NitroBox: Lightning-Fast Sandbox for Large-Scale RL Training

Anonymous Author(s)
Affiliation
Address
email

Abstract

Recent work on agentic training efficiency has mainly focused on improving learning algorithms, rollout strategies, or model–hardware co-design. However, these approaches overlook an important source of training latency: the sandbox that runs the environments. Recent studies show that creating, executing, and resetting containers accounts for an important portion of the rollout process in RL training, substantially slowing down overall training. This overhead largely stems from the centralized management mechanisms (e.g., DAEMON) of existing sandboxes, along with additional features (e.g., multi-tenancy) that are unnecessary for reinforcement learning (RL). Moreover, existing sandboxes are designed for general-purpose isolated execution, making them a poor fit for the requirements of RL rollout and training. We propose NitroBox, a novel sandbox system purpose-built for agentic RL that significantly accelerates the rollout and training process. Technically speaking, we first comprehensively redesign NitroBox's lifecycle to eliminate the dependency on DAEMON, so that every sandbox instance is fully independent during creation, startup, and reset (e.g., via rootless full-namespace isolation). Furthermore, we introduce a copy-on-write mechanism to enable fast startup across multiple instances. Second, we knock off features that are time-consuming but unnecessary for RL — such as multi-tenancy — while preserving the core functionality and security guarantees needed for RL training. Finally, we introduce some RL-nature features to further improve efficiency, such as optional CRIU-based process checkpoint saving and restoring for mid-trajectory branching. Our experiments show that NitroBox is faster than Docker across the entire lifecycle: 10.6 $\times$ faster in sandbox creation, 24.6$\times$ in reset, and 19.1 $\times$ in deletion. When training Qwen2.5-Coder-32B-Instruct with GRPO at $c = 32$ concurrency, end-to-end wall-clock training time drops from 40.6 hours to 28.4 hours when replacing Docker with NitroBox, while preserving the same task pass rate.

1 Introduction

With recent progress in LLM agents, there is an increasing paradigm shift and emphasis on agentic RL for LLM post-training, where the LLM model is equipped with agent scaffolding, including tool calls and memory. The agent interacts with the environment to collect execution trajectories, which are then used to fine-tune LLMs for solving real-world tasks.

Most recent work on agentic RL mainly focuses on improving model performance by proposing new algorithms [25, 33, 15, 32], building RL training infrastructure [26, 29, 11, 12], and synthesizing useful environments [14, 27, 13, 21, 31]. Aside from training performance, efficiency is also a key bottleneck, especially since agentic training is resource-demanding and requires a very large number of rollouts. Without an efficient training pipeline, it is nearly impossible to train good models under limited resources, let alone tune the hyperparameters for RL, where efficient training is critical.

Submitted to 40th Conference on Neural Information Processing Systems (NeurIPS 2026). Do not distribute.

Recent work on agentic training efficiency has mainly focused on improving learning algorithms [25, 33, 15, 32], rollout strategies [20, 7, 16], or model–hardware co-design [38, 30, 10, 37]. However, these approaches overlook an important source of training latency: the sandbox that runs the environments. Creating, executing, and resetting containers accounts for a significant portion of the rollout process in RL training.

Existing works typically use off-the-shelf containers, such as Docker [17] and Podman [23]; however, these are not purpose-built for RL training. They rely on centralized management mechanisms (e.g., DAEMON), which introduce additional wall time when starting multiple parallel instances and include many features (e.g., multi-tenancy) that are unnecessary for reinforcement learning. As a result, sandbox operations have become one of the major sources of training overhead, especially during the rollout phase. Despite this, there is very limited research on this important topic. The most relevant work is SWE-MiniSandbox [34]. It attempts to bypass Docker entirely but reverts to chroot+mount-namespace and tarball-based reset, sacrificing isolation properties (network, PID, IPC) that are required for safe concurrent rollouts. As such, the community still lacks a fully functional, secure, and efficient sandbox for hosting RL environments in post-training.

To bridge this gap, we propose NitroBox, a novel sandbox purpose-built for agentic RL. Technically, we first propose a new sandbox system design that eliminates the dependency on DAEMON, the centralized sandbox management mechanism. DAEMON introduces notable latency when managing multiple sandbox instances due to its centralized locking. By removing DAEMON, we enable every sandbox to operate fully asynchronously with independent lifecycles. Besides this, to enable efficient sandbox inheritance, we introduce a copy-on-write mechanism for the file system, allowing an environment to be easily duplicated into multiple copies. This is especially useful for RL training, where multiple sandbox instances are typically created from the same template. Beyond the new system design, we also remove certain sandbox features that are time-consuming but unnecessary for RL, while preserving the core functionality and security guarantees needed for RL training. This includes features like multi-tenancy support. Finally, we integrate specific support features for RL training, such as CRIU-based process checkpoint saving and restoring. This allows us to efficiently restore previous trajectories and continue generation from any intermediate state, which is required by many modern agentic RL methods.

Our experiments demonstrate that NitroBox substantially outperforms Docker across the entire sandbox lifecycle. Compared to Docker, NitroBox achieves $10.6\times$ faster sandbox creation, $24.6\times$ faster reset, and $19.1\times$ faster deletion. On Terminal-Bench 2.0 evaluation with 88 tasks at 16-way concurrency, end-to-end wall-clock time drops from 1962s to 880s (2.23× speedup); When training Qwen2.5-Coder-32B-Instruct with GRPO at $c = 32$ concurrency, end-to-end wall-clock training time drops from 40.6 hours to 28.4 hours when replacing Docker with NitroBox, while preserving the same task pass rate.

Contributions. This paper makes the following contributions:

- We characterize the sandbox-lifecycle bottleneck in modern agentic RL training through profiling and survey across recent frameworks, and identify a structural mismatch between Docker's design assumptions and the agentic RL workload.
- We design and implement **NitroBox**, an RL-specific sandbox that comprehensively improves Docker operation speed across the entire lifecycle. NitroBox introduces a new system design together with many RL-specific features to minimize overhead in sandbox operations.
- We evaluate NitroBox across three regimes: micro-benchmarks (10.6–24.6$\times$ improvement on individual lifecycle operations), end-to-end agentic benchmarks (up to 2.23$\times$ speedup on Terminal-Bench 2.0), and full RL training (1.43$\times$ wall-clock reduction at identical Pass@1), all while preserving necessary security features.

