Characterizing and Provisioning the Environment Bubble in Agentic-RL LLM Rollout

Sathishkumar Sivashanmugam
Amazon Web Services
satsivas@amazon.com

Abstract

Reinforcement-learning (RL) post-training of tool-using LLM agents leaves expensive rollout GPUs idle while each trajectory waits on CPU-side environment work: sandbox cold-start, code execution, retrieval, and external API calls. We term this idle the *environment bubble* and measure it directly on 8×A100 hardware using a 20 Hz NVML profiler that avoids the kernel-residency pitfall in the utilization counter and attributes each idle instant to one of three states: env-wait, straggler, or pipeline. Our central finding is that the magnitude of the bubble is governed by the environment tool's cold-start cost rather than by the RL system. Two real-GPU regimes bracket this behavior. Under an aggressive injected cold-start (`warm_s`=8 s, representative of heavy tools such as fresh containers, VM or browser spin-up, and a costly `env.reset`), env-wait is large and decreases from 0.80 at k=4 to 0.23 at k=64. Under a real `python:3.11-slim` Docker code-execution backend, whose cold-start we measure at 0.3–1.2 s, env-wait is small, decreasing from 0.049 to 0.008 over the same range, while effective utilization rises to 0.42–0.55. The large values therefore require heavy tool cold-start; under realistic sub-second tools only a few percent of idle is recoverable, and the tool latency of the target system should be measured before optimization. Increasing the number of sandbox workers reduces the bubble, but the benefit diminishes as cold-start decreases. We model the tier as a closed finite-source ($M/M/k//C$) queue; because the model overstates the achievable worker savings, we size k from the measured idle-vs-k curve. A lightweight next-tool-call predictor anticipates calls on real rollout token streams at AUC 0.81 without an additional forward pass. Finally, on real hardware speculative prewarming does not recover meaningful GPU idle in either regime, and a naive simulator (a persistent warm pool evaluated with too few seeds) can indicate otherwise. This is a measurement and reality-check study, and each reported value is labeled by how it was obtained.

1 Introduction

The rollout GPU pool is the scarce, expensive resource in reinforcement-learning (RL) post-training of tool-using LLM agents. Generation dominates step time, and for agentic workloads it is increasingly serialized behind *environment* work that never touches the GPU, such as `env.reset`, sandboxed code execution, retrieval, and external tool or API calls. While the GPU waits for a tool to return, it produces no tokens and earns no reward. We term this idle the *environment bubble*.

This paper presents a measurement-first study on real 8×A100 hardware and states only what the data support. The central result is a mechanism rather than a single number: the magnitude of the bubble is governed by the environment tool's cold-start cost, not by the RL system. We establish this by bracketing the bubble between two real-GPU regimes (Figure 2): an aggressive injected cold-

start (`warm_s`=8 s, representative of heavy tools such as fresh containers, VM or browser spin-up, and a costly `env.reset`), in which env-wait is large and decreases with the worker count k; and a real Docker code-execution backend with sub-second cold-start, in which env-wait is small. The bubble is real and is dominated by env-wait (Figure 1), but its magnitude is determined by the tool, so the tool latency of the target system should be measured before optimization. Section 3 reports the measurements.

This paper makes four contributions. First, we build an attributed env-bubble profiler and characterize the bubble on real hardware (Sections 2, 3), bracketing it between an aggressive injected regime and a real Docker backend that differ by more than an order of magnitude. Second, we analyze provisioning (Section 4): a closed $M/M/k//C$ queue accounts for the shape of the curve but overstates the achievable worker savings, so the measured curve is the appropriate guide, and its value diminishes as cold-start decreases. Third, we present a real-trace tool-call anticipator (Section 5): a logistic regression on lexical decode features attains AUC 0.81 on real Qwen2.5-3B transcripts without an additional forward pass. Fourth, we report a negative result (Section 6): on real hardware, speculative prewarming recovers no meaningful idle in either regime, and a simulator with a persistent pool and too few seeds produces a false positive. Each result is labeled by provenance (*measured*, *offline real data*, or *simulation*). An earlier version of this work reported large prewarming gains and a fixed idle headline; these did not reproduce on hardware and have been removed.

2 Setup and the Attributed Profiler

In PPO- and GRPO-style post-training, each rollout step interleaves T GPU generation segments with T CPU environment steps. Environment execution is heterogeneous and heavy-tailed: a search may take 100 ms, a Python sandbox 1–3 s, and a container cold-start 10–20 s. Most of the cost is cold-start; once warm, a sandbox serves subsequent calls cheaply.

