safeguards.py` imposes strict hard limits on agent actions. Valid actions are capped at maximums of 16000m CPU, 32Gi Memory, and 100 total replicas. The `max_punish` reward function mathematically penalizes the agent for attempting to exceed these boundaries.

System instability primarily stems from "ghost simulations." If a Python process is killed mid-execution (`SIGKILL`), the SimKube Simulation CR is orphaned, leaving hanging pods that pollute future observations. A hard purge (`kubectl delete simulations.simkube.io -all`) is frequently necessary. Additionally, a historical bug involving driver job failures attempting to mutate terminating namespaces continues to occasionally manifest during high-concurrency trace loads.

12.6 Recommended Next Steps

The current observation vector produced by `observe/reader.py` is limited to primary pod counts (e.g., ready, pending, total). Future developers should expand this state space to explicitly expose container state reasons (like `OOMKilled` or `CrashLoopBackOff`) to the RL agent, providing much richer context than binary phase counts.

The current framework restricts both RL and LLM agents to 7 discrete

actions, such as `bump_cpu_small` (+500m) or `scale_down_replicas`. Extending `env/actions/ops.py` to allow dynamic, continuous action inputs (e.g., specifying an exact percentage resource increase) would allow for far more granular infrastructure optimization.

The current LLM benchmark records latency, steps to solve, and tool calls, outputting to `report.json`. Moving forward, the benchmarking harness should penalize models that achieve target replica counts but wildly overprovision memory or CPU, integrating the `cost_aware_v2` reward concepts directly into the LLM evaluation summaries.

The most critical unaddressed technical debt is fully implementing `run_experience_collection (manifest)` within `runner/dist_run.py`. Wiring this to upload structured transition tuples to an S3 URI will enable asynchronous, decoupled replay buffer generation across large EC2 fleets.

With both LLMs and RL architectures sharing the exact same Gymnasium observation space and 7-action vector, the arena is perfectly positioned for strict A/B testing. Future teams should plot the learning curves and final convergence efficiency of the DQN directly against the zero-shot prompting capabilities of models like Claude 3.5 Sonnet.

Chapter 13

Recommendations and Future Work

13.1 Recommendations for ACRL

ACRL can immediately leverage the `agent/llm_agent.py` implementation alongside the FastMCP server. By exposing tools like `describe_deployment` and `get_events`, engineers can quantitatively benchmark how models like Gemini 2.5 Flash Lite or Claude 3.5 Sonnet handle complex, continuous Kubernetes failure states (such as `OOMKilled` vs `FailedScheduling` errors) autonomously.

The infrastructure backbone is mature. The Gymnasium integration (`env/simkube_gymenv.py`) allows any modern RL framework to wrap the simulator seamlessly. Similarly, the S3 job dispatch protocol and the `sync_server.py` aggregator are robust, having successfully passed end-to-end validation tests for multi-node gradient synchronization.

While the Deep Q-Network effectively validates the concept of a closed-loop K8s optimization gym, it remains prototype code. The network currently operates on a highly constrained 5-dimensional observation space and a 7-discrete-action space (`noop`, `bump_cpu_small`, `bump_mem_small`, `scale_up_replicas`, etc.). Extensive tuning is required before such an agent could be trusted in production environments.

Significant engineering time should be directed toward building out the Flask-based telemetry dashboard detailed in the `TRAINING_SERVER_README.md` design specifications. A central visual coordinator will drastically lower the overhead required to monitor federated multi-worker RL runs across AWS.

13.2 Short-Term Future Work

Effort should be directed to the `ops/hooks.py` pre-start and post-end logic to implement more rigorous programmatic cleanups. Currently, a KWOK control plane crash permanently stalls the polling `worker.py` loop on EC2 instances. Adding health timeouts to the driver will drastically improve distributed uptime.

The benchmarking scenarios registered in `benchmark/scenarios/index.json` currently classify problems into `cpu_insufficient`, `mem_insufficient`, and `replica_deficit`. Short-term work must expand this index to include multi-component failures, such as networking bottlenecks or persistent volume attachment limits.

The current startup flow requires manual verification of the cluster state via `make preflight` to avoid hanging resources. Automating this pre-flight sequence natively into the `runner/train.py` initialization sequence will prevent operators from accidentally launching simulations on unstable clusters.

To fully realize the distributed system design, developers should implement the Flask telemetry dashboard outlined in `TRAINING_SERVER_README.md`. Linking this interface to the `experience_collection` job types will provide operators with real-time visualization of remote agent exploration metrics.

13.3 Medium-Term Future Work

Future teams should expand `agent/agent.py` to support Proximal Policy Optimization (PPO) or advanced Sarsa variants. The current `cost_aware_v2` reward function is heavily shaped, providing continuous negative penalties for resource waste. PPO typically handles continuous, dense reward shaping more stably than a standard DQN architecture.

The current observation vector in `observe/reader.py` is simplistic, extracting primary pod counts (ready vs pending). Medium-term work should expand `state_dim` to ingest precise container resource utilization metrics, Kubelet disk pressure warnings, and API server latencies, allowing models to deduce failure causes beyond basic pod-phase transitions.

SimKube has the capability to model complex time-series behaviors. The arena should eventually grade agents on their ability to predict and remediate slow memory leaks over thousands of steps, rather than just rapidly addressing immediate out-of-memory (`OOMKilled`) crashes generated in the static `demo/traces/` library.

