POLAR: AGENTIC RL ON ANY HARNESS AT SCALE

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning for language agents increasingly depends on custom harnesses that manage long-running context, multi-turn tool use and multi-agent orchestration. However, porting these harnesses into RL environment interfaces remains difficult and often loses important training signals. We bridge this gap with POLAR, a rollout framework for scalable asynchronous RL over arbitrary agent harnesses. POLAR treats the agent harness as a black box: it proxies LLM API calls, records token-level model interactions, and reconstructs token-faithful trajectories for training. Each rollout node efficiently manages runtime prewarming, agent execution, trajectory reconstruction, and evaluation in parallel, exposing asynchronous service endpoints that can be consumed by independent trainers at scale. This decoupled design makes POLAR agnostic to agent harnesses, training infrastructure, and RL algorithms while improving compute utilization for long-running agent workloads. We validate POLAR by training agents on software-engineering tasks with popular coding harnesses. Using simple GRPO, POLAR improves Qwen3.5-4B by 22.6, 4.8, 0.6 and 6.2 points on SWE-Bench Verified with the Codex, Claude Code, Qwen Code and Pi harnesses, respectively. We further demonstrate POLAR for offline data generation over custom harnesses and ablate trajectory reconstruction strategies.

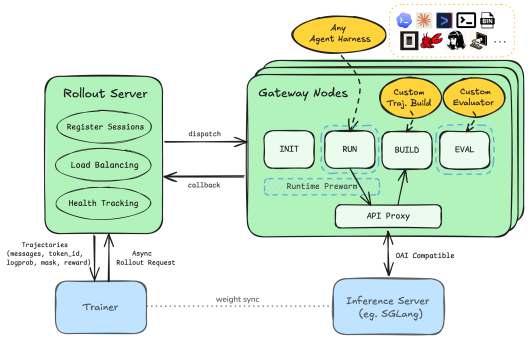


Figure 1: **POLAR architecture overview.** POLAR runs an existing agent harness inside an isolated runtime and places a model API proxy between the harness and the inference server. The proxy forwards model calls, records token-level request and response data, and reconstructs RL trajectories, while rollout gateways asynchronously handle runtime prewarming, harness execution, evaluation, and trainer callbacks. This decoupled design allows POLAR to treat agents as a black-box environment, seamlessly scaling across different training frameworks.

1 INTRODUCTION

Reinforcement learning for large language model is moving beyond short, single-step tasks toward agentic settings (Zhang et al., 2026a; Tan et al., 2025) that require sustained interaction with external environments, such as code repositories (Jimenez et al., 2024; Pan et al., 2024), web browsers (Zhou et al., 2023; Deng et al., 2023), and even full operating systems (Xie et al., 2024; Wang et al., 2025), through iterative tool use (Shao et al., 2024; Guo et al., 2025; Patil et al., 2025). These settings often produce long-horizon trajectories with dozens of interaction steps and tens of thousands of tokens.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

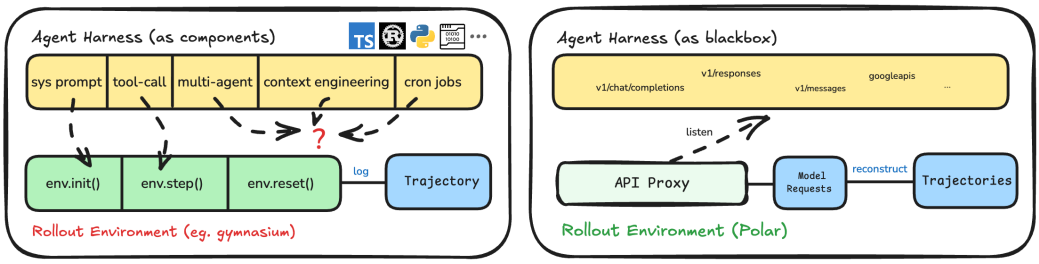


Figure 2: **POLAR uses the model API proxy as the rollout boundary.** Traditional rollout frameworks usually require the agent or harness logic to be rewritten behind a framework-owned environment API. This makes the trainer depend on harness-specific integration code and can miss details of the native execution path. POLAR instead keeps the harness unchanged and places a provider-compatible proxy at the LLM API boundary; the proxy records prompts, sampled tokens, log probabilities, and responses, then reconstructs trainer-ready trajectories outside the harness.

This shift makes the training target itself a central systems challenge for agentic RL. Traditional RL often assumes the training target can be exposed through a simple, standardized interface (Brockman et al., 2016), allowing researchers to focus mainly on the RL algorithm. In agentic RL, however, the training target is often a complex software system (OpenAI, 2026; Anthropic, 2026). It may involve heterogeneous environments (NVIDIA, 2025), various external tools (Zhang et al., 2026b; 2024), and long-running workflows (Jimenez et al., 2024), and may be implemented in different languages or even distributed as a closed-source binary (Anthropic, 2026).

This creates substantial integration burdens. For example, in building agentic RL systems, SkyRL-Agent (Cao et al., 2025) and PRIME-RL (Prime Intellect, 2026) integrate agent execution directly into the RL pipeline, requiring users to adapt their agents to the RL infrastructure rather than allowing the infrastructure to accommodate existing agent implementations. This makes the design less flexible: every new agent or harness often requires one framework-specific integration. Some recent systems attempt to make this integration less intrusive. For instance, Agent Lightning (Luo et al., 2025) and rLLM (Tan et al., 2025) reduce this burden by introducing standard tracing interfaces and LLM-call capture mechanisms, but still require agents to conform to prescribed interfaces. Thus, these systems lower the cost of integration but do not fully eliminate it. These issues are likely to become more severe as agent harnesses grow increasingly complex and, in some cases, even do not expose their internal implementation, making conventional RL integration difficult or even infeasible. Motivated by these issues, we explore the following central question:

Can we train agents with RL without opening the box?

That is, without touching their harnesses or forcing them to conform to an RL framework. The key observation is that, although agents differ widely in their internal implementations, every LLM-based agent must talk to a model. This model API boundary provides a common interface that exists outside the agent itself. Instead of integrating with the agent harness, we can train by listening to the agent's LLM calls: capturing its prompts, sampled tokens, log probabilities, and responses, and converting them into RL trajectories. In this view, an agent can be treated as a black box while still becoming trainable.

Building on this intuition, we present POLAR: an agentic RL infrastructure that could train **any** agents as black boxes. The name POLAR reflects both its roots in **PrOrL A**gent serv**R** (Zhang et al., 2026a) and its role in connecting the two "poles" of agent training and deployment: the training environment and the product harness. Instead of treating the agent harness as the RL interface, POLAR uses the agent's LLM API traffic as the interface. Through listening to its model calls through a proxy and converts them into trajectories and rewards for training, the agent runs unchanged.

In addition, POLAR separates runtime setup, agent execution, trajectory reconstruction, evaluation, and trainer callbacks behind asynchronous service boundaries. This allows slow and long-tail agent rollouts to scale independently from GPU training, exposing a trainer-agnostic rollout-as-a-service

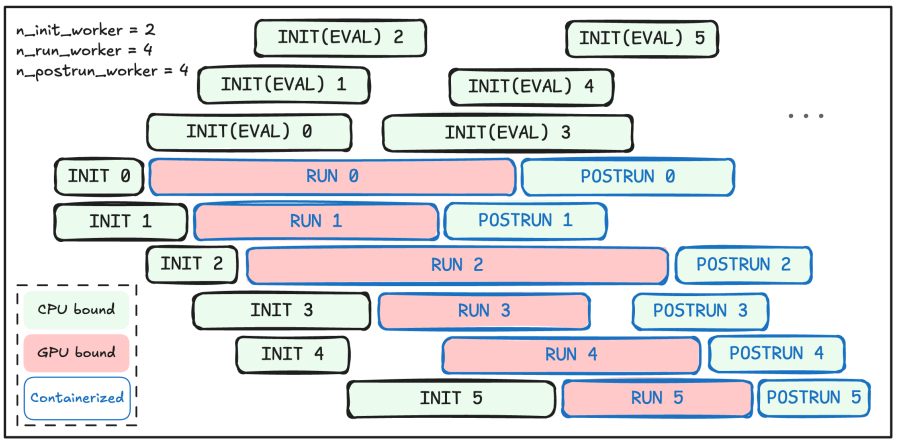


Figure 3: **Gateway-level asynchronous staging in POLAR.** A gateway separates runtime initialization, ready buffering, harness execution, and post-run trajectory and evaluation work into isolated worker pools. Runtime preparation and evaluator prewarm proceed off the critical path, so CPU-heavy runtime setup and long-tail evaluation do not block active GPU-bound agent run.

interface for scaling efficient RL infrastructures (Zhang et al., 2026a). In summary, the main contributions of this work are:

- **Proxy based rollout and reconstruction over agent harnesses.** We propose a paradigm using the agent's LLM API payloads as the RL rollout interface, allowing existing harnesses to serve directly as RL environments without internal code change.
- **Rollout-as-a-service architecture for scaling RL infrastructures.** POLAR separates task submission, runtime setup, harness execution, trajectory reconstruction, evaluation, and trainer callbacks behind asynchronous service boundaries, natively scaling with modern RL infrastructures.
- **Token-faithful trajectory reconstruction.** POLAR converts raw model requests into token-faithful traces for training. We provide conservative per-request reconstruction and prefix merging for heavy rolllouts, while leaving registry-based extensible interfaces.
- **End-to-end validation on real-world coding harnesses.** We validate POLAR with RL training on various popular harnesses for software-engineering tasks, and further demonstrate offline SFT data generation with a custom coding harness.

2 POLAR

We target agentic RL tasks where a policy is exercised through an existing harness rather than a custom rollout loop. A task starts from an instruction and a runtime; the harness calls a model endpoint while using tools, editing files, spawning sub-agents, or managing context (compaction, injection, replacement, etc.). After execution, an evaluator assigns an outcome or trace-level reward. The rollout system must preserve the native model interactions as trainer-ready traces: prompt context, sampled assistant tokens, optional behavior-policy log probabilities, loss masks, rewards, and provenance.

2.1 ARCHITECTURE

POLAR has two core components: a rollout server and gateway nodes. The rollout service coordinates tasks and global scheduling. Gateway nodes execute sessions, host the model proxy, construct trajectories, and run evaluation. This split keeps durable task management separate from per-session execution and capture.

Rollout server. The rollout service accepts a `TaskRequest` and expands it into `num_samples` independent sessions. A session is the scheduling unit: it has a session ID, task ID, timeout budget, runtime specification, agent specification, trajectory builder, evaluator, and callback URL. The service dispatches sessions to gateway nodes, persists compact terminal results, exposes task status through polling, and accepts gateway callbacks when sessions finish. Sec. B.7 gives a representative payload.

Gateway node. A gateway owns the lifecycle of each session. It starts the runtime, prepares the harness, runs the harness commands, builds trajectories from captured completions, evaluates the output, tears down resources, and returns the result. The same gateway also hosts the proxy endpoint used by the harness for model calls. This co-location keeps completion capture tied to the session registry and avoids a separate trace-collection service.

Training frameworks are independent from POLAR servers. And the service boundaries natively supports efficient asynchronous RL at scale. Fig. 5a shows one such example with Slime: a background worker submits POLAR tasks, receives task-completion callbacks, converts traces into Slime `Sample` objects, and applies trajectory-aware reward post-processing.

2.2 HARNESS AND PROXY CAPTURE

POLAR observes native harnesses by routing their model calls through the gateway proxy. A harness is configured through its normal environment variables or config files so that its model base URL points to the gateway.

For each incoming model request, the gateway performs four steps.

1. **Detect the provider API.** Detection uses the request path and headers to distinguish Anthropic Messages, OpenAI Chat Completions, OpenAI Responses, and Google `generateContent`-style calls.
2. **Normalize the request.** A provider transformer converts roles, content parts, tool definitions, tool choices, stop controls, and generation parameters into the OpenAI Chat Completions shape consumed by local inference servers. The transformer also adds fields needed for training, such as `logprobs=true`.
3. **Capture token-level data.** The gateway forwards the normalized request to the inference servers and stores a completion record containing the request messages, response messages, prompt token IDs, sampled response token IDs, finish reason, and log probabilities from inference backends.
4. **Return the provider shape.** The response is transformed back to the schema expected by the harness. For streaming requests, our implementation obtains a non-streaming upstream response and emits a synthetic provider-shaped stream. This simplifies faithful token capture while preserving compatibility with harnesses that expect server-sent events.

The proxy boundary is intentionally below the agent framework. It does not need to understand how the harness plans, manages tools, or decides when to stop. It only needs to preserve API compatibility and record enough information to reconstruct training samples.

2.2.1 HARNESS ADAPTER

A harness adapter in POLAR is small by design. It may install configuration, register MCP servers or skills, write provider settings, and return the shell commands that run the agent. A generic `shell` command harness can be used for wrapped agent execution. We also integrate popular agent harnesses as shortcuts like `claude_code`, `codex`, `gemini_cli`, `qwen_code`, `opencode` and `pi`.

2.2.2 RUNTIME INTERFACE

Runtimes implement a common interface for `start`, `stop`, `exec`, upload, download, and cancellation. Our first release supports Docker and rootless Apptainer for HPC setup. Because gateway code only depends on the runtime interface, a task can change isolation backend without friction.

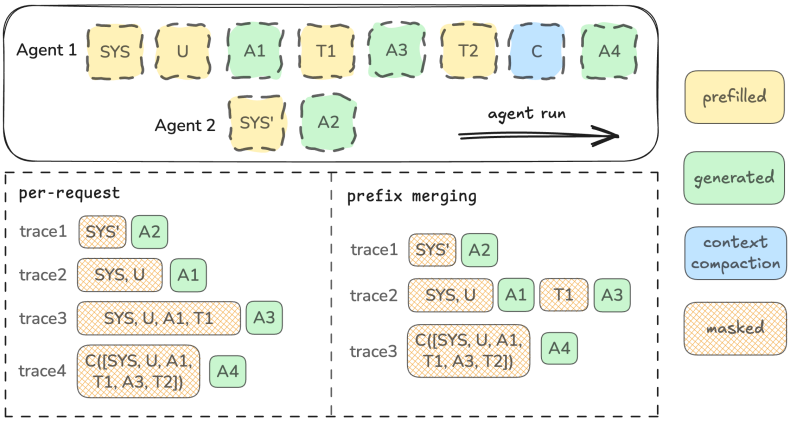


Figure 4: **Trajectory reconstruction example.** The visualized session contains a three-turn main agent that undergoes one harness-level context compaction and spawns one subagent. The per-request builder keeps each captured model call as an independent trace. Prefix merging instead recovers append-only conversation chains where valid, while compaction and subagent boundaries naturally form separate chains. Within each merged trace, POLAR prefix merging algorithm copies only sampled assistant tokens as trainable tokens and masks canonical interstitial tokens, preserving behavior-policy fidelity while reducing trainer-facing samples.

2.3 ASYNCHRONOUS ROLLOUT STAGING

Long-horizon harness rollouts mix several different costs: runtime startup, dependency preparation, harness execution, evaluator setup, test execution, patch application, and teardown. POLAR keeps these costs from blocking one another through stage-isolated execution inside each gateway (Fig. 3).

2.3.1 IN-NODE WORKER POOLS.

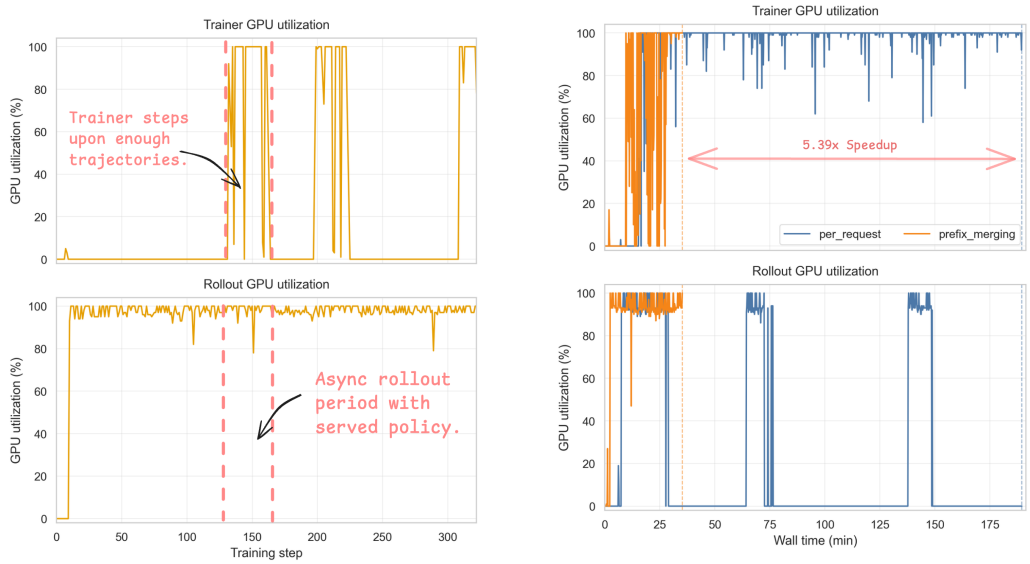
Each gateway uses isolated worker pools for `INIT`, `RUNNING`, and `POSTRUN`, plus a bounded `READY` buffer. `INIT` starts the runtime and executes prepare actions. `READY` holds initialized runtimes until a run slot is available. `RUNNING` executes the harness. `POSTRUN` builds trajectories, runs evaluators, executes post-run hooks, sends callbacks, and tears down resources. The ready buffer allows CPU-heavy runtime preparation to proceed in the background without blocking GPU-bound agent execution.

2.3.2 EVALUATOR PREWARM AND TIMEOUTS.

When an evaluator requests a clean runtime, the gateway begins preparing that runtime during the agent run. Each session also carries one shared deadline; if a harness times out after model calls have been captured, the gateway still enters post-run so partial traces can be recovered with terminal timeout status.

2.4 TRAJECTORY RECONSTRUCTION

The trajectory builder interface converts an ordered `CompletionSession` into a `Trajectory`. A completion session is the stored sequence of proxy-captured model calls for one harness session. A trajectory contains one or more `Trace` objects, each with prompt token IDs, response token IDs, a loss mask, prompt messages, response messages, tool definitions, log probabilities, reward, and metadata. Sec. B.8 shows a representative trainer-facing trace. Custom trajectory strategies can be seamlessly added to the registry, and we provide two strategies, per request and prefix merging, compared in Fig. 4.



(a) Async RL.

(b) GPU utilization with different reconstruction strategies.

Figure 5: **POLAR improves GPU utilization across the rollout-training boundary.** *(a)* Shows an asynchronous RL pipeline enabled by POLAR services. The rollout server keeps inferencing with existing policy, while trainer steps only if receiving batch size of evaluated trajectory groups. *(b)* Shows a span of 3 training steps under the same workload and topology, prefix merging emits fewer trainer updates than per-request reconstruction and substantially accelerates the training process.

2.4.1 PER REQUEST

The `per_request` builder is the conservative baseline as shown in the bottom left of Fig. 4: every completion becomes one trace. This is lossless with respect to individual calls, but it can fragment a coherent multi-turn agent session into many short samples. For complex coding harnesses, solving a single coding problem can produce hundreds of such traces, which increases the burden on downstream trainers.

2.4.2 TOKEN FAITHFUL PREFIX MERGING

The `prefix_merging` builder reconstructs longer traces when parts of a harness session preserve append-only conversation histories as shown in the bottom right of Fig. 4. It does not assume that the whole session is a single conversation. Instead, for a session with completions $C_1, \ldots, C_T$, where completion C_i has prompt token sequence p_i, raw sampled response token sequence a_i, response log probabilities ℓ_i, and prompt/response messages m_i, POLAR partitions the completions into ordered chains

$$\mathcal{G} = \{G_1, \ldots, G_J\}, \qquad G_j = (C_{i_1^j}, C_{i_2^j}, \ldots, C_{i_{K_j}^j}),$$

with $i_1^j < i_2^j < \cdots < i_{K_j}^j$. A new completion can join an existing chain only when a normalized message-level grouping key identifies it as a candidate continuation and the strict token-prefix relation holds against the last prompt in that chain. For adjacent completions C_{i_m} and $C_{i_{m+1}}$ inside one chain, this check is

$$p_{i_{m+1}}[1 : |p_{i_m}|] = p_{i_m}.$$

Thus sub-agents, parallel agent branches, context compaction, prompt rewriting, or independent tool-mediated conversations naturally form additional chains rather than being forced into one global trace.

Merging is then applied independently to each chain. Consider one chain $G = (C_{i_1}, \ldots, C_{i_K})$ and write $p_m = p_{i_m}$, $a_m = a_{i_m}$, and $\ell_m = \ell_{i_m}$. The main challenge is that p_{m+1} contains a canonical

server rendering of the previous assistant turn plus the interstitial context inserted by the harness before the next generation prompt. The previous assistant body must not be copied from this canonical rendering, because the behavior-policy tokens are the raw sampled tokens a_m. Let e denote the end-of-turn token ID. For two adjacent completions in the chain, define the canonical tail

$$t_m = p_{m+1}[|p_m| + 1 :].$$

We locates the first e in t_m. If a_m already ends with e, the interstitial u_m is the suffix after that e; otherwise u_m starts at that e so the assistant turn is still closed before the next prompt context. The token sequence represented by this chain is

$$z^{(j)} = p_1 \,||\, a_1 \,||\, u_1 \,||\, a_2 \,||\, u_2 \,||\cdots||\, a_K.$$

The emitted trajectory therefore contains one trace $\tau^{(j)}$ per chain, with the first prompt p_1 stored as the trace prompt and the remaining suffix $a_1||u_1||\cdots||a_K$ stored as the trace response. The explicit loss mask is one on tokens copied from sampled responses a_m and zero on tokens copied from canonical interstitials u_m. Real response log-probability entries are copied for a_m tokens. Interstitial slots receive synthetic log-probability entries so `response_logprobs` stays aligned with `response_ids`; trainability is controlled by `loss_mask`.

This construction gives a simple correctness invariant in every emitted trace:

Every trainable token matches the behavior policy during rollout, and any non-generated tokens are masked out.

Fig. 5b compares the GPU utilizations of the 2 strategies above with the same configurations.

2.5 Evaluation and Reward Propagation

Evaluators are registry-backed custom strategies that run after trajectory construction. They receive the trajectory, session artifacts, and optionally refreshed runtime context. Built-in evaluators include a session-completion reward, a configurable test-on-output evaluator, and a SWE-Bench/SWE-Gym harness evaluator. An outcome reward can be broadcast to every trace, whereas tasks with process rewards may need per-trace assignment. The evaluator registry allows straightforward extension to custom rule-based verification, agent-as-judge scoring, and task-specific reward shaping.

3 Experiments

We validate POLAR in two settings: online RL rollout and offline SFT data generation. The experiments test whether unchanged harnesses can produce trainable traces for both reward-driven and supervised training.

3.1 SWE-Gym GRPO on Coding Harnesses

We run standard GRPO over four representative coding harnesses with POLAR. Starting from the same Qwen3.5-4B base checkpoint, we run standard GRPO training on the `SkyRL-v0-293-data` SWE-Gym dataset,[1] with POLAR and Slime. We use the training split for policy optimization and reserve evaluation for SWE-Bench Verified (Jimenez et al., 2024). All experiments use `prefix_merging` to convert llm traffics from harnesses into trainable traces. And we use `swebench_harness` to score the final edition patch in a fresh runtime. The main training hyperparameters are listed in Tab. 4.

Fig. 6 shows the training reward for Codex, Claude Code, Qwen Code, and Pi. The Codex run begins near zero reward and rises steadily over training, with the last ten steps averaging 54.5% pass@1 reward compared with 9.5% over the first ten steps. Claude Code also improves substantially, rising from 28.8% over the first ten steps to 67.0% over the last ten steps. Qwen Code and Pi start from stronger native-harness priors: Qwen Code is noisier but rises from 61.6% to 66.0%, while Pi improves more clearly from 61.6% to 76.2% over the same first and last ten-step windows. These curves are consistent with the benchmark result in Tab. 1: POLAR improves the same 4B base model

[1] `https://huggingface.co/datasets/NovaSky-AI/SkyRL-v0-293-data`

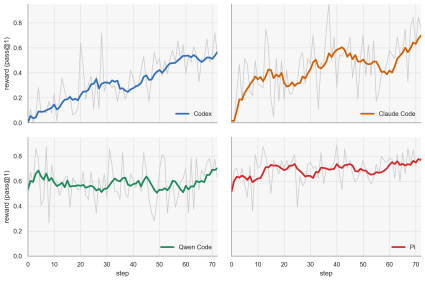


Figure 6: **SWE-Gym GRPO training curves.** Each panel shows the per-step outcome reward, equivalent to rollout pass@1, for one of four evaluated coding harnesses. RL improves reward across harnesses, with the clear gains on execution paths involving complex prompting, orchestration, or unfamiliar tool schemas.

Harness	Base	POLAR RL	Gain
Codex	3.8%	**26.4%**	**22.6** ↑
Claude Code	29.8%	**34.6%**	**4.8** ↑
Qwen Code	34.6%	**35.2%**	**0.6** ↑
Pi	34.2%	**40.4%**	**6.2** ↑

Table 1: **SWE-Bench Verified evaluation.** All rows start from the same Qwen3.5-4B base model and are trained over listed harnesses. Scores are pass@1 over the full benchmark, running on corresponding harnesses.

under all four evaluated harnesses. The largest absolute gain appear in Codex, likely due to unfamiliar tool schemas.

The evaluation isolates the value of harness-native RL. Under Codex, the 4B base model reaches only 3.8% pass@1 before training, but the POLAR-trained checkpoint reaches 26.4%, a 22.6 point absolute gain. This large jump is expected: Codex presents an unfamiliar action protocol, context policy, and patch-submission style to a Qwen model that was not originally trained as a Codex-native policy. POLAR keeps that harness unchanged and attaches the reward to the actual sampled tokens flowing through the Codex execution path, so GRPO optimizes the behavior the model must use at evaluation time. Under the native Qwen Code harness, the base model is already much stronger at 34.6%, and POLAR still improves it to 35.2%, a 0.6 point gain. Claude Code also improves from 29.8% to 34.6%, adding 4.8 points, and Pi improves from 34.2% to 40.4%, adding 6.2 points. These results show that harness-native RL can deliver large adaptation gains for unfamiliar execution paths while still preserving gains when the base checkpoint is already well aligned with the harness.

Trajectory builder ablation. We ablate the trajectory builder under identical model, hardware, and topology settings, changing only whether captured completions are emitted as `per_request` traces or merged by `prefix_merging`. Fig. 5b shows a partial utilization profile. Over same three training steps, `prefix_merging` reduces the trainer stream from 1,185 request-level updates to 218 merged-trace updates, cutting wall-clock time from 189.5 to 35.2 minutes (5.39×). `prefix_merging` keeps rollout GPUs active with 87.7% average rollout utilization, compared with 20.4% average utilization for `per_request` over the same period.

We also tried `per_request` with outcome-reward broadcasting to every emitted trace, but observed significant reward hacking. The issue is noisy credit assignment: request-level traces can receive session-level credit without proper session normalization or an advanced process reward model. Those mechanisms are outside the scope of this work, but providing examples and tools for session normalization and PRM-style credit assignment is on our roadmap.

3.2 OFFLINE DATA GENERATION

Beyond serving online RL rollouts, POLAR can be repurposed as a distributed *offline* data-generation service: a fixed checkpoint and harness are fanned out across the cluster, every session is journaled to disk, and the resulting traces are filtered and post-processed for downstream training. The same primitives that make POLAR useful for RL—per-session container isolation, automatic retry, and gateway-mediated scheduling.

Case study: SWE-Gym SFT trajectories. We used POLAR to generate a supervised fine-tuning corpus of agentic software-engineering trajectories. The setup is intentionally minimal: a single 8×H100 SGLang serve job hosting Qwen3.5-122B-A10B (TP=8, `max_model_len`=32,768) drives the `pi-coding-agent` v0.67.68 harness against 1,638 instances drawn from seven SWE-Gym repositories. Each task runs in its own Apptainer SIF built from the SWE-Gym reference image

Repo	Attempts	Accepted	Rate
getmoto/moto	343	184	53.6%
python/mypy	257	101	39.3%
conan-io/conan	71	27	38.0%
pydantic/pydantic	81	24	29.6%
iterative/dvc	219	45	20.5%
pandas-dev/pandas	477	98	19.7%
dask/dask	141	25	17.7%
Total	**1,638**	**504**	**30.8%**

Table 2: Per-repository acceptance rates for SFT data generated by POLAR with Qwen3.5-122B-A10B and the `pi` harness on SWE-Gym. "Accepted" means the agent's patch passed both `FAIL_TO_PASS` and `PASS_TO_PASS` tests in the SWE-Bench evaluator.

with Node.js 22 and the harness layered on top, so the agent's tool calls (`bash`, `read`, `edit`, `write`) execute against a fresh checkout of the target commit. Submission uses `max_concurrent`=5–8, `max_retries`=1, and a per-task timeout of 3,600 seconds; trajectories that finished with `empty_generation` are retried once and the rest accepted as-is.

A trajectory is accepted into the SFT corpus if and only if the SWE-Bench evaluation harness reports the agent's final patch as resolving every `FAIL_TO_PASS` test while leaving every `PASS_TO_PASS` test green. With this single-bit filter, POLAR produced **504 accepted trajectories from 1,638 attempts** (30.8% acceptance), at a cost of roughly 64 GPU-hours on the interactive partition. Per-repository acceptance varies substantially with task difficulty (Table 2): bug-fix heavy repositories like `getmoto/moto` accept at over 50%, while data-frame and dataflow workloads with longer test suites accept below 20%.

Released format. Each accepted row contains the SWE-Gym instance metadata (`instance_id`, `repo`, `problem_statement`, `base_commit`, `version`) and the full multi-turn conversation as a list of OpenAI-style messages with `role`, `content`, `tool_calls`, and `tool_call_id` fields, terminated by the assistant turn that produced the accepted patch. Trajectories are long: an average of 104 messages per session and 51 assistant turns, with a long tail above 200 turns. The corpus is released as a HuggingFace dataset under an Apache-2.0 license, with a 90/10 train/test split stratified by repository so that every repo is represented in both splits.

We deliberately kept the filter narrow—a single binary verifier from the existing SWE-Bench harness—to keep the case study reproducible. The same POLAR deployment can be re-used for richer offline pipelines without changing the runtime: rejection sampling falls out of running multiple completions per prompt and keeping only those that pass the verifier; verifier-training data falls out of retaining the rejected trajectories alongside the accepted ones; preference data falls out of pairing accepted and rejected traces from the same prompt. Scaling the present run to the full 2,438-instance SWE-Gym set, swapping in stronger teachers, or adding additional harnesses (e.g. `codex` or `claude_code`) requires no changes to the orchestration code—only additional submitter shards and the corresponding checkpoint.

4 CONCLUSION

POLAR treats agent test-time environments as a first-class part of the RL system rather than an implementation detail to be ported into the trainer. Its central design choice is to move the integration boundary to the model endpoint: the harness runs normally, the proxy observes token-level model traffic, and the rollout service turns completed executions into trainable trajectories and rewards. This separation lets rollout scale independently from training and inference, while preserving the behavior of non-standard harnesses whose value often lies in their engineering details. We believe POLAR opens a new paradigm for scaling agentic RL infrastructure in the modern era, and we are actively developing and maintaining the framework as the ecosystem evolves.

AI USE STATEMENT

Generative AI tools assisted with drafting, language editing, and LaTeX formatting. The authors retain responsibility for research decisions, verification, and the accuracy of the manuscript and its reported findings.

ETHICS STATEMENT

Experiments with POLAR use public models and software-engineering benchmarks, without human-subject studies or collection of sensitive personal data. Training capable coding agents can nevertheless enable unintended or harmful actions. Deployment should therefore combine isolated execution and outcome verification with access restrictions and safeguards appropriate to the intended application.

REPRODUCIBILITY STATEMENT

POLAR's architecture, trajectory reconstruction, and experimental procedures are described in the main text. Appendix sections B.6, B.7, and B.8 supply training hyperparameters and representative task and trace formats. Section 3.2 details the offline generation configuration, filtering criteria, and released data format needed to reproduce that workflow.

REFERENCES

Agent Lightning Team. No more retokenization drift: Returning token ids via the openai compatible api matters in agent rl. https://blog.vllm.ai/2025/10/22/agent-lightning.html, 2025. URL https://blog.vllm.ai/2025/10/22/agent-lightning.html. vLLM blog.

Anthropic. Claude code. https://www.anthropic.com/product/claude-code, 2026. Accessed: 2026-05-06.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Shiyi Cao, Dacheng Li, Fangzhou Zhao, Shuo Yuan, Sumanth R. Hegde, Connor Chen, Charlie Ruan, Tyler Griggs, Shu Liu, Eric Tang, et al. Skyrl-agent: Efficient rl training for multi-turn llm agent, 2025. URL https://arxiv.org/abs/2511.16108.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments. https://github.com/harbor-framework/harbor, 2026. URL https://github.com/harbor-framework/harbor. GitHub repository.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Xufang Luo, Yuge Zhang, Zhiyuan He, Zilong Wang, Siyun Zhao, Dongsheng Li, Luna K. Qiu, and Yuqing Yang. Agent lightning: Train any ai agents with reinforcement learning, 2025. URL https://arxiv.org/abs/2508.03680.

NVIDIA. Nemo gym: An open source library for scaling reinforcement learning environments for llm. https://github.com/NVIDIA-NeMo/Gym, 2025. GitHub repository.

OpenAI. Codex: Ai coding partner from openai. https://openai.com/codex/, 2026. Accessed: 2026-05-06.

OpenClaw-RL Contributors. Openclaw-rl: Scalable rl in real-world agentic settings, 2026. URL https://arxiv.org/abs/2603.10165.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym, 2024. URL https://arxiv.org/abs/2412.21139.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=2GmDdhBdDk.

Prime Intellect. Prime-rl: Async rl training at scale. https://github.com/PrimeIntellect-ai/prime-rl, 2026. URL https://github.com/PrimeIntellect-ai/prime-rl. GitHub repository.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Sijun Tan, Michael Luo, Colin Cai, Tarun Venkat, Kyle Montgomery, Aaron Hao, Tianhao Wu, Arnav Balyan, Manan Roongta, Chenguang Wang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. rllm: A framework for post-training language agents. https://github.com/rllm-org/rllm, 2025. URL https://github.com/rllm-org/rllm. GitHub repository.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, et al. Opencua: Open foundations for computer-use agents. *arXiv preprint arXiv:2508.09123*, 2025.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Hao Zhang, Mingjie Liu, Shaokun Zhang, Songyang Han, Jian Hu, Zhenghui Jin, Yuchi Zhang, Shizhe Diao, Ximing Lu, Binfeng Xu, Zhiding Yu, Jan Kautz, and Yi Dong. Prorl agent: Rollout-as-a-service for rl training of multi-turn llm agents, 2026a. URL https://arxiv.org/abs/2603.18815.

Shaokun Zhang, Jieyu Zhang, Jiale Liu, Linxin Song, Chi Wang, Ranjay Krishna, and Qingyun Wu. Offline training of language model agents with functions as learnable weights. In *Forty-first International Conference on Machine Learning*, 2024.

Shaokun Zhang, Yi Dong, Jieyu Zhang, Jan Kautz, Bryan Catanzaro, Andrew Tao, Qingyun Wu, Zhiding Yu, and Guilin Liu. Nemotron-research-tool-n1: Exploring tool-using language models with reinforced reasoning. In *The Fourteenth International Conference on Learning Representations*, 2026b. URL https://openreview.net/forum?id=yiE16lWzDj.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Huang, Jeff Sun, Chao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, et al. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

Zilin Zhu, Chengxing Xie, Xin Lv, and Contributors. slime: An llm post-training framework for rl scaling. https://github.com/THUDM/slime, 2025. URL https://github.com/THUDM/slime. GitHub repository.

A Appendix

B Related Work

A compact checklist of rollout-system design choices is provided in Tab. 3 in the appendix. This section focuses on the qualitative differences behind that comparison.

B.1 Agent RL Systems

The first wave of LLM RL infrastructure largely assumed that rollout generation was a Python function owned by the trainer. This assumption is increasingly strained by multi-turn agents, where interaction spans many model calls and environment actions. ProRL Agent (Zhang et al., 2026a) introduced a service boundary for multi-turn agent rollouts, separating sandbox setup, agent execution, and reward computation from the training process. POLAR inherits the same high-level idea that rollout should be a service, but changes the integration contract. Instead of implementing an agent handler inside the rollout service, the user supplies a harness adapter that prepares configuration and launches the native executable. The model proxy then observes the harness from outside.

SkyRL-Agent (Cao et al., 2025) is a full-stack system for efficient RL training and evaluation of multi-turn, long-horizon agents, with SkyRL-Gym providing tool-use environments through a Gymnasium-style interface. SkyRL's strength is efficient training once tasks are represented in its environment and agent abstractions. POLAR is complementary: it targets the earlier systems problem of running a pre-existing harness whose internal event loop, tool formatting, and context policy should remain unchanged.

PRIME-RL (Prime Intellect, 2026) focuses on large-scale asynchronous RL with trainer-inference separation, stale-policy step semantics, and support for verifiers environments. Slime (Zhu et al., 2025; Zheng et al., 2024) similarly connects Megatron training with SGLang rollout engines and exposes customizable data-generation interfaces. These systems address the policy-optimization and inference-scaling side of the pipeline. POLAR is not a replacement trainer. It is a rollout substrate that can feed asynchronous trainers with trajectories from heavier harnesses than typical verifiable-reward functions.

B.2 Low-Intrusion Agent Instrumentation

Agent Lightning (Luo et al., 2025) proposes a training-agent disaggregation architecture and a unified data interface for converting agent execution into trainable transitions. rLLM (Tan et al., 2025) similarly aims to train agents across frameworks with minimal code changes, using tracked clients, decorators, workflow abstractions, and proxy support to collect token IDs and log probabilities. Both systems recognize that researchers should not have to rewrite complete applications to train them.

POLAR differs in the chosen minimum integration point. For many coding and terminal agents, the most reliable interface is not an SDK callback graph but the provider API endpoint already used by the harness. The gateway proxy therefore becomes the observation device: it accepts Anthropic, OpenAI Chat, OpenAI Responses, and Google-style requests; translates them to the local inference backend; and records the token-level fields needed by the trainer. This choice is narrower than general observability instrumentation, but it is robust to harnesses implemented as command-line programs, package-managed tools, or binaries.

B.3 SWE Task Evaluation and Benchmark

Harbor (Harbor Framework Team, 2026) evaluates agents such as Claude Code, OpenHands, Codex CLI, and related systems in containerized environments, supports parallel execution through local and cloud providers, and converts native agent logs into evaluation trajectories. This evaluation-first design is highly aligned with POLAR's harness-native motivation. The difference is the model boundary and training data contract. Harbor launches each harness with provider-specific configuration and does not provide a gateway that translates model-provider protocols or mediates the harness's model traffic. As a result, model substitution is limited by what the native harness and external endpoint already support: for example, evaluating a Qwen checkpoint through Claude Code requires an

Anthropic-compatible endpoint outside Harbor. POLAR instead places a proxy at this boundary, so the same style of harness execution can yield token IDs, log probabilities, loss masks, and rewards that are directly consumable by an RL trainer.

SWE-bench (Jimenez et al., 2024) established real GitHub issue resolution as a benchmark requiring repository understanding, editing, and executable validation. SWE-Gym (Pan et al., 2024) extends this direction with training environments, verifiers, and trajectories for software-engineering agents. These workloads are a natural stress test for rollout infrastructure because they combine expensive runtime setup, sparse patch-level rewards, long-tail execution time, and many opportunities for harness-side state to diverge from a clean evaluator state.

B.4 TOKEN FIDELITY AND RETOKENIZATION DRIFT

The training signal in agent RL is only correct if it is attached to the tokens sampled by the behavior policy. This is difficult in agent harnesses because provider APIs may return text, tool-call JSON, reasoning fields, or streamed events rather than the exact token IDs and log probabilities used by the inference backend. The vLLM and Agent Lightning discussion of retokenization drift emphasizes that decoding and re-encoding a transcript can produce different token IDs from the original generation (Agent Lightning Team, 2025). POLAR follows the same token-fidelity principle but applies it to arbitrary harness rollouts: generated assistant tokens are copied from inference responses, non-generated interstitial tokens are taken from canonical prompt tokenization, and the loss mask marks only behavior-policy tokens as trainable.

B.5 FRAMEWORK COMPARISON

System	Async RL Support	Async Rollout Staging	Rollout as Service	Agent Harness Agnostic
POLAR	✓	✓	✓	✓
PRORL AGENT (Zhang et al., 2026a)	✓	✓	✓	✗
SkyRL-Agent (Cao et al., 2025)	✓	✓	✗	●
PRIME-RL (Prime Intellect, 2026)	✓	✗	✗	✗
Agent Lightning (Luo et al., 2025)	●	✗	●	●
rLLM (Tan et al., 2025)	●	✗	✗	✗
OpenClaw-RL (OpenClaw-RL Contributors, 2026)	✓	✗	✗	●

Table 3: **Comparing rollout-system design choices.** We find that modern rollout infrastructures should meet following criteria: async RL support means training can consume rollouts while generation continues under explicit policy-version or staleness handling; async rollout staging means rollout execution is decomposed into independently scheduled runtime-preparation, execution, post-run reconstruction/evaluation, and cleanup stages; rollout as service means a durable task API that is separable from a specific trainer loop; and native-harness agnosticism means a CLI, SDK, or application harness can be trained without being reimplemented as the framework's environment. ✓ denotes first-class support, ● denotes partial or planned support, and ✗ denotes that we did not find the property as a primary design contract in the referenced code or documentation.

The partial marks in Tab. 3 avoid treating adjacent mechanisms as absent. SkyRL exposes custom generators and Harbor integration, but native harnesses are not the default unit of rollout. Agent Lightning provides a rollout store, queue, runner control plane, and broad framework instrumentation, while its main boundary is trace/workflow observability rather than staged execution of opaque harness processes. rLLM includes fully asynchronous training and a model gateway that captures token IDs and log probabilities, but its rollout service abstraction is narrower than a distributed runtime lifecycle service. OpenClaw-RL decouples serving, rollout collection, judging, and training for real-world agent settings; its support is organized around OpenClaw and specific terminal, GUI, SWE, and tool-call recipes rather than arbitrary native harness submission.

B.6 SWE-GYM GRPO HYPERPARAMETERS

Hyperparameter	Value
Base checkpoint	`Qwen/Qwen3.5-4B`
Training data	`NovaSky-AI/SkyRL-v0-293-data`, train split, 293 tasks
Trainer	Slime asynchronous GRPO
Epochs	1
Rollout batch size	4
Samples per prompt	16
Trace construction	`prefix_merging`
Optimizer	Adam
Learning rate	1×10^{-6}
Weight decay	0.1
TIS	Enabled

Table 4: **Training hyperparameters for the SWE-Gym GRPO experiments.** The table reports ordinary policy-optimization and rollout parameters from `examples/swegym_slime_grpo`; cluster topology and worker placement are omitted.

B.7 REPRESENTATIVE TASK PAYLOAD

Representative Polar Task Payload

```
{
  "task_id": "polar-swegym-{rollout_id}-{group_index}",
  "instruction": "Fix the issue in /polar/session/workspace.",
  "num_samples": 8,
  "timeout_seconds": 1200,
  "runtime": {
    "backend": "docker",
    "image": "{sample.metadata.runtime_image}",
    "network": "host",
    "workdir": "/polar/session/workspace",
    "prepare": [
      {
        "type": "exec",
        "command": "prepare repository, harness, and dependencies"
      }
    ]
  },
  "agent": {
    "harness": "codex",
    "model_name": "{served_model_name}"
  },
  "builder": {
    "strategy": "prefix_merging"
  },
  "evaluator": {
    "strategy": "swebench_harness",
    "refresh_runtime": true,
    "config": {
      "repo_dir": "/testbed",
      "patch_command": "cd /polar/session/workspace && git add -A &&
    git diff --cached --binary",
      "instance": "{sample.metadata.instance}"
    }
  },
  "callback_url": "http://{trainer_host}:{callback_port}/callback/
    task_result",
```

```
  "metadata": {
    "group_id": "{rollout_group_id}",
    "policy_version": "{policy_version}",
    "rollout_step": "{rollout_step}"
  }
}
```

B.8 Representative Trace

Representative Trainer-Facing Trace

```
{
  "prompt_ids": [151644, 872, "..."],
  "response_ids": [9211, 374, 264, "...", 151645, 271, 151644],
  "loss_mask": [1, 1, 1, "...", 0, 0, 1],
  "response_logprobs": [
    {
      "token": "The",
      "token_id": 785,
      "logprob": -0.21
    },
    {
      "token": " patch",
      "token_id": 10042,
      "logprob": -0.44
    },
    {
      "token_id": 151645,
      "logprob": 0.0
    }
  ],
  "prompt_messages": [
    {"role": "system", "content": "agent harness system prompt"},
    {"role": "user", "content": "task instruction and current tool
    state"}
  ],
  "response_messages": [
    {"role": "assistant", "content": "sampled assistant turn"}
  ],
  "tools": null,
  "finish_reason": "stop",
  "reward": 1.0,
  "metadata": {
    "session_id": "session-123",
    "task_id": "polar-swegym-0001",
    "builder": "prefix_merging",
    "harness": "codex"
  }
}
```

B.9 Service API Summary

The rollout service exposes a small asynchronous API:

- `POST /rollout/task/submit`: submit a non-blocking task request.
- `GET /rollout/task/{task_id}`: poll task status, partial results, and final results.
- `GET /rollout/status`: inspect task states, node states, and pending sessions.
- `POST /callbacks/session_result`: receive gateway session callbacks.

- `POST /nodes/register` and `POST /nodes/{node_id}/heartbeat`: maintain gateway membership and scheduling metrics.

The gateway exposes a control surface for session creation, status, and deletion, plus a catch-all proxy surface for provider-style model requests. Session deletion is used by the rollout pipeline as best-effort cleanup after a terminal result has been persisted.