TACO: A NON-INTRUSIVE RL FRAMEWORK FOR ONLINE, BLACK-BOX AGENT SERVICES

Anonymous authors
Paper under double-blind review

ABSTRACT

A growing number of deployed agents operate at the *service level*: they run online as black boxes, managing their own loops, tools, and model providers, with fixed configurations after deployment. Existing reinforcement learning (RL) frameworks, however, generally assume the agent loop can be relocated into the RL runtime, making it difficult to improve deployed agents without rebuilding them. We present TACO, a non-intrusive RL framework that trains such agents directly in their existing execution environments. TACO requires no changes to the agent source code; instead, before the service starts, it uses only a single configurable field the agent passes to the model unchanged (e.g., the model endpoint or API key) to carry a *worker identifier* and track rollout identity. To improve training efficiency, TACO packs rollout turns using bounded token-level *edit distance* to avoid repeated context computation, and overlaps rollout with training while draining admitted tasks before weight synchronization. We evaluate TACO on three diverse agents: a ReTool-style math agent, a TritonForge-style kernel agent, and ZeroClaw, a long-running Rust service. Integrating them with TACO requires only 122–334 lines of adapter code. On the multi-turn math agent, packing reduces processed tokens by $3.0\times$, and combining packing with asynchrony gives a $2.51\times$ end-to-end speedup. TACO is now available on a commercial cloud AI platform as a selectable built-in training image for service-level agentic RL.

1 INTRODUCTION

A modern agent is no longer a single model call. It runs a multi-turn loop, calls tools, executes code in a sandbox, and often keeps state across a session (Yao et al., 2023; Yang et al., 2024; Wang et al., 2025b). A growing share of agents are *service-level*: online and already deployed as processes, containers, or long-running services, with their own loops, tools, memory, model providers, and dependencies. From the outside they are black boxes, and their configurations are fixed after deployment. Improving the model behind such an agent in its own environment is a practical goal, and reinforcement learning is a natural instrument (Christiano et al., 2017; DeepSeek-AI, 2025).

Mainstream RL frameworks assume the agent loop can move inside the training runtime. This gives fine-grained control but requires rewriting the loop against a framework API, re-declaring tool schemas, migrating the sandbox, and rebuilding the container. Agents may resist these changes: they use other languages, pin dependencies, own their memory models, and expose only a service interface. Migration also changes the production environment where we want to improve the agent's performance. The question is therefore *how to train the model behind an existing, unmodified agent*.

Recent work decouples agent execution from RL training through a disaggregated architecture, allowing existing agents to run in their own runtimes while a separate trainer consumes their rollouts (Luo et al., 2025). We take this separation as given. What remains unclear is how such a design behaves once the agent is a service-level black box.

Telling concurrent rollouts apart. Recovering token IDs only helps if we know which rollout each request belongs to. The framework can track rollout identity for LLM calls it makes, but agent-issued calls carry no such identity. A deployed agent is configured once before any rollout exists. When several rollouts run concurrently, their requests arrive interleaved on one endpoint. Without

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.

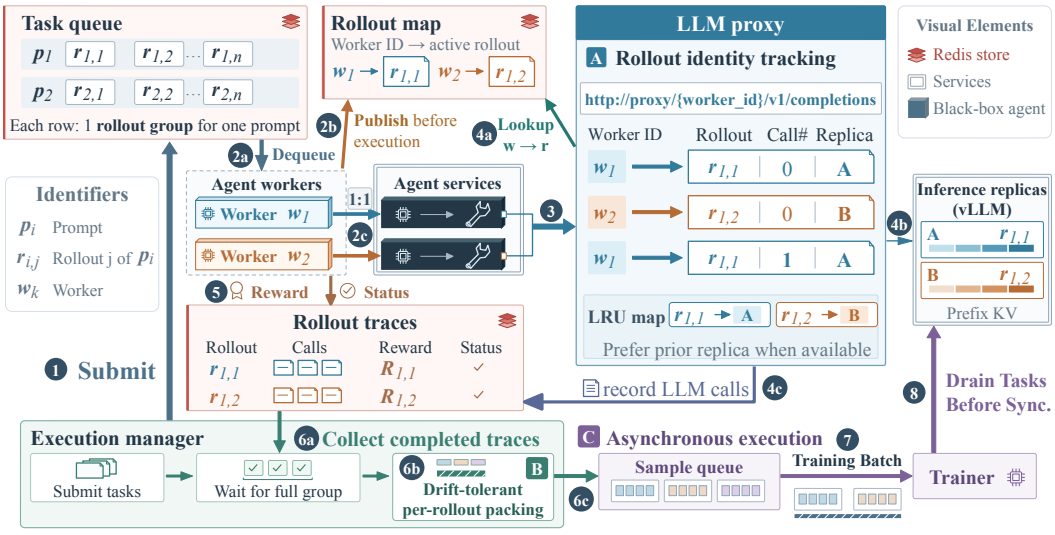

Figure 1: TACO's asynchronous pipeline for training black-box agent services.

separating them, we cannot reconstruct trajectories, group the n rollouts of a prompt for GRPO (Shao et al., 2024), or assign the correct reward.

Deciding when two turns can be merged. A T-turn rollout re-encodes its growing history T times. Merging its turns removes this redundancy, but is lossless only when the recorded history and its replay in the next prompt have identical token IDs. Opaque agents often break this condition. They may rewrite, summarise, or trim the history. Even unchanged history can acquire different token IDs when the next prompt is reconstructed and re-tokenized after appending tool outputs. In the ReTool-style math agent, exact matching succeeds on only 0.16% of merge attempts; 99.6% of failures have a located response boundary but nonidentical token prefixes.

Overlapping rollout with training. Asynchronous RL overlaps generation with training, normally assuming the runtime can monitor and cancel rollouts before changing weights. External agent tasks may be long-tailed, can fail inside opaque containers, and expose no checkpoint of their loops, tools, memory, or side effects, so they cannot be easily suspended and resumed.

TACO addresses these challenges in the agents' existing execution environments (Figure 1). It requires only one configuration change before the service starts: a configurable field that the agent passes to the model unchanged, such as the LLM endpoint path, model name, API key, or a dedicated field added for this purpose, while the agent retains its source code, tools, and runtime dependencies. TACO is now deployed on a commercial cloud AI platform as a selectable built-in training image, providing a non-intrusive RL framework for service-level agents.

In TACO, each worker has its own service instance that executes rollouts sequentially, while multiple worker–service pairs run concurrently. Each service carries its worker's identifier in the configured field, allowing the framework to associate each LLM request with the worker's current rollout. TACO then packs compatible turns from the same rollout into training sequences under a bounded token-level edit distance. For asynchronous execution, TACO overlaps rollout with training while draining admitted tasks before weight synchronization, so that each rollout completes under the same model weights. A queue, a proxy, and an agent server coordinate execution outside the agent.

