ECHORL: POLICY-EVOLVING ENVIRONMENT-STATE CACHES FOR AGENTIC REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic reinforcement learning (RL) trains policies that act inside real software environments, such as checking out repositories, installing dependencies, running builds, and invoking tools. Group-based learning algorithms, such as GRPO, sample multiple rollouts per task, causing the same deterministic environment operations to be repeated across rollouts and thereby incurring a considerable fraction of the end-to-end rollout latency. Caching environment operations is a natural solution, but policy evolution across training steps raises two challenges: (i) cached execution results may become invalid under updated context, producing incorrect trajectories and biased policy gradients; and (ii) the cache must continually adapt to retain the environment states most likely to be revisited by future policies. To this end, we present EchoRL, a runtime layer that reuses environment work. We make two contributions. First, we reduce environment latency by casting reuse as caching a deterministic transition function keyed by logical state and tool version, and show that under an explicit determinism assumption cached and uncached rollouts follow the same distribution, introducing no additional bias into the underlying policy-gradient estimator. Second, we introduce advantage-guided cache retention to anticipate environment-state reuse under evolving policies, improving cache hit rates in subsequent training iterations. Across all evaluated workloads, EchoRL significantly reduces per-rollout environment latency by $2.1\times$–$8.3\times$ compared with No Cache while preserving trajectory correctness.

1 INTRODUCTION

Reinforcement learning (RL) has become a central approach for transforming large language models (LLMs) into agents capable of reasoning, acting, and interacting with external systems (Christiano et al., 2017; Ouyang et al., 2022; Feng et al., 2026). Unlike conventional language-model post-training, agentic RL places a live execution environment in the training loop (Tian et al., 2025; Dong et al., 2026; Gao et al., 2026). To complete a task, an agent may inspect and modify files, check out repositories, install dependencies, compile software, query databases, or invoke domain-specific tools. Each rollout therefore alternates between model inference and environment execution: the policy generates an action, the environment executes it and returns an observation, and the policy continues generation conditioned on the resulting interaction history.

Collecting such interactive trajectories is substantially more expensive than generating text alone. In particular, group-based RL algorithms, *e.g.*, GRPO (Shao et al., 2024), DAPO (Yu et al., 2025) and GSPO (Zheng et al., 2025), sample multiple stochastic rollouts at a nonzero temperature for each task to construct a relative-advantage baseline. Although these rollouts may eventually explore different solutions, they frequently share identical or overlapping action prefixes. Consequently, the same deterministic environment operations, such as cloning a repository, installing a package, building a project, or running an inspection command, are repeatedly executed across rollouts. Since these operations can take seconds or even minutes, redundant environment execution accounts for a considerable fraction of end-to-end rollout latency and leaves expensive model-serving resources waiting for tool results (Zhang et al., 2026; Yu et al., 2026).

Caching previously executed environment work is a natural way to eliminate this redundancy, but applying caching to agentic RL raises two fundamental challenges. The first is *correctness*. Tool outputs are generally stateful: the result of an action depends not only on its explicit arguments, but

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

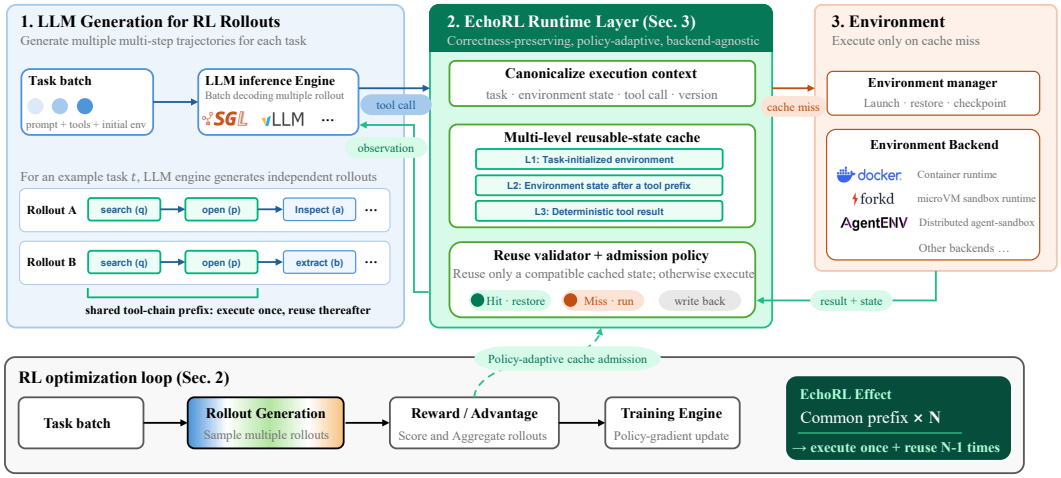

Figure 1: EchoRL within the agentic RL training framework. Positioned between the LLM inference engine and environment backends, EchoRL reuses compatible environment states and tool results across rollouts while adapting cache retention to policy updates. It reduces redundant environment execution without altering rollout sampling or the underlying RL optimization algorithm.

also on the environment state produced by all preceding actions. Moreover, changes to tool implementations, dependencies, or execution configurations may invalidate previously recorded results. Reusing an output under a mismatched state can silently alter the observations received by the policy, causing cached and uncached rollouts to follow different distributions and potentially biasing the resulting policy-gradient estimates.

The second challenge is *adaptivity*. The rollout distribution is non-stationary because the policy changes after every training update. A state that is frequently visited by the current policy may become irrelevant after subsequent updates, whereas a less frequent state associated with a promising trajectory may become increasingly popular. Under a finite cache budget, an effective admission and eviction policy must therefore anticipate which environment states are likely to be revisited by future policies, rather than merely retaining those that were frequently accessed in the past. Conventional recency- or frequency-based policies (Kumar et al., 2026) do not exploit the structured manner in which an RL policy evolves during training.

To address these challenges, we present EchoRL, a runtime layer for correctness-preserving and policy-aware reuse of environment computation in agentic RL, shown in Figure 1. EchoRL models tool execution as a deterministic transition from a logical environment state and an action to a successor state and observation. Cached transitions are identified by their logical state, tool invocation, and tool version, ensuring that a cached result is returned only when the corresponding transition would have been executed under an equivalent environment context. Under an explicit determinism assumption, this form of reuse is observationally equivalent to executing the transition again. Cached and uncached rollouts consequently follow the same distribution, leaving the policy-gradient estimator unchanged.

While this formulation establishes when environment reuse is valid, a finite cache budget requires deciding which reusable states to retain as the policy evolves. To guide this decision, EchoRL uses trajectory advantages to anticipate future state visitation. The rewards and estimated advantages of sampled trajectories provide a forward-looking signal of how their visitation probabilities are likely to change. Under policy-gradient optimization, actions with positive advantages are reinforced and are therefore more likely, all else being equal, to be sampled by subsequent policies; actions with negative advantages are suppressed and become less likely to recur. EchoRL propagates the signal to the environment states visited along each trajectory. States on positive-advantage trajectories receive higher admission and retention priority, while states associated with negative-advantage trajectories become stronger eviction candidates. In this way, the cache evolves together with the policy and prioritizes environment states that future rollouts are more likely to revisit.

Importantly, EchoRL reuses only environment-side computation. Rewards and advantages are consumed only as cache-management signals and are not returned to future rollouts or substituted into

the RL objective. This separation allows EchoRL to accelerate trajectory collection without modifying the learning algorithm, the policy update, or the statistical contents of collected trajectories. The runtime is also independent of the underlying sandbox mechanism and can operate over container- or microVM-based backends such as Docker (Merkel et al., 2014), forkd[1], and AgentENV[2]. Our anonymous code is available here[3].

In summary, this work makes the following contributions:

- We formulate environment reuse in agentic RL as caching a deterministic transition function keyed by logical environment state, tool invocation, and tool version.
- We design EchoRL, a runtime layer that combines three-level caching with cost-based admission and advantage-guided retention to anticipate future reuse under evolving policies.
- We evaluate EchoRL on Terminal-Bench, EgoSchema, and SkyRL-SQL, demonstrating up to $8.3\times$ speedup in per-rollout environment latency while achieving nearly identical performance with and without caching.

2 Problem Formulation

Let $x \sim \mathcal{D}$ denote an agentic task sampled from a training distribution $\mathcal{D}$. Each task is associated with an initial environment state $s_0(x)$. At interaction step t, the policy π_θ generates an action

$$a_t \sim \pi_\theta(\cdot \mid h_t), \tag{1}$$

where h_t contains the task specification and the interaction history up to step t, including previously generated tokens, tool calls, and environment observations. In an LLM agent, a_t may correspond to a structured tool invocation embedded in a longer sequence of tokens. The action is executed by an environment transition function

$$(s_{t+1}, o_{t+1}) = \mathcal{T}_{v_t}(s_t, a_t), \tag{2}$$

where s_t is the complete environment state, o_{t+1} is the observation returned to the policy, and v_t identifies the version of the invoked tool and its relevant execution dependencies. The state s_t may include filesystem contents, installed packages, running processes, database contents, and other information that can affect subsequent tool execution.

A rollout of length H is written as $\tau = (x, s_0, a_0, o_1, s_1, \ldots, a_{H-1}, o_H, s_H)$, and receives a reward $R(\tau)$. Agentic RL optimizes

$$J(\theta) = \mathbb{E}_{x\sim\mathcal{D},\, \tau\sim\pi_\theta}\left[R(\tau)\right]. \tag{3}$$

Group-based algorithms, *e.g.,* GRPO (Shao et al., 2024), DAPO (Yu et al., 2025) and GSPO (Zheng et al., 2025), sample N rollouts $\{\tau_i\}_{i=1}^{N}$ for the same task and estimate their advantages relative to other rollouts in the group. Abstracting away algorithm-specific clipping and importance-weighting terms, the resulting policy-gradient estimator has the form

$$\widehat{g}(\theta) = \frac{1}{N}\sum\nolimits_{i=1}^{N}\sum\nolimits_{t=0}^{H_i-1} \widehat{A}_{i,t}\nabla_\theta \log \pi_\theta(a_{i,t} \mid h_{i,t}), \tag{4}$$

where $\widehat{A}_{i,t}$ denotes the estimated advantage assigned to action $a_{i,t}$. As rollouts for the same task often share action prefixes, the same environment transition in Eqn. (2) may be executed repeatedly.