The framework currently registers `GeminiProvider` and `AnthropicProvider` in `agent/providers/`. To ensure industry-wide relevance, this factory should be expanded to include OpenAI's models as well as lightweight open-source counterparts (such as Llama 3) running locally to measure performance versus cost trade-offs.

13.4 Long-Term Research Directions

The trajectory of this project is to scale from single-deployment optimization loops into full cluster-level remediation. Instead of operating purely on explicitly defined applications (like the `web` deployment flag), an advanced agent would act as an autonomous drop-in replacement for the Vertical Pod Autoscaler and Cluster Autoscaler combined.

Sim-Arena's efficacy is bound by the quality of its trace files. While there are currently 100 generated traces in `demo/traces/`, deeper integration with synthetic trace generators (like `sk-gen`) would allow the environment to procedurally generate novel failure conditions on the fly during the RL training loops, preventing agent overfitting.

Moving policies learned within SimKube to a real production environment entails a significant sim-to-real gap. Research must be directed into creating formally verified safe-action wrappers—significantly more advanced than the current `runner/safeguards.py` bounds checking—to guarantee an LLM or RL agent cannot inadvertently destabilize a live cluster while attempting remediation.

The repository's 100 MessagePack execution traces (`demo/traces/`) and

problem categorizations (`cpu_insufficient`, `mem_insufficient`, `replica_deficit`) found `benchmark/scenarios/index.json` constitute a novel dataset. Long-term, this suite should be published to establish an open, standardized gymnasium for evaluating both RL heuristics and LLM tool-calling capabilities in cloud infrastructure remediation.

Chapter 14

Conclusion

14.1 Project Summary

This project addressed a growing challenge in modern cloud infrastructure: how to safely evaluate intelligent agents that attempt to diagnose and remediate Kubernetes workload issues. While Kubernetes is widely used in industry, it is highly complex, and small configuration mistakes involving CPU, memory, or replica counts can lead to pending pods, unstable services, or wasted resources. Testing automated fixes directly on production clusters is risky, expensive, and impractical.

To address this problem, the team developed Sim-Arena, a training and benchmarking platform built on top of SimKube. Sim-Arena provides a controlled environment where machine learning agents can interact with trace-driven Kubernetes simulations. Agents observe workload health, choose from a set of corrective actions, receive rewards based on outcomes, and learn how to solve the problem more effectively over multiple episodes.

The final system supports multiple approaches to agent decision-making. Reinforcement learning agents such as DQN baselines were implemented for iterative training, while large language model agents were evaluated through tool-based interfaces that expose cluster state. Because both approaches use the same environment and action definitions, Sim-Arena enables more meaningful comparison between different classes of intelligent systems.

In addition to local experimentation, the project also explored distributed execution through an S3-based job protocol and EC2 worker infrastructure. This helped reduce training bottlenecks and demonstrated a path toward larger experiment campaigns.

14.2 Final Assessment

The project successfully produced a functional prototype and research platform. By the end of the year, the team had built a system capable of loading traces, launching simulations, collecting observations, applying actions, computing rewards, and logging results. Supporting documentation, setup guides, and operational runbooks were also completed to improve long-term usability, open-source use, and sponsor handoff.

At the same time, several areas remain open for future work. Some distributed components remain prototype-level, and broader benchmarking across larger scenario sets would strengthen future evaluations. The current observation space and action space were intentionally simplified, meaning more state representations and more realistic interventions remain important next steps.

Despite these limitations, the project provides clear value. Sim-Arena is a functinal software platform that future teams can extend. It also establishes a shared evaluation framework for comparing reinforcement learning and LLM-based agents under consistent conditions. Most importantly, it gives ACRL a practical foundation for continued research in Kubernetes automation.

Acknowledgements

We would like to sincerely thank Applied Computing Research Labs (ACRL) for sponsoring this project and for providing the opportunity to work on a challenging and meaningful systems problem.

We are especially grateful to Dr. David Morrison for his guidance, technical insight, and continued support throughout the year. His feedback helped shape the direction and practical relevance of this work.

We also thank our faculty advisor, Prof. Erin Talvitie, for her mentorship, encouragement, and support throughout the Clinic process.

Finally, we thank every member of the team for their dedication, collaboration, and persistence throughout the year, as well as the broader open-source community whose tools and projects made this work possible.

Bibliography

[1] Applied Computing Research Labs. "SimKube: A Record-and-Replay Kubernetes Simulator". In: *ACRL Open Source Projects* (2024). Accessed on September 16, 2025. Available at https://simkube.dev/.

[2] Farama Foundation. *Gymnasium*. https://gymnasium.farama.org/. Accessed on May 5, 2026.

[3] Google and the Cloud Native Computing Foundation. "Kubernetes Documentation". In: *Kubernetes Project* (2014). Accessed on September 16, 2025. Available at https://kubernetes.io/docs/home/.

[4] Carlos E Jimenez et al. "SWE-bench: Can Language Models Resolve Real-world Github Issues?" In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=VTF8yNQM66.

[5] Saya Kim-Suzuki et al. "Graphing the Kubernetes State Space to Generate Synthetic Data". In: *Final Report for ACRL, Harvey Mudd College Clinic Program* (May 2025). Advisor: Professor Beth Trushkowsky, Liaison: Dr. David Morrison.

[6] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533. URL: https://doi.org/10.1038/nature14236.

[7] Alex Nichol et al. "Gotta Learn Fast: A New Benchmark for Generalization in RL". In: *arXiv preprint arXiv:1804.03720* (2018).

We acknowledge that we used LLMs throughout the writing process for reference, but the text included in this document is our own.