2 Related Work

RL infrastructure for LLM agents. A growing body of work scales agentic RL training to thousands of concurrent rollouts. ProRL Agent [36] introduces a rollout-as-a-service decoupling that exposes rollout collection as an HTTP service independent of the trainer; SkyRL-Agent [4] adopts a Kubernetes-based architecture with asynchronous pipeline dispatchers, achieving 1.55 times speedup over native batching; and RollArt [8] pushes further with disaggregated infrastructure separating

System	Approach	Rootless	Reset	Docker img	Persist. shell	Checkpoint	Net iso
General-purpose container							
Docker	ns+cgroup, daemon	✗	recreate (∼600ms)	native	✗	✗	✓
Podman	ns+cgroup, daemonless	✓	recreate	✓	✗	○CRIU	✓
gVisor	user-space kernel	○	recreate	OCI	✗	✗	✓
microVM							
Firecracker	KVM microVM	✓	memory snapshot	(conv.)	✗	✓	✓
microsandbox	libkrun microVM	✓	recreate (∼200ms)	OCI	✗	✗	✓
Lightweight namespace							
bubblewrap	mount/PID/user ns	✓	recreate	✗	✗	✗	✓
isolate	full ns + cgroup	✓	per-task	✗	✗	✗	✓
Agent / production							
e2b	Firecracker (cloud)	✓	recreate	✓	○	✗	✓
AIO Sandbox	Docker (all-in-one)	✗	recreate	✓	○	✗	✓
RL-specific sandbox							
SWE-MiniSandbox	mount ns + chroot	✓	tarball decompress	✗	✓	✗	✗
ProRL SingularityRT	Singularity, custom	✓	recreate	○	✓	✗	○
nitrobox	**full ns + overlayfs**	✓	**O(1) rename (∼12ms)**	**+ Compose**	✓	**✓CRIU**	✓

Table 1: Sandbox runtime comparison along dimensions critical for agentic RL training.

GPU-bound generation from CPU-bound environment interaction, profiling SWE-bench [14] training runs to identify environment lifecycle as a primary bottleneck. At the production scale, rStar2-Agent [24] reports up to 45K concurrent tool calls per training batch. While these systems support a wide range of lifetime patterns, none is specifically designed for the high-frequency, repeatedly-reset lifecycle of agentic RL rollouts.

RL-specific sandbox and environment infrastructure. A handful of recent works explicitly target the sandbox layer for agentic RL. SWE-MiniSandbox [34], which abandons containers in favor of mount namespaces, chroot, and tarball-based environment caching for SWE-bench training. While motivated by similar concerns, SWE-MiniSandbox provides only filesystem isolation—it shares the host PID, network, and IPC namespaces, leading to port conflicts and process visibility between concurrent rollouts—and its tarball-based reset scales linearly with environment size. OSGym [21] addresses the GUI computer-use case with a hardware-aware orchestration layer that achieves 1420 trajectories per minute via XFS reflink CoW disk provisioning and pre-warmed runner pools.

Agent execution sandbox platforms. A new infrastructure category has emerged around running LLM-generated code in production agent products. Commercial offerings include E2B [6], Modal [19], Daytona [5], which trade off across isolation primitive (gVisor, Firecracker, Kata Containers), startup latency (25–200 ms), and persistence model (ephemeral vs. checkpoint-restore). Recent open-source entries include microsandbox [35] (libkrun-based local-first microVMs), AIO Sandbox [3] (an all-in-one Docker container bundling browser, shell, IDE, and Jupyter), OpenSandbox [2]. These platforms are designed for hostile multi-tenant code execution from end users, leading to design choices (full microVM isolation, daemon-mediated lifecycles, cloud-only deployment) that are over-provisioned for the within-job, mutually-trusted RL workload that nitrobox targets.

Container runtimes and isolation primitives. Docker [17] is the dominant container runtime built on Linux namespaces and cgroups, mediated by a privileged daemon. Stronger isolation is offered by gVisor [9] (user-space syscall interception), Firecracker [1] (KVM-based microVMs powering AWS Lambda), and Kata Containers [22] (VM-backed container runtime). *All of these treat sandboxes as long-lived execution units.*

3 Nitrobox

3.1 Nitrobox's Overview

Fig. 1 summarizes the design of NitroBox. NitroBox is built around two principles. First agentic RL requires a high-frequency execution lifecycle rather than a general-purpose container lifecycle. During rollout, workers repeatedly create environments, execute many short agent actions, reset state at episode boundaries, and delete environments after use. These operations lie on the rollout hot path,

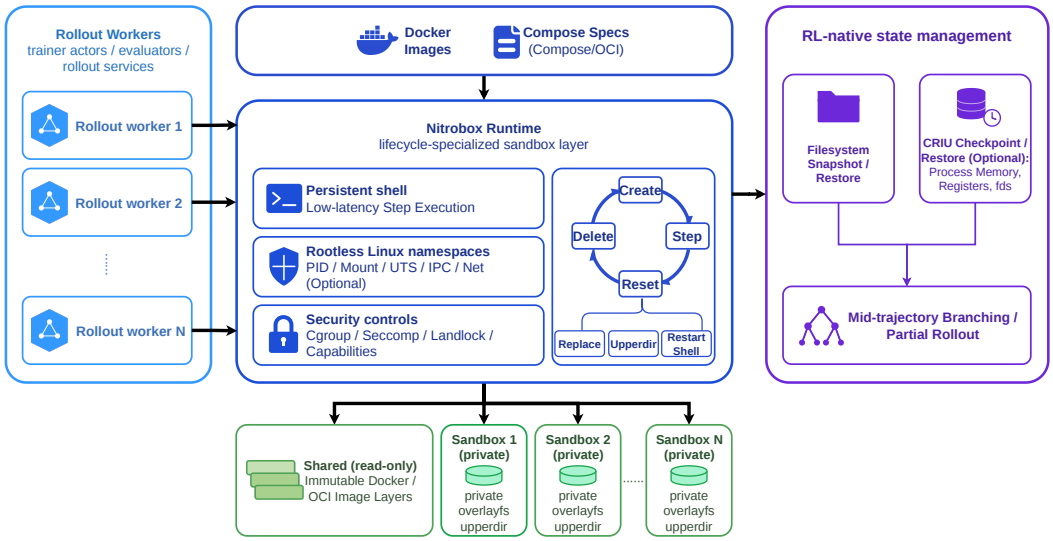

Figure 1: Overview of NitroBox. NitroBox specializes the sandbox lifecycle for agentic RL rollout: workers create, execute, reset, snapshot, and checkpoint independent rootless namespace sandboxes without routing the per-episode path through a Docker daemon. Immutable image layers are shared across instances, while each sandbox keeps a private copy-on-write upper layer that can be replaced for fast reset. Optional filesystem snapshots and CRIU checkpoints provide RL-native rollback and mid-trajectory branching.