Testbed. Unless noted, all hardware measurements use one `p4d.24xlarge` (8×A100-40GB, 96 vCPU) running vLLM [9] with `Qwen/Qwen2.5-3B-Instruct` at TP$=8$, driving C=64 concurrent ReAct rollouts against a bounded k-worker sandbox tier. The tier is driven either by a parameterized latency injection (`warm_s`=8 s, lognormal `serve_s`=0.3 s, c_v=1.2) or by the real Docker backend (Section 3). The existence and attribution of the idle are real GPU behavior; only its magnitude is set by the tool's latency distribution and by k.

Profiler and the NVML pitfall. A 20 Hz sampler records, at each instant, whether the GPU is active (at least one resident kernel), the number of outstanding environment requests $o(t)$, and the number of still-generating sequences $g(t)$. Each idle instant is classified as *env-wait* ($o(t) \geq 1$), *straggler* ($o(t)$=0, $g(t) \geq 1$), or *pipeline* ($o(t)$=$g(t)$=0). `nvmlDeviceGetUtilizationRates` reports whether at least one kernel was resident rather than SM occupancy, so a near-empty batch reads as busy; we therefore use the counter only to distinguish idle from active, and separately report *effective utilization* (active-fraction times mean batch occupancy), which exposes the waste within near-empty batches. The attribution is stable across busy-thresholds and sampling rates (artifact).

3 Characterizing the Bubble on Real Hardware

Two measured findings shape the remainder of the paper.

The bubble is dominated by env-wait. Figure 1 shows the attributed idle for a k=64 rollout. Generation phases are punctuated by synchronized troughs in which all C=64 rollouts block on the sandbox tier simultaneously; the straggler and pipeline components are negligible. The idle originates in the environment tier rather than in scheduling, which identifies the tier as the object to provision.

The bubble is governed by k, not a constant. As k increases (5 seeds; full table in Appendix, Table 1), env-wait decreases monotonically from 0.8047 ± 0.0233 at k=4 to 0.2278 ± 0.0406 at k=64, and effective utilization rises from 0.0036 to 0.1177. No single bubble percentage exists: the same workload spans a factor of $3.5\times$ in idle according to the number of provisioned sandbox

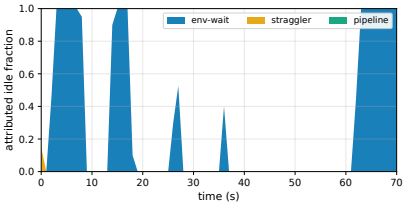

Figure 1: Attributed idle over a real 8×A100 rollout (k=64, 1 Hz, aggressive-cold-start regime). The idle is almost entirely *env-wait*, appearing as synchronized troughs in which all 64 rollouts block simultaneously.

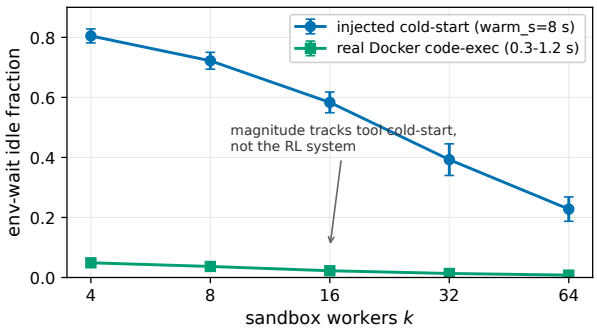

Figure 2: **The magnitude of the bubble tracks the tool's cold-start cost.** Env-wait vs. k for two real-GPU regimes: an aggressive injected cold-start (`warm_s`=8 s), which ranges $0.80 \rightarrow 0.23$, and a real Docker code-execution backend (cold-start 0.3–1.2 s p50), which ranges $0.049 \rightarrow 0.008$. Both decrease with k, but the magnitudes differ by more than an order of magnitude.

workers. This is the central measured result, and it identifies provisioning (Section 4) as the primary lever.

Overlap is a prerequisite. Under synchronous wave rollout, measured idle is 0.7993 ± 0.0791 at k=64 (0.42 env-wait and 0.37 pipeline stalls at the barrier); absent overlap between generation and environment execution, provisioning does not help. The bubble is also not an artifact of high tensor parallelism: at k=64, env-wait decreases with TP, from 0.324 ± 0.017 at TP= 1 to 0.236 ± 0.015 at TP= 8 (3 seeds, Figure 3 in Appendix A).