Our contributions are three mechanisms for training opaque service-level agents and an empirical study of their training cost and accuracy.

Training deployed agents without entering their loop. One configuration field forwarded unchanged by each service instance, together with a worker identifier placed in that field, lets the proxy recover rollout identity from interleaved requests. We train three black-box agents without modifying their source: a Python math agent, a kernel-generation agent, and ZeroClaw, a long-running Rust service. Integrating them with TACO requires only 122–334 lines of adapter code (§3.1, §4.2).

Drift-tolerant trajectory packing. We merge turns under a bounded token-level edit distance to absorb small history changes and re-tokenization drift. On the math agent, exact-prefix matching succeeds on only 0.16% of merge attempts, leaving rollouts almost entirely unmerged; our tolerant packing reduces processed tokens by $3.0\times$ over unpacked training and improves step time by $1.75\times$ over exact-prefix packing. Packing also reduces the optimizer update count at a fixed PPO mini-batch size. Reducing that size increases the update count and recovers accuracy comparable to unpacked training on the same dataset (§3.2, §4.3).

Asynchronous execution for non-preemptible agent tasks. We overlap rollout with training and drain admitted tasks before weight synchronization, without preempting in-flight agent tasks. Asynchrony gives $1.60\times$ speedup with packing and $1.14\times$ without it: packing shortens actor updates and reduces the time rollout GPUs wait for synchronization, making most training overlap with rollout generation. Together, packing and asynchrony give a $2.51\times$ end-to-end speedup (§3.3, §4.4).

2 Problem setting

Which side issues the model call. An *in-framework* loop is called by the framework, which gets token IDs directly and can often control rollout execution. An *SDK-managed* agent issues model calls itself, but through an SDK or wrapper provided by the framework. A *service-level* agent issues its model calls through a provider configured before training and is otherwise opaque to the framework: it owns its loop, tools, memory, dependencies, and execution environment, and may be implemented in another language. TACO targets the third case. Our deployment assigns a dedicated service instance to each worker and each instance executes one rollout at a time, while multiple worker–service pairs may run concurrently. To recover rollout identity in this execution model, a model-routing field, such as the endpoint address, model name, or API key, is configured before startup and forwarded unchanged to the model provider. Under these conditions, a fixed worker identifier carried in that field can be mapped unambiguously to the worker's current rollout while preserving the agent's execution environment.

What one training step does. A step draws a batch of prompts and expands each into n agent *rollouts*. A rollout runs the agent to completion through a variable number of LLM calls interleaved with tool calls, and its rewards are computed outside the agent by a user-provided function or a remote evaluator. We call one turn of a rollout a *transition*: the prompt the agent sent, the response the model produced, possibly containing a tool call, and the tool output; the agent then folds the entire transition into the next prompt. A T-turn rollout therefore has T transitions, and the n rollouts of the same prompt form one GRPO group (Shao et al., 2024). In our outcome-reward setting, each rollout receives a scalar reward normalized across the group to produce its advantage. This advantage is shared by all transitions in the rollout. Process rewards, if available, can instead produce transition-specific advantages. The *response mask* marks where each advantage applies: 1 on model-generated tokens and 0 on prompt, tool output, and system content.

Packing is an optimization, not a precondition. Training directly on transitions is already sufficient for outcome-reward GRPO: group normalization produces one rollout-level advantage, which is shared by all transitions in the rollout and applied only to their response tokens. However, the cost is redundant computation. Unmerged, a T-turn rollout becomes T sequences whose prompts include the entire preceding history, causing earlier context to be repeatedly re-encoded and processed in the actor's forward and backward passes. When the recorded history is a token-exact prefix of the next prompt, packing can merge the transitions losslessly into one sequence with prompt P_1 and continuation $R_1{+}T_1{+}\ldots{+}R_T$, where R_i is the model response, possibly containing a tool call, and T_i is the tool output. Only the R_i tokens are masked with 1, so a single causal pass covers every generated token under its original causal context. Masked-out positions still supply keys and values to later positions that attend to them, so packing avoids repeated context computation. Section 3.2 extends this idea to histories with bounded token differences.

3 Design

Figure 1 shows how TACO turns externally executed rollouts into training batches. On the rollouter, an *external execution manager* uses an agent client to submit n rollouts per prompt to a Redis

task queue. Workers then dequeue the tasks and pass them to their associated black-box agent services, each configured with its own LLM endpoint. As a service executes a task, its model requests pass through the proxy, which recovers their rollout identity, routes them to inference replicas, and records token IDs and log probabilities. After execution, the worker writes the task's status, answer, and reward to Redis. Meanwhile, the agent client polls task statuses and retrieves the call records of completed rollouts. Once all n rollouts of a prompt finish, the external execution manager packs their traces, assembles the training records into a *rollout group*, and places that group in a separate training-sample queue. In parallel, the trainer starts an actor update after collecting B rollout groups from the queue. Rollout generation continues while the trainer updates the model, with periodic weight synchronization making the new weights available to the inference replicas.

3.1 Tracking rollout identity across turns for black-box agent services

Recovering identity through a fixed configuration field. The framework cannot tag LLM calls it does not issue, so it must recover identity from information the agent forwards unchanged. Conversation history is fragile because an agent may summarise, compact, or trim it, and request metadata may also be lost when the agent rebuilds model requests. TACO instead uses a configuration field forwarded unchanged to the model provider: an endpoint address, model name, API key, or a dedicated field added for this purpose (§2). For a long-running agent service, these fields may remain fixed for its lifetime, making them unsuitable for carrying a task-varying rollout identifier. TACO therefore assigns each worker a stable *worker identifier* w, places it in one such field, and recovers rollout identity through a separate worker-to-rollout mapping stored in Redis.

Mapping a worker to its current rollout. Each worker is associated with one agent service instance, which may run as a separate process or container. The service is configured once with a field carrying w and executes one rollout at a time. This serialization is local to each worker–service pair, while multiple pairs may run concurrently. Before passing a task to the service, the worker writes $w \mapsto r$ to Redis, where r is the task's rollout identifier. Every request carrying w therefore maps unambiguously to the worker's current rollout. For example, an endpoint path configured as `http://proxy/worker_id/`w`/v1/chat/completions` carries the same w for the service's lifetime. The proxy extracts w, looks up r, strips the identifier, and forwards the request to an inference replica while recording its prompt tokens, response tokens, and response log probabilities under r. Calls sharing the same r are then reassembled into one rollout trace for training.

However, if a service is shared across workers, the worker identifier alone is no longer sufficient. TACO instead places a rollout identifier r in a routing field when the worker–service connection allows that field to vary by rollout. Otherwise, TACO falls back to placing r in the initial prompt and relying on the agent to replay it across turns. This fallback requires faithful history replay and may fail if the agent rewrites or trims its context (Appendix D).