so NitroBox implements them as lightweight sandbox operations rather than as full container life cycle events.

The center of Figure 1 shows the runtime components that support this lifecycle. Each sandbox is a rootless Linux-namespace environment with a persistent shell and explicit security controls. Rootless namespaces isolate the process tree, mount table, hostname, IPC state, and optionally network state, while allowing ordinary rollout workers to create and reset sandboxes without root privileges. The persistent shell supports low-latency command execution across the many short steps that appear in code and terminal-agent trajectories. Security controls such as cgroups, seccomp, Landlock, capability restrictions, and controlled device or network access preserve the containment needed for RL execution.

The bottom of Figure 1 shows the filesystem design. NitroBox accepts Docker/OCI images and Compose-style specification as compatibility inputs, but represents each sandbox filesystem as a copy-on-write overlay. Immutable image layers are shared across sandbox instances as read-only lower layers, while each sandbox receives a private overlayfs upper directory for episode-local mutations. Creating multiple environments therefore reuses the same image state rather than copying or unpacking it for every sandbox. This copy-on-write structure makes reset a first-class operation. At an episode boundary, NitroBox terminates the sandbox shell, replaces the dirty private upper/work directories with fresh ones, and restarts the shell under the same sandbox configuration. The immutable image layers and security policy are reused. Thus, reset is implemented as local filesystem state replacement plus shell restart, rather than destroying and recreating a container.

Finally, the right side of Figure 1 shows NitroBox's RL-native state management. Filesystem snapshot/restore records and restores the sandbox-local copy-on-write state, enabling rollback to intermediate filesystem states. Optional CRIU-based checkpoint/restore additionally captures process memory, registers, and file descriptors, enabling mid-trajectory branching when restoring files alone is insufficient. Together, these mechanisms turn reset, rollback, and branching into explicit runtime operations aligned with agentic RL rollout.

3.2 Lifecycle-Specialized Sandbox Runtime

The key design choice in NitroBox is to separate the rollout hot path from the general-purpose container control path. In a conventional container stack, creating an environment, executing a command, resetting by reaction, and tearing the environment down are mediated by global container-management components such as the Docker daemon, containerd, and runc. This design is appropriate for long-running services and multi-application orchestration, but it is mismatched to agentic RL, where the same operations occur at episode frequency and directly add to rollout latency.

NitroBox instead treats each environment as a local rollout sandbox. Once image layers have been prepared, a rollout worker can create, execute, reset, snapshot, and delete the sandbox using local namespace and filesystem primitives. Image acquisition and layer preparation are treated as amortized setup costs, while the repeated operations in the training loop are optimized as the runtime hot path. This is why NitroBox preserves Docker/OCI image compatibility without routing every episode boundary through the Docker container lifecycle.

This specialization relies on two mechanisms. First, the filesystem separates shared immutable image state from private episode state: all writes go to the sandbox's overlayfs upper directory, so reset can be implemented by replacing that upper directory rather than recreating the full container. Second, command execution goes through a persistent shell inside the sandbox, amortizing namespace entry and environment setup across the many short actions in an agentic trajectory. Together, these mechanisms turn container-scale lifecycle events into a lightweight rollout operation.

3.3 Nitrobox's Implementation

NitroBox is implemented as a three-layer runtime. The Python layer exposes the user-facing `Sandbox` and `SandboxConfig` APIs, including Docker compatibility constructors that translate common Docker SDK and `docker run` options into NitroBox's native configuration. The Rust extension `nitrobox._core` implements the syscall-heavy sandbox path, including namespace creation, overlayfs mounting, pivot/chroot setup, volume and device mounts, cgroup attachment, and security controls. A side-band Go binary, `nitrobox-core`, manages image build and pull through an embedded BuildKit service.

Sandbox creation first resolves the base filesystem. Local rootfs paths are used directly; Docker/OCI images are resolved into ordered layer directories that serve as shared overlayfs lower layers. Each sandbox creates its own environment directory containing private `upper` directory, `work` directory, and rootfs mount point. In rootless mode, the Rust init path creates the user-namespace sandbox and mounts overlayfs inside it. In rootful mode, NitroBox prepares the filesystem and cgroup state before spawning the sandbox process. In both cases, the sandbox is isolated with Linux namespaces and configured with the requested resource and security policies.

Command execution and reset are optimized for rollouts. Each sandbox contains a persistent shell connected through stdin/stdout pipes; `run()` dispatches commands to this already-initialized shell, avoiding a full container or namespace setup path for each short agent action. Reset terminates the shell, replaces the dirty private `upper`/`work` directories with fresh ones, and restarts the shell under the same sandbox configuration, reusing immutable image layers and security policy across episodes.

NitroBox implements branching with the same filesystem abstraction. Filesystem snapshot/restore copies and restores the sandbox's private upper layer, capturing only sandbox-local mutations. For full process-state rollback, `CheckpointManager` combines this filesystem state with CRIU dump/restore, preserving process memory, registers, file descriptors, and pipe metadata. Image build, pull, and layer-resolution operations remain off the per-episode hot path through a side-band BuildKit service, after which sandbox lifecycle operations run on local filesystem paths rather than through Docker daemon, containerd, or runc.

4 Evaluation

We evaluate NitroBox to answer four questions: **Q1** (lifecycle cost): how does NitroBox compare against existing sandbox runtimes on individual lifecycle operations? **Q2** (workload breakdown): where does the per-call advantage translate into wall-clock savings under realistic agentic workloads? **Q3** (concurrency stability): does the advantage persist as rollout concurrency scales toward production-

scale RL training? **Q4** (end-to-end RL training): does NitroBox preserve task pass rate while reducing total training time on a real GRPO pipeline? We answer Q1 with a sandbox-lifecycle micro-benchmark (Section 4.2), Q2 with a per-phase breakdown on four agent benchmarks (Section 4.3), Q3 with a concurrency sweep on SWE-bench Verified (Section 4.4), and Q4 with an end-to-end RL training run (Section 4.5).

4.1 Evaluation Setting

Hardware. All measurements run on a single GCP `m1-megamem-96` instance (96 vCPU Intel Xeon @ 2.0 GHz, 1.4 TiB DDR4, 4×375 GB local NVMe in RAID-0), running Linux 6.17 on Ubuntu 24.04 with rootless user namespaces enabled. RL training experiments additionally use 8×NVIDIA B300 GPUs.

Compared sandbox runtimes. We compare NitroBox against eleven sandbox runtimes spanning four families: *general-purpose containers* (**Docker** as canonical baseline, **Podman** rootless, bare `runc`, **gVisor** [9]); *microVM-based runtimes* (**Firecracker** [1] in snapshot and cold-boot regimes, **e2b** [6] as hosted Firecracker, and **CubeSandbox** [28]); and *RL-specific or namespace-based research sandboxes* (**SWE-MiniSandbox** [34], **OpenSandbox** [2], and **AIO Sandbox** [3]). Although microVM runtimes operate in a stronger isolation regime, we include them to characterize the full performance–isolation tradeoff space.

Workloads. The micro-benchmark (Section 4.2) measures `create`, `exec`, `reset`, and `delete` on a pre-pulled `ubuntu:22.04` image, reporting mean and standard deviation over $N = 10$ trials after a 5-trial warm-up. End-to-end evaluation (Section 4.3) spans four agent benchmarks: Terminal-Bench 2.0 (88 shell tasks, ∼200 MB images), R2E-Gym-Lite (88 Python repo-debugging tasks, ∼700 MB images), SWE-bench Verified (500 GitHub-issue tasks, ∼1 GB images), and OSWorld (100 desktop GUI tasks across nine applications). RL training setup is described in Section 4.5.

4.2 Lifecycle micro-benchmark.

Table 2 reports single-instance lifecycle latency across all backends. NitroBox completes a full create-exec-reset-delete cycle in 45.1 ms, compared to 635.6 ms for Docker rootful—a $14\times$ aggregate reduction, with $10.6\times$ on `create`, $24.6\times$ on `reset`, and $19.1\times$ on `delete`. The advantage stems from three orthogonal design choices, each visible in a different operation: NitroBox avoids the daemon stack on `create`, amortizes per-command setup via persistent shell on `exec`, and replaces full container teardown with overlayfs upperdir-rename on `reset`. Two outliers warrant comment. SWE-MiniSandbox [34], despite its container-free design, pays 355 ms create and 455 ms reset due to tarball-based environment extraction—an artifact of trading OCI semantics for filesystem simplicity. Podman rootless exhibits 10-second reset and delete latencies dominated by `slirp4netns` teardown, illustrating the cost of bolting rootlessness onto a daemon-oriented runtime rather than designing for it from the start. In aggregate, NitroBox is the only backend that maintains tens-of-milliseconds latency uniformly across all four operations.

4.3 E2E Evaluation

We evaluate NitroBox against Docker, e2b, and gVisor on four agentic benchmarks: Terminal-Bench 2.0 [18] (shell tasks), R2E-Gym-Lite [13] (Python repository debugging), SWE-bench Verified [14] (GitHub issue resolution), and OSWorld [31] (GUI desktop tasks). We distinguish two cache regimes: A *cold* run wipes both the Docker daemon image store and the local sandbox cache and a *hot* run is launched after a cold run with no cleanup. SWE-bench Verified always exercises the image-build path: its harness rebuilds per trial, so we report a single number corresponding to the build path on a pre-warmed cache. For E2B, sandbox templates are built remotely and persisted in the E2B control plane, so cold and hot collapse to the same measurement.

Table 3 reports total wall-clock and pass count across the four benchmarks. NitroBox is the fastest backend on every SWE-style benchmark in this single-machine regime, with end-to-end speedups over Docker of $2.2\times$ on Terminal-Bench 2.0 hot (880 s vs. 1962 s), $2.0\times$ cold, $1.5/1.3\times$ on R2E-Gym-Lite cold/hot, and $1.38\times$ on SWE-bench Verified. The advantage is largest when sandbox lifecycle dominates the workload (Terminal-Bench hot); when execution dominates the wall-clock (SWE-bench Verified), e2b's hosted infrastructure narrows the gap by trading sandbox lifecycle for

Table 2: **Single-instance lifecycle latency (ms, mean $\pm$ std over N=10 trials, lower is better).** *create* measures sandbox spawn through ready-for-exec; *exec* is the mean of 20 sequential `echo` commands; *reset* is teardown plus respawn; *delete* is teardown only. All container backends share a pre-pulled `ubuntu:22.04` image.