Let $p_j = (a_0, a_1, \ldots, a_{j-1})$ be a tool prefix that transforms s_0 into s_j. Without caching, every rollout that generates p_j must execute the composed transition

$$s_j = \mathcal{T}^{(j)}(s_0, p_j) = \mathcal{T}_{v_{j-1}}\left(\cdots \mathcal{T}_{v_1}\left(\mathcal{T}_{v_0}(s_0, a_0), a_1\right)\cdots, a_{j-1}\right). \tag{5}$$

If the same prefix has already been executed, the environment work in Eqn. (5) can instead be replaced by restoring its cached successor state:

$$\widehat{s}_j = \mathrm{Restore}\big(m(s_j)\big), \qquad \widehat{s}_j \equiv s_j, \tag{6}$$

[1] https://github.com/deeplethe/forkd

[2] https://github.com/kvcache-ai/AgentENV

[3] https://anonymous.4open.science/r/EchoRL

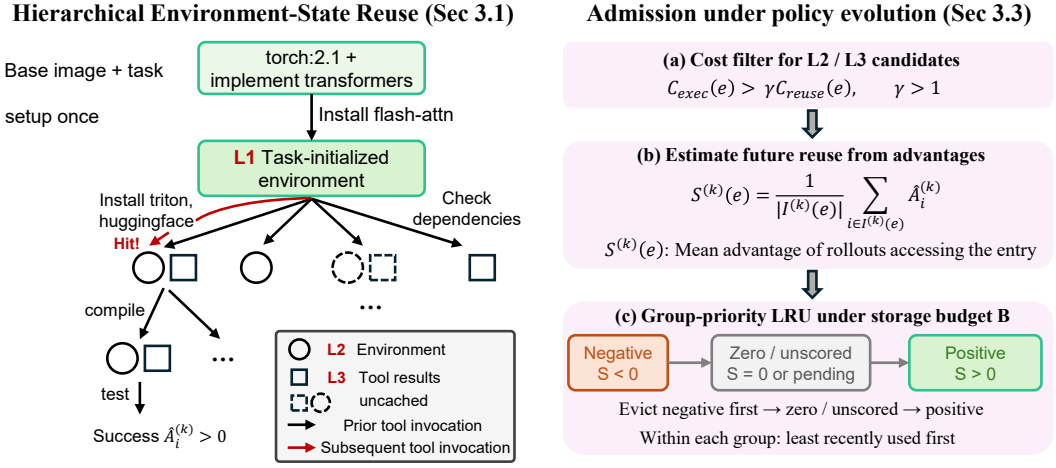

Figure 2: Overview of EchoRL. (Left) Three-level caching reuses task-initialized environments (L1), intermediate environment states after tool prefixes (L2), and deterministic tool results (L3). (Right) Cost-based admission and advantage-guided LRU adapt cache retention to policy evolution.

where $\equiv$ denotes logical equivalence with respect to future tool execution. Importantly, the new rollout still generates its own tokens and actions and receives the corresponding observations, while EchoRL skips only the repeated environment execution.

Let $C_{\text{exec}}(s_j, a_j)$ denote the cost of directly executing a tool action a_j under state s_j, and let $C_{\text{reuse}}(s_{j+1})$ denote the cost of locating and restoring its cached environment state, the benefit of caching the environment state is therefore:

$$C_{\text{reuse}}(s_{j+1}) = C_{\text{lookup}}(s_{j+1}) + C_{\text{restore}}(s_{j+1}),\ \Delta C(s_{j+1}) = C_{\text{exec}}(s_j, a_j) - C_{\text{reuse}}(s_{j+1}). \quad (7)$$

This benefit can be substantial for costly environment operations such as compiling FlashAttention.

Given a storage budget B, our objective is to retain the environment states that maximize the expected execution time saved by future rollouts, while preserving the original rollout distribution:

$$\max_{\mathcal{C}} \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{s_j \in \tau} \mathbf{1}[s_j \in \mathcal{C}] \Delta C(s_j) \right] \quad \text{s.t.} \quad \text{Size}(\mathcal{C}) \le B, \quad P_{\mathcal{C}}(\tau \mid x, \pi_\theta) = P_{\text{exec}}(\tau \mid x, \pi_\theta). \quad (8)$$

The objective favors both frequently revisited and costly environment states, while the distributional constraint ensures no additional bias into the underlying policy-gradient estimator.

3 EchoRL: Policy-Evolving Environment-State Cache

EchoRL organizes reusable environment work into three levels, which capture reuse at progressively finer granularities: task initialization, intermediate environment states, and deterministic tool results. The overview structure and admission strategy are shown in Figure 2.

3.1 Hierarchical Environment-State Reuse

L1: Task-initialized environments. The L1 cache stores the environment produced by task initialization. Let s_x^{base} denote the environment created directly from the base image associated with task x, and let $\mathcal{I}_{u_x}$ denote the task initialization procedure with configuration version u_x. The initialized state is

$$s_x^{(0)} = \mathcal{I}_{u_x}\left(s_x^{\text{base}}\right). \quad (9)$$

Initialization may include cloning a repository, checking out a fixed commit, downloading task data, installing dependencies, or compiling common libraries. L1 stores a snapshot handle

$$\mathcal{C}_{\text{L1}}[x, u_x] = m_1\left(s_x^{(0)}\right). \quad (10)$$

All rollouts of task x can begin from independent instances restored or forked from this snapshot:

$$s_{x,i}^{(0)} = \text{Fork}\left(m_1\left(s_x^{(0)}\right)\right), \qquad i = 1, \ldots, N. \quad (11)$$

Without L1, the initialization is repeated for every rollout. L1 generally has a high reuse rate because all rollouts for the same task share the same initial environment. However, L1 may be unnecessary when the provided Docker image already contains the complete task-specific state and its instantiation cost is negligible. In that case, $\mathcal{I}_{u_x}$ is effectively the identity operation and EchoRL starts directly from the provided image.

L2: Environment States after tool prefixes. L2 captures intermediate states produced as the agent executes tools. For a tool prefix $p_t = (a_0, \ldots, a_{t-1})$, let $s(p_t)$ denote the resulting environment state. The prefixes observed across rollouts form a tree $\mathcal{T}_x$: the root corresponds to the task-initialized state, and each edge corresponds to a tool invocation.

Because the physical environment state may be large, EchoRL represents it using a logical-state descriptor (detailed extensively in Appendix A.1)

$$\ell(s) \in \mathcal{S}_L, \tag{12}$$

which captures the state relevant to subsequent tool execution. For the transition $(s', o) = \mathcal{T}_v(s, a)$, L2 uses the key k_2 and stores the cache:

$$k_2(s, a, v) = \big(x, \ell(s), a, v\big), \quad \mathcal{C}_{\mathrm{L2}}[k_2(s, a, v)] = \big(\ell(s'), m_2(s')\big), \tag{13}$$

where $m_2(s')$ identifies a snapshot or copy-on-write branch from which s' can be materialized.

When another rollout reaches a logically equivalent state and issues the same action under the same tool version, EchoRL returns o and continues from

$$\widehat{s}' = \mathrm{Restore}\left(m_2(s')\right), \qquad \ell(\widehat{s}') = \ell(s'). \tag{14}$$

The tool action is still generated by the current rollout, but its environment execution is skipped. L2 is particularly beneficial for expensive, state-mutating operations such as dependency installation, source compilation, data preprocessing, and service initialization.

L3: Deterministic tool results. L3 caches the results of deterministic operations performed from an L2-identified environment state. Let r be a deterministic action satisfying

$$\mathcal{T}_v(s, r) = (s', o), s' = s \ \vee \ s' \neq s \quad \text{s.t.} \forall s'', o' : \mathcal{T}_v(s, r) = (s'', o') \Rightarrow s'' = s' \tag{15}$$

L3 uses the key k_3 and stores only the resulting observation:

$$k_3(s, r, v) = \big(x, \ell(s), r, v\big), \quad \mathcal{C}_{\mathrm{L3}}[k_3(s, r, v)] = o. \tag{16}$$

An L3 lookup is performed only after EchoRL has established the corresponding L2 logical state. This state-conditioned lookup prevents a result obtained from one environment from being reused in another environment with different contents. On a hit, EchoRL immediately returns

$$\widehat{\mathcal{T}}_v(s, r) = (s', \mathcal{C}_{\mathrm{L3}}[k_3(s, r, v)]) \tag{17}$$

without executing the tool.

L3 is useful when an operation is deterministic but computationally expensive. Examples include a SQL `SELECT` query over a huge database, decoding or inspecting a large video, and scanning a large repository. For L3 reuse to be valid, all relevant inputs, e.g., the database, video contents, query parameters, and decoder version, must be represented by $\ell(s)$, r, or v.

3.2 Correctness of Environment Reuse

EchoRL must ensure its three cache levels are functionally equivalent to direct execution.

Logical-state equivalence. We write

$$s \equiv \widetilde{s} \quad \Longleftrightarrow \quad \ell(s) = \ell(\widetilde{s}) \tag{18}$$

when two physical environments represent the same logical state for future cacheable operations.

Assumption 1 (**Deterministic transitions**). For a fixed logical state, action, and tool version, environment execution produces a unique observation and successor logical state. Specifically, if $\ell(s) = \ell(\widetilde{s})$ and $\mathcal{T}_v(s, a) = (s', o)$ and $\mathcal{T}_v(\widetilde{s}, a) = (\widetilde{s}', \widetilde{o})$ then

$$o = \widetilde{o} \quad \text{and} \quad \ell(s') = \ell(\widetilde{s}'). \tag{19}$$

The version descriptor v must include every execution dependency that can affect the transition. Operations involving uncontrolled randomness, external mutable services, current time, or unrecorded network responses bypass the cache unless these inputs are explicitly captured.

Definition 1 (**Transparent cache hit**). A cache hit is transparent if it returns the same observation and successor logical state as direct execution:

$$\left(o^{\text{cache}}, \ell(s^{\text{cache}})\right) = \left(o^{\text{exec}}, \ell(s^{\text{exec}})\right). \tag{20}$$

Theorem 1 (Correctness of the three-level cache). Suppose that: (i) an L1 entry is reused only when the task identity and initialization version match; (ii) an L2 entry is reused only when its logical state, action, and tool version match; and (iii) an L3 entry is reused only for an action verified to be deterministic and read-only under the matched L2 state.

Under Assumption 1, all cache hits in EchoRL are transparent. Therefore, for any fixed task x and agent policy π_θ,

$$P_{\text{EchoRL}}(\tau \mid x, \pi_\theta) = P_{\text{exec}}(\tau \mid x, \pi_\theta). \tag{21}$$

Proof sketch. Please refer to Appendix A.2.

3.3 Forward-Looking Cache Admission under Policy Evolution

EchoRL combines a cost-based admission filter with advantage-guided retention and least-recently-used (LRU) eviction.

Cost-based admission. For a candidate entry e, let $C_{\text{exec}}(e)$ denote the environment execution cost avoided by a cache hit and $C_{\text{reuse}}(e)$ the cost of serving that hit. For L2, C_{reuse} includes snapshot lookup and restoration; for L3, it includes retrieving and returning the cached tool output. An entry is eligible for admission only if

$$C_{\text{exec}}(e) > \gamma C_{\text{reuse}}(e), \qquad \gamma > 1, \tag{22}$$

where γ specifies the minimum required execution-to-reuse cost ratio. This filter excludes inexpensive operations for which caching provides little benefit. The specific implementation is described in Appendix B.5.

Advantage-guided retention (AGR). Let $\mathcal{I}_k(e)$ denote the rollouts that access entry e during training iteration k. For an L2 entry, these are the rollouts visiting the corresponding environment state; for an L3 entry, they are the rollouts requesting the same read-only tool result in that state. EchoRL assigns each entry a future-reuse score using the mean advantage of these rollouts:

$$S^{(k)}(e) = \frac{1}{|\mathcal{I}^{(k)}(e)|} \sum\nolimits_{i \in \mathcal{I}^{(k)}(e)} \widehat{A}_i^{(k)}. \tag{23}$$

A positive score indicates that the entry is associated with trajectories reinforced by the policy update, whereas a negative score indicates an association with suppressed trajectories. The score therefore serves as a directional estimate of future reuse rather than a calibrated visitation probability. Scores are updated after each training iteration; entries not accessed in that iteration retain their previous scores, and new entries remain unscored until their rollout advantages become available.

LRU eviction with advantage-based priority. Among cost-eligible entries, EchoRL prioritizes positive-score entries for admission and retention. When the storage budget is exhausted, it evicts entries in the following order: negative-score entries first, zero-score or unscored entries next, and positive-score entries last. Within each group, it evicts the least recently used entry. Equivalently, the eviction candidate is ordered lexicographically:

$$e_{\text{evict}} = \arg\min\left(g(e), t_{\text{last}}(e)\right). \tag{24}$$

$g(e) = 0$ for negative $S^{(k)}(e)$, $g(e) = 1$ for zero or unavailable $S^{(k)}(e)$, and $g(e) = 2$ for positive $S^{(k)}(e)$. The timestamp $t_{\text{last}}(e)$ records the most recent access.

Pending admissions follow the reverse group priority, and a lower-priority candidate does not displace a higher-priority resident entry. Entries currently being restored or required by retained snapshots are excluded from eviction. Eviction removes cached payloads while preserving the lightweight metadata needed to identify logical states.

This mechanism combines the simplicity of LRU with the direction of policy improvement: execution costs determine whether an entry is worth caching, advantages guide retention across policy updates, and recency resolves eviction decisions within each priority group.

4 EXPERIMENTS

In this section, we conduct empirical experiments to demonstrate the EchoRL efficacy. We seek to address the following questions. **RQ1:** How much does environment reuse accelerate rollout collection? **RQ2:** Does caching preserve execution correctness and training behavior? **RQ3:** How do the three cache levels and advantage-guided retention (AGR) contribute to performance? **RQ4:** Does AGR improve reuse across policy updates? **RQ5:** What is the overhead of using EchoRL?

4.1 EXPERIMENTAL SETUP

Workloads and training configuration. We consider three workloads with different environment-execution characteristics. Terminal-Bench (Merrill et al., 2026) involves multi-step terminal interaction, including filesystem operations, dependency installation, and command execution. SkyRL-SQL (Liu et al., 2025a) involves iterative SQL query generation and execution in database environments, where agents use execution feedback to refine subsequent queries. EgoSchema (Mangalam et al., 2023) involves video understanding, with environment work arising from the video-processing tools exposed to the agent. We integrate EchoRL with Slime (Zhu et al., 2025), an agentic RL framework, using SGLang (Zheng et al., 2024) for rollout generation and Docker (Merkel et al., 2014) for sandbox execution. We train Qwen-3.5 9B model (Qwen Team, 2026) with GRPO, sampling 8 rollouts per task at temperature 0.7. Experiments run on 4×NVIDIA H20 141GB HBM3e GPUs with tensor parallelism, an Intel(R) Xeon(R) Platinum 8468V CPU at 2.4GHz with 192 threads and 2TB of memory. We provide other experiment setups, like training details, dataset information and task prompt in Appendix C.

Baselines. We compare EchoRL against two baselines: *no-cache*, which executes all environment operations directly, and TVCache (Kumar et al., 2026), which reuses environments associated with matching tool-call prefixes. Unlike TVCache, EchoRL features a multi-level environment cache for more effective reuse of recurring training states, an advantage-guided mechanism that raises the policy's cache hit rate during training, and a backend-agnostic design that supports heterogeneous environment backends.

4.2 EXPERIMENT RESULTS

In this subsection, we aim to answer the five research questions posed above. Additionally, in Appendix D, we investigate how different environment backends affect environment latency and the effectiveness of EchoRL across different LLM scales.

Table 1: Tool-call latency and environment execution time per rollout (median).

Workload (Time unit)	**Tool Call (Time unit / call)**			**Env. Execution Time (Time unit / rollout)**		
	No Cache	TVCache	EchoRL	No Cache	TVCache	EchoRL
EgoSchema (s)	6.0	3.4 (1.8×)	**2.9 (2.1×)**	12.1	7.3 (1.7×)	**5.8 (2.1×)**
Terminal-Bench (s)	27.8	5.7 (4.9×)	**4.5 (6.2×)**	209.8	42.0 (5.0×)	**35.7 (5.9×)**
SkyRL-SQL (ms)	63.7	8.4 (7.6×)	**7.5 (8.5×)**	117.6	16.7 (7.0×)	**14.2 (8.3×)**

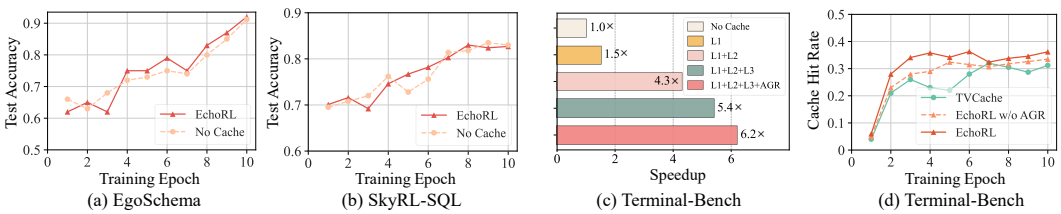

Figure 3: EchoRL evaluation. (a–b) Test accuracy remains comparable to uncached training. (c) Tool-call speedup over No Cache with successive cache levels and advantage-guided retention (AGR). (d) AGR improves cache hit rates across training epochs.

Environment execution performance (RQ1). Table 1 compares median values of tool-call latency and environment execution time per rollout across the three workloads. EchoRL achieves the lowest time on both metrics in every workload, with tool-call speedups of $2.1\times$–$8.5\times$ and per-rollout environment speedups of $2.1\times$–$8.3\times$ relative to No Cache. Compared with TVCache, EchoRL shows consistent improvement across all three workloads, which demonstrates the effectiveness of EchoRL in reducing environment execution latency.

Correctness and training behavior (RQ2). Figure 3 compares held-out test performance with and without EchoRL on EgoSchema and SkyRL-SQL. Across these two datasets, test performance remains comparable at matched training iterations. Minor differences may reflect stochastic rollout sampling and policy optimization, with no evident systematic degradation from caching. These results are consistent with Section 3.2: under the stated assumptions, EchoRL reuses deterministic environment computation while preserving the trajectory distribution for a fixed policy. Thus, the observed efficiency gains are achieved without an apparent loss in learning quality.

Ablation study (RQ3). Figure 3(c) isolates the contributions of EchoRL's cache hierarchy and advantage-guided retention (AGR) on Terminal-Bench. Relative to No Cache, L1 alone achieves a $1.5\times$ tool-call speedup by reusing task-initialized environments. Adding L2 increases the speedup to $4.3\times$, highlighting the benefit of restoring intermediate environment states instead of re-executing shared tool-call prefixes. Incorporating L3 further raises the speedup to $5.4\times$ by reusing deterministic tool results. Finally, enabling AGR yields a $6.2\times$ speedup, reducing tool-call latency by approximately 13% relative to the full hierarchy without AGR. These results demonstrate that the three cache levels provide complementary reuse opportunities, while policy-aware retention delivers additional gains beyond the cache hierarchy alone.

Cache retention under policy evolution (RQ4). Figure 3(d) compares cache hit rates across training epochs on Terminal-Bench. All methods exhibit a rapid initial increase as reusable entries accumulate, followed by fluctuations as training progresses. EchoRL generally maintains higher hit rates than both TVCache and its variant without AGR, although the gap varies across epochs. By the final epoch, EchoRL reaches a hit rate of approximately 36%, compared with 33% without AGR and 31% for TVCache. The comparison with EchoRL without AGR isolates the benefit of advantage-guided retention within the same cache hierarchy. The observed improvement supports using trajectory advantages to prioritize entries for future reuse, helping the cache remain effective as the rollout distribution changes with policy updates.

Overhead of EchoRL (RQ5). Figure 4 compares the latency distributions of EchoRL's cache queries, snapshot creation, and snapshot restoration with that of full tool execution on EgoSchema. Nearly half (49.4%) of full tool calls exceed the maximum observed latency of every cache-related operation, including cache queries, snapshot creation, and restoration. This gap indicates that EchoRL can reuse expensive environment computations at comparatively low overhead, providing substantial opportunities to reduce environment execution time on cache hits.

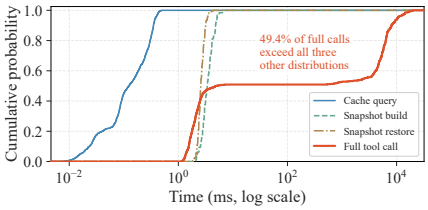

Figure 4: Time cost cumulative distributions of snapshot-creation/restoration, cache-query and full tool-call latencies.

5 Related Work

Reinforcement learning for LLM agents. Reinforcement learning has become an important paradigm for transforming LLMs from passive text generators into agents that reason, invoke tools, and interact with external environments. While early work primarily applied RL to improve reasoning or align model outputs (DeepSeek-AI, 2025), recent systems such as ReTool (Feng et al., 2026), Search-R1 (Jin et al., 2025), RAGEN (Wang et al., 2025), and Agent Lightning (Luo et al., 2025) extend this paradigm to multi-turn interaction, optimizing tool selection and long-horizon behavior from outcome-based rewards. These approaches focus mainly on learning objectives, credit assignment, and policy optimization. EchoRL is complementary: it leaves the learning algorithm and sampled trajectories unchanged, while reducing the systems cost of collecting those trajectories by safely reusing deterministic environment computation.

Caching in LLM serving. Caching has been extensively studied for accelerating LLM inference. KV caching avoids recomputing attention states for previously processed tokens, while systems such as vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024) improve its memory management and enable KV-state sharing across requests with common prefixes. Prompt Cache (Gim et al., 2024), Mooncake (Qin et al., 2026), LMCache (Liu et al., 2025b), and CacheBlend (Yao et al., 2025) further extend reuse across structured prompts, storage tiers, serving instances, or independently cached document contexts. These techniques cache model-side representations or generated responses and therefore do not capture the mutable external state induced by tool execution. Most closely related to EchoRL, TVCache represents tool-call histories as a graph and reuses tool outputs and selected sandbox snapshots through exact prefix matching (Kumar et al., 2026). **Different from TVCache**, EchoRL proposes a multi-level environment cache that better reuses recurring states during training, an advantage-guided mechanism that improves the policy's cache hit rate throughout training, and a design that is compatible with heterogeneous environment backends.

Sandboxing, Snapshotting and Environment Forking. Sandboxing, snapshotting, and environment forking provide distinct but complementary capabilities for agent execution. Sandboxing establishes an isolated execution boundary in which untrusted tool calls can safely modify files, launch processes, and install dependencies. Docker containers (Merkel et al., 2014) and Firecracker microVMs (Agache et al., 2020) represent two common design points with different isolation and virtualization costs. Snapshotting captures an environment at a particular execution point so that its filesystem, processes, and, in some systems, memory state can later be restored. Representative mechanisms range from container checkpointing with CRIU (Andrijauskas et al., 2024) to full microVM snapshots. Environment forking builds on such state-capture mechanisms to create multiple independently writable descendants from a common state, typically using copy-on-write sharing to support parallel exploration. Recent agent-oriented runtimes such as forkd and AgentENV expose fast snapshot, restore, and fork primitives for speculative execution, fault recovery, and parallel RL rollouts, while Crab (Wu et al., 2026) adaptively selects checkpoint granularity according to the OS-visible effects of agent actions. These systems primarily optimize the mechanisms for isolating, capturing, and materializing execution states. EchoRL is orthogonal: it determines whether an environment transition is semantically valid for reuse and which reusable states should remain cached as the policy evolves. It can therefore employ Docker, forkd, AgentENV, or similar systems as its underlying environment backend.

6 Conclusion

We presented EchoRL, a runtime layer for reducing redundant environment computation in agentic reinforcement learning. EchoRL reuses only deterministic environment transitions while leaving policy-generated tokens, log-probabilities, rewards, and advantages untouched. By keying cached transitions on logical environment states and tool versions, EchoRL preserves the rollout distribution, and therefore avoids biasing policy-gradient estimation, under an explicit determinism assumption. Moreover, our empirical analysis reveals how environment-state reuse evolves as the policy is updated, enabling a policy-aware cache-admission strategy that retains states likely to be revisited in future training iterations. Together, these mechanisms substantially improve cache hit rates and reduce rollout latency without changing the underlying RL algorithm or execution backend. We believe EchoRL provides a practical foundation for integrating correctness-preserving, adaptive environment reuse into large-scale agentic RL systems.

AI USE STATEMENT

In this work, we used generative AI tools to assist with writing and editing the manuscript and, in a coding-assistant capacity, with parts of the implementation and the measurement and analysis scripts. We did not use generative AI tools to generate research ideas. All intellectual contributions pertaining to the central ideas, methodological design, experimental work, and analytical insights originate from the authors.

ETHICS STATEMENT

EchoRL is a training infrastructure that reuses deterministic environment work and caches neither model outputs nor user data. Environment state is isolated per task and tenant, snapshots scrub credentials, and sandboxes remain disposable. Our validation uses public benchmarks and involves no human subjects or personal data. Within foreseeable scope, we identify no additional risks.

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, 2020.

Fabio Andrijauskas, Igor Sfiligoi, Diego Davila, Aashay Arora, Jonathan Guiang, Brian Bockelman, Greg Thain, and Frank Würthwein. Criu-checkpoint restore in userspace for computational simulations and scientific applications. In *EPJ Web of Conferences*, volume 295, pp. 07046. EDP Sciences, 2024.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Guanting Dong, Hangyu Mao, Kai Ma, Licheng Bao, Yifei Chen, Zhongyuan Wang, Zhongxia Chen, Jiazhen Du, Huiyang Wang, Fuzheng Zhang, et al. Agentic reinforced policy optimization. In *International Conference on Learning Representations*, volume 2026, pp. 16981–17017, 2026.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms. In *International Conference on Learning Representations*, volume 2026, pp. 37909–37926, 2026.

Jingyue Gao, Yanjiang Guo, Chen Xiaoshuai, and Jianyu Chen. Proceedrl: Process critic with explorative demonstration reinforcement learning for llm agentic reasoning. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 16281–16295, 2026.

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338, 2024.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan O Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training LLMs to reason and leverage search engines with reinforcement learning. In *Second Conference on Language Modeling*, 2025.

Abhishek Vijaya Kumar, Bhaskar Kataria, Byungsoo Oh, Emaad Manzoor, and Rachee Singh. Tv-cache: A tool-value cache for post-training LLM agents. In *Forty-third International Conference on Machine Learning*, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Shu Liu, Alan Zhu, Sumanth Hegde, Shiyi Cao, Shuo Yuan, Samion Suwito, Tyler Griggs, Matei Zaharia, Joseph E Gonzalez, and Ion Stoica. Skyrl-sql: Multi-turn sql data agents via rl. In *First Workshop on Multi-Turn Interactions in Large Language Models*, 2025a.

Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, et al. Lmcache: An efficient kv cache layer for enterprise-scale llm inference. *arXiv preprint arXiv:2510.09665*, 2025b.

Xufang Luo, Yuge Zhang, Zhiyuan He, Zilong Wang, Siyun Zhao, Dongsheng Li, Luna K Qiu, and Yuqing Yang. Agent lightning: Train any ai agents with reinforcement learning. *arXiv preprint arXiv:2508.03680*, 2025.

Karttikeya Mangalam, Raiymbek Akshulakov, and Jitendra Malik. Egoschema: A diagnostic benchmark for very long-form video language understanding. *Advances in Neural Information Processing Systems*, 36:46212–46244, 2023.

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

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35: 27730–27744, 2022.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Heyi Tang, Feng Ren, Teng Ma, Shangming Cai, Yineng Zhang, Mingxing Zhang, et al. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *ACM Transactions on Storage*, 22(4):1–38, 2026.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Wanxin Tian, Shijie Zhang, Kevin Zhang, Xiaowei Chi, Chun-Kai Fan, Junyu Lu, Yulin Luo, Qiang Zhou, Yiming Zhao, Ning Liu, et al. Seea-r1: Tree-structured reinforcement fine-tuning for self-evolving embodied agents. *Advances in Neural Information Processing Systems*, 38:78458–78499, 2025.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, et al. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025.

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.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the twentieth European conference on computer systems*, pp. 94–109, 2025.

Peng Yu, Bo Liu, Shaomin Tang, Dongdong Li, and Weiwei Lin. Awto: A latency-optimized task offloading scheme for llm-driven agentic workflows on heterogeneous edge. *Future Generation Computer Systems*, pp. 108415, 2026.

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.

Zili Zhang, Yinmin Zhong, Chengxu Yang, Chao Jin, Bingyang Wu, Xinming Wei, Yuliang Liu, and Xin Jin. Heddle: A distributed orchestration system for agentic rl rollout. *arXiv preprint arXiv:2603.28101*, 2026.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, et al. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

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

A Technical Details

A.1 The Logical-State Descriptor

Computing $\ell(s)$ by hashing the complete physical sandbox state would introduce substantial overhead, because a sandbox may contain a large filesystem, running processes, and in-memory state. EchoRL instead constructs logical-state descriptors incrementally from the task-initialized state and the deterministic transitions that produce each successor state. Implementation details are provided in Sec B.3.