Routing one rollout to the same inference replica. The proxy also maintains an LRU mapping from r to an inference replica, routing later calls of the same rollout to that replica. Because successive turns often share a long prefix, this increases the chance of a prefix-cache hit. Thus, the rollout identifier r supports both trace collection and replica affinity.

3.2 Drift-tolerant trajectory packing

When the history already recorded from earlier turns is a token-exact prefix of the next prompt, packing is lossless: it concatenates responses and new tool output, applies loss only to model tokens, and preserves their causal contexts (Figure 2(a)). Re-tokenization and small history edits often break this condition even when turns retain nearly the same context. TACO therefore allows a bounded token-level edit distance τ between the recorded history and its replay in the next prompt.

The packer keeps an *accumulated sequence*, initialized to the first turn's prompt and response. For a T-turn rollout, the packer attempts to merge each later turn into this sequence, making $T - 1$ merge attempts in total. At each attempt, it takes the last a tokens of the accumulated sequence as an *anchor* and searches for this anchor within a local window of size h around the expected end of the replayed history in the next prompt. The rightmost match gives a candidate prompt prefix, which is compared with the accumulated sequence using token-level edit distance computed

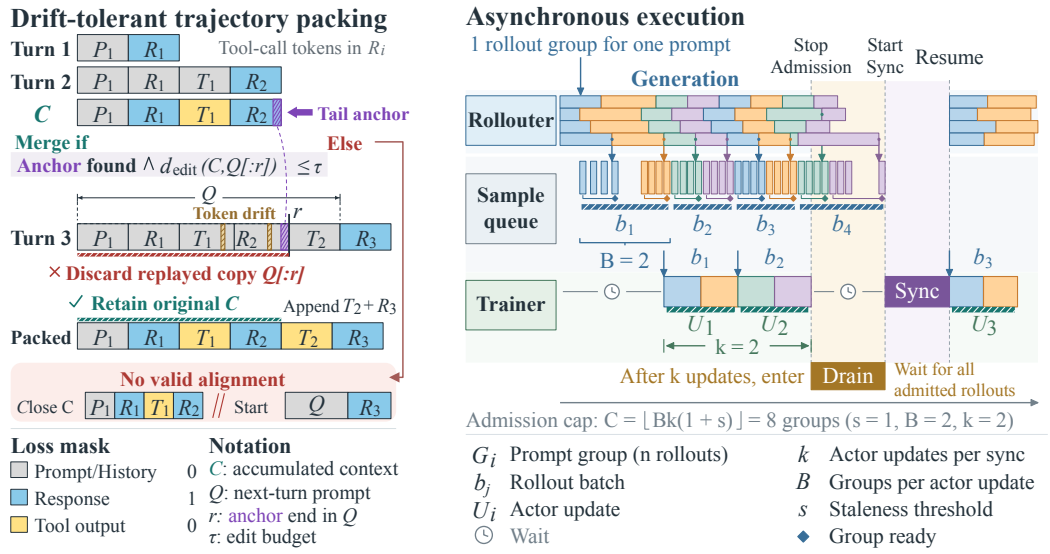

Figure 2: TACO mechanisms: (a) drift-tolerant packing; (b) asynchronous execution.

by a standard banded dynamic program (Levenshtein, 1966; Ukkonen, 1985). The search window and edit-distance tolerance handle different forms of drift: h limits how far the replayed-history boundary may shift, while τ limits token differences within the candidate prefix. A merge therefore requires both an anchor within the search window and an edit distance within τ.

If the edit distance is within τ, the packer keeps the accumulated history and appends the remaining prompt suffix, including new tool output (mask 0) and the next response (mask 1). Earlier responses also keep their original rollout log probabilities for asynchronous training (Figure 2(a)). Otherwise, the current accumulated sequence ends, and a new one is initialized to the next turn's prompt and response under the same rollout identifier. Thus, large rewrites or summarization form new segments, while small differences under τ can be merged; τ=0 corresponds to exact-prefix matching. Packed sequences that exceed the model's length limit are truncated, and Appendix B gives the full pseudocode.

Packing changes optimization granularity: fewer training sequences at a fixed mini-batch size give fewer optimizer updates, each covering more response tokens. Section 4.3 evaluates the effects of edit-distance tolerance and fewer optimizer updates on training accuracy and efficiency. Appendix C discusses the context approximation introduced by nonzero tolerance.

3.3 Asynchronous Execution with Opaque Agents

Admission, draining, and weight synchronization. The scheduling unit is a *rollout group* for one prompt (Figure 2(b)), whose n tasks are admitted together. Packing within each rollout may change the number of training sequences in a group, but not the group count. The trainer consumes B groups per actor update and synchronizes every k actor updates, thus consuming Bk groups between two weight synchronizations. With staleness threshold s, the admission cap is $C = \lfloor Bk(1+s) \rfloor$.

After each synchronization, the counter starts at q, the number of queued groups generated under earlier parameter versions, and increases by one per newly admitted group. Once the counter reaches C, the external execution manager stops admissions and waits for submitted tasks to finish. Therefore, at most $C - q$ new groups can be admitted before submissions pause.

Because an opaque agent may not expose a checkpoint for its loop, tools, memory, or side effects, TACO drains already-submitted tasks before synchronizing weights. To bound drain latency and waiting time, each task has a configured timeout. If a task times out, TACO marks it as failed and invalidates its worker–rollout mapping, so the proxy rejects any late model requests. A retry uses a fresh rollout identifier and a new worker connected to a backup service instance; reusing the original worker identifier requires stopping the old service execution first. The failed task's partial trace and late results are excluded from training. Once draining completes and the resulting groups enter

the training-sample queue, TACO clears the inference KV cache, transfers the updated weights, increments the parameter version, and resumes submissions (Appendix E). This ensures that all model calls used to form a rollout are served by the same model weights. Stale groups already in the training-sample queue remain queued during synchronization, and each records the parameter version used for generation, so the trainer can determine its staleness when consumed. Between weight synchronizations, rollout generation overlaps with training.

An actor update may contain multiple optimizer updates. Following VeRL's convention, the PPO mini-batch size m is specified at the rollout-group level; with n rollouts per group, our trainer uses mn training sequences per optimizer update. It collects sequences from the B rollout groups and truncates their count to a multiple of mn before distributing them into mini-batches. Packing reduces the sequence count without changing B, so reducing m can increase the number of optimizer updates to better align with unpacked training (§4.3).

3.4 Implementation

Our implementation builds on VeRL's trainer and inference replicas (Sheng et al., 2025; Zhao et al., 2023; Kwon et al., 2023), which we retain largely unchanged. Our changes are concentrated on the rollout side: a Redis backend for tasks and results, an LLM proxy, an agent server, an external execution manager, and a packing stage. The external execution manager converts collected rollouts into VeRL's `DataProto`, and rewards are computed by a user function or returned from a remote judge. Ray schedules the framework components across hosts (Moritz et al., 2018), while agent services may run externally.

4 Evaluation

We evaluate TACO along three dimensions: integration with black-box agents, the efficiency and accuracy of trajectory packing, and the additional benefit of asynchronous execution.

4.1 Setup

The main experiments use 16 NVIDIA H20 GPUs on two hosts. Asynchronous execution uses one 8-GPU host for rollout generation and the other for training; synchronous execution uses all 16 GPUs for each phase in turn. We initialize from a Qwen2.5-7B cold-start checkpoint produced with ReTool's recipe (Qwen Team, 2025; Feng et al., 2025), train on DAPO-Math-17k (Yu et al., 2025), and evaluate on AIME-2024 (Mathematical Association of America, 2024). Each training step consumes B=80 rollout groups with GRPO n=8. The 2×2 comparison varies packing and execution mode, using τ=128 and PPO mini-batch size m=72. The edit-distance budget comparison changes only τ to 0 or 20; the optimizer ablation changes only m to 24 in asynchronous packed training. All runs use token-mean loss without a KL term. Asynchronous runs synchronize weights every actor update (k=1), with staleness threshold s=0.5 (§3.3).

Main timing comparisons use 200 observed training steps; Appendix F reports repetitions at their observed endpoints. We evaluate AIME mean@8 at steps 0, 20, …, 200 and compare learning curves and final accuracy at step 200. Appendices A and E provide configurations and timing definitions.

We integrate three black-box agents. The **ReTool-style math agent** uses the model and datasets above, writes and executes Python in a sandbox, and is used for all packing and execution-mode comparisons. The **TritonForge-style kernel agent** (Li et al., 2025a) uses an AutoTriton checkpoint (Li et al., 2025b) as its base model. It trains on 50 KernelBench problems (Ouyang et al., 2025) and validates on 18 held-out problems. For each task, the agent produces one Triton kernel (Tillet et al., 2019), which is scored by an external evaluation service. **ZeroClaw** (ZeroClaw Labs, 2026) is a Rust service with its own model provider, tools, memory, and gateway. It uses Qwen2.5-7B-Instruct and trains on Calc-X (Kadlčík et al., 2023), with evaluation on 20 held-out test tasks.

4.2 Training black-box agents

All three integrations require no agent source changes and 122–334 lines of adapter code (Table 1). VeRL implements the Python tool within its own loop; TACO reuses the existing math agent and its

Table 1: Agent source changes and adapter LOC, including reward logic.

Agent	Agent ΔLOC	Adapter LOC	VeRL integration
ReTool-style math agent	0	122	built-in loop + tool adapter
TritonForge-style kernel agent	0	270	not implemented
ZeroClaw (Rust service)	0	334	requires an external bridge

tools. The adapter returns answers and rewards without reproducing the agent's loop or environment (Appendix G).

ZeroClaw's Rust runtime owns its loop and issues model calls through the proxy; most of its 334 adapter lines handle HTTP/WebSocket communication. The adapter for the kernel agent obtains rewards from an external evaluator that compiles and checks kernels while preserving the agent's generation logic.

All three integrations complete end-to-end RL training. With m=24, training improves the math agent's AIME mean@8 from 0.03 to 0.33 over 200 steps. The kernel agent's validation reward increases from 0.338 to 0.704 over 300 steps. ZeroClaw's reward rises from 0.50 to 0.80 over 70 training steps, peaking at 0.85 at step 20. Appendix G.2 plots the two additional validation curves.

4.3 Drift-tolerant trajectory packing

Exact matching leaves almost all turns unmerged. For each rollout group, the *compression ratio* is the number of transitions divided by the number of training sequences after packing. Averaged over all rollout groups produced during the 200 training steps, the mean compression ratios at τ=0, 20, and 128 are 1.001, 2.564, and 2.828, respectively; mean rejection rates are 99.84%, 1.39%, and 0.78% (Figure 3; aggregation in Appendix E). At τ=0, exact-prefix matching rejects 215,383 merge attempts. Among these rejections, 99.58% have an anchor but a nonidentical token prefix, 0.42% have no anchor in the search window, and one comes from a prompt that is too short.

The math agent replays history without trimming or summarization, yet exact matching almost always fails. Even when the replayed history's text is unchanged, appending responses and new tool outputs can still alter its token IDs when the next prompt is re-tokenized. Drift-tolerant packing accepts these small differences and avoids repeated history computation. Compared with exact matching, packing with τ=128 reduces the number of training sequences from about 1720 to 650 per step and cuts step time from 272 s to 155 s, a 1.75× speedup.

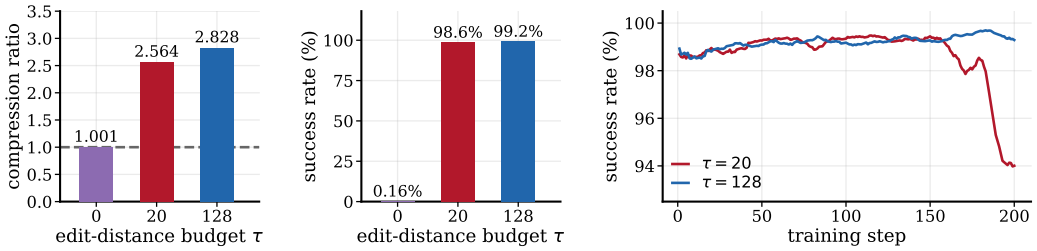

Figure 3: Edit-distance budgets. Left: mean compression ratio. Middle: mean merge success rate. Right: merge success rate over training, smoothed over 10 steps (τ=0 is below the plotted range).

We use τ=128 in the main experiments because its merge success remains high over all 200 training steps, whereas τ=20 degrades near the end. Up to step 150, rejection rates for τ=20 and 128 are 0.83% and 0.86%; after step 185 they become 5.89% and 0.66%. At τ=20, accepted merges increasingly approach the edit-distance limit, suggesting that the smaller tolerance becomes insufficient as context drift grows. However, a larger tolerance still requires the anchor to fall within the search window; otherwise, packing starts a new training sequence (Appendix F).

Packing reduces actor-update cost. In asynchronous training, packing reduces processed tokens from 4.21M to 1.41M per step and actor-update time from 257 s to 87 s, a $\mathbf{2.97}\times$ speedup (Table 2). Because only 8 GPUs are used for training, the packed batch still maintains sufficient per-GPU workload, keeping token throughput near 2.0k tokens/s with and without packing. The $2.98\times$ reduction in processed tokens therefore closely matches the reduction in actor-update time.

In synchronous training, processed tokens fall from 4.78M to 1.83M per step and actor-update time falls from 137 s to 63 s, a $\mathbf{2.16}\times$ speedup. Here, all 16 GPUs are used for training, so the smaller packed batch lowers the per-GPU workload enough to reduce throughput from 2.19k tokens/s without packing to 1.81k with packing. As a result, actor-update time improves by $2.16\times$, less than the $2.61\times$ reduction in processed tokens. Reported speedups are computed from unrounded mean timings (Appendix E).

Adjusting mini-batch size recovers accuracy. Figure 4 shows that, at the same PPO mini-batch size, the unpacked learning curves are generally above the packed curves in both synchronous and asynchronous training. Packing also reduces the number of optimizer updates within each actor update, which may contribute to this gap. Following §3.3, with m=72 and n=8, each optimizer mini-batch contains mn=576 training sequences. Asynchronous packed training produces about 650 sequences per step, which are truncated to 576 and therefore give one optimizer update. Unpacked training produces about 1890 sequences and averages 2.80 updates. The same pattern appears in synchronous training: packed training performs one optimizer update, while unpacked training averages 2.75 (Appendix E.1).

To test this explanation, we reduce m to 24 for asynchronous packed training while keeping the packing algorithm and other parameters unchanged. This run produces 640–656 packed sequences per step; truncation again leaves 576. With mn=24 $\times$ 8=192 training sequences per mini-batch, 576 retained sequences correspond to three optimizer updates, comparable to the 2.80 updates in unpacked training. Final AIME mean@8 at step 200 rises from 0.306 to $\mathbf{0.331}$, nearly matching the unpacked result of 0.335.

The accuracy recovery suggests that fewer optimizer updates contribute to the gap between packed and unpacked training. However, packing also places more response tokens in each sequence, so reducing the mini-batch size changes both the optimizer update count and the number of response tokens per update; this ablation measures their joint effect. End-to-end training takes 9.94 h at m=24 versus 8.72 h at m=72, still $\mathbf{2.21}\times$ faster than synchronous unpacked training at 21.94 h. Appendix F reports the unpacked repetitions.

4.4 Asynchronous execution and composition

End-to-end speedup and pipeline balance. Asynchronous execution overlaps rollout generation with training. With packing, it reduces wall-clock time from 13.92 h to 8.72 h, a $\mathbf{1.60}\times$ speedup (Table 2). Without packing, wall-clock time falls from 21.94 h to 19.18 h, only a $\mathbf{1.14}\times$ speedup; together, packing and asynchrony give a $\mathbf{2.51}\times$ speedup over the synchronous unpacked baseline under the same total GPU budget. Because asynchronous execution uses half the GPUs for rollout generation, its benefit depends on hiding enough training time through overlap.

Packing changes the pipeline balance. With packing, the actor update takes 87 s; the trainer waits for samples during 41.7% of step time, while the rollouter is idle for only 13.6%. Rollout generation

Table 2: Efficiency on 16 H20 GPUs for a 200-step training budget. Per-step times are means in seconds; Upd/step counts optimizer updates (Appendix E).

Configuration	Wall	Step	Queue wait	Gen.	Actor upd.	Tok/step	Upd/step
async + packing	**8.72 h**	155	67	–	87	**1.41M**	1.00
async + unpacking	19.18 h	345	85	–	257	4.21M	2.80
sync + packing	13.92 h	244	–	149	63	1.83M	1.00
sync + unpacking	21.94 h	386	–	170	137	4.78M	2.75
async + packing, m=24	9.94 h	178	78	–	98	1.56M	**3.00**

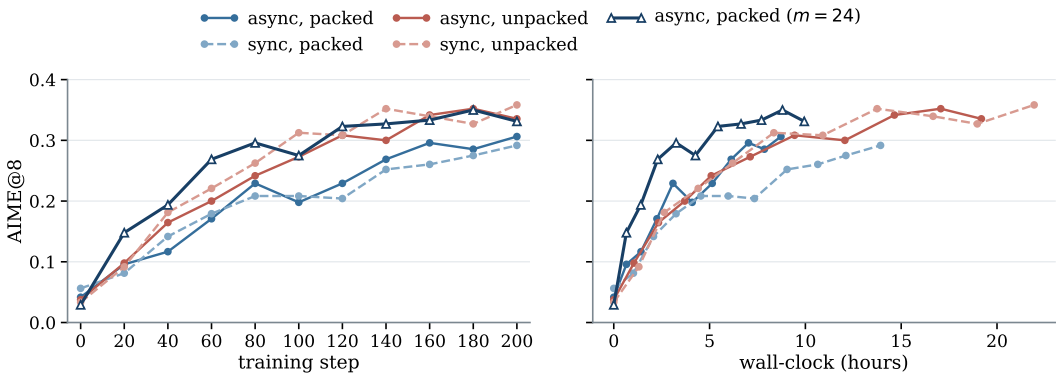

Figure 4: AIME mean@8 at steps 0–200, every 20 steps. Blue/red: packing/unpacking; solid/dashed lines: async/sync; circles/triangles: m=72/m=24. Wall-clock timing follows Appendix E.

therefore limits throughput, and most training can overlap with it. Without packing, the actor update grows to 257 s. Trainer wait falls to 23.9%, while rollouter idle rises to 49.8% because generation reaches the admission cap and waits for the trainer to consume queued samples. Training thus becomes the bottleneck and cannot be fully hidden behind rollout. Packing shortens training enough that most of it can overlap with rollout, increasing the end-to-end benefit of asynchronous execution.

The drain-and-synchronization time shows the same pipeline behavior. Across 200 asynchronous training steps, this phase averages 1.72 s with packing (95th percentile 11.62 s; maximum 40.38 s), but only 0.035 s without packing (95th percentile 0.064 s; maximum 0.119 s). Most of the difference comes from drain waiting: the pause/drain phase averages about 1.70 s and 0.011 s, respectively. With packing, rollout generation is the bottleneck, so long-running tasks may still be active when the trainer is ready for synchronization. Without packing, the longer actor update gives those tasks more time to finish, so almost no extra waiting is needed before synchronizing weights (Appendix E).

5 Related Work

RL frameworks and external agents. VeRL (Sheng et al., 2025) supports distributed RLHF; ROLL (Wang et al., 2025a) coordinates rollout, environment, and reward workers. rLLM (Tan et al., 2025) abstracts agents and environments, while SkyRL (Griggs et al., 2025) separates training, generation, and environment execution. Agent Lightning (Luo et al., 2025) decouples agent execution from training. TACO instead focuses on rollout identification through persistent service configurations and reconstruction of service-issued calls.

Trajectory reconstruction and packing. Agent Lightning's 2025 aggregation uses exact token-prefix matching and reports failures from re-tokenization and history changes (Agent Lightning Team, 2025). TACO locates boundaries with token anchors and tolerates bounded token-level edits in the preceding history. We compare it with exact prefix matching and study its interaction with optimizer update counts (§4.3).

Asynchronous RL. Generation–training overlap and off-policy learning are established (Mnih et al., 2016; Espeholt et al., 2018; Hu et al., 2024). AReaL (Fu et al., 2025) decouples generation from training, controls staleness, and supports interruptible generation across updates. TACO drains external agent tasks before synchronization, preserving one parameter version per rollout.

6 Conclusion

TACO trains black-box agents in their existing runtimes without source changes. Rollout identification enables trace collection, while drift-tolerant packing and asynchronous execution give a 2.51× speedup on the math agent. Accuracy recovery with smaller mini-batches shows how packing interacts with optimization granularity. Together, these results show that service-level agents can be trained efficiently without moving their execution into the RL runtime.

AI USE STATEMENT

We used generative AI tools to assist with manuscript outlining, drafting and revision, literature search, checking methodological descriptions against source code, interpreting experimental logs, and preparing or revising figures and plotting scripts. The numerical results reported in this paper come from our experimental runs. We reviewed AI-assisted text and figures, checked cited work against the original sources, and checked quantitative claims against code and experiment records. We take responsibility for the final text, methods, results, and artifacts.

ETHICS STATEMENT

TACO studies reinforcement learning infrastructure for existing agent services. Our evaluation focuses on task performance and computational efficiency. Recording agent trajectories can expose sensitive information in prompts and tool outputs, and optimizing task rewards does not by itself ensure safe agent behavior. Deployments should therefore use authorized data, protect collected trajectories, and restrict tool permissions and code execution to appropriate environments. Safe deployment in high-stakes settings requires further evaluation beyond the performance measurements reported here.

REPRODUCIBILITY STATEMENT

To support reproducibility, we describe our methods, experimental setup, and evaluation protocol in the main text and appendix. We report the model initialization, training configurations, and computational resources, and provide algorithmic and implementation details for trajectory packing and agent integration. We also specify batch assembly procedures and the measurement and aggregation of timing and learning metrics to support replication of the reported comparisons under the same experimental conditions. We submit our code with sensitive information removed as supplementary material.

REFERENCES

Agent Lightning Team. Adopting the trajectory level aggregation for faster training. `https://agent-lightning.github.io/posts/trajectory_level_aggregation/`, 2025. December 17.

Paul F. Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018.

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. *arXiv preprint arXiv:2504.11536*, 2025.

Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. AReaL: A large-scale asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025.

Tyler Griggs, Sumanth Hegde, Eric Tang, Shu Liu, Shiyi Cao, Dacheng Li, Charlie Ruan, Philipp Moritz, Kourosh Hakhamaneshi, Richard Liaw, Akshay Malik, Shishir G. Patil, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Evolving SkyRL into a highly-modular RL framework. `https://novasky-ai.notion.site/skyrl-v01`, 2025. June 26.

Horace He and Thinking Machines Lab. Defeating nondeterminism in LLM inference. *Thinking Machines Lab: Connectionism*, 2025. doi: 10.64434/tml.20250910. https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Zilin Zhu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, Weikai Fang, Xianyu, Yu Cao, Haotian Xu, and Yiming Liu. OpenRLHF: An easy-to-use, scalable and high-performance RLHF framework. *arXiv preprint arXiv:2405.11143*, 2024.

Marek Kadlčík, Michal Štefánik, Ondrej Sotolar, and Vlastimil Martinek. Calc-X and calcformers: Empowering arithmetical chain-of-thought through interaction with symbolic systems. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 12101–12108. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.742. URL https://aclanthology.org/2023.emnlp-main.742/.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. 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 (SOSP)*, 2023.

Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

Haonan Li, Keyu Man, Partha Kanuparthy, Hanning Chen, Wei Sun, Sreen Tallam, Chenguang Zhu, Kevin Zhu, and Zhiyun Qian. TritonForge: Profiling-guided framework for automated Triton kernel optimization. *arXiv preprint arXiv:2512.09196*, 2025a.

Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, and Maosong Sun. AutoTriton: Automatic Triton programming with reinforcement learning in LLMs. *arXiv preprint arXiv:2507.05687*, 2025b. URL https://arxiv.org/abs/2507.05687.

Yingru Li. Rollout correction. VeRL documentation, https://verl.readthedocs.io/en/latest/algo/rollout_corr.html, 2025.

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.

Mathematical Association of America. AIME 2024. https://maa.org/maa-invitational-competitions/, 2024. American Invitational Mathematics Examination. Dataset mirror maintained by Maxwell-Jia: https://huggingface.co/datasets/Maxwell-Jia/AIME_2024.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, 2016.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018. https://github.com/ray-project/ray.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Qwen Team. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115v2*, 2025.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

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

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys)*, 2025.

Sijun Tan, Michael Luo, Colin Cai, Tarun Venkat, Kyle Montgomery, Aaron Hao, Tianhao Wu, Arnav Balyan, Manan Roongta, Chenguang Wang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. rLLM: A framework for post-training language agents. Official project announcement, 2025.

Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, 2019.

Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1–3): 100–118, 1985.

Weixun Wang, Shaopan Xiong, Gengru Chen, Wei Gao, Sheng Guo, Yancheng He, Ju Huang, Jiaheng Liu, Zhendong Li, Xiaoyang Li, Zichen Liu, Haizhou Zhao, Dakai An, Lunxi Cao, Qiyang Cao, Wanxi Deng, Feilei Du, Yiliang Gu, Jiahe Li, Xiang Li, Mingjie Liu, Yijia Luo, Zihe Liu, Yadao Wang, Pei Wang, Tianyuan Wu, Yanan Wu, Yuheng Zhao, Shuaibing Zhao, Jin Yang, Siran Yang, Yingshui Tan, Huimin Yi, Yuchi Xu, Yujin Yuan, Xingyao Zhang, Lin Qu, Wenbo Su, Wei Wang, Jiamang Wang, and Bo Zheng. Reinforcement learning optimization for large-scale learning: An efficient and user-friendly scaling library. *arXiv preprint arXiv:2506.06122*, 2025a.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025b.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

ZeroClaw Labs. ZeroClaw: Autonomous AI assistant infrastructure. `https://github.com/zeroclaw-labs/zeroclaw`, 2026.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch FSDP: Experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

A EXPERIMENTAL CONFIGURATION

Table 3 supplements the setup in §4.1. The policy-ratio clipping interval $[0.8, 1.28]$, learning rate 10^{-6}, and disabled KL reward and loss terms follow VeRL's ReTool recipe.

Table 3: Additional math-agent configuration shared across the main comparisons.

Learning rate / PPO epochs	10^{-6} / 1
Per-turn prompt / response limits	16K / 4K tokens
Packed prompt / continuation limits	16K / 16K tokens
Matcher	τ=128; anchor 10 tokens; search range 32 tokens
Entropy coefficient	0
Policy ratio clipping	asymmetric: ϵ_{low}=0.2, ϵ_{high}=0.28; $[0.8, 1.28]$
Offload	off for the async trainer; on when colocated
Seeds / checkpointing	data and trainer seed 42 / disabled
Workers / async client timeout	128 / 300 s
Inference	vLLM 0.12; prefix caching; GPU memory utilization 0.8

The kernel agent uses 10 prompts per batch, n=12, a configured PPO mini-batch size 30, and prompt/response limits of 8K/16K tokens. ZeroClaw uses the Calc-X data packaged in Agent Lightning's example, with 32 prompts per batch, n=8, and a configured PPO mini-batch size 64.

B PACKING ALGORITHM AND ALIGNMENT COST

For call i in a rollout of T calls, let P_i, R_i, and L_i denote its prompt tokens, response tokens, and response log probabilities. Model-generated tool-call tokens belong to R_i; tool output appears in a later prompt. Each output segment contains a prompt P, a continuation Y, a loss mask μ, and log probabilities L aligned with Y. The accumulated tokens are $S = P + Y$. The mask selects model-generated tokens only; zeros inserted into L at masked positions are placeholders.

Below, $+$ denotes concatenation, $|\cdot|$ denotes token count, and slices use zero-based, end-exclusive indices. For any nonnegative integer j, 0^j and 1^j are constant vectors of length j. The edit-distance budget is τ; the anchor-search tolerance is $h = 32$ tokens.

1. Initialize $P = P_1$, $Y = R_1$, $\mu = 1^{|R_1|}$, $L = L_1$, and $S = P + Y$.
2. For each call $i = 2, \ldots, T$, let $Q = P_i$ and use the last $a = \min(10, |S|)$ tokens of S as the anchor. Search for its rightmost exact occurrence in Q between indices $\max(0, |S| - a - h)$ and $\min(|Q|, |S| + a + h)$. Let b be the end index of that occurrence, so $Q[:b]$ is the candidate prefix.
3. Reject the merge if $|Q| < |S| - \tau$, no anchor is found, or the token-level Levenshtein distance between S and $Q[:b]$ exceeds τ.
4. On acceptance, let $D = Q[b:]$ be the new prompt suffix and append $Y \leftarrow Y + D + R_i$, $\mu \leftarrow \mu + 0^{|D|} + 1^{|R_i|}$, and $L \leftarrow L + 0^{|D|} + L_i$. Update $S = P + Y$.
5. On rejection, emit (P, Y, μ, L) and initialize a new segment from call i with $P = Q$, $Y = R_i$, $\mu = 1^{|R_i|}$, $L = L_i$, and $S = P + Y$. Emit the final segment after the loop.

The edit-distance budget applies independently to each merge attempt, rather than being accumulated across a rollout. If packing produces multiple training segments, each segment keeps the rollout's identifier and reward before the usual length filtering and truncation.

Cost and diagnostics. For each candidate prefix, the banded edit-distance calculation evaluates at most $2\tau + 1$ cells per token position, requiring $O((\tau + 1)\max(|S|, b))$ work. The computation terminates early once no alignment within τ remains possible. The search window h limits where the anchor can be found, whereas τ limits token edits within the matched prefix; increasing one does not remove the other constraint. We record three rejection causes: a prompt that is too short, a missing anchor, and an edit distance above τ.

C PACKING AND POLICY IMPORTANCE RATIOS

An old log probability depends on the model snapshot, token context, and execution engine. For response token $a = R_{i,j}$ in call i, let P_i denote the recorded prompt tokens and $R_{i,<j}$ the preceding response tokens. Its original context is $H = (P_i, R_{i,<j})$. Let $\widetilde{P}_i$ be the prefix before this response in the packed sequence, giving packed context $\widetilde{H} = (\widetilde{P}_i, R_{i,<j})$. We hold filtering and truncation fixed in this comparison. Write π^{T} and π^{I} for probabilities evaluated by the training and inference engines. Let θ_v be the parameters used to generate the rollout, θ_p a frozen trainer snapshot used for the PPO denominator, and θ the parameters being optimized. The trainer snapshot is taken before an actor update or at the start of a synchronization interval. It can differ from θ_v when queued rollouts were generated by an earlier policy version. Separately, even identical weights can produce different probabilities across engines because of kernels, precision, or batching.

Unpacked reference. Without packing, each response is trained under its original prompt. The superscripts *roll* and *train* indicate whether the denominator uses recorded rollout probabilities or probabilities recomputed by the trainer. The corresponding PPO ratios (Schulman et al., 2017) are

$$\rho^{\mathrm{unpack,roll}}(a) = \frac{\pi^{\mathrm{T}}_{\theta}(a \mid H)}{\pi^{\mathrm{I}}_{\theta_v}(a \mid H)}, \qquad \rho^{\mathrm{unpack,train}}(a) = \frac{\pi^{\mathrm{T}}_{\theta}(a \mid H)}{\pi^{\mathrm{T}}_{\theta_p}(a \mid H)}. \tag{1}$$

Both sides use H, although the rollout-based ratio can still be affected by parameter staleness and numerical differences between engines.

Packing with recorded rollout log probabilities. The proxy records response log probabilities during generation, and packing preserves them. The denominator therefore retains the original context:

$$\rho^{\mathrm{pack,roll}}(a) = \frac{\pi^{\mathrm{T}}_{\theta}(a \mid \widetilde{H})}{\pi^{\mathrm{I}}_{\theta_v}(a \mid H)}. \tag{2}$$

Compared with the unpacked ratio, packing changes the numerator's context. When $\widetilde{H} \neq H$, this introduces a context mismatch in addition to any staleness or training–inference discrepancy.

Packing with trainer-recomputed old log probabilities. After packing, the trainer can evaluate the denominator on the same training sequences using θ_p and hold these probabilities fixed during the actor update:

$$\rho^{\mathrm{pack,train}}(a) = \frac{\pi^{\mathrm{T}}_{\theta}(a \mid \widetilde{H})}{\pi^{\mathrm{T}}_{\theta_p}(a \mid \widetilde{H})}. \tag{3}$$

The forward pass reads the full packed context; policy loss is applied to every response position with mask 1, including responses from earlier merged turns. Tool outputs and other mask-0 positions remain in the context but do not contribute directly to the policy loss. Both sides now use the same context and engine, so the ratio is 1 at $\theta = \theta_p$, aside from numerical error. This removes context and engine mismatch within the ratio, but does not make packed and unpacked training identical: the response and reward were obtained under H, and changing H to $\widetilde{H}$ can still change the policy gradient.

Denominator choices in TACO. Through its VeRL-based asynchronous trainer, TACO exposes `use_rollout_log_probs` and `compute_prox_log_prob`. Enabling only the former uses recorded rollout probabilities directly. Enabling both retains those records and recomputes the PPO denominator with a trainer snapshot frozen at the first actor update after synchronization. The trainer saves this snapshot on CPU, temporarily restores it for later denominator computations in the interval, and restores the current parameters before optimization. The next interval refreshes the snapshot. This gives one proximal policy per synchronization interval; it does not recover each queued rollout's generation policy. With k=1, the snapshot is the policy immediately before each actor update. Disabling `use_rollout_log_probs` instead recomputes probabilities with the current pre-update parameters. Thus, asynchronous training can also use the same context and engine on both sides of Equation 3.

In §4.3, reducing the PPO mini-batch size brings the optimizer update count closer to that of unpacked training and recovers comparable final accuracy, while keeping the packing algorithm and denominator source unchanged. This suggests that fewer optimizer updates explain much of the accuracy gap, although context mismatch may also contribute. Because changing the mini-batch size also changes the number of response tokens per optimizer update, these effects are not fully isolated. A more controlled comparison could separate these effects and independently vary the denominator computation.

Optional rollout correction. Like VeRL, TACO also supports correction weights between recorded rollout probabilities and trainer-recomputed probabilities (Li, 2025). For packed data, the unmodified token-level weight is

$$w(a) = \frac{\pi^{\mathrm{T}}_{\theta_p}(a \mid \widetilde{H})}{\pi^{\mathrm{I}}_{\theta_v}(a \mid H)}, \qquad w(a)\rho^{\mathrm{pack,train}}(a) = \rho^{\mathrm{pack,roll}}(a). \tag{4}$$

Here $w(a)$ corrects for the difference between the generating policy and the frozen trainer policy, including differences in context, inference engine, and policy version. In contrast, $\rho^{\mathrm{pack,train}}$ measures the change from the frozen trainer policy to the current policy under the same packed context and engine. Their product recovers $\rho^{\mathrm{pack,roll}}$ before PPO clipping or truncation of the correction weights. After clipping or truncation, the resulting objective may differ from directly clipping $\rho^{\mathrm{pack,roll}}$. Our asynchronous experiments do not use this additional importance-weight correction.

At τ=0, accepted merges preserve the original token context, so packed and unpacked ratios differ only through numerical variation and later truncation. For $\tau > 0$, the edit-distance budget bounds token changes but does not bound their effect on policy probabilities or gradients. Trainer recomputation and batch-invariant kernels (He & Thinking Machines Lab, 2025) can reduce numerical mismatch, but cannot remove the context difference introduced by tolerant packing.

D IDENTIFIER CARRIERS

Worker-scoped carriers. The proxy accepts three fixed configuration fields, in the following order:

1. **Endpoint path:** `http://proxy/worker_id/`w`/v1`. The client appends `/chat/completions`; the proxy removes `/worker_id/`w before forwarding. The agent must preserve the base path.
2. **Model name:** `<name>#`w. The proxy splits at the last separator and restores the original model name.
3. **API key:** `<key>#`w. The proxy restores the original Authorization header before forwarding.

All three use the worker-to-rollout mapping in §3.1. Each worker has exclusive use of its service instance, executes tasks and their LLM calls sequentially, and publishes the mapping before passing a task to the service. The proxy accepts a worker identifier with a live mapping. Reusing that identifier after a timeout requires the previous service execution to have stopped or been isolated.

Rollout-scoped paths. For agents with a configurable endpoint per rollout, the path `/request_id/`r`/v1` carries the rollout identifier directly. The proxy removes this path component and records every call under r. If multiple workers share one service, every model request must carry the identifier of the rollout it belongs to, so the proxy can distinguish concurrent tasks.

Prompt-marker fallback. The fallback prefixes the initial user task with a request identifier enclosed in configurable delimiters; the proxy removes the marker before inference. Recovery requires faithful replay of the marker in the message inspected by the proxy. Adapter details appear in Appendix G.

E Optimizer update counts and timing definitions

E.1 Optimizer updates per actor update

The fractional counts in Table 2 are averages over actor updates. With one PPO epoch, N valid training sequences after filtering, and mn sequences per optimizer mini-batch, an actor update performs $\lfloor N/(mn) \rfloor$ optimizer updates after truncating the sequence count to a multiple of mn; all measured main-run batches have $N \geq mn$. Here m is the configured PPO mini-batch size and n is the number of rollouts per prompt, as in §3.3.

At m=72 and n=8, each optimizer update uses 576 sequences. Across the primary asynchronous unpacked run's 200 actor updates, 40 perform two optimizer updates and 160 perform three, giving an average of $(40 \times 2 + 160 \times 3)/200 = 2.80$ optimizer updates per actor update. The synchronous unpacked run has 59, 133, and 8 actor updates with two, three, and four optimizer updates, respectively, giving an average of $(59 \times 2 + 133 \times 3 + 8 \times 4)/200 = 2.745$, rounded to 2.75.

E.2 Timers and aggregation

Actor-update time measures policy forward/backward passes and optimizer work. **Queue wait** measures asynchronous batch collection from the training-sample queue; **generation** measures the synchronous rollout phase. **Step time** includes batch collection and training. Asynchronous **synchronization time** is logged separately and includes draining outstanding tasks and transferring weights.

Wall-clock time runs from the recorded run start to the completion of the model update at training step 200, before that checkpoint's validation. It includes all work before this endpoint, such as earlier validation and synchronization. Figure 4 uses each checkpoint's training-completion time, with the initial model at time zero.

Trainer idle ratio is queue-wait time divided by step time, computed per step and then averaged. Rollouter idle ratio is the idle portion of each parameter-version interval, then averaged across intervals. Token throughput divides total processed tokens by total actor-update time and the number of training GPUs. Speedups use unrounded timings.

Compression is the number of recorded calls divided by the resulting training-sequence count; rejection rate is the fraction of merge attempts rejected. These ratios are computed per rollout group, combined within each step using rollout-count weights, and averaged over steps. Count metrics are summed within each step. Unless stated otherwise, reported means use training steps 1–200; the merge-success curve uses a 10-step moving average.

F Edit-distance budget comparison and repeat variation

Comparing edit-distance budgets. All three settings in Figure 3 use m=72, the same 8:8 GPU split, search range 32, and inference GPU memory utilization 0.8, and are compared over the first 200 training steps. The τ=128 setting is the main asynchronous packed run. Table 4 reports means over training steps, including each step's largest accepted distance, and final AIME mean@8 at step 200.

Table 4: Edit-distance budgets with shared configuration. AIME is measured at step 200. Rejected distances are censored; accepted distances indicate how much of the budget successful merges use.

τ	Updates/step	Compression	Rejected merges	Step time	Max. distance	AIME
0	2.43	1.001	99.84%	272 s	0	0.348
20	1.00	2.564	1.39%	138 s	16.8	0.277
128	1.00	2.828	0.78%	155 s	30.1	0.306

Rollout-to-training KL peaks at 0.0018, 0.0165, and 0.0108 for τ=0, 20, and 128, respectively. Mean clip fraction over the final twenty steps is below 0.003 in every run. At the same mini-batch

size, exact matching leaves more sequences to train on: τ=0 averages 2.43 optimizer updates per actor update, whereas τ=20 and 128 each perform one.

We observe a longer mean step time at τ=128 than at τ=20 (155 versus 138 s). The former also generates longer responses (547 versus 522 tokens per turn) and processes more training tokens (1.41M versus 1.31M). Step time depends on this generated workload as well as packing, so it does not isolate the efficiency of the matching algorithm.

Rejection conditions. The logged counters distinguish a missing anