CUA-SANDBOX: EFFICIENT ENVIRONMENTS FOR COMPUTER-USE REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning enables computer-use agents to improve through interaction with real software environments, including websites and desktop applications. However, conventional deployments replicate an initialized runtime for each independent rollout, even when trajectories use the same software, incurring repeated memory and initialization costs as the number of parallel environments grows. Does an independent computer-use environment require an independent execution runtime? Our key observation is that trajectories require independent mutable state, while initialized application runtimes can be reused across concurrently evolving environments, making state the natural unit of environment independence. Guided by this observation, we introduce CUA-Sandbox, which separates private state capsules from shared runtimes through state-scoped execution and transactional lifecycle operations, including resets and branches, while retaining the original software interfaces and task evaluators. Experiments show comparable or improved task success relative to Docker, while substantially reducing rollout and resource costs. CUA-Sandbox achieves up to a 6.20× increase in rollout throughput, a 9.2× reduction in per-environment memory, and a 504× reduction in incremental storage.

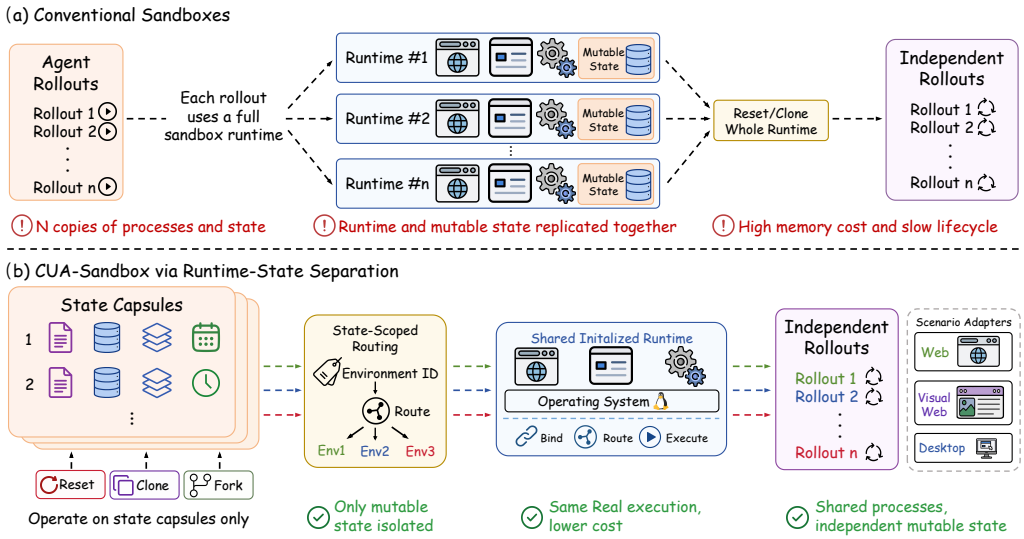


Figure 1: **Environment independence through runtime–state separation.** (a) Conventional sandboxes allocate a complete initialized runtime and mutable state to each rollout, repeating process memory and lifecycle work. (b) CUA-Sandbox shares initialized processes while storing each environment's mutable resources in a private state capsule. State-scoped bindings route actions to the correct capsule, allowing trajectories to evolve independently. Reset, clone, and fork operate on capsule state instead of rebuilding the runtime. Scenario adapters retain the original web, visual-web, and desktop interaction interfaces, so runtime reuse preserves execution in real software.

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.

1 INTRODUCTION

Computer-use agents (CUAs) operate websites, applications, and operating systems through the same interfaces available to people. Their scope has expanded from web tasks (Zhou et al., 2024; Koh et al., 2024) to desktop and mobile systems (Xie et al., 2024; Rawles et al., 2025), including long-horizon workflows that span multiple applications (Yuan et al., 2026b; Wang et al., 2025). Learning these behaviors increasingly relies on reinforcement learning (RL), where agents improve through repeated interaction with executable software environments (Xi et al., 2024; Wang et al., 2026a). Each rollout changes files, sessions, databases, or application state, so parallel trajectories require independent mutable environments.

Providing this independence is expensive. Conventional systems typically instantiate a complete container, virtual machine, or application stack for each environment. Rollouts repeatedly initialize the same software even when they differ only in the state produced by their actions. Memory and lifecycle costs therefore grow with the number of environments, limiting the experience that a training system can collect under a fixed resource budget. This coupling raises a basic question: *does an independent computer-use environment require an independent execution runtime?*

Existing approaches address different parts of this cost. Learned world and experience models approximate transitions (Wang et al., 2026b; Xiao et al., 2026; Chen et al., 2025), while code-generated environments expand the supply of executable tasks (Tu et al., 2026; Dong et al., 2026; Cao et al., 2026). These approaches can provide cheaper experience, but their fidelity is bounded by the dynamics and dependencies they capture. Real software introduces persistent sessions, hidden service state, rendering semantics, and distributed side effects that matter over long trajectories. Containers and microVMs preserve real execution (Merkel, 2014; Agache et al., 2020), and recent systems reduce provisioning and checkpointing costs (Lu et al., 2026b; Yuan et al., 2026a; Wu et al., 2026). Yet a separate initialized runtime commonly remains the unit of environment independence.

Our observation is that application code and initialized services can remain common across rollouts, while their mutable state must diverge. Figure 1 illustrates this distinction. We develop **CUA-Sandbox**, which maintains a private state capsule for each logical environment and binds execution to the appropriate capsule. The system coordinates state transitions with ongoing requests, allowing reset and branching without rebuilding the complete runtime. Task-relevant writes remain private, and resources that cannot be safely shared use private backends.

We evaluate CUA-Sandbox on WebArena, VisualWebArena, and OSWorld through paired inference, matched agent post-training, and systems measurements. The inference study tests whether changing the execution backend preserves task success, while the post-training study examines whether agents retain the benefits of supervised fine-tuning and on-policy reinforcement learning. Rollout and operation-level measurements quantify throughput, resource use, and lifecycle costs. Component ablations separate the savings from copy-on-write, the overhead of managing non-database state, and the correctness effects of transactional publication. Together, these experiments show that runtime reuse reduces environment costs while maintaining measured agent performance.

- **Runtime–state separation.** We separate environment-specific state and bindings from reusable application runtimes.
- **CUA-Sandbox for runtime reuse.** We introduce state capsules, state-scoped execution, and transactional lifecycles across web and desktop environments while retaining their interaction and evaluation interfaces.
- **Efficiency and task fidelity.** Paired inference and post-training evaluations support task fidelity across web and desktop environments. Experiments show comparable or improved task success relative to Docker, while substantially reducing rollout and resource costs. CUA-Sandbox achieves up to a 6.20× increase in rollout throughput, a 9.2× reduction in per-environment memory, and a 504× reduction in incremental storage.

2 RELATED WORK

Computer-Use Agents. Early web-based computer use introduced interactive environments and demonstrations. WebShop (Yao et al., 2022) framed product search and purchase as interaction, while

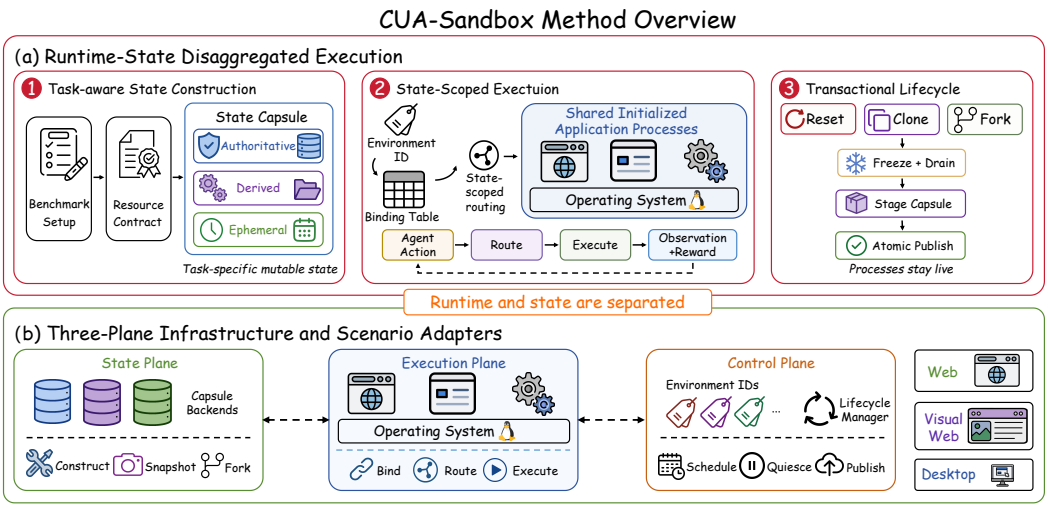


Figure 2: **CUA-Sandbox architecture and execution flow.** (a) Task-aware construction turns benchmark setup and resource contracts into capsules of authoritative, derived, and ephemeral state. Environment IDs and binding tables route actions through shared initialized processes to produce observations and rewards. Reset, clone, and fork freeze and drain ongoing work, stage successor capsules, and atomically publish their bindings while processes remain live. (b) The state plane manages capsule backends; the execution plane binds and executes actions; the control plane schedules environments and coordinates lifecycle transitions. Scenario adapters expose this infrastructure to web, visual-web, and desktop tasks.