Backend	create	exec	reset	delete	Class
runc (rootful, direct)	39.1 ± 0.1	28.3 ± 2.3	208.9 ± 5.1	171.3 ± 0.0	runtime alone, no daemon
Firecracker (snapshot)	76.7 ± 0.7	53.8 ± 0.1	192.4 ± 4.0	16.5 ± 2.8	microVM, paused-VM restore
e2b (cloud, snapshot)	117.0 ± 41.8	60.5 ± 11.6	202.3 ± 11.2	108.2 ± 1.6	remote Firecracker, snapshot
gVisor (runsc, KVM)	145.4 ± 0.1	29.2 ± 1.7	249.1 ± 11.3	98.9 ± 14.0	user-space kernel (Sentry)
Docker (rootful)	155.1 ± 4.9	48.9 ± 0.7	295.7 ± 10.9	135.9 ± 6.9	rootful container, daemon stack
Podman (rootless)	156.2 ± 0.6	186.8 ± 6.4	10,457 ± 2	10,339 ± 52	rootless, slirp4netns teardown
SWE-MiniSandbox	355.0 ± 0.2	104.1 ± 0.1	455.2 ± 0.1	100.3 ± 0.0	namespace + chroot, no images
OpenSandbox	666.8 ± 24.8	1,011.3 ± 0.2	1,778 ± 29	1,113 ± 19	HTTP API → Docker + execd
Firecracker (cold boot)	1,383 ± 55	53.8 ± 0.1	1,488 ± 45	16.7 ± 2.4	microVM, full kernel boot
CubeSandbox (cold boot)	859.6 ± 33.6	12.4 ± 1.0	953.0 ± 33.1	102.0 ± 4.9	KVM microVM, full boot
AIO Sandbox	3,713 ± 21	838.6 ± 8.4	3,941 ± 10	213.8 ± 5.5	container + bundled services
NITROBOX	**14.6 ± 1.0**	**11.4 ± 0.0**	**12.0 ± 0.6**	**7.1 ± 0.3**	**rootless ns + overlayfs, in-process**

Benchmark	Tasks	c	Docker	**nitrobox**	e2b	gVisor	Pass (D/N/E/G)
SWE-style task evaluation							
Terminal-Bench 2.0 (cold)	88	16	1,961.1 s	**980.0 s**	2,469 s†	4,341 s	82 / 82 / 75 / 63
Terminal-Bench 2.0 (hot)	88	16	1,961.8 s	**880.0 s**	2,469 s†	3,616 s	82 / 82 / 75 / 69
SWE-bench Verified	499	20/16	1,920.4 s	**1,395.0 s**	1,483 s	2,674 s	492 / 492 / 489 / 489
R2E-Gym-Lite (cold)	88	16/8	363.5 s	**240.0 s**	722.0 s†	385.2 s	88 / 87 / 87 / 88
R2E-Gym-Lite (hot)	88	16/8	124.5 s	**95.0 s**	132.4 s	168.7 s	88 / 88 / 87 / 88
GUI agent evaluation							
OSWorld	100	10	2,653.7 s	**2,280.0 s**	—	—	80 / 80 / — / —

Table 3: **End-to-end wall-clock and pass-count across agentic benchmarks under four sandbox runtimes.**

Benchmark	Env_Setup (s)				Verifier (s)				Teardown (s)			
	Docker	nbx	e2b	gVisor	Docker	nbx	e2b	gVisor	Docker	nbx	e2b	gVisor
Terminal-Bench 2.0 (cold)	19.8	17.8	**1.2**	14.2	57.5	**25.6**	44.2	381.4	10.4	0.5	**0.1**	1.9
Terminal-Bench 2.0 (hot)	3.3	**0.7**	1.2	1.7	63.5	**25.4**	44.4	249.5	15.6	1.1	**0.1**	7.9
SWE-bench Verified	23.6	**1.2**	1.6	3.5	33.8	30.5	**30.2**	68.7	9.0	4.0	**0.1**	1.8
R2E-Gym-Lite (cold)	40.6	35.4	**29.2**	42.0	6.5	6.6	28.4	9.3	12.2	**0.1**	**0.1**	12.2
R2E-Gym-Lite (hot)	**2.0**	3.1	4.0	3.2	**5.6**	9.1	6.8	8.9	10.8	0.3	**0.1**	11.0
OSWorld	23.2	**17.7**	—	—	23.0	**23.2**	—	—	0.3	**0.3**	—	—

Table 4: Per-phase mean time across four sandbox runtimes (seconds, lower is better).

cloud parallelism. Task pass-count remains within ± 1 of Docker on every benchmark, confirming that lifecycle speedups do not come at the cost of correctness.

As shown in Table. 4, we decompose the wall clock into environment setup, verifier execution, and teardown, which reveals where the gain comes from. Teardown is consistently 14–120$\times$ faster on lifecycle-heavy benchmarks. For Setup in SWE-Bench, we observe a 19.7$\times$ speedup as BuildKit's layer cache eliminates per-task pulls; benchmarks with smaller per-task images yield more modest setup speedups. For Verifier execution, the speed-up depends on what dominates the verifier wall-clock: when Python test runner startup dominates, NitroBox's persistent shell yields up to 2.5$\times$ by amortizing per-command setup; on long compute-bound test suites such as SWE-bench Verified, the gap narrows to $\sim$1.1$\times$ since the test runtime itself dominates.

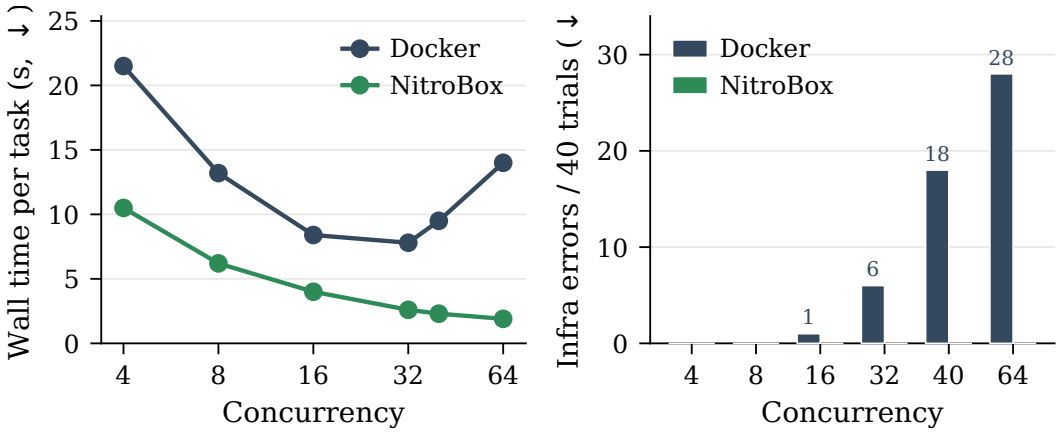

Figure 2: Concurrency scaling on SWE-bench Verified. Left: wall time per task as a function of concurrency. NitroBox keeps improving with diminishing returns; Docker bottoms out around $c = 16$–32 before climbing again under daemon contention. Right: number of trials (out of 40) that fail with infrastructure errors. Docker's daemon-routed lifecycle starts losing trials at $c = 16$ and the failure rate rises with concurrency; NitroBox's daemon-less lifecycle stays at zero across the full sweep.

4.4 Concurrency Scaling and Infrastructure Stability

The end-to-end numbers above are taken at modest concurrency ($c \leq 20$), where Docker's daemon is already saturated but rarely fails. RL training, however, demands rollout concurrencies that scale with available cores. Figure 2 sweeps SWE-bench Verified concurrency from $c = 4$ to $c = 64$ on the same hardware. NitroBox's per-task wall time keeps improving across the sweep with diminishing returns at high c, while Docker reaches a shallow optimum around $c = 16$–32 and starts to climb again under daemon contention. The two backends also diverge in stability: occasional infrastructure errors begin to appear in Docker from $c = 16$, accumulate as concurrency grows, and reach $18/40$ failed trials at $c = 40$; NitroBox remains at zero infrastructure errors across every concurrency we measured. The combination is what makes NitroBox usable as a drop-in sandbox for a high-concurrency RL training pipeline, which we measure next.

4.5 End-to-End RL Training

Setup. We train Qwen2.5-Coder-32B-Instruct with GRPO using NVIDIA's NeMo-RL framework on the Nemotron synthetic task corpus, swapping only the sandbox runtime between runs. Each policy step collects a batch of 32 rollouts at concurrency $c = 32$ with group size $K = 8$; rollouts share the trained policy via a co-located vLLM server, with environment commands dispatched to either Docker or NitroBox. We use the NeMo-RL default optimizer recipe (AdamW, lr 5×10^{-6}, cosine schedule, warmup ratio 0.1, BF16, DeepSpeed ZeRO-3) and train for 487 policy steps on a single 8×NVIDIA-B300 node. Evaluation follows the Terminal-Bench 1.0 protocol: we report Average Pass@1 over three independent generations at temperature 0.6 on the public test set. The only change between the two runs is the sandbox under-the-hood — model, data, optimizer, seeds, and rollout concurrency are held constant.

Results. Figure 3 (left) plots the GRPO mean-group reward against training wall-clock for both backends. Both runs start near 0.42 and converge to a plateau around 0.61 on the same trajectory; the only difference is the rate of advance along the wall-clock axis. NitroBox reaches the plateau in ≈ 28.4 hours versus Docker's ≈ 40.6 hours, a $1.43\times$ wall-clock reduction at identical training reward. Figure 3 (right) decomposes a single training step: rollout-phase sandbox time falls from 4.2 minutes to 2.7 minutes, while the GPU-side training step is unchanged at ≈ 0.8 minutes; the bottleneck shifts from sandbox lifecycle to GPU compute. Table 5 summarises wall-clock, GPU-hours, throughput, and final Pass@1 across the two backends.

The infrastructure-stability gap from Section 4.4 also has a direct training consequence. Repeating the same run at $c = 64$, Docker fails to complete due to the daemon-level infrastructure errors documented

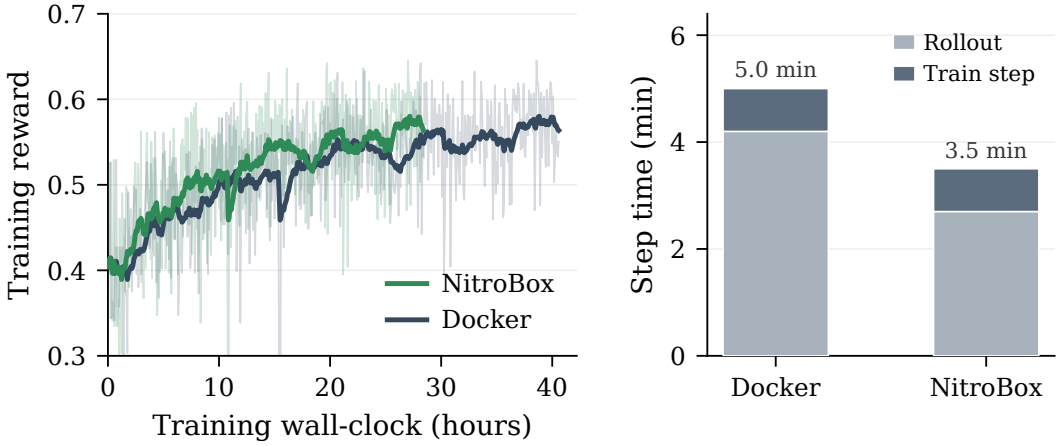

Figure 3: End-to-end GRPO training of Qwen2.5-Coder-32B-Instruct, swapping only the sandbox runtime. Left: training reward (GRPO mean group score) vs. wall-clock hours, raw signal in light tone and EMA-smoothed curve in bold. Both backends traverse the same reward trajectory; NitroBox reaches the same plateau in $1.43\times$ less wall-clock time. Right: per-step time decomposed into rollout (sandbox) vs. training (forward + backward + optimizer) on $8\times$B300. NitroBox cuts the rollout component from 4.2 to 2.7 minutes ($1.56\times$) while the GPU-side step is unchanged.

Backend	c	Wall-clock	GPU-hours	Steps/h	Final Pass@1
Docker	32	40.6 h	324.8	12	16.0 ± 1.5
NitroBox	32	28.4 **h**	227.2	17	16.2 ± 1.3
Docker	64	*run aborted (infrastructure errors, see Sec. 4.4)*			
NitroBox	64	$\sim$17.0 **h**	$\sim$136	$\sim$29	16.5 ± 1.4

Table 5: End-to-end GRPO training on Qwen2.5-Coder-32B-Instruct, evaluated on Terminal-Bench 1.0. Both backends converge to statistically equivalent Pass@1; NitroBox delivers the same final model in $1.43\times$ less wall-clock at $c=32$. Raising rollout concurrency to $c=64$ pushes the wall-clock advantage to $\sim 2.4\times$, but Docker cannot finish a $c=64$ run in this environment due to the infrastructure errors documented in Section 4.4.

in Figure 2, while NitroBox completes in ≈ 17.0 hours — another $1.67\times$ reduction over the $c=32$ NitroBox run. In other words, the daemon-less sandbox not only accelerates each rollout but also unlocks rollout concurrencies that the daemon-routed baseline simply cannot reach.

5 Limitation and Conclusion

Agentic RL training is increasingly bottlenecked not by GPU computation but by rollout efficiency, which is dominated by the sandbox that hosts each environment. In this work, we presented NitroBox, a sandbox runtime purpose-built for *agentic RL training*, which exhibits a high-frequency, repeatedly-reset lifecycle. NitroBox redesigns the sandbox lifecycle along three axes: eliminating the centralized daemon dependency, stripping features unnecessary for RL, and adding RL-native capabilities, all while preserving Docker image and Compose compatibility for drop-in adoption. Across micro-benchmarks, NitroBox achieves up to $24.6\times$ speedup over Docker on individual lifecycle operations; on end-to-end agentic benchmarks, it delivers up to $2.23\times$ wall-clock reduction while preserving task completion rate; and within a full GRPO training pipeline, it yields a $1.43\times$ wall-clock reduction at identical Pass@1, demonstrating that lifecycle speedups translate directly into training-time savings without affecting convergence. NitroBox also has limitations: our evaluation focuses on a representative set of agentic RL workloads, and broader coverage across more diverse environments and training frameworks remains valuable future work. We hope NitroBox motivates further investigation into the sandbox infrastructure that plays an increasingly central role in modern agentic RL post-training pipelines.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[2] Alibaba. OpenSandbox: A general-purpose sandbox platform for AI applications. `https://github.com/alibaba/OpenSandbox`, March 2026. Released March 3, 2026; accessed 2026-05-06.

[3] ByteDance Agent Infra. AIO sandbox: All-in-one sandbox for AI agents. `https://github.com/agent-infra/sandbox`, 2025. Accessed: 2026-05-06.

[4] 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. *arXiv preprint arXiv:2511.16108*, 2025.

[5] Daytona. Daytona: Sandbox infrastructure for AI agents. `https://www.daytona.io/`, 2026. Accessed: 2026-05-06.

[6] E2B. E2B: The enterprise AI agent cloud. `https://e2b.dev/`, 2026. Accessed: 2026-05-06.

[7] Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, et al. Rollpacker: Mitigating long-tail rollouts for fast, synchronous rl post-training. *arXiv preprint arXiv:2509.21009*, 2025.

[8] Wei Gao, Yuheng Zhao, Tianyuan Wu, Shaopan Xiong, Weixun Wang, Dakai An, Lunxi Cao, Dilxat Muhtar, Zichen Liu, Haizhou Zhao, et al. Rollart: Scaling agentic rl training via disaggregated infrastructure. *arXiv preprint arXiv:2512.22560*, 2025.

[9] Google. gVisor: Application kernel for containers. `https://github.com/google/gvisor`, 2018. Accessed: 2026-05-06.

[10] Zhenyu Han, Ansheng You, Haibo Wang, Kui Luo, Guang Yang, Wenqi Shi, Menglong Chen, Sicheng Zhang, Zeshun Lan, Chunshi Deng, et al. Asyncflow: An asynchronous streaming rl framework for efficient llm post-training. *arXiv preprint arXiv:2507.01663*, 2025.

[11] Jian Hu. Reinforce++: A simple and efficient approach for aligning large language models. *arXiv preprint arXiv:2501.03262*, 2026.

[12] Jian Hu, Xibin Wu, Zilin Zhu, Xianyu, Weixun Wang, Dehao Zhang, and Yu Cao. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. *arXiv preprint arXiv:2405.11143*, 2024.

[13] Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*, 2025.

[14] 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 *The twelfth international conference on learning representations*, 2023.

[15] Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025.

[16] Han Lu, Zichen Liu, Shaopan Xiong, Yancheng He, Wei Gao, Yanan Wu, Weixun Wang, Jiashun Liu, Yang Li, Haizhou Zhao, et al. Part ii: Roll flash–accelerating rlvr and agentic training with asynchrony. *arXiv preprint arXiv:2510.11345*, 2025.

[17] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2, 2014.

[18] Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

[19] Modal Labs. Modal sandboxes: Running untrusted code in Functions. `https://modal.com/docs/guide/sandboxes`, 2026. Accessed: 2026-05-06.

[20] Moonshot AI. Meet Kimi K2.6: Advancing open-source coding. `https://forum.moonshot.ai/t/meet-kimi-k2-6-advancing-open-source-coding/369`, April 2026. Released April 21, 2026; accessed 2026-05-06.

[21] Zengyi Qin, Jinyuan Chen, Yunze Man, Shengcao Cao, Ziqi Pang, Zhuoyuan Wang, Xin Sun, Gen Lin, Han Fang, Ling Zhu, et al. Osgym: Super-scalable distributed data engine for generalizable computer agents. *arXiv preprint arXiv:2511.11672*, 2025.

[22] Alessandro Randazzo and Ilenia Tinnirello. Kata containers: An emerging architecture for enabling mec services in fast and secure way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 209–214. IEEE, 2019.

[23] Red Hat. Podman: A tool for managing OCI containers and pods. `https://podman.io/`, 2018. Accessed: 2026-05-06.

[24] Ning Shang, Yifei Liu, Yi Zhu, Li Lyna Zhang, Weijiang Xu, Xinyu Guan, Buze Zhang, Bingcheng Dong, Xudong Zhou, Bowen Zhang, et al. rstar2-agent: Agentic reasoning technical report. *arXiv preprint arXiv:2508.20722*, 2025.

[25] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y.K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[26] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

[27] Shuang Sun, Huatong Song, Lisheng Huang, Jinhao Jiang, Ran Le, Zhihao Lv, Zongchao Chen, Yiwen Hu, Wenyang Luo, Wayne Xin Zhao, et al. Swe-world: Building software engineering agents in docker-free environments. *arXiv preprint arXiv:2602.03419*, 2026.

[28] Tencent Cloud. CubeSandbox: Instant, concurrent, secure & lightweight sandbox for AI agents. `https://github.com/TencentCloud/CubeSandbox`, 2026. Accessed: 2026-05-06.

[29] Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. TRL: Transformers Reinforcement Learning, 2020.

[30] Bo Wu, Sid Wang, Yunhao Tang, Jia Ding, Eryk Helenowski, Liang Tan, Tengyu Xu, Tushar Gowda, Zhengxing Chen, Chen Zhu, et al. Llamarl: A distributed asynchronous reinforcement learning framework for efficient large-scale llm training. *arXiv preprint arXiv:2505.24034*, 2025.

[31] 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.

[32] Zhuolin Yang, Zihan Liu, Yang Chen, Wenliang Dai, Boxin Wang, Sheng-Chieh Lin, Chankyu Lee, Yangyi Chen, Dongfu Jiang, Jiafan He, et al. Nemotron-cascade 2: Post-training llms with cascade rl and multi-domain on-policy distillation. *arXiv preprint arXiv:2603.19220*, 2026.

[33] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

[34] Danlong Yuan, Wei Wu, Zhengren Wang, Xueliang Zhao, Huishuai Zhang, and Dongyan Zhao. Swe-minisandbox: Container-free reinforcement learning for building software engineering agents. *arXiv preprint arXiv:2602.11210*, 2026.

[35] Zerocore AI. microsandbox: Self-hosted microVM sandboxes for AI agents. `https://github.com/microsandbox/microsandbox`, 2026. Accessed: 2026-05-06.

[36] Hao Zhang, Mingjie Liu, Shaokun Zhang, Songyang Han, Jian Hu, Zhenghui Jin, Yuchi Zhang, Shizhe Diao, Ximing Lu, Binfeng Xu, et al. Prorl agent: Rollout-as-a-service for rl training of multi-turn llm agents. *arXiv preprint arXiv:2603.18815*, 2026.

[37] Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Huazuo Gao, Jiashi Li, Liyue Zhang, Panpan Huang, Shangyan Zhou, Shirong Ma, et al. Insights into deepseek-v3: Scaling challenges and reflections on hardware for ai architectures. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 1731–1745, 2025.

[38] Yinmin Zhong, Zili Zhang, Xiaoniu Song, Hanpeng Hu, Chao Jin, Bingyang Wu, Nuo Chen, Yukun Chen, Yu Zhou, Changyi Wan, et al. Streamrl: Scalable, heterogeneous, and elastic rl for llms with disaggregated stream generation. *arXiv preprint arXiv:2504.15930*, 2025.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state three concrete claims—10.6–24.6× speedups on individual sandbox lifecycle operations, up to 2.23× end-to-end speedup on agentic benchmarks, and 1.43× wall-clock reduction on a full GRPO training pipeline at identical Pass@1—each of which is directly supported by experiments in Sections 4.2, 4.3, and 4.5 respectively.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 5 (Limitation and Conclusion) explicitly discusses that our evaluation focuses on a representative set of agentic RL workloads and that broader coverage across more diverse environments and training frameworks remains future work.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: This is a systems paper that does not contain theoretical results, theorems, or formal proofs.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Section 3.3 describes the NitroBox implementation (three-layer Python/Rust/Go runtime, overlayfs and namespace primitives), Section 4.1 specifies hardware (GCP m1-megamem-96, Linux 6.17, Ubuntu 24.04) and the eleven baseline runtimes compared, and Section 4.5 specifies the training framework (NeMo-RL), model (Qwen2.5-Coder-32B-Instruct), optimizer recipe, and evaluation protocol.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: To preserve anonymity during review, code is not released with the submission. The benchmarks used (Terminal-Bench 2.0, R2E-Gym-Lite, SWE-bench Verified, OSWorld) and the base model (Qwen2.5-Coder-32B-Instruct) are publicly available. Code release is planned upon publication.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 4.1 specifies the workload and trial configuration ($N = 10$ trials with 5-trial warm-up); Section 4.5 specifies the optimizer (AdamW, lr 5×10^{-6}, cosine schedule, warmup ratio 0.1, BF16, DeepSpeed ZeRO-3), batch and group sizes ($c = 32$, $K = 8$), training length (487 policy steps), and evaluation protocol (Average Pass@1 over 3 generations at temperature 0.6).

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Table 2 reports mean $\pm$ standard deviation over $N = 10$ trials for the lifecycle micro-benchmark, and Table 5 reports Pass@1 with $\pm$ standard deviation over three independent generations.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Section 4.1 specifies the hardware (single GCP m1-megamem-96 instance with 96 vCPU Intel Xeon, 1.4 TiB DDR4, 4$\times$375 GB NVMe RAID-0); Section 4.5 specifies the GPU configuration (8$\times$NVIDIA-B300 node) and reports total wall-clock and GPU-hours per training run in Table 5.

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics?

   Answer: [Yes]

   Justification: The research conforms with the NeurIPS Code of Ethics. The work concerns sandbox infrastructure for RL training and does not involve human subjects, sensitive data, or applications that raise ethical concerns.

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: This work is foundational systems infrastructure for accelerating RL training pipelines and does not have direct positive or negative societal impact beyond enabling more efficient research workflows.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse?

    Answer: [N/A]

    Justification: This paper does not release new datasets or pre-trained models. The contribution is a sandbox runtime, which poses no high risk of misuse.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All baseline systems (Docker, Podman, gVisor, Firecracker, e2b, Kata Containers, microsandbox, AIO Sandbox, OpenSandbox, CubeSandbox, SWE-MiniSandbox), benchmarks (Terminal-Bench, R2E-Gym, SWE-bench, OSWorld), models (Qwen2.5-Coder-32B-Instruct), and frameworks (NeMo-RL, vLLM) are cited with their original references in the bibliography.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

Justification: No new datasets, models, or pre-trained checkpoints are released with this submission. The NitroBox runtime itself will be documented and released upon publication.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This work does not involve crowdsourcing or research with human subjects.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This work does not involve human subjects and therefore does not require IRB approval.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research?

    Answer: [N/A]

    Justification: LLMs were not used as a core component of the research methodology. The paper trains an LLM (Qwen2.5-Coder-32B-Instruct) as a workload to evaluate the proposed sandbox runtime, but the core contribution—the NitroBox runtime—does not involve LLMs in its design or implementation.