Reality check with a real Docker backend. We additionally evaluate a real code-execution backend: a pool of `python:3.11-slim` Docker containers with real create/start cold-start (0.3–1.2 s p50) and `docker exec` service, at the same C=64 and TP= 8. Under this lightweight tool the bubble is small and continues to decrease with k (Figure 2): env-wait is 0.0490 ± 0.0065 at k=4 and 0.0079 ± 0.0059 at k=64, while effective utilization rises from 0.220 to 0.542 as concurrent decode covers the short tool calls. Synchronous idle is 0.5516 ± 0.1485 at k=64, and the TP sweep is flat at approximately 1.5–2%.

4 Provisioning via a Closed Finite-Source Queue

Because additional workers reduce the bubble, the design question is how many to provision. The environment tier is a *closed* queue: a blocked sequence issues no new request until it resumes, so arrivals self-throttle and the in-flight population is bounded by the concurrency C. An open $M/G/k$ model overstates the load; the appropriate model is the finite-source $M/M/k//C$ machine-repair queue [5], which yields idle $I(k) = \mathbb{E}[n]/C$ with a floor of $\bar{s}/(\lambda + \bar{s})$ at k=C (a sequence still blocks for its own service).

Model versus measured curve: a cautionary mismatch. Overlaying $I(k)$ on the measured curve (Appendix A, Figure 4) reproduces the endpoints and the direction but is not a quantitative fit: the exponential model reaches its floor by $k \approx 16$, whereas the measured curve continues to decrease to k=64 (maximum difference 0.335), the signature of cold-start-dominated service. Consequently there is no inexpensive provisioning knee: the model suggests that a small k suffices, but on hardware k=16 still leaves 0.58 idle, and only $k \approx C$ attains the floor. The model's savings estimates (3× to 18×, of which the frequently cited 8× is a single operating point) assume exponential service and are not realized, so we do not report a savings headline and instead size k from the measured curve. The real backend reinforces this conclusion: with a sub-second tool, env-wait is a few percent at any k, so provisioning helps but its value is limited.

5 Predicting Tool Calls on Real Rollout Traces

An agent trajectory follows a regular reason-then-act grammar, so a lightweight model can anticipate the next tool call during generation. The anticipator is a systems component rather than an ML contribution: a logistic regression on decode-time features (position within the grammar, tokens since the last newline, and running marker counts). It outputs $P(\text{tool call within } H \text{ tokens})$ without an additional forward pass and runs on the CPU, off the critical path. We generated real Qwen2.5-3B ReAct rollout transcripts on the GPU testbed and evaluated the anticipator with a grouped-by-trajectory split (H=16, 5 seeds). On 248 real Docker-backend transcripts, at a base rate of 0.291, it attains AUC $\mathbf{0.808 \pm 0.017}$ (Appendix A, Figure 5), consistent with the AUC ≈ 0.83 obtained offline on the public Hermes function-calling corpus and on the injected-backend transcripts. The signal is thus present in the live decode distribution, not only in curated datasets, and it persists under the real backend. Transfer across corpora is weak (leave-one-corpus-out AUC 0.42–0.76, one fold below chance); we therefore regard the anticipator as a deployment-time signal to be calibrated on the target policy's own traces rather than as a corpus-transferable artifact.

6 A Negative Result and a Measurement Pitfall

We anticipated that the predictor would enable *speculative prewarming*: initiating a warm sandbox during generation so that cold-start is paid off the critical path. On real hardware this does not succeed, for a reason that generalizes.

Prewarming did not reduce idle. Using an on-hardware reclamation-attribution design, in which for each seed and pool discipline we run no-prewarm, an oracle prewarmer, and a volume-matched random-fire null, at k=64 (Table 2 and Figure 6) the oracle's idle is not lower than no-prewarm in either the persistent (0.2645 vs. 0.2541) or the TTL-reaped (0.2852 vs. 0.2791) discipline, and the random-fire null is no worse than informed timing. At C=64, k=64 the tier is not cold-start-bound, so no cold-start idle remains to hide. A naive simulator reports the opposite because of two confounds: a persistent (never-reaped) warm pool allows poorly timed prewarms to accumulate standing workers that appear to hide all cold-start, and too few seeds (n=3) obscure the noise. Reaping stale workers on a TTL and using a sufficient number of seeds removes both.

The negative persists with a real sandbox backend. To exclude an injected-latency artifact, we repeated the attribution against the real Docker code-execution backend (`python:3.11-slim` containers, measured cold-start p50 0.3–1.2 s, `docker exec` service). Prewarming recovers essentially nothing (Figure 7). Below k=64 the oracle is not below baseline (persistent: k=8, 0.0351 vs. 0.0318; k=16, 0.0256 vs. 0.0262), with warm-hit rates near zero (0.4–0.8%), because concurrent decode covers the short env-wait windows. Only at k=64/persistent (warm-hit 82.3%) does the oracle fall below baseline (0.0100 vs. 0.0196, approximately 1 pp), and even there the random-fire null captures roughly half (0.0149), a capacity effect rather than anticipation; the TTL discipline shows no gain. Across both backends, provisioning reduces the bubble whereas prewarming does not.

7 Related Work, Scope, and Conclusion

Reactive agentic-RL systems constitute the closest prior art. ROSE [1] harvests idle serving GPUs, Heddle [2] schedules trajectories, ARL-Tangram [3] right-sizes the CPU and reward tier, RollArt [4] over-provisions to hide `env.reset`, Libra [6] routes on tool-return signals, and MORI [7] offloads work into idle windows. Our contribution is a measured, attributed characterization, a closed-queue sizing rule, and an explicit prewarming negative. Training frameworks [8] and serving systems [9] are orthogonal, and queueing texts [5] provide the model. The principal threat to validity is that the injected regime is a proxy; the real Docker backend, however, exhibits the same trends.

Conclusion. The environment bubble is real and dominated by env-wait, but its magnitude is governed by the tool's cold-start cost rather than by the RL system, ranging from $0.80 \rightarrow 0.23$ under an aggressive proxy to a few percent under a real sub-second backend, so tool latency should be measured first. Provisioning is the primary lever, with diminishing value as cold-start decreases; the anticipator attains AUC 0.81; and prewarming recovers no meaningful idle in either regime.

References

[1] W. Gao et al. ROSE: Rollout On Serving GPUs via Cooperative Elasticity for Agentic RL. 2026.

[2] Z. Zhang et al. Heddle: A Distributed Orchestration System for Agentic RL Rollout. 2026.

[3] B. Xiao et al. ARL-Tangram: Unleash the Resource Efficiency in Agentic Reinforcement Learning. 2026.

[4] W. Gao et al. RollArt: Disaggregated Multi-Task Agentic RL Training at Scale. 2025.

[5] D. Gross, J. F. Shortle, J. M. Thompson, C. M. Harris. Fundamentals of Queueing Theory. 4th ed., Wiley, 2008.

[6] K. Chen, X. Tan, J. Li, H. Xu. Libra: Efficient Resource Management for Agentic RL Post-Training. 2026.

[7] T. Xia et al. Idleness is Relative: Exploiting Tool-Call Idle Windows for Offloading with MORI. 2026.

[8] G. Sheng et al. HybridFlow: A Flexible and Efficient RLHF Framework. EuroSys 2025.

[9] W. Kwon et al. Efficient Memory Management for LLM Serving with PagedAttention. SOSP 2023.

A Additional Figures and Tables

Table 1: Injected aggressive-cold-start regime: env-bubble idle and effective utilization vs. worker count k (real 8×A100, TP= 8, C=64, 5 seeds).

k	env-wait (mean±std)	effective util	idle total
4	0.8047 ± 0.0233	0.0036 ± 0.0025	0.8097
8	0.7221 ± 0.0281	0.0072 ± 0.0031	0.7278
16	0.5834 ± 0.0346	0.0173 ± 0.0015	0.5864
32	0.3926 ± 0.0527	0.0618 ± 0.0068	0.3944
64	0.2278 ± 0.0406	0.1177 ± 0.0118	0.2316

Table 2: On-hardware reclamation attribution, aggressive-cold-start regime (k=64, 3 seeds). Oracle idle is not below baseline in either discipline, so speculative prewarming does not reduce idle.

discipline	idle (no-prewarm)	idle (oracle)	idle (random-fire)
persistent	0.2541	0.2645	0.2461
reaped (TTL)	0.2791	0.2852	0.2679

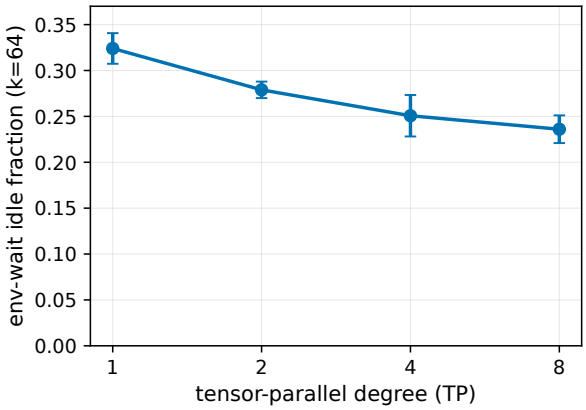

Figure 3: Env-wait vs. tensor-parallel size (k=64, C=64, 3 seeds). Idle *decreases* with TP, so the bubble is not a high-TP artifact.

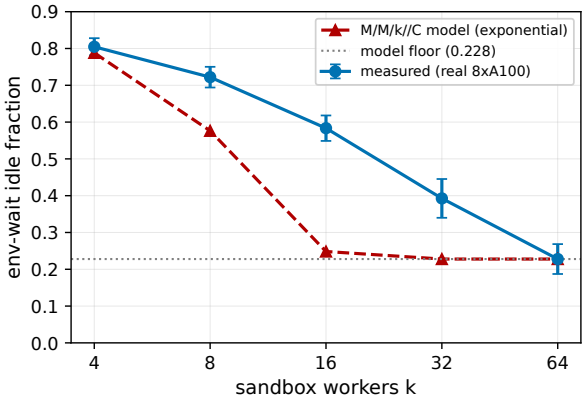

Figure 4: Closed-queue model $I(k)$ vs. measured env-wait (Table 1). The endpoints and direction agree, but the exponential model reaches its floor by $k \approx 16$ while the measured curve continues to decrease to k=64 (maximum difference 0.335). Under cold-start-dominated service there is no inexpensive knee, so the model would under-provision; the measured curve is the operational guide.

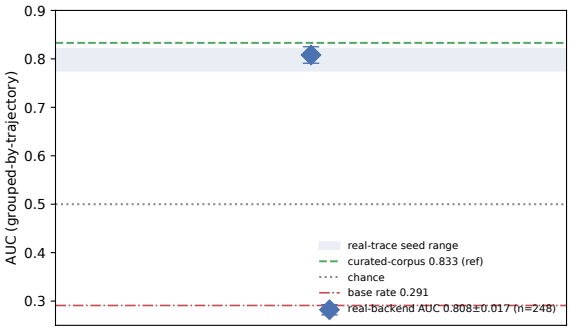

Figure 5: Next-tool-call anticipation on real Qwen rollout traces (real-backend AUC 0.808 ± 0.017) against the curated-corpus reference 0.833 and chance.

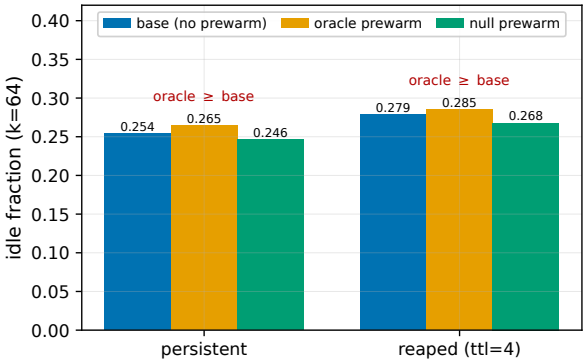

Figure 6: Speculative prewarming does not reduce idle on real hardware (aggressive-cold-start regime): the oracle arm is not better than no-prewarm, and the random-fire null is not worse than the oracle.

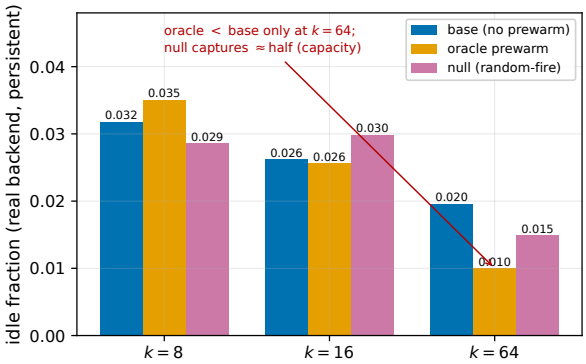

Figure 7: Real Docker backend prewarming attribution (persistent pool) across k. The oracle prewarm is not below the no-prewarm baseline until $k{=}64$, and even there the random-fire null captures approximately half of the reduction, so the negative persists on a real backend.