Mind2Web (Deng et al., 2023) collected trajectories across diverse websites. WebArena (Zhou et al., 2024) established self-hosted, stateful websites with functional evaluation; VisualWebArena (Koh et al., 2024) added visually grounded tasks, and WorkArena (Drouin et al., 2024) extended evaluation to enterprise workflows. WebVoyager (He et al., 2024) studied interaction on live websites, while SeeClick (Cheng et al., 2024) advanced visual grounding across interfaces. Beyond browsers, OSWorld (Xie et al., 2024) and Windows Agent Arena (Bonatti et al., 2024) covered desktop systems, AndroidWorld (Rawles et al., 2025) targeted mobile devices, and OSWorld 2.0 (Yuan et al., 2026b) introduced long-horizon professional workflows. Recent general computer-use agents include UI-TARS (Qin et al., 2025) and OpenCUA (Wang et al., 2025), which scale data and model training, followed by Qwen-UI-Agent (Zhou et al., 2026) and Qwen-CUA (Lu et al., 2026a) across web, desktop, and mobile platforms. However, broader platforms, longer horizons, and richer state require many complete environment instances, making execution infrastructure a growing bottleneck for scalable training and evaluation.

Reinforcement Learning Environments. Early agentic RL relied on manually constructed executable environments. AgentGym (Xi et al., 2024) unified diverse domains, τ-bench (Yao et al., 2024) modeled stateful tool–agent–user interactions, and AppWorld (Trivedi et al., 2024) exposed controllable applications with executable evaluation. Their domain-specific interfaces, tasks, initialization, and verifiers make construction and large-scale instantiation costly. To reduce authoring cost, ScaleEnv (Tu et al., 2026) synthesized interactive environments, Gym-Anything (Aggarwal et al., 2026) converted arbitrary software into agent environments, and Agent-World (Dong et al., 2026) discovered executable tool ecosystems and generated verifiable tasks. GUI-GENESIS (Cao et al., 2026), ClawGym (Bai et al., 2026), and CUA-Gym (Wang et al., 2026a) similarly scaled construction for software-interacting agents. Another line reduced rollout cost through simulation. DreamGym (Chen et al., 2025) distilled environment dynamics into an experience model, while Agent World Model (Wang et al., 2026b), WebWorld (Xiao et al., 2026), Qwen-AgentWorld (Zuo et al., 2026), and SearchEyes (Jiao et al., 2026) built learned or structured simulated worlds. Dockerless (Zeng et al., 2026) replaced execution-based rewards with environment-free verification. Yet simulation and execution-free feedback omit application dynamics, hidden state, and real side effects, creating a persistent sim-to-real gap; high-fidelity agent learning therefore remains dependent on costly, stateful software execution at scale.

Agent Execution Infrastructure. Docker (Merkel, 2014) established containers as the standard substrate for reproducible, isolated execution. SOCK (Oakes et al., 2018) and SAND (Akkus et al., 2018) reduced provisioning overhead through specialization and sandbox sharing, while Firecracker (Agache et al., 2020) introduced lightweight microVMs for dense isolation. For serverless functions, SEUSS (Cadden et al., 2020) and Catalyzer (Du et al., 2020) reused initialized state through snapshots and sandbox forking, and Groundhog (Alzayat et al., 2023) restored a shared runtime between sequential invocations. Agent workloads, however, are long-running and stateful. WebServ (Lu et al., 2026b) applied block-level copy-on-write to web environments, SWE-MiniSandbox (Yuan et al., 2026a) combined kernel isolation with reusable caches, and Orchard (Peng et al., 2026) exposed reusable sandbox-lifecycle primitives. AgentCgroup (Zheng et al., 2026), Crab (Wu et al., 2026), Sandlock (Wang & Zheng, 2026), and SpecBox (Zhang et al., 2026) improved resource control, checkpointing, confinement, and scheduling, while the Rollout Infrastructure Tax (Graviet et al., 2026) quantified agent-rollout systems overhead. Yet these systems still allocate a separate runtime to each concurrent environment, whereas prior runtime sharing assumes stateless or sequential execution. Supporting concurrently diverging mutable states without replicating complete runtimes remains an open systems problem for large-scale agent training.

3 CUA-SANDBOX

3.1 OVERVIEW

CUA-Sandbox separates the mutable state of a logical environment from the runtime that executes its actions. Each environment maintains a private *state capsule*, while initialized application processes are reused across environments. During execution, environment-specific bindings direct actions to the corresponding capsule. Resetting or branching an environment therefore operates on its state, while the shared runtime remains initialized.

Consider N environments running the same application stack. A conventional deployment couples each environment's state to its own initialized processes. CUA-Sandbox instead makes the state and its bindings the environment-specific components:

$$\mathcal{E}_i^{\mathrm{rep}} = \langle P_i, B, S_i \rangle, \qquad \mathcal{E}_i^{\mathrm{CUA}} = \langle P, B, K_i, \Phi_i \rangle. \tag{1}$$

Here, P_i denotes a private initialized runtime and P the shared runtime. Immutable software and base data B may already be shared in either design. Capsule K_i represents trajectory state S_i, and Φ_i binds application resources to that capsule. The change concerns environment realization; task setup, interfaces, and evaluation remain the same.

Figure 2 outlines the execution flow. The system constructs a capsule from the task's state requirements, binds an action and its resulting observations to that capsule, and coordinates state transitions when the environment is reset or branched. We describe capsule construction in Section 3.2, state-scoped execution in Section 3.3, and lifecycle management in Section 3.4. These mechanisms target two requirements: mutations remain private to their environment, and execution preserves the task-relevant behavior of the underlying software.

3.2 ENVIRONMENT STATE CAPSULES

The first step is to identify the mutable resources that determine future observations and rewards. A capsule collects these resources into one independently managed environment state, allowing the application runtime to be reused without merging trajectories.

Identifying mutable state. A task resource contract specifies the databases, user files, browser profiles, service state, and interface resources required by a task family. It is constructed from task setup, evaluator dependencies, and application-state specifications. For each resource, the contract selects a backend adapter and a lifecycle policy. For example, two shopping rollouts can use the same application server while resolving database and session access to separate environment state. Resources without a fine-grained adapter fall back to a coarse private backend; filesystem copy-on-write covers writes outside the fine-grained file entries. This fallback trades sharing efficiency for a larger private state boundary.

Constructing state capsules. We distinguish three classes of state according to how they must be preserved. *Authoritative state*, such as databases and user-created files, is stored as a private delta over

immutable base data. *Derived state*, such as caches and search indexes, uses private namespaces and may be reconstructed when rebuilding is cheaper than copying. *Ephemeral state*, such as temporary IPC endpoints and display resources, is recreated when a capsule is activated. Their backends differ, but all three belong to the same logical environment.

Each capsule has an environment identity, a generation number, and a manifest recording its component versions and dependencies. Backend adapters expose a common interface for preparation, quiescence, snapshotting, forking, resetting, activation, and destruction. The manifest commits only after all required components are ready, so a database branch, file overlay, and application profile advance as one generation. Appendix D gives the resource and binding representations.

3.3 State-Scoped Execution

State capsules define what remains private; state-scoped execution determines how shared processes access it. Every action executes with an environment identity and a binding table that selects the corresponding data branches, service namespaces, and interface endpoints.

Binding actions to state. Before an action executes, the control plane resolves the environment's current bindings and acquires a generation lease. For authoritative resource k, the resulting view is

$$V_{i,k}^{g} = b_k \oplus \delta_{i,k}^{g}, \tag{2}$$

where b_k is immutable base data, $\delta_{i,k}^{g}$ is the private branch for environment i at generation g, and $\oplus$ denotes backend-specific overlay composition. Reads use the composed view, while writes terminate in the private branch. Derived services and ephemeral interfaces are selected through their environment-specific namespaces and endpoints. The action's causal requests, callbacks, and background work inherit the same context; observations and evaluator access use the same generation.

Maintaining isolation. Runtime sharing requires complete resource bindings, disjoint writable state, and a consistent generation throughout an operation. If W_i^g denotes the writable state reachable under an environment's bindings, the isolation requirement is

$$W_i^g \cap W_j^{g'} = \emptyset, \qquad i \neq j. \tag{3}$$

Immutable bases may be shared, but each environment's writes must resolve to its own branches and namespaces. A context becomes ready only after every contracted resource has a binding. Generation-scoped leases keep ongoing work attached to one version and reject stale requests after a transition. These requirements depend on covering the task's mutable resources and propagating context through their adapters; sharing a process alone does not establish isolation.

The binding granularity follows the application. Web services can select private state per request, while browser and desktop interactions require session-scoped profiles, configuration, display, and message-bus endpoints. Both cases expose the original interaction interface to the agent. Because execution can be asynchronous or nondeterministic, fidelity is evaluated through task-relevant observations, evaluator-visible state, and final outcomes.

3.4 Lifecycle and Runtime Reuse

Once execution is bound to private state, lifecycle operations can replace or branch that state without rebuilding the shared runtime. The control plane coordinates these operations with ongoing execution so that actions never observe a partially updated capsule.

State-level operations. Reset constructs a clean successor capsule from the immutable base. Snapshot records an immutable version of the current capsule, while clone and fork create copy-on-write descendants that share unchanged data and diverge as new actions modify them. These operations preserve the initialized application processes and act on the capsule's component backends.

Atomic transitions. A transition freezes the environment's route and drains in-flight requests and background work. Backends stage a successor capsule and its bindings. After all components commit and pass readiness checks, the control plane atomically publishes the new generation:

$$\mathrm{active}[i] \leftarrow \langle g+1, \Phi_i^{g+1} \rangle. \tag{4}$$

Table 1: **Inference performance with matched execution backends.** Success rates (%) are unweighted site/category means. Shaded rows show CUA-Sandbox; bold marks pairwise maxima. Site-level scores appear in Appendix B.

Model	Backend	WebArena	VisualWebArena	OSWorld
GPT-5.6-Terra	Docker	21.89	28.34	67.29
	CUA-Sandbox	**22.43**	**29.06**	**70.62**
Qwen3.5-9B	Docker	9.67	10.52	24.17
	CUA-Sandbox	**10.49**	**13.54**	**27.66**
Muse-Glimmer-30B	Docker	23.14	25.36	24.40
	CUA-Sandbox	**25.63**	**25.80**	**34.39**
Gemma-4-31B-IT	Docker	24.97	19.07	32.61
	CUA-Sandbox	**26.34**	**20.36**	**35.54**

Table 2: **Qwen3.5-4B performance before and after training.** Success rates (%) follow Base $\rightarrow$ SFT $\rightarrow$ RL under each training backend. Final checkpoints share the reference evaluation pipeline. Bold marks final RL scores; Δ is the RL gain over SFT in percentage points.

Benchmark	Training backend	Base	SFT	RL	Δ (pp)
WebArena	Docker	8.25	9.85	**12.93**	+3.08
	CUA-Sandbox	8.25	10.10	**13.05**	+2.95
VisualWebArena	Docker	18.13	23.84	**27.03**	+3.19
	CUA-Sandbox	18.13	24.07	**27.03**	+2.96
OSWorld	Docker	14.96	21.61	**25.48**	+3.87
	CUA-Sandbox	14.96	22.16	**26.04**	+3.88

A failed transition retains the previous published generation. Together with generation-scoped leases, this protocol prevents delayed actions or evaluator requests from crossing a state transition and prevents execution against mixed component versions.

Reusing execution slots. The number of materialized capsules need not equal the number of active execution contexts. CUA-Sandbox maintains N logical environments and allocates Q physical execution slots as needed; inactive environments remain as capsules. Rebinding a slot quiesces its current context, attaches the target capsule, recreates ephemeral resources, and updates the route before passing a readiness barrier. A slot therefore has no permanent environment identity. The state plane manages capsules, the execution plane applies their bindings, and the control plane schedules slots and publishes lifecycle transitions. A memory and storage model appears in Appendix E.

4 Experiments

We evaluate whether runtime sharing preserves agent performance, improves rollout efficiency, and reduces environment costs. We then ablate the state and lifecycle mechanisms to separate their contributions. Tables 1–3 and Figures 3–4 address these questions; detailed evaluation settings and scaling analysis are deferred to the appendix.

4.1 Experimental Setup

Benchmarks and models. We evaluate WebArena, VisualWebArena, and OSWorld using their original task interfaces and evaluators (Zhou et al., 2024; Koh et al., 2024; Xie et al., 2024). The inference study uses GPT-5.6-Terra, Qwen3.5-9B, Muse-Glimmer-30B, and Gemma-4-31B-IT. Post-training uses WebArena, VisualWebArena, and OSWorld, with Qwen3.8-27B as the teacher and Qwen3.5-4B as the student.

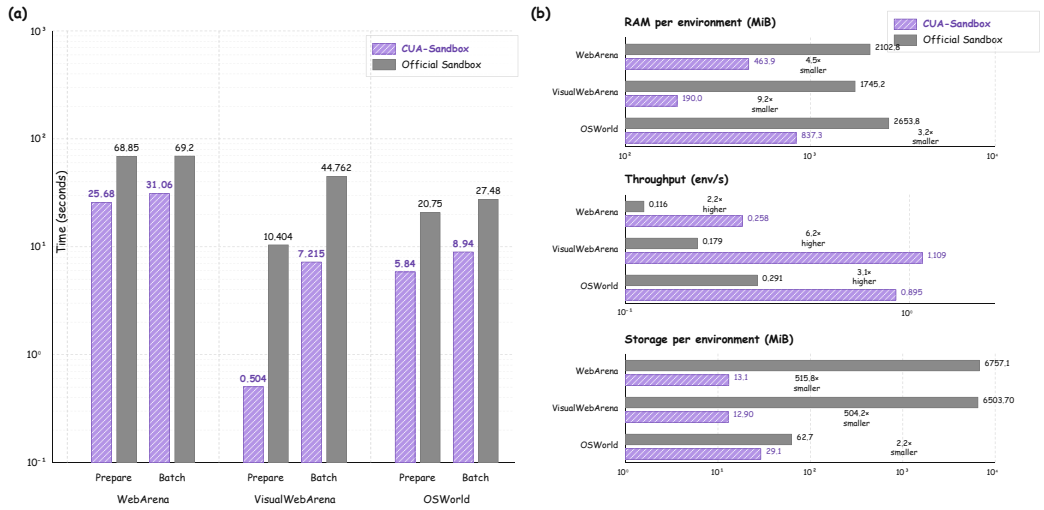


Figure 3: **Rollout efficiency with eight concurrent environments.** (a) Preparation and batch wall-clock time. (b) Per-environment memory, throughput, and incremental storage. Purple hatched bars show CUA-Sandbox; gray bars show the Docker baseline (Official Sandbox). Axes are logarithmic. Lower time and resource costs and higher throughput are better. Table 6 lists the measurements.

Backends and controls. We compare CUA-Sandbox with the conventional Docker-based environment for each benchmark under matched experimental conditions. Each inference pair uses the same model, prompts, decoding settings, interaction horizon, task interface, and official evaluator, so that the execution backend is the primary experimental variable. For post-training, both comparisons start from the same Qwen3.5-4B base checkpoint. Successful teacher trajectories are collected independently under each backend to construct the corresponding SFT data, after which on-policy RL is performed using the same backend as trajectory collection. Across the two training runs, we match the data and rollout budgets, task sampling procedure, decoding configuration, optimization hyperparameters, reward function, and interaction horizon. Final checkpoints are evaluated through the same reference evaluation pipeline, ensuring that differences in measured task performance are not introduced by changes in prompts, evaluators, or training budgets.

Metrics and compute. We report task success and its unweighted mean across sites or task categories, keeping each benchmark separate. The rollout study fixes concurrency at eight environments and measures preparation time, batch wall-clock time, throughput, memory, and incremental storage. Operation-level measurements separately cover resource use and five lifecycle operations. All runs use a host with 8 NVIDIA RTX PRO 5000 Blackwell GPUs (72 GB each), 192 physical CPU cores, 1.5 TiB memory, and two approximately 3.5 TB NVMe drives, running CUDA 13.0. Appendix C.1 gives decoding and evaluation details.

4.2 Agent Performance

Runtime sharing preserves task performance across agents. CUA-Sandbox scores above Docker in all twelve model–benchmark pairs (Table 1). For GPT-5.6-Terra, success increases from 21.89% to 22.43% on WebArena-Lite, from 28.34% to 29.06% on VisualWebArena, and from 67.29% to 70.62% on OSWorld. The same direction holds for Qwen3.5-9B, Muse-Glimmer-30B, and Gemma-4-31B-IT despite their different starting scores. This consistency supports runtime reuse across agent families and interaction interfaces under the tested prompts and evaluators.

The efficiency gains carry through agent post-training. The CUA-Sandbox student finishes 0.12 percentage points above Docker on WebArena, matches it on VisualWebArena, and finishes 0.56 points above it on OSWorld (Table 2). OSWorld provides the closest comparison of the RL stage: success rises from 22.16% to 26.04% with CUA-Sandbox and from 21.61% to 25.48% with Docker, gains of 3.88 and 3.87 points. Together with the web results, these matched runs show that the cheaper execution backend retains the measured benefit of on-policy interaction.

Table 3: **Where runtime sharing saves resources and lifecycle work.** Entries are conventional-environment costs divided by CUA-Sandbox costs (×); larger is better. The first two rows report resource reductions, and the remaining rows report operation speedups. Shaded bold cells mark the largest factor per metric.

	WebArena			VisualWebArena			OSWorld
Metric	Shopping Admin	Shopping	GitLab	Shopping	Reddit	Classifieds	Desktop
Memory	20.6	46.5	**51.2**	1.92	22.0	3.70	1.45
Storage	60.1	543.6	6.3	391.2	**9,535**	3.09	3.67
Create	1.70	1.15	8.82	**15.17**	8.75	6.89	2.98
Reset	1.20	1.03	5.51	4.99	**7.29**	2.66	3.17
Restore	1.20	1.21	6.62	5.78	**7.39**	2.52	2.90
Fork	1.10	1.25	9.37	**17.84**	10.18	6.65	2.98
Clone	2.13	1.17	**7.47**	2.10	1.96	3.30	1.93

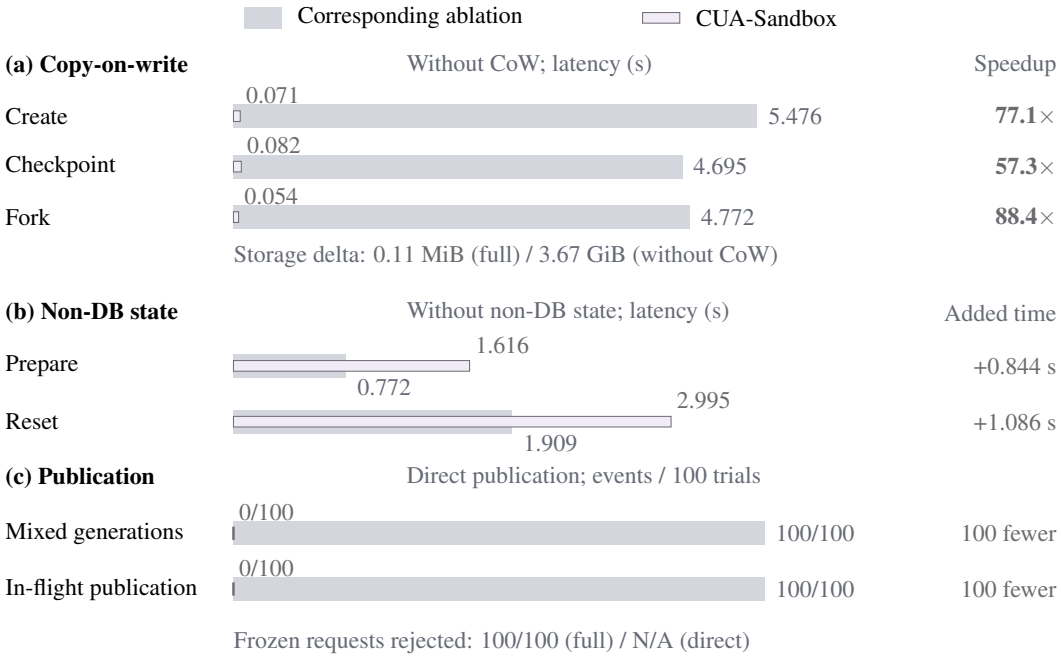


Figure 4: **Component ablations separate efficiency costs from lifecycle correctness.** Thin purple bars show CUA-Sandbox; wide gray bars show the named ablation. Bar lengths use a shared linear scale within each panel; labels give exact values. (a) Copy-on-write on a 3.6 GB Shopping capsule. (b) Non-DB state management on read-only WebArena task 21. (c) Publication events over 100 trials; purple origin markers denote zero events, and N/A denotes an inapplicable comparison.

4.3 Rollout Efficiency

Preparation savings increase end-to-end rollout throughput. VisualWebArena shows the largest throughput gain: preparation falls from 10.404 to 0.504 s, batch time falls from 44.762 to 7.215 s, and throughput rises from 0.179 to 1.109 environments/s (Figure 3). The 20.64× preparation speedup becomes a 6.20× batch speedup, showing that initialization accounts for only part of the rollout cost. Throughput also increases by 2.22× on WebArena and 3.08× on OSWorld. Runtime reuse therefore improves the complete measured batch, beyond the environment setup step.

Higher throughput comes with a smaller per-environment footprint. On WebArena, memory falls from 2,102.8 to 463.9 MiB while incremental storage falls from 6,757.1 to 13.1 MiB. VisualWebArena shows a similarly strong reduction, with memory falling from 1,745.2 to 190.0 MiB and incremental storage falling from 6,503.70 to 12.90 MiB. OSWorld memory falls from 2,653.8 to 837.3 MiB, while incremental storage falls from 62.7 to 29.1 MiB. The larger web storage savings are consistent with sharing initialized application state that would otherwise be materialized per environment. These results concern eight concurrent environments; maximum density remains unmeasured.

4.4 Environment Resource and Lifecycle Costs

Memory and storage savings peak on different workloads. WebArena GitLab reduces memory by 51.2× but storage by 6.3×; VisualWebArena Reddit reduces memory by 22.0× and storage by 9,535× (Table 3). A workload with the largest memory benefit need not have the largest storage benefit. The two metrics capture different costs of maintaining independent environments, and the private state that remains after runtime sharing depends on the application.

Lifecycle speedups depend on both the operation and the application. VisualWebArena Shopping achieves a 15.17× create speedup and a 17.84× fork speedup, whereas WebArena Shopping improves by 1.15× and 1.25× on the same operations. Its reset speedup is only 1.03×. These differences delimit the benefit of runtime reuse: avoiding process initialization helps most where it dominates the original operation, while capsule preparation and readiness checks still incur latency.

4.5 Ablation Study

We remove copy-on-write, non-DB state management, and transactional publication in turn to separate their effects on cost and correctness. Appendix A.1 provides controls and measurement details, including task selection, readiness checks, and stress-test conditions. More experients are provided in Appendix F

Copy-on-write removes the dominant state-materialization cost. With the same 3.6 GB Shopping capsule, removing copy-on-write raises create latency from 0.071 to 5.476 s and fork latency from 0.054 to 4.772 s (Figure 4). The storage delta rises from 0.11 MiB to 3.67 GiB. Because the capsule and lifecycle operations are otherwise unchanged, this ablation identifies state materialization as the principal cost avoided by copy-on-write.

Managing non-DB state adds adapter overhead. On WebArena task 21, including non-DB state adds 0.844 s to preparation and 1.086 s to reset; all four HTTP readiness probes return status 200. This cost is the work required to manage resources beyond the database. The task is read-only, so the comparison isolates adapter overhead rather than isolation under state-changing actions.

Transactional publication prevents mixed-generation execution. In 100 stress-test trials, the full protocol exposes no mixed generations and never publishes while requests remain in flight. Direct publication produces both events in all 100 trials. The full protocol also rejects all 100 requests issued while the route is frozen. These controls show why state reuse requires a coordinated transition: successor state must become visible only after ongoing execution has drained.

5 Conclusion

CUA-Sandbox separates the mutable state of computer-use environments from their initialized execution runtime. State capsules, environment-scoped bindings, and transactional lifecycle operations enable reuse while retaining the original software interfaces and evaluators. Across web and desktop workloads, the measured resource and lifecycle savings translate into higher rollout throughput, with maintained task performance and similar post-training improvements. The results support runtime reuse as a practical systems component for agent training. The ablations clarify this tradeoff: copy-on-write avoids state materialization, while non-database adapters incur measurable overhead and transactional publication protects lifecycle consistency. The approach therefore depends on complete resource contracts and binding coverage, with private backends retained wherever safe sharing cannot be established. Broader mutation workloads, longer trajectories, and concurrency sweeps are needed to characterize the remaining sharing boundary, stress state isolation under diverse write patterns, and determine how these gains translate into achievable environment density.

AI USE STATEMENT

AI tools assisted with language editing, manuscript organization, and LaTeX presentation.

REFERENCES

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)*, pp. 419–434. USENIX Association, 2020.

Pranjal Aggarwal, Graham Neubig, and Sean Welleck. Gym-Anything: Turn any software into an agent environment. *arXiv preprint arXiv:2604.06126*, 2026.

Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 923–935. USENIX Association, 2018.

Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient request isolation in FaaS. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pp. 398–415. ACM, 2023. doi: 10.1145/3552326.3567503.

Fei Bai, Huatong Song, Shuang Sun, Daixuan Cheng, Yike Yang, Chuan Hao, Renyuan Li, Feng Chang, Yuan Wei, Ran Tao, Bryan Dai, Jian Yang, Wayne Xin Zhao, and Ji-Rong Wen. ClawGym: A scalable framework for building effective claw agents. *arXiv preprint arXiv:2604.26904*, 2026.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Jang, and Zack Hui. Windows agent arena: Evaluating multi-modal OS agents at scale. *arXiv preprint arXiv:2409.08264*, 2024.

James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: Skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–15. ACM, 2020. doi: 10.1145/3342195.3392698.

Yuan Cao, Dezhi Ran, Mengzhou Wu, Yuzhe Guo, Xin Chen, Ang Li, Gang Cao, Gong Zhi, Hao Yu, Linyi Li, Wei Yang, and Tao Xie. GUI-GENESIS: Automated synthesis of efficient environments with verifiable rewards for GUI agent post-training. *arXiv preprint arXiv:2602.14093*, 2026.

Zhaorun Chen, Zhuokai Zhao, Kai Zhang, Bo Liu, Qi Qi, Yifan Wu, Tarun Kalluri, Sara Cao, Yuanhao Xiong, Haibo Tong, Huaxiu Yao, Hengduo Li, Jiacheng Zhu, Xian Li, Dawn Song, Bo Li, Jason Weston, and Dat Huynh. Scaling agent learning via experience synthesis. *arXiv preprint arXiv:2511.03773*, 2025.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. SeeClick: Harnessing GUI grounding for advanced visual GUI agents. *arXiv preprint arXiv:2401.10935*, 2024.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a generalist agent for the web. *arXiv preprint arXiv:2306.06070*, 2023.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, Jiajie Jin, Yutao Zhu, Hanbin Wang, Fangyu Lei, Qinyu Luo, Mingyang Chen, Zehui Chen, Jiazhan Feng, Ji-Rong Wen, and Zhicheng Dou. Agent-World: Scaling real-world environment synthesis for evolving general agent intelligence. *arXiv preprint arXiv:2604.18292*, 2026.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, Léo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. WorkArena: How capable are web agents at solving common knowledge work tasks? *arXiv preprint arXiv:2403.07718*, 2024.

Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 467–481. ACM, 2020. doi: 10.1145/3373376.3378512.

Daniel Thi Graviet, Lovre Pesut, Ivan Dagelic, Vedran Jukic, and Ivan Burazin. The rollout infrastructure tax in coding-agent reinforcement learning. *arXiv preprint arXiv:2607.01415*, 2026.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. WebVoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*, 2024.

Zhengbo Jiao, Yiming Cheng, Yilei Jiang, Kaituo Feng, Rui Huang, Tianyi Jiang, Juanxi Tian, Jiapeng Li, Qunzhong Wang, Tailai Chen, Qianshan Wei, Chuan Xiao, Shanyu Rong, Yangfu Li, Yanhan Zhou, Yunpu Ma, Yifan Zhang, and Xiangyu Yue. SearchEyes: Towards frontier multimodal deep search intelligence via search world simulation. *arXiv preprint arXiv:2607.05943*, 2026.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. VisualWebArena: Evaluating multimodal agents on realistic visual web tasks. *arXiv preprint arXiv:2401.13649*, 2024.

Dunjie Lu, Shuai Bai, Tianyi Bai, Sicheng Fan, Chang Gao, Jian Guan, Feng Hu, Mianqiu Huang, Xingyang Huang, Yizhen Jiang, Yuheng Jing, Dehui Kong, Ning Li, Dayiheng Liu, Shixuan Liu, Zheng Liu, Que Shen, Bowen Wang, Junli Wang, Chencan Wu, Rui Xie, Tianbao Xie, Zhihui Xie, Haiyang Xu, An Yang, Tao Yu, Wenzhen Yuan, Xi Zhang, Zhenru Zhang, Mingkang Zhu, Zhaoqing Zhu, Yizhong Cao, Kai Dang, Binyuan Hui, Kaixin Li, Junyang Lin, Haiquan Wang, Zekun Wang, Yiheng Xu, Fan Yan, Mengqi Yuan, Danyang Zhang, Jiajun Zhang, Zhipeng Zhang, Fan Zhou, and Fan Zhou. Qwen-CUA: Native computer use for (almost) everything. *arXiv preprint arXiv:2608.02352*, 2026a.

Yuxuan Lu, Ziyi Wang, Jing Huang, Hui Liu, Jiri Gesi, Yan Han, Shihan Fu, Tianqi Zheng, Xianfeng Tang, Chen Luo, Yisi Sang, Jin Lai, and Dakuo Wang. WEBSERV: A full-stack and RL-ready web environment for training web agents at scale. *arXiv preprint arXiv:2510.16252*, 2026b.

Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2014.

Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 57–70. USENIX Association, 2018.

Baolin Peng, Wenlin Yao, Qianhui Wu, Hao Cheng, Xiao Yu, Rui Yang, Tao Ge, Alessandro Sordoni, Xingdi Yuan, Yelong Shen, Pengcheng He, Tong Zhang, Zhou Yu, and Jianfeng Gao. Orchard: An open-source agentic modeling framework. *arXiv preprint arXiv:2605.15040*, 2026.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, Xiaojun Xiao, Kai Cai, Chuang Li, Yaowei Zheng, Chaolin Jin, Chen Li, Xiao Zhou, Minchao Wang, Haoli Chen, Zhaojian Li, Haihua Yang, Haifeng Liu, Feng Lin, Tao Peng, Xin Liu, and Guang Shi. UI-TARS: Pioneering automated GUI interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. AndroidWorld: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573*, 2025.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. *arXiv preprint arXiv:2407.18901*, 2024.

Dunwei Tu, Hongyan Hao, Hansi Yang, Yihao Chen, Yi-Kai Zhang, Zhikang Xia, Yu Yang, Yueqing Sun, Xingchen Liu, Furao Shen, Qi Gu, Hui Su, and Xunliang Cai. ScaleEnv: Scaling environment synthesis from scratch for generalist interactive tool-use agent training. *arXiv preprint arXiv:2602.06820*, 2026.

Bowen Wang, Dunjie Lu, Junli Wang, Tianyi Bai, Shixuan Liu, Zhipeng Zhang, Haiquan Wang, Hao Hu, Tianbao Xie, Shuai Bai, Dayiheng Liu, Que Shen, Junyang Lin, and Tao Yu. CUA-Gym: Scaling verifiable training environments and tasks for computer-use agents. *arXiv preprint arXiv:2605.25624*, 2026a.

Cong Wang and Yusheng Zheng. Sandlock: Confining ai agent code with unprivileged linux primitives, 2026. URL `https://arxiv.org/abs/2605.26298`.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, Zhennan Shen, Zhuokai Li, Ryan Li, Xiaochuan Li, Junda Chen, Boyuan Zheng, Peihang Li, Fangyu Lei, Ruisheng Cao, Yeqiao Fu, Dongchan Shin, Martin Shin, Jiarui Hu, Yuyan Wang, Jixuan Chen, Yuxiao Ye, Danyang Zhang, Dikang Du, Hao Hu, Huarong Chen, Zaida Zhou, Haotian Yao, Ziwei Chen, Qizheng Gu, Yipu Wang, Heng Wang, Diyi Yang, Victor Zhong, Flood Sung, Y. Charles, Zhilin Yang, and Tao Yu. OpenCUA: Open foundations for computer-use agents. *arXiv preprint arXiv:2508.09123*, 2025.

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent world model: Infinity synthetic environments for agentic reinforcement learning. *arXiv preprint arXiv:2602.10090*, 2026b.

Tianyuan Wu, Chaokun Chang, Lunxi Cao, Wei Gao, and Wei Wang. Crab: A semantics-aware checkpoint/restore runtime for agent sandboxes. *arXiv preprint arXiv:2604.28138*, 2026.

Zhiheng Xi, Yiwen Ding, Wenxiang Chen, Boyang Hong, Honglin Guo, Junzhe Wang, Dingwen Yang, Chenyang Liao, Xin Guo, Wei He, Songyang Gao, Lu Chen, Rui Zheng, Yicheng Zou, Tao Gui, Qi Zhang, Xipeng Qiu, Xuanjing Huang, Zuxuan Wu, and Yu-Gang Jiang. AgentGym: Evolving large language model-based agents across diverse environments. *arXiv preprint arXiv:2406.04151*, 2024.

Zikai Xiao, Jianhong Tu, Chuhang Zou, Yuxin Zuo, Zhi Li, Peng Wang, Bowen Yu, Fei Huang, Junyang Lin, and Zuozhu Liu. WebWorld: A large-scale world model for web agent training. *arXiv preprint arXiv:2602.14721*, 2026.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *arXiv preprint arXiv:2404.07972*, 2024.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. WebShop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems*, 2022.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Danlong Yuan, Wei Wu, Enhan Zhao, 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*, 2026a.

Mengqi Yuan, Zilong Zhou, Xinzhuang Xiong, Weiming Wu, Jiayang Sun, Jiamin Song, Kaiqian Cui, Bowen Wang, Haoyuan Wu, Yitong Li, Dunjie Lu, Haikong Lu, Qi Zhen, Xinyuan Wang, Jiaqi Deng, Yuhao Yang, Cheng Chen, Boyuan Zheng, Alex Su, Xiao Yu, Hao Zou, Saaket Agashe, Xing Han Lu, Manpreet Kaur, Zhengyang Qi, Vincent Sunn Chen, Frederic Sala, Dayiheng Liu, Junyang Lin, Zhou Yu, Yu Su, Siva Reddy, Xin Eric Wang, Peng Qi, Tianbao Xie, and Tao Yu. OSWorld 2.0: Benchmarking computer use agents on long-horizon real-world tasks. *arXiv preprint arXiv:2606.29537*, 2026b.

Wenhao Zeng, Yuling Shi, Xiaodong Gu, Chao Hu, Chaofan Wang, Yuhao Cui, Hongting Zhou, Mengnan Qi, Jianqiao Wangni, Zhaojian Yu, Shuzheng Gao, Kai Cai, and Shilin He. Dockerless: Environment-free program verifier for coding agents. *arXiv preprint arXiv:2606.28436*, 2026.

Yihui Zhang, Tianyu Wo, Jinghao Wang, Xiaoyang Sun, Menghao Zhang, Cangzhou Yuan, Li Li, Chunming Hu, Albert Y. Zomaya, and Renyu Yang. SpecBox: Speculative sandbox scheduling for efficient LLM agent serving. *arXiv preprint arXiv:2607.23933*, 2026.

Yusheng Zheng, Jiakun Fan, Quanzhi Fu, Yiwei Yang, Wei Zhang, and Andi Quinn. AgentCgroup: Understanding and controlling OS resources of AI agents. *arXiv preprint arXiv:2602.09345*, 2026.

Hanzhang Zhou, Panrong Tong, Xu Zhang, Quyu Kong, Chenglin Cai, Tianyu Xia, Gongjie Zhang, Jianan Zhang, Long Li, Long Chen, Lei Wang, Gaole Dai, Pengxiang Li, Liangyu Chen, Yue Wang, and Steven Hoi. Qwen-UI-Agent technical report: Toward next-generation real-world centric foundation GUI agents. *arXiv preprint arXiv:2607.28227*, 2026.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2024.

Yuxin Zuo, Zikai Xiao, Li Sheng, Fei Huang, Jianhong Tu, Yuxuan Liu, Tianyi Tang, Xiaomeng Hu, Yang Su, Qingfeng Lan, Yantao Liu, Qin Zhu, Yinger Zhang, Bowen Yu, Haiquan Zhao, Haiyang Xu, Jianxin Yang, Jiayang Cheng, Junyang Wang, Lianghao Deng, Mingfeng Xue, Tianyi Bai, Yang Fan, Yubo Ma, Yucheng Li, Zeyu Cui, Zhihai Wang, Zhihui Xie, Zhuorui Ye, An Yang, Dayiheng Liu, Jingren Zhou, and Ning Ding. Qwen-AgentWorld: Language world models for general agents. *arXiv preprint arXiv:2606.24597*, 2026.

A Supplementary Ablations and Execution Configurations

We first report the controls and interpretation of the component ablations in Figure 4, then compare additional execution configurations. Each ablation removes one mechanism from complete CUA-Sandbox; the separate Docker comparison evaluates the execution backend as a whole.

A.1 Component Ablation Details

Copy-on-write. Both configurations use the same 3.6 GB Shopping MySQL capsule. We measure create, checkpoint, and fork latency, together with the physical storage delta for a capsule operation. Removing copy-on-write raises the three latencies from 0.071, 0.082, and 0.054 s to 5.476, 4.695, and 4.772 s, respectively. Storage grows from 0.11 MiB to 3.67 GiB. Figure 4 computes speedups from these reported latency values; storage retains its original units.

Non-database state. The comparison uses WebArena task 21 and removes non-DB state management while retaining the rest of the system. Preparation takes 1.616 s with the complete system and 0.772 s after removal; reset takes 2.995 and 1.909 s. The resulting overheads are 0.844 and 1.086 s, and all four HTTP readiness probes return status 200. Because this task is read-only, the comparison measures adapter overhead rather than isolation during state-changing interaction.

Transactional publication. Over 100 stress-test trials, we compare the full freeze/drain and atomic-publication protocol with direct publication. The complete protocol yields no mixed-generation events and no publication while requests remain in flight; direct publication yields both events in every trial. The full protocol also rejects all 100 requests issued while the route is frozen. Frozen-request rejection does not apply to direct publication and is reported as N/A.

A.2 Additional Execution Configurations

Table 4 complements the component ablations with absolute launch, storage, and memory measurements for WebServ Incus, fresh Docker, and DB/reflink configurations. Launch latency is averaged over eight subsequent runs. DB/reflink reports marginal memory growth, while the other entries report working-set memory; the two quantities are not directly comparable.

Table 4: **Absolute costs of additional execution configurations.** Launch latency is the mean of eight subsequent runs. Storage is per environment; † marks marginal memory growth, while the other memory entries are working-set measurements.

Configuration	Launch (s)	Storage	Memory (GiB)
WebServ Incus	1.781	28.01 MiB	1.74
CUA-Sandbox: fresh Docker	12.323	6.621 GiB	1.795
CUA-Sandbox: DB/reflink	1.730	12.50 MiB	0.201†

B Detailed Experimental Results

B.1 Task Success by Site and Category

Table 5 reports the paired site and task-category scores used to compute the benchmark means in Table 1. Each entry retains the same model and evaluation protocol across the two backends.

Table 5: **Detailed task success by site and category.** Each cell reports Docker / **CUA-Sandbox** success rates (%) for the same model and evaluation protocol. Benchmark means appear in Table 1; bold distinguishes the CUA-Sandbox result in each pair.

(a) WebArena-Lite

Model	Shopping	CMS	GitLab
GPT-5.6-Terra	20.86 / **21.98**	20.90 / **20.90**	23.90 / **24.40**
Qwen3.5-9B	8.02 / **8.79**	9.89 / **9.89**	11.11 / **12.78**
Muse-Glimmer-30B	24.60 / **27.22**	17.03 / **18.13**	27.78 / **31.55**
Gemma-4-31B-IT	25.13 / **27.81**	20.88 / **22.32**	28.89 / **28.89**

(b) VisualWebArena

Model	Classifieds	Reddit	Shopping
GPT-5.6-Terra	27.35 / **27.78**	25.71 / **25.71**	31.97 / **33.69**
Qwen3.5-9B	13.18 / **15.38**	5.71 / **5.71**	12.66 / **19.53**
Muse-Glimmer-30B	23.93 / **24.78**	19.52 / **20.00**	32.62 / **32.62**
Gemma-4-31B-IT	19.66 / **21.37**	15.24 / **16.32**	22.32 / **23.39**

(c) OSWorld

Model	OS	Office	Daily	Professional	Workflow
GPT-5.6-Terra	75.00 / **83.33**	72.36 / **74.30**	56.16 / **57.42**	79.59 / **81.64**	53.34 / **56.43**
Qwen3.5-9B	25.00 / **29.17**	20.43 / **23.11**	37.16 / **37.56**	30.61 / **36.73**	7.65 / **11.72**
Muse-Glimmer-30B	45.83 / **50.00**	12.58 / **16.15**	28.21 / **34.81**	24.49 / **53.06**	10.87 / **17.94**
Gemma-4-31B-IT	70.83 / **70.83**	7.76 / **8.55**	45.35 / **45.35**	22.45 / **34.69**	16.68 / **18.28**

B.2 Absolute Rollout Measurements

Table 6 provides the measurements underlying Figure 3 at eight-way concurrency. Resource costs are per environment; throughput and batch time describe the complete concurrent workload.

Table 6: **Exact rollout measurements for Figure 3.** All settings use eight concurrent environments. Memory and incremental storage are per environment; CUA denotes CUA-Sandbox. Preparation and batch wall-clock time are in seconds.

Metric	WebArena		VisualWebArena		OSWorld	
	Docker	**CUA**	Docker	**CUA**	Docker	**CUA**
Prepare (s)	68.85	25.68	6.964	0.633	20.75	5.84
Batch (s)	69.20	31.06	9.104	2.138	27.48	8.94
Throughput (env/s)	0.116	0.258	0.220	0.936	0.291	0.895
Memory (MiB)	2102.8	463.9	328.9	188.4	2653.8	837.3
Storage (MiB)	6757.1	13.1	4879.05	12.52	62.7	29.1

C Experimental Setup and Evaluation Protocols

C.1 Model Settings

During inference, Qwen3.5-9B used `enable_thinking=true`, temperature 1.0, top-$p = 0.9$, and 512 maximum output tokens for OSWorld, and temperature 1.0, top-$p = 0.9$, and 384 maximum output tokens for web tasks. Gemma-4-31B-IT used `enable_thinking=true`, temperature 1.0, top-$p = 0.95$, and 4096 maximum output tokens for OSWorld or 1000 for web tasks. GPT-5.6-Terra used temperature 1.0, top-$p = 0.9$, 1000 maximum output tokens, and `reasoning_effort=low`. Muse-Glimmer-30B used temperature 1.0, top-$p = 0.9$, and an output limit of 1000 tokens.

C.2 More Details

To evaluate the framework itself, we first focus on the supervised fine-tuning (SFT) stage. We use Qwen3.8-27B as the teacher model to generate 5k successful trajectories from the same training-task pool under both the conventional sandbox and CUA-Sandbox. We retain trajectories that successfully pass the corresponding task evaluator and use them to independently fine-tune Qwen3.5-4B for 2 epochs, obtaining one SFT checkpoint for each execution environment. Next, we perform on-policy RL training starting from the corresponding SFT checkpoints using 1,600 training instances for 1 epoch under both environment backends. We set the RL batch size to 16 and the rollout count to 8, resulting in 128 parallel logical environments per update. Both runs use the same task sampler, decoding configuration, maximum interaction horizon, reward function, and optimization hyperparameters, with the execution environment being the only difference.

C.3 Benchmarks and Evaluation Protocols

WebArena. WebArena Zhou et al. (2024) is a realistic and reproducible benchmark for evaluating autonomous web agents on long-horizon tasks. It provides fully functional, self-hosted web applications spanning e-commerce, social discussion, collaborative software development, and content management, together with auxiliary tools and knowledge resources. The benchmark contains 812 natural-language tasks and evaluates agents based on the functional correctness of the resulting website state rather than exact action-sequence matching. In our experiments, we focus on the Shopping, CMS, and GitLab domains, which involve state-changing interactions and therefore require isolated and resettable execution environments.

VisualWebArena. VisualWebArena Koh et al. (2024) extends web-agent evaluation to visually grounded tasks that require joint reasoning over images and textual webpage content. It contains 910 tasks across three self-hosted environments: Classifieds, Shopping, and Reddit. The Shopping and Reddit environments are inherited from WebArena, while Classifieds introduces an additional visually rich domain. Completing these tasks requires agents to interpret natural-language instructions, perceive visual information on webpages, and execute multi-step browser actions. We evaluate CUA-Sandbox on all three domains using the standard multimodal interaction protocol.

OSWorld. OSWorld Xie et al. (2024) is a benchmark for evaluating multimodal agents on open-ended computer-use tasks in real desktop environments. It provides a scalable execution environment with task initialization and execution-based evaluation, and includes 369 tasks involving real web and desktop applications, operating-system file operations, and cross-application workflows. Each task specifies an initial environment state together with a task-specific evaluator that directly checks the resulting system state. In our experiments, we report results across the OS, Office, Daily, Professional, and Workflow categories.

Agents and observations. WebArena agents receive the task instruction and the accessibility tree. VisualWebArena agents receive the current page screenshot, the task-image captions, and the screenshot with Set-of-Marks (SoM) element identifiers (i.e., the Image + Captions + SoM configuration), and emit SoM-based actions. OSWorld agents receive the task instruction and the current desktop screenshot and interact through the benchmark's mouse, keyboard, and terminal action interfaces.

Execution and compute. All experiments are run on the host described in Section 4.1: 8 NVIDIA RTX PRO 5000 Blackwell GPUs (72 GB each), 192 physical CPU cores, and 1.5 TiB of system memory. We use the benchmark defaults for the maximum interaction horizon (30 steps for WebArena and VisualWebArena) and keep sampling parameters, context limits, image limits, and timeout values fixed within each benchmark. Evaluation workers may run in parallel, but the concurrency limit is held constant across the two environment implementations so that the comparison does not conflate isolation with scheduling capacity.

Metrics and failure handling. We first compute task success within each benchmark site or task category. For group g, containing the evaluated task set $\mathcal{T}_g$, the group-level success rate is

$$\mathrm{SR}_g = \frac{\#\{\text{successful evaluated tasks in } g\}}{\#\{\text{evaluated tasks in } g\}}.$$

The benchmark-level score reported in the main tables is the unweighted mean over the corresponding sites or task categories:

$$\mathrm{BenchmarkScore} = \frac{1}{|\mathcal{G}|} \sum_{g \in \mathcal{G}} \mathrm{SR}_g,$$

where $\mathcal{G}$ is the set of reported sites or categories. Thus, each site or category contributes equally to the benchmark-level score, irrespective of the number of tasks it contains. This is a macro average over groups rather than a task-weighted micro average.

For WebArena, $\mathcal{G}$ consists of Shopping, CMS, and GitLab; for VisualWebArena, it consists of Classifieds, Reddit, and Shopping; and for OSWorld, it consists of OS, Office, Daily, Professional, and Workflow. The corresponding group-level scores are reported separately in Appendix B.

For WebArena and VisualWebArena, the official task-specific program/HTML, string-match, and image-based evaluators are used as applicable; for OSWorld, we use the official task evaluator. The same judge model and endpoint are used for both sides of a pair.

A transport retry may re-execute a crashed worker, but it does not create an additional task or otherwise change the evaluation set. Unresolved environment, browser, or evaluator errors for an evaluated task are recorded as failures. Tasks excluded from evaluation are reported separately and are not assigned a synthetic score.

In addition to task success, we record environment preparation time, per-step environment communication time, timeout counts, and the physical storage occupied by each environment's mutable state. Unless otherwise specified, each reported measurement is repeated five times under the same experimental configuration, and we report the arithmetic mean across the five runs.

D Implementation Details

Resource contracts. For a task family τ, let $\mathcal{L}_\tau$ denote its logical resource keys. The contract

$$\Gamma_\tau = \{(k, c_k, a_k, p_k) \mid k \in \mathcal{L}_\tau\}$$

associates each key with a state class c_k, backend adapter a_k, and lifecycle policy p_k. Keys identify databases, file trees, profiles, queues, displays, message-bus sessions, or other task-relevant resources. Agents operate only on these logical keys and do not observe the associated physical backend names.

The resource contract defines the sharing boundary conservatively. A task-relevant mutable resource is eligible for runtime sharing only when its accesses can be completely resolved through an environment-scoped binding. Resources for which such a binding cannot be established are assigned a private backend or a coarser private execution context. They are never forced into the shared runtime merely to increase reuse. Consequently, extending CUA-Sandbox to a setting with additional mutable resources may reduce the amount of sharing, but does not require weakening the environment-isolation contract.

Capsule metadata. A capsule generation is represented as

$$K_i^g = \left\langle I_i^g, \Delta_i^{\mathrm{auth}}, N_i^{\mathrm{der}}, P_i^{\mathrm{eph}}, M_i^g \right\rangle.$$

Here, I_i^g records the environment, branch, and generation identity; Δ_i^{auth} holds authoritative state deltas; N_i^{der} records private namespaces and reconstruction rules; P_i^{eph} describes ephemeral resources to recreate; and M_i^g records component versions and dependencies. The composite manifest coordinates their publication as a single logical generation.

This representation separates persistent task state from derived and ephemeral execution state without assuming that all mutable state is shareable. State that can affect future observations, evaluator-visible results, or externally visible side effects must either be represented in the capsule, explicitly scoped by the active execution context, or retained within a private component.

Binding tables. Each published generation has one binding for every contracted key:

$$\mathrm{dom}(\Phi_i^g) = \mathcal{L}_\tau.$$

An entry

$$\Phi_i^g(k) = \left\langle b_k, \delta_{i,k}^g, n_{i,k}^g, e_{i,k}^g \right\rangle$$

selects immutable base data, a private data branch, a private service namespace, and a private interface endpoint; fields that do not apply are null.

The trusted execution context combines the environment identity with this table, and a generation-scoped lease pins the selected version for the duration of execution. A context becomes executable only after all task-relevant contracted resources have valid bindings. Thus, for writable state W_i^g reachable by environment i at generation g, the required isolation invariant is

$$W_i^g \cap W_j^{g'} =, \qquad i \neq j.$$

Process sharing alone is therefore insufficient to establish isolation. Any task-relevant mutable process state must either be keyed by the active environment context or remain inside a private execution component. If this condition cannot be established for a particular application component, the component is conservatively excluded from the shared boundary.

Web adapters select databases, files, queues, and caches; visual-web adapters additionally bind browser profiles and sessions; desktop adapters bind filesystem, application-profile, configuration, display, message-bus, and window state. These benchmark-specific adapters differ only in which resources are included in the contract. The isolation rule itself is unchanged across settings: all task-relevant writable state remains private to one logical environment generation, while only resources satisfying the sharing contract may be reused.

Asynchronous and background execution. Environment identity and generation apply to the complete causal execution context rather than only to the foreground agent action. Asynchronous requests, callbacks, and tracked background work spawned by an admitted operation inherit the same environment context and generation lease and therefore resolve state through the same binding table.

During reset, clone, fork, or slot rebinding, new work for the affected environment is first quiesced and existing in-flight work is drained before a successor generation is published. Delayed work carrying a stale generation cannot be rebound to the successor state. Background activity that cannot be reliably attributed, scoped, or quiesced is instead placed outside the shared execution boundary and uses a private or coarser-grained backend.

Thus, unsupported asynchronous behavior or application-specific state can reduce the achievable degree of runtime reuse, but does not require relaxing the isolation invariant.

E Scaling Analysis

The separation of logical environments from execution slots gives a resource model for runtime reuse. The following accounting compares the two designs under the same logical state and separates shared initialization from private execution and capsule costs.

Table 7: Concurrency scaling measurements at 4 and 16 active environments. Lower is better for preparation time, batch wall time, RAM, and storage; higher is better for throughput.

Benchmark	Backend	Concurrency	Env. prep. avg./env (s)	Batch wall (s)	Throughput (env/s)	RAM / env (MiB)	Storage / env (MiB)
WebArena	CUA-Sandbox	4	20.360	21.340	0.188	492.9	13.00
WebArena	Docker	4	44.600	44.800	0.089	2112.3	6757.10
WebArena	CUA-Sandbox	16	44.238	49.372	0.324	436.1	13.05
WebArena	Docker	16	120.771	121.483	0.132	2082.6	6778.84
VisualWebArena	CUA-Sandbox	4	1.434	5.735	0.697	15.9	0.44
VisualWebArena	Docker	4	6.208	24.831	0.161	514.8	4880.04
VisualWebArena	CUA-Sandbox	16	1.366	21.851	0.732	15.6	0.99
VisualWebArena	Docker	16	4.803	76.848	0.208	349.1	4903.34
OSWorld	CUA-Sandbox	4	5.490	8.310	0.481	851.0	28.60
OSWorld	Docker	4	17.870	23.950	0.167	2737.7	68.90
OSWorld	CUA-Sandbox	16	7.268	10.545	1.517	836.5	29.52
OSWorld	Docker	16	18.742	54.293	0.295	2647.0	233.47

Memory. Let m_P denote the proportional-set memory of initialized processes and the common runtime, m_S the private memory of an active slot, $m(K_i)$ the resident memory of capsule i, and m_{ctrl} the control-plane overhead. Excluding fixed host-OS memory common to both designs,

$$\begin{aligned} \mathcal{M}_{\text{rep}}(N) &= N(m_P + m_S) + \sum_{i=1}^{N} m(K_i), \\ \mathcal{M}_{\text{CUA}}(N, Q) &= m_P + Q m_S + \sum_{i=1}^{N} m(K_i) + m_{\text{ctrl}}. \end{aligned} \tag{5}$$

Under this decomposition, the avoided replication is

$$\mathcal{M}_{\text{rep}} - \mathcal{M}_{\text{CUA}} = (N-1)m_P + (N-Q)m_S - m_{\text{ctrl}}. \tag{6}$$

The first term captures shared runtime initialization; the second captures the additional benefit when some trajectories remain inactive. Even when $Q = N$, runtime sharing can reduce memory, while each environment still incurs private state and execution costs. Actual savings therefore depend on the workload and its sharing boundary.

Storage and lifecycle cost. Let d_{img} be the immutable image already shared by either design, d_{init} the initialized writable state materialized for one complete runtime, and $d(\Delta_i)$ the subsequent private data written by environment i. With capsule metadata cost d_{meta},

$$\begin{aligned} \mathcal{D}_{\text{rep}}(N) &= d_{\text{img}} + N d_{\text{init}} + \sum_{i=1}^{N} d(\Delta_i), \\ \mathcal{D}_{\text{CUA}}(N) &= d_{\text{img}} + d_{\text{init}} + \sum_{i=1}^{N} d(\Delta_i) + d_{\text{meta}}. \end{aligned} \tag{7}$$

This comparison attributes savings to initialized writable state, not to duplicating immutable images. Likewise, lifecycle operations avoid rebuilding the complete runtime but still pay for quiescence, state preparation, rebinding, and readiness checks. Their latency depends on the selected backends and the amount of private state, so the operation-specific measurements in Table 3 complement this accounting. The model describes the expected sources of savings; it does not by itself establish throughput or environment density as concurrency increases.

Concurrency scaling. Table 7 evaluates whether the resource and throughput advantages of CUA-Sandbox persist as execution concurrency increases from 4 to 16. Across all three workloads, CUA-Sandbox maintains lower batch latency, higher throughput, and substantially lower per-environment memory and storage than Docker at both concurrency levels.

The advantage remains substantial at concurrency 16: CUA-Sandbox achieves 2.45×, 3.52×, and 5.14× higher throughput on WebArena, VisualWebArena, and OSWorld, respectively. Per-environment memory is reduced by 4.78×, 22.38×, and 3.16× on the same workloads. These results indicate that the efficiency gains are not limited to the eight-way setting used in the main rollout experiment, but persist across both lower and higher tested concurrency levels.

Table 8: Ablation of copy-on-write (CoW) and runtime sharing for eight logical environments. Runtime RAM measures the memory occupied by the application runtime, while Agent/Browser PSS reports the proportional-set size of the agent and browser processes. Writable storage includes environment-specific writable state. Lower is better.

Configuration	Active runtime containers	Runtime RAM (MiB)	Agent/Browser PSS (MiB)	Writable storage (GiB)
Official Docker	8 Web	15,560	6,998	52.99
CoW + independent runtime	8 Web + 8 MySQL	13,882	7,045	24.65
CoW + shared runtime	1 Web + 1 MySQL	**3,261**	7,291	**23.19**

F RUNTIME-SHARING ABLATION.

Table 8 separates the effects of copy-on-write state management from runtime sharing. Introducing CoW while retaining independent runtimes reduces writable storage from 52.99 to 24.65 GiB, but only modestly reduces runtime memory from 15,560 to 13,882 MiB. Sharing the initialized runtime further reduces runtime RAM to 3,261 MiB, a 4.26× reduction relative to the CoW configuration with independent runtimes and a 4.77× reduction relative to the official Docker baseline.

In contrast, Agent/Browser PSS remains approximately unchanged across the three configurations (6,998–7,291 MiB). This indicates that the memory reduction does not arise from shrinking the agent or browser workload, but from eliminating replicated application-runtime processes. Writable storage changes only slightly when moving from independent to shared runtimes (24.65 to 23.19 GiB), consistent with CoW accounting for most of the state-storage reduction while runtime sharing primarily removes replicated process memory.

G CASE STUDIES: STATE-SCOPED EXECUTION

We illustrate the method in Section 3 through two task workflows: updating a shopping cart and editing a desktop document. Each follows capsule construction, state-scoped interaction, and reset. The workflows are design illustrations rather than recorded evaluation trajectories.

Case 1: An independent cart over a shared web runtime

Scenario. Two environments start from the same shopping state. Environment A adds a product to its cart; B should retain its own cart.
Shared / private. The initialized web server is shared. Each environment retains its own database branch, session state, cache namespace, and browser profile.

State sketch. A: initial cart → updated cart → initial cart.
B: initial cart throughout, under its own capsule bindings.

Phase I: Execute against the selected environment state.

Step 1: Bind the browser action

Action. A issues an "add to cart" interaction through the original interface.
Binding. Resolve A's current generation and lease its database, session, and cache bindings before servicing the request.

Step 2: Keep the mutation and observation together

State access. Resolve the cart update through A's database branch and session. Any resulting cache access or background work must retain A's context.
Expected behavior. A sees the added product; B reads its separate state. The evaluator uses A's generation to inspect A's cart.

Phase II: Reset one environment without resetting the other.

Step 3: Stage a clean successor for A

Transition. Freeze A's route and drain its in-flight work. Prepare a clean database branch, session state, and derived-state namespace as one successor capsule.
Boundary. Keep the previous generation published until its successor is ready; B's bindings are not part of this transition.

Step 4: Publish bindings and resume interaction

Activation. Publish A's new generation atomically, reject stale-generation requests, and resume A only after readiness checks.
Expected behavior. A returns to the task's initial cart state; B retains its own state. The shared web server remains initialized.

Analysis. The sharing boundary is the request context, not the cart itself. Connection reuse, cache lookup, and asynchronous work must respect that context; an adapter that cannot do so requires a private backend.

Figure 5: **Web case study: request-scoped state separation.** The example follows a cart mutation and an independent reset through shared execution, private state access, and generation publication. Purple headers mark binding and staging; gray headers mark state effects and activation. Expected outcomes illustrate the method's requirements and are not measured execution results.

Case 2: Document editing with a private state capsule

Scenario. Two desktop environments open the same initial document. A edits and saves it; B should continue to see its own copy.
Shared / private. Compatible runtime components are reusable. Documents, profiles, configuration, displays, and message-bus endpoints remain private to each session.

State sketch. A: initial document → saved edit → initial document.
B: initial document throughout; its session remains separate.

Phase I: Bind a desktop interaction to one session.

Step 1: Activate A's complete session context

Binding. Attach A's file and profile state, select its display and message-bus endpoints, and establish readiness before accepting input.
Boundary. Any process whose mutable state cannot be isolated or safely rebound must remain within a private backend.

Step 2: Edit, save, and observe through A's bindings

Action. A edits the document and saves through the desktop interface.
Expected behavior. The save reaches A's file overlay. Screenshots come from A's display, and evaluation reads A's document; B's document is unchanged.

Phase II: Reset state before reusing an execution slot.

Step 3: Quiesce the session and stage its successor

Transition. Freeze A's route and drain pending input, saves, and background work. Stage clean document and profile state and recreate ephemeral endpoints.
Boundary. An open document buffer or file handle must not survive into the successor with stale state; use a private backend where necessary.

Step 4: Publish and verify the rebound context

Activation. Publish only the complete successor bindings and pass readiness checks before routing input or collecting screenshots.
Expected behavior. A observes the initial document through the new session. B retains its state, and delayed requests cannot access A's successor generation.

Analysis. The capsule couples document state with the session that exposes it. File isolation alone is insufficient: observations and evaluation must follow the same environment bindings. Components that cannot be safely rebound remain in the coarse private backend described in Section 3.2.

Figure 6: **Desktop case study: session-scoped state separation.** The workflow follows a document edit through private file and profile state, environment-specific display endpoints, and a coordinated reset. Observations and evaluation follow the same capsule. The private-session boundary also covers components whose in-memory state or open handles cannot be safely rebound.