For an L1 task-initialized environment, EchoRL computes

$$\ell\left(s_x^{(0)}\right) = \mathrm{Hash}\left(x \,\|\, d_x^{\mathrm{image}} \,\|\, d_x^{\mathrm{init}} \,\|\, d_x^{\mathrm{config}}\right), \tag{25}$$

where d_x^{image} is the digest of the base image, d_x^{init} identifies the task-initialization procedure, and d_x^{config} captures relevant task inputs and runtime configuration.

For each subsequent tool invocation, EchoRL first constructs a canonical transition descriptor

$$q_t = \mathrm{Serialize}\left(\mathrm{name}(a_t), \mathrm{args}(a_t), \mathrm{cwd}(a_t), \mathrm{env}(a_t), v_t\right), \tag{26}$$

where serialization preserves all semantically relevant action information in a deterministic representation. The successor logical state is then derived recursively as

$$\ell(s_{t+1}) = \mathrm{Hash}\left(\ell(s_t) \,\|\, q_t\right). \tag{27}$$

For a deterministic read-only action, the environment state is unchanged and EchoRL retains

$$\ell(s_{t+1}) = \ell(s_t). \tag{28}$$

This construction identifies a logical state by the initial environment and the exact sequence of deterministic mutations that produced it. It does not require EchoRL to repeatedly hash the entire sandbox. Moreover, it is deliberately conservative: two different tool sequences that happen to produce equivalent physical states may receive different descriptors, resulting in a missed reuse opportunity, but they cannot produce an incorrect cache hit merely because their physical states appear similar.

The version descriptor v_t covers all dependencies that can affect tool execution. For example, for a source installation of FlashAttention, v_t may include the FlashAttention source commit, Python, PyTorch, CUDA, compiler, and build-configuration versions. When a transition produces an L2 snapshot, EchoRL records the association

$$\ell(s_{t+1}) \longmapsto m_2(s_{t+1}), \tag{29}$$

allowing a later rollout that reaches the same logical descriptor to materialize the corresponding sandbox directly.

Actions with uncontrolled inputs, such as mutable remote services, unversioned mounted files, system time, or unrecorded randomness, are marked as non-cacheable unless those inputs are incorporated into q_t or v_t. Once such an action mutates the environment, EchoRL also conservatively prevents reuse of descendant states whose determinism cannot be established.

A.2 Proof sketch of EchoRL Correctness

L1 restores a state logically equivalent to the result of executing the same task initializer. For L2, an exact match of $(\ell(s), a, v)$ and Assumption 1 guarantee that the cached and directly executed transitions produce identical observations and equivalent successor states. For L3, the matched logical state guarantees the same tool input, while the read-only property guarantees that returning the cached observation does not omit any environment mutation.

The trajectory-level result follows by induction. Cached and uncached rollouts begin from equivalent initial states. If their interaction histories are identical at step t, the agent policy samples the next action from the same conditional distribution. A transparent cache hit then produces the same observation and successor logical state, so the histories remain identical at step $t + 1$. This establishes Eqn. (21).

Consequently, EchoRL introduces no additional bias into the gradient estimator used by the underlying RL algorithm:

$$\mathbb{E}\left[\widehat{g}_{\mathrm{EchoRL}}(\theta)\right] = \mathbb{E}\left[\widehat{g}_{\mathrm{exec}}(\theta)\right]. \tag{30}$$

This result states that caching does not alter the estimator; it does not require the estimator used by the original RL algorithm to be unbiased in an absolute sense.

B Code Organization and Environment-State Reuse

This section describes the released implementation of EchoRL. The repository separates policy optimization from environment execution: Slime provides policy sampling and the Megatron training backend, whereas EchoRL implements task adapters, environment-state contracts, and cache management. A cache entry stores an environment observation and, when required by the action semantics, its successor-state snapshot. Generated policy tokens, rewards, and gradients are computed for each rollout and are not cache payloads.

B.1 Repository organization and principal interfaces

```
echorl/                  Cache contracts, execution, and task adapters
  contracts.py           Action identity and logical state cursors
  executor.py            Tool execution and cache-hit handling
  coordinator.py         Ownership, admission, eviction, and feedback
  artifacts.py           Content-addressed snapshots and private restores
  backends.py            Filesystem-backed state interface
  sql.py, rl_sql.py       SQL environment and Slime rollout integration
  terminal.py            Docker environment and private verifier
  rl_terminal.py         Terminal-Bench rollout integration
  ego/                   Video tools, explicit state, and rollout hooks
configs/                 Task, sampling, and cache configuration
scripts/                 Dependency/data preparation and training
    launchers
  sql/, ego/, terminal/ Task-specific launch, evaluation, and export
    tools
tests/                   Contract, lifecycle, coverage, and adapter tests
docker/                  Pinned training-container recipe
docs/                    Setup documentation
```

Third-party source checkouts, datasets, model weights, and generated artifacts are placed in the separate `vendor/`, `data/`, `models/`, and `runs/` directories, respectively. These directories are excluded from the source release. `dependencies.json` identifies external source revisions, while the launchers record configuration and source identities for each run.

The principal launch interfaces are `scripts/sql/run_local.py`, `scripts/ego/run.py`, and `scripts/terminal/run.py`. Additional no-cache epoch interfaces are provided by `scripts/sql/train_epochs.py` and `scripts/ego/run_epochs.py`. The video rollout and retention hooks reside in `echorl/ego/generate.py` and `echorl/ego/feedback.py`; `echorl/terminal_screen.py` implements training-task rollout screening.

Source and principal interface	Responsibility
`echorl/contracts.py`: `Action`, `ToolPolicy`, `StateCursor`, `transition_key`	Represent an action, declare reuse conditions, and identify the logical state from which the action is issued.
`echorl/executor.py`: `StateBackend`, `Executor.execute`	Define the backend contract and dispatch a fresh action through cache lookup, execution, optional snapshot creation, or state restoration.
`echorl/coordinator.py`: `Coordinator.claim`, `publish`, `commit_feedback`	Coordinate concurrent requests, reserve storage, admit entries, and update retention scores after successful optimizer updates.
`echorl/artifacts.py`: `ArtifactStore.capture`, `restore`, `release`	Serialize quiescent filesystem state into immutable blobs, reconstruct private state, and manage blob references.
`echorl/sql.py`: `SQLBackend.action`, `query_rows`, `reward`	Classify and execute SQLite queries and evaluate final SQL independently of the observation cache.
`echorl/ego/runtime.py`: `ToolSession`, `VideoBackend.checkpoint`, `restore`	Map video tool calls to cache actions and serialize or reload explicit video application state.
`echorl/terminal.py`: `command_policy`, `TerminalBackend.start`, `raw`, `grade`	Create rollout-local containers, classify shell commands, execute them, and invoke the task's private verifier after the policy terminates.
`echorl/schedule.py`: `EpochSchedule`; `echorl/rl_sql.py` and `echorl/rl_terminal.py`: `generate_one`, `normalize_and_record`	Assign tasks to updates, collect multi-turn trajectories, and record the scalar GRPO feedback associated with each sample.

B.2 Mapping a task to an executable environment

Data preparation produces manifests containing task identities, policy-visible inputs, and adapter-specific metadata. The rollout adapter constructs a backend for each sample and passes it to an `Executor`. The backend interface specifies `execute`, `checkpoint`, `restore`, `fingerprint`, `fork`, and `close`. Snapshot operations are used only for adapters whose declared capabilities support them; the observation-only adapters do not provide general state restoration.

SkyRL-SQL. Each manifest row maps a question to a SQLite path and database digest. `SQLBackend` opens request-local connections in read-only, immutable mode. An SQLite authorizer rejects writes and operations such as database attachment and extension loading. `classify_query` inspects an `EXPLAIN` statement and the functions it invokes; queries outside the deterministic function whitelist bypass reuse. The namespace combines the task identity with `SQLBackend.version`, which incorporates the database digest and relevant runtime versions. An eligible query changes no database state, so its cache entry contains the formatted observation without a successor snapshot.

EgoSchema. A manifest row identifies an immutable video by filename, path, and SHA-256, together with its question, answer options, and separate reference label. `ToolSession` constructs a `VideoBackend`; `VideoEnv` stores application state in the rollout-local entry of `media.STATES`. The cache namespace incorporates the video digest, frozen tool-model revision, frame count, vision seed, and deterministic-inference configuration. `load_video_into_sandbox` and `preprocess` are classified as state-changing actions. Caption retrieval and visual queries are classified as read-only actions under the adapter's seeded inference contract. The policy model and the frozen model serving video tools have distinct roles.

Terminal-Bench. A manifest row selects a task directory, and an image lock selects the Docker image used by `TerminalBackend.start`. Each rollout operates in an isolated environment with the resource limits specified in `task.toml`. The cache namespace identifies the task, base image, initialization configuration, and relevant tool and runtime versions. At L1, EchoRL caches the task-initialized environment and restores or forks an independently writable instance for each rollout, avoiding repeated setup. When the supplied image already contains the complete task-specific state and instantiation is inexpensive, the rollout can start directly from that image.

During interaction, EchoRL tracks the logical state induced by the generated tool-call prefix. At L2, a deterministic state-changing command, such as a dependency installation with pinned inputs or a reproducible compilation, can produce a reusable successor-state snapshot. On a cache miss, the command executes in the rollout's current environment; if the transition satisfies the cost-based admission rule and the backend can faithfully capture its relevant state, EchoRL stores the returned observation, successor logical-state descriptor, and snapshot handle. The entry is keyed by the task identity, pre-action logical state, tool invocation, and tool version. On a matching hit, EchoRL materializes a private instance of the cached successor state, checks its logical-state descriptor, and returns the stored observation before the policy continues. Subsequent mutations affect only that rollout's instance. Snapshot reuse requires the backend to preserve all filesystem and execution state relevant to future tool calls; transitions requiring unsupported state capture are not eligible for L2 reuse. At L3, deterministic commands can reuse their observations under the matched logical state without restoring a successor snapshot. Reuse is therefore not restricted to the initial workspace: it can also occur after a compatible L2 restoration or a directly executed deterministic transition. Commands with uncontrolled inputs bypass the cache, and mutations whose successor state cannot be identified soundly disable subsequent reuse. L2 and L3 entries follow the cost-based admission and advantage-guided retention rules in Section 3.3. Every tool call is still generated by the current policy, and the task's private verifier runs on the final rollout-local environment to compute a fresh reward.

B.3 Action identity, logical state, and cache dispatch

An `Action` contains a tool name, structured arguments, and a `ToolPolicy`. Its identity includes the tool-policy version and seed contract. `canonical` uses a type-preserving representation: dictionaries have sorted keys, but SQL text, shell commands, free text, and paths are not semantically rewritten. With H denoting SHA-256 over this representation, the implementation forms the transition key

$$k_t = H(\texttt{transition-v1},\, n,\, z_t,\, I(a_t)), \tag{31}$$

where n is the backend namespace, z_t is the cursor's logical state, and $I(a_t)$ is the action identity. Initially, $z_0 = H(\texttt{initial}, n)$. Read-only actions leave z_t unchanged; other actions advance it to $H(z_t, I(a_t))$. Thus, the logical identifier represents declared transition history rather than a hash of an arbitrary live process. The original action position is tracked separately. Restoration is eager: the implementation does not maintain a queue of deferred mutations.

Reuse additionally requires a cacheable cursor, a deterministic or explicitly seeded policy, replay safety, a known effect, and a backend satisfying the required capabilities. `Coordinator.claim` returns one of four states: a pinned hit, ownership of a missing entry, a wait on an existing builder, or a bypass when admission is unavailable. A single event loop and an asynchronous lock protect the coordinator; this is not a distributed locking service. Builders hold renewable leases and unique ownership tokens. Publication checks the token and lease again, so an expired builder cannot publish over a newer owner. Pins prevent eviction while an entry is being consumed.

For a read-only hit, `Executor.execute` returns a deep copy of the stored observation. For a state-changing hit, it first restores the successor snapshot, then advances and checks the logical successor before returning the observation. The event log denotes state-restoring hits as L2 and observation-only hits as L3. On a miss or bypass, the action executes on the current backend. Uncontrolled state-changing actions, non-cacheable state-changing results, and exceptions during such actions disable further reuse. An optional shadow audit forks the pre-action state, re-executes the action, and compares both its observation and the resulting state fingerprint; mismatches invalidate the entry and raise an error. The default shadow-audit fraction is zero.

B.4 SNAPSHOT CONSTRUCTION AND PRIVATE RESTORATION

EchoRL stores backend-specific snapshot handles for L1 task-initialized environments and L2 successor states. A reusable snapshot must capture all state relevant to subsequent tool execution and support restoration into an independently writable environment. The backend establishes a consistent checkpoint boundary before capture; the runtime associates the snapshot with its logical-state descriptor and execution namespace. An L2 entry additionally stores the observation returned by the transition that produced the snapshot.

The filesystem implementation, `DirectoryBackend`, captures files and quiescent application state. Its backend must finish the action and quiesce its state before `ArtifactStore.capture` traverses the snapshot tree. Capture records relative paths, file contents and modes, directory names and modes, and the root mode. Symlinks and non-regular filesystem objects are rejected. Process memory, open file descriptors, sockets, device state, and external services are outside this filesystem snapshot contract; transitions that depend on such state require a backend capable of preserving it before they can be reused through L2.

Each regular file is streamed to a staging file while its content is hashed. Capture checks the reserved byte limit and detects size or modification-time changes during copying. After flushing and synchronizing the staging file, a new blob is published by atomic replacement under its content digest and made read-only. Existing blobs with the same digest are reused, with reference counts tracking ownership. The resulting `Snapshot` is a metadata manifest referencing these immutable blobs. Its logical byte count sums the referenced file sizes; physical storage can share identical blobs across snapshots.

`ArtifactStore.restore` reconstructs a new private directory. It checks that reconstructed paths remain under that directory and attempts a copy-on-write reflink for each file. Where reflinks are unavailable, `copy_private_file` performs an ordinary copy. Writable hard links to cached blobs are never used. File modes are restored, followed by directory modes after their children have been written. `DirectoryBackend.restore` then replaces the rollout's active root and removes its previous private tree. Blob references are released on eviction or cleanup; failed deletions remain accounted for until retried. The lightweight snapshot-health check verifies blob existence and size; it is not a full content rehash on every cache hit.

For EgoSchema, `VideoBackend.checkpoint` writes the complete explicit application state to `state.json` before invoking filesystem capture. After preprocessing, this state includes the immutable video's path, duration, sampled frame timestamps and base64-encoded JPEGs, and generated segment captions. Restoration reloads that JSON into the receiving environment's own state entry. The MP4 itself is referenced by its namespace digest rather than copied into every snapshot.

For Terminal-Bench, snapshot construction and restoration operate through the environment backend described in Appendix B.2. L1 retains a snapshot of the initialized task environment, from which rollouts obtain private instances. L2 retains intermediate snapshots after eligible deterministic commands, including the resulting filesystem changes and any execution state required for subsequent commands. Snapshot eligibility depends on the backend's capture capabilities: a filesystem copy alone is insufficient when correctness requires live process or other runtime state. Commands whose required state cannot be captured faithfully execute without L2 reuse.

On an L2 hit, the runtime first matches the task namespace, pre-action logical state, command, and tool version. It then restores or forks a private environment from the cached successor snapshot and verifies the associated successor logical-state descriptor before returning the cached observation. The rollout continues from that restored environment, with later writes isolated from both the cached snapshot and other rollouts. Cached snapshots remain immutable, and entries in active use are pinned against eviction. This restoration contract is independent of whether the backend represents a snapshot as filesystem artifacts or a native environment checkpoint. Read-only L3 hits return only the cached observation and require no successor-state restoration. SkyRL-SQL uses observation-only entries over an immutable database and therefore does not require environment snapshots.

B.5 ADMISSION, RETENTION, AND INTEGRATION WITH TRAINING

Admission is conditional on both resource reservations and execution cost. The executor first compares measured execution time with an explicit backend reuse-cost estimate. When a state-changing

action requires a snapshot, snapshot construction time is added to this estimate before `publish` repeats the cost test. In both cases, publication requires $t_{\text{exec}} > \gamma\, \widehat{t}_{\text{reuse}}$. This is an admission rule, not a measurement of end-to-end training performance. The coordinator enforces global and per-task memory/disk limits, a metadata limit, an in-flight disk limit, and a bound on concurrent builds.

After a successful optimizer update, `commit_feedback` assigns each retained entry the mean scalar advantage of distinct rollouts that visited its key in that update. Duplicate update identifiers do not reapply feedback. The `advantage_lru` policy groups scores by sign and orders entries by recency within each group. New entries have zero score: under the current admission path, unpinned negative- and zero-score entries can be evicted to admit a new entry, whereas positive-score entries are protected from that admission. Time-to-live expiration and invalidation are handled separately. A plain `lru` mode is also available.

The SQL and Terminal-Bench `generate_one` functions, and `echorl/ego/generate.py`'s `generate`, obtain fresh policy tokens and log probabilities from the rollout engine. Environment observations are appended as new conversation turns with zero policy-loss mask. The training hooks record the scalar GRPO advantages and visited cache keys; an optimizer commit must exist before cache-retention feedback is applied. Epoch-oriented entry points synchronize updated policy weights to the rollout engine before held-out evaluation, which disables environment reuse. SQL's checkpoint-enabled path can serialize unpinned observation entries and retention metadata for resumption; this mechanism is separate from filesystem snapshots and from model/optimizer checkpoints. The dedicated no-cache epoch entry points and the Terminal-Bench trainer do not use that cache-resumption path.

C Additional Experiment Settings

C.1 Training Configuration and Resource Allocation

This section specifies the training configuration implemented by the released launchers. All three tasks use Qwen3.5-9B, initialized from the same pretrained model revision, and the Slime training framework with a Megatron optimization backend and SGLang rollout inference. Configuration values describe the execution protocol; they do not imply that every supported configuration has been evaluated. Dataset manifests and the resolved configuration saved in each run directory identify the configuration of an individual experiment.

C.1.1 Hardware allocation and execution order

Each training job uses four GPUs on a single node. The first two selected GPUs are allocated to the trainable policy, and the remaining two are allocated to one rollout inference engine. Both components use tensor parallelism of degree two. Pipeline, context, expert, and expert tensor parallelism each have degree one; sequence parallelism is enabled for training. The recorded local deployment uses NVIDIA H20-3e GPUs with 143,771 MiB of device memory per GPU. This hardware description identifies the deployment and is not a minimum memory requirement inferred from profiling.

Component	Parallelism
Policy optimization	Tensor parallelism: 2
Policy rollout and held-out inference	One engine; tensor parallelism: 2

The training loop is synchronous. It first broadcasts the initial policy weights to the rollout engine. Each update then generates a rollout batch, waits for one optimization step to complete, records the update, and broadcasts the updated weights. Although the training API is named `async_train`, the outer loop waits for its result before proceeding. Held-out evaluation in the epoch-based interfaces follows the final weight broadcast of that epoch and uses the existing rollout engine.

EgoSchema additionally serves a frozen copy of Qwen3.5-9B for its video tools. This service shares the two rollout GPUs and is not updated by policy optimization. The policy and video-tool engines use SGLang static memory fractions of 0.25 and 0.20, respectively. SQL and Terminal-Bench use a policy engine fraction of 0.65. These settings control SGLang memory allocation; they are not measurements of total device-memory consumption.

C.1.2 Optimization and rollout parameters

Training uses the GRPO advantage estimator and the Adam optimizer with a constant learning rate of 10^{-6}, $\beta_1 = 0.9$, and $\beta_2 = 0.98$. One optimization step is performed per rollout batch. Entropy regularization, KL penalties, attention dropout, and hidden dropout are disabled in these configurations. The reward post-processing hooks apply within-group reward normalization for optimization while retaining raw rewards for reporting. Training uses BF16 parameters, FP32 gradient accumulation and all-reduce, and FP32 attention softmax. Full activation recomputation is enabled with uniform recomputation groups of one layer. Dynamic batching uses the per-device token budgets listed below.

Let B denote the number of task prompts in a rollout batch and G the number of sampled trajectories per prompt. The nominal global training batch is BG trajectories. It is distinct from both the inference request concurrency and the token budget used to construct training microbatches.

Parameter	SkyRL-SQL	EgoSchema	Terminal-Bench
Prompts per update, B	8	2	8
Trajectories per prompt, G	8	8	8
Nominal trajectories per update, BG	64	16	64
Training temperature	0.7	0.7	0.7
Maximum agent turns	5	5	30
Maximum context tokens	32,768	4,096	32,768
Generated-token budget per trajectory	8,192	1,024	16,384
Training tokens per GPU (batching budget)	32,768	4,096	32,768
Lower/upper clipping parameters	0.20/0.20	0.20/0.28	0.20/0.20
Weight decay	0.01	0.10	0.01
SGLang concurrency	32	8	32

SQL and Terminal-Bench enable the model's thinking mode; EgoSchema disables it. SQL allows 32 concurrent environment rollouts, whereas Terminal-Bench limits environment rollout concurrency to 16 even though its inference engine allows 32 running requests. Terminal-Bench additionally caps each generation request at 4,096 tokens. EgoSchema caps each request at the smaller of 512 tokens, the remaining context capacity, and the remaining trajectory budget. These per-request limits do not replace the trajectory-level token limits in the table.

The EgoSchema video tools use 16 sampled frames. Their frozen inference service uses tensor parallelism of degree two, a context limit of 16,384 tokens, at most four running requests, and a total-token capacity of 32,768. Video-tool requests use temperature zero, and at most 256 generated tokens, with thinking disabled. The service also disables CUDA graphs and radix caching and enables deterministic inference.

C.1.3 Dataset coverage and epoch construction

The epoch interfaces train for ten epochs. Each real task contributes exactly G trajectories per epoch. A local random generator seeded with the training seed plus the zero-based epoch index determines task order. If the final batch is incomplete, these two adapters insert padding slots with zero training loss. Padding is excluded from task coverage checks and reward summaries. The EgoSchema epoch adapter instead requires an even number of training videos and rejects incomplete batches.

Dataset configuration	Train	Held out	Updates/epoch	Real rollouts/epoch
SkyRL-SQL	653	1,034	82	5,224
EgoSchema	400	100	200	3,200
Terminal-Bench	71	18	9	568

SkyRL-SQL retains the dataset's official training and validation splits. EgoSchema uses a custom video-disjoint partition of publicly labeled videos. The release data-preparation path defaults to 400 training and 100 validation videos. Terminal-Bench uses a custom task-disjoint 71/18 train/test split. Optional pre-training screening executes one rollout per training task using the initial policy, records

task failures, and performs no optimization. When a screening report is supplied, the accepted training manifest is fixed and shared across comparison arms; the test manifest is unchanged. For example, 70 retained training tasks imply nine updates and 560 real trajectories per epoch. The retained task count must be reported with the corresponding run.

The EgoSchema paired tool-timing entrypoint is a separate protocol. Its default full stage uses 16 selected training videos for two epochs, with $B = 2$ and $G = 4$, giving 16 updates per arm. It does not perform held-out validation. This configuration should be distinguished from the ten-epoch video-disjoint training protocol above.

C.1.4 Evaluation, reward accounting, and persistence

Epoch-based held-out evaluation samples one trajectory per validation or test task at temperature zero after synchronizing the latest policy weights. Evaluation does not update the policy.

For epoch e, the reported mean training reward is the sample-weighted average of raw task rewards over real trajectories,

$$\overline{r}_e = \frac{1}{N_e G} \sum_{i=1}^{N_e} \sum_{j=1}^{G} r_{e,i,j}.$$

It is computed before GRPO reward normalization and excludes padding. Consequently, it is not the unweighted average of batch means when the final batch contains fewer real tasks. The reporting code checks task coverage against the manifest and aggregates batches associated with completed optimizer updates. Task accuracy or success rate is recorded separately from mean reward, since invalid-format penalties can make these quantities differ.

C.1.5 Cache settings

The no-cache condition disables EchoRL environment-state reuse. In the EchoRL condition, SQL configure 2 GiB of global payload RAM, 64 MiB per task, 128 MiB for metadata with no disk payload budget. The EgoSchema configuration uses the same RAM limits and additionally allocates an 8 GiB disk budget, a 512 MiB per-task disk budget, and a 1 GiB in-flight disk budget. The Terminal-Bench configure 100 GiB of global payload RAM, 4 GiB per task, 128 MiB for metadata with 128 GiB disk payload budget. The cache retention parameter is $\gamma_{\text{cache}} = 2$.

C.2 Task Prompts and Illustrative Interaction Formats

This section specifies the policy-facing interfaces for SkyRL-SQL, EgoSchema, and Terminal-Bench 2. The system messages below reproduce the released prompt text; line wrapping is only for typesetting. The subsequent task instances and responses are constructed examples illustrating the interface, rather than benchmark trajectories or empirical results. Each example distinguishes the user input, policy output, environment observation, and final response. Reference labels and private verification code are not included in the policy-visible prompt.

All three interfaces use the Qwen chat template. SQL and Terminal-Bench enable the model's thinking mode in their default configuration, whereas the video policy prompt disables it. For readability, the examples show decoded message bodies and omit tokenizer-specific delimiters such as `<|im_start|>` and `<|im_end|>`. Brief text inside `<think>` or the video `thought` field illustrates syntax only.

Reading guide. Pale yellow denotes system instructions and task prompts; pale blue denotes intermediate policy outputs; light gray denotes environment observations and file contents; pale green denotes final responses.

C.2.1 SkyRL-SQL

Prompt construction. `scripts/prepare_manifests.py` preserves the official dataset's `prompt` message list, and `echorl/rl_sql.py`'s `generate_one` applies the tokenizer's chat template to that list. The system message reproduced below is shared by the prepared train and validation splits. Its schema, knowledge, and question placeholders remain literal in the system message; the corresponding values are supplied by the following dataset user message. EchoRL

does not replace this prompt with a separate system instruction. The example embedded in the system message is part of the upstream prompt.

```
Task Overview:
You are a data science expert. Below, you are provided with a database
   schema and a natural language question. Your task is to understand
  the schema and generate a valid SQL query to answer the question
  within limited turns. You should breakdown the problem, draft your
  reasoning process, and generate the solution.

Database Engine:
SQLite

Database Schema:
{db_details}
This schema describes the database's structure, including tables,
  columns, primary keys, foreign keys, and any relevant relationships
  or constraints.

External Knowledge:
{external_knowledge}

Question:
{question}

Instructions:
- Make sure you only output the information that is asked in the
  question. If the question asks for a specific column, make sure to
  only include that column in the SELECT clause, nothing more.
- The generated query should return all of the information asked in
  the question without any missing or extra information.
- Before generating the final SQL query, please think through the
  steps of how to write the query. It should include detailed
  considerations such as analisying questions,
summarizing relevant findings, brainstorming new ideas, verifying the
  accuracy of the current steps, refining any errors, thinking of how
  to call SQL tools, and revisiting previous steps.


Format:
- Conduct thinking inside <think>...</think> blocks every time you get
   new observation or information.
- You can use SQL tool written within a single <sql>your sql</sql>
  block to explore or verify. SQL tool output will be shown as
  dataframe inside <observation>...</observation>. Based on this
  observation, you can think again and refine.
- The returned dataframe will be truncated in 50 rows if observation
  is too long.
- If you find no further exploration is needed or reaches max turns,
  you MUST directly provide the final SQL query solution inside <
  solution>...</solution>.

------------------------ START OF EXAMPLE ------------------------
Question: how many pigs are in the farm?
    Database Schema:
    Table: animals
    - id (INTEGER, PRIMARY KEY)
    - species (TEXT)
    - age (INTEGER)
    - name (TEXT)

<think>I am querying how many pigs are in the farm. I will begin by
  checking if the 'animals' table exists and contains entries with
  species = 'pig'.</think>
```

```
<sql>SELECT COUNT(*) FROM animals WHERE species = 'pig';</sql>
<observation>
+----------+
| COUNT(*) |
+----------+
|   12     |
+----------+
</observation>
<think>The result indicates that there are 12 pigs in the farm. Since
  the question asks for how many pigs, I can now output the final SQL
  as the solution.</think>
<solution>SELECT COUNT(*) FROM animals WHERE species = 'pig';</
  solution>
------------------------ END OF EXAMPLE ------------------------
```

Illustrative task input. Consider an SQLite fixture with three staff records: `(1, 'Engineering')`, `(2, 'Engineering')`, and `(3, 'Sales')`. These records reside in the environment; the user message provides the schema and question in the dataset's field-labelled format:

```
{db_details}:CREATE TABLE staff (
    id INTEGER PRIMARY KEY,
    department TEXT
);;
 {external_knowledge}: ;
 {question}: How many staff members work in Engineering?
```

Illustrative tool request and observation. An intermediate policy message contains one SQL block:

```
<think>I will check the number of Engineering staff.</think>
<sql>SELECT COUNT(*) FROM staff
WHERE department = 'Engineering';</sql>
```

`generate_one` extracts the SQL block and calls `Executor.execute(SQLBackend.action(query))`. The adapter represents query rows as a `frozenset` and renders a pandas dataframe without its index. It therefore does not preserve SQL column names or row order in this observation. For the fixture above, the next user message contains:

```
<observation> 0
 2
<reminder>You have 4 turns left to complete the task.</reminder></
  observation>
```

The reminder shown corresponds to the first tool interaction under the default five-turn budget. `echorl/sql_messages.py`'s `observation_turn` wraps this text in a new user turn and appends the next assistant header without retokenizing previously sampled policy tokens.

Illustrative final response and evaluation. The final artifact is a SQL query, not the numeric answer alone:

```
<think>The count query answers the question.</think>
<solution>SELECT COUNT(*) FROM staff
WHERE department = 'Engineering';</solution>
```

The rollout stops on a completed solution block. `reward` invokes the upstream format validator and independently executes the predicted and reference queries. A valid solution receives reward one when their result sets match and zero otherwise; invalid completion format receives minus one. This

reward computation does not read from the observation cache. The default interface allows at most five generation turns, 8,192 generated tokens, and a 32,768-token context; these are configuration limits, not properties of the dataset.

C.2.2 EGOSCHEMA

Prompt construction. `echorl/ego/generate.py`'s `generate` supplies its `SYSTEM` constant as the system message. The user message concatenates the question, five indexed answer options, and the video filename. The policy receives video observations through tool calls; frames are sent to a separate frozen vision model rather than directly inserted into the policy's initial message.

```
You are a video question-answering agent. Return a JSON object only:
{"thought":"one short sentence", "actions":[{"tool":"name","inputs":"
  string"}], "final_answer":null}.
Keep thought under 40 words. Tool inputs MUST be JSON strings,
  including tuple arguments.
When ready, return {"thought":"brief reasoning", "actions":[], "
  final_answer":0} with an answer from 0 to 4.
First call load_video_into_sandbox with the provided video filename,
  then preprocess with inputs="".
You may put these two actions in one response; they execute
  sequentially. Observe the video evidence before answering.
preprocess samples 16 frames across the actual video and describes two
   temporal segments, indexed 0 and 1.
Available tools after preprocessing:
caption_retrieval inputs="(0, 1)": return captions for the segment
  range.
object_memory_querying inputs="a question": inspect sampled video
  frames for objects/actions.
segment_localization inputs="event description": locate the event in
  sampled frames.
visual_question_answering inputs="('a question', 0)": inspect a
  temporal segment (0 or 1).
The observations may be incomplete. Make your best choice using the
  evidence. Do not emit Markdown.
```

Illustrative task input. Assume that a hypothetical clip depicts its wearer rinsing a cup at a sink. The user message has the following form:

```
What does the camera wearer do with the cup?
Options:
0: Places it in a cupboard.
1: Fills it with coffee.
2: Rinses it under running water.
3: Hands it to another person.
4: Packs it into a bag.
Video filename: example_video.mp4
```

Illustrative tool request and observation. The policy may issue the two required setup actions in one JSON response:

```
{"thought":"I will inspect the clip before answering.",
 "actions":[
   {"tool":"load_video_into_sandbox",
    "inputs":"example_video.mp4"},
   {"tool":"preprocess","inputs":""}
 ],
 "final_answer":null}
```

These actions execute sequentially. The rollout adapter joins their returned strings with a newline and supplies the concatenation as the next user message. An illustrative observation for a 12-second clip is:

```
Loaded example_video.mp4
{"duration_s":12.0,"segments":2,"captions":[
  "The wearer holds a cup above a sink.",
  "The wearer rinses the cup under running water."
]}
```

Here the captions are illustrative, not measured tool outputs. In the actual implementation, `decode_frames` samples 16 frames by default and `VideoEnv` partitions them into two temporal segments. Each segment is captioned using the following fixed request to the frozen vision model:

```
Describe the actions, objects and their temporal order in these
  egocentric video frames. Be factual and concise.
```

Each request interleaves text of the form `Frame at <time> seconds:` with base64 JPEG image content. `vision` uses temperature zero, seed 1234, a 256-token response limit, and disabled thinking. Further policy actions use string-valued arguments; for example, a segment-specific query is expressed as:

```
{"thought":"I will inspect the second segment.",
 "actions":[{"tool":"visual_question_answering",
             "inputs":"('What happens to the cup?', 1)"}],
 "final_answer":null}
```

Illustrative final response and evaluation. Once sufficient evidence has been observed, the policy emits an integer option index and an empty action list:

```
{"thought":"The observed action is rinsing the cup.",
 "actions":[],"final_answer":2}
```

The integer is compared with the reference label retained in sample metadata. A correctly formatted correct answer receives reward one; an incorrect answer or absence of a final answer receives zero; an invalid response format receives minus two. The supplied training scripts use a five-turn tool loop, a 1,024-token policy-response budget, and a 4,096-token policy context. The `thought` field is part of the task's JSON protocol and is separate from the model-specific thinking mode, which is disabled for this policy interface.

C.2.3 TERMINAL-BENCH

Prompt construction. `echorl/rl_terminal.py`'s `task_rows` constructs two messages: `SYSTEM_PROMPT` from `echorl/terminal.py`, followed by the exact contents of the selected task's `instruction.md`. Each benchmark instance thus supplies its own user instruction under the same terminal interaction protocol. Filesystem outputs and persistent processes belong to the task's isolated container, not to the host repository.

```
You are an agent solving a task in a Linux terminal. Use the bash tool
   to inspect and modify the environment. Each command runs in a new
  bash shell in the initial working directory; files and background
  processes persist within your task. Use explicit cd or absolute
  paths. Return exactly one tool call per turn in this format:
<tool_call>{"name":"bash","arguments":{"command":"your shell command
  ","timeout_sec":60}}</tool_call>
When finished, respond with <done>summary</done>. You have no access
  to the private verifier. Do not create or modify /tests or /logs/
  verifier. Complete the actual task in the environment. Tool timeouts
   are capped at 300 seconds per call.
```

Illustrative task input. The following synthetic instruction illustrates the task interface; it is not presented as an official Terminal-Bench instance. Assume a fixture at `/app/input.txt` containing three lines, `pear`, `apple`, and `pear`, in that order:

```
Read /app/input.txt and write its distinct lines in
lexicographic order to /app/output.txt.
Each output line must end with a newline.
Do not modify /app/input.txt.
```

Illustrative tool requests and observations. A policy message first inspects the task input:

```
<think>I will inspect the input file.</think>
<tool_call>{"name":"bash","arguments":{
"command":"cat /app/input.txt","timeout_sec":60}}
</tool_call>
```

`parse_call` parses the tool-call JSON outside the reasoning segment. The backend runs the command with a timeout and returns exit status, standard output, and standard error. The rollout adapter serializes that result into a new user message:

```
Tool result:
<tool_response>
{"exit_code":0,"stdout":"pear apple pear","stderr":""}
</tool_response>
```

The next policy message creates the requested output file:

```
<think>I will write the distinct sorted lines.</think>
<tool_call>{"name":"bash","arguments":{
"command":"sort -u /app/input.txt > /app/output.txt",
"timeout_sec":60}}</tool_call>
```

```
Tool result:
<tool_response>
{"exit_code":0,"stdout":"","stderr":""}
</tool_response>
```

The first command can be eligible for initial-workspace observation reuse when the input file appears in the startup inventory. The redirection in the second command is outside the reuse whitelist: the command executes normally, and subsequent commands in that rollout bypass the cache. Eligibility alone does not imply admission or a cache hit.

Illustrative final response and evaluation. The policy terminates with a completion marker:

```
<think>The requested output file has been written.</think>
<done>Created /app/output.txt with the distinct sorted lines.</done>
```

For this fixture, the intended file contents are:

```
apple
pear
```

The text inside `<done>` is not itself the correctness criterion. After policy termination, `TerminalBackend.grade` copies the private tests into the container and runs the task's original verifier, which produces `reward.txt` and structured test output. Verifier setup failures are reported as infrastructure errors; a verifier timeout after tests have started has a separate state and a zero reward. No verifier output or reward is reused. The default rollout budget is 30 turns, 16,384

generated tokens, and a 32,768-token context, with at most 4,096 generated tokens per turn. Shell-call timeouts are clamped to 1–300 seconds and also bounded by the remaining per-task agent budget. Environment observations receive zero policy-loss mask; policy-generated tokens retain their sampling log probabilities.

D Additional Experiment Results

D.1 Environment Backend Influence

EchoRL is compatible with different environment backends, allowing its environment-state reuse mechanisms to operate independently of the underlying execution infrastructure. To examine the influence of backend choice, we evaluate AgentENV and forkd as alternatives to Docker for environment snapshotting, execution, and restoration. However, our current evaluation setting offers limited opportunities to exploit their respective strengths: forkd is particularly suited to rapidly creating many environment instances from a shared snapshot, while AgentENV targets large-scale agent training in distributed environments. In our single-machine setting with eight rollouts per task, neither backend provides a performance advantage over the default Docker backend (Figure 5). These results are specific to the scale and configuration of our evaluation and do not establish the relative performance of these backends at larger scales. We leave evaluation under higher rollout concurrency and distributed training to future work.

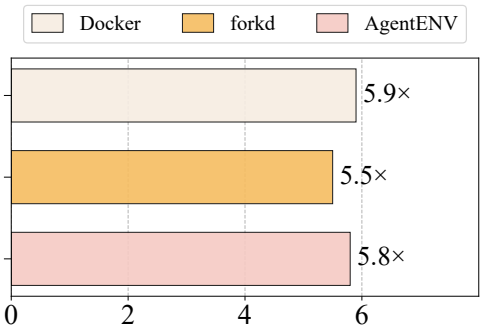

Figure 5: Environment execution speedup over No Cache on Terminal-Bench across different environment backends.

D.2 Effectiveness across Model Scales

To assess whether the benefits of EchoRL extend beyond the Qwen3.5-9B model used in our main experiments, we additionally evaluate Qwen3.5-4B and Qwen3.5-27B, covering both smaller and larger policy models within the same model family. As shown in Table 3, EchoRL reduces environment execution time across all three model scales, indicating that its effectiveness is not restricted to a particular parameter count. This observation is consistent with EchoRL's design: it reuses environment states and deterministic tool results through an interface that is independent of the policy model's architecture and size. Nevertheless, the magnitude of the benefit can vary with the trajectories generated by each model and the resulting opportunities for environment reuse. These results support the applicability of EchoRL across the evaluated model scales; broader validation across model families remains an avenue for future work.

Table 3: Environment execution time per rollout on Terminal-Bench across different LLMs.

LLMs	**Env. time (s/rollout)**		
	No Cache	EchoRL	Speedup
Qwen3.5-4B	184.6	33.1	5.6×
Qwen3.5-9B	209.8	35.7	5.9×
Qwen3.5-27B	317.2	47.3	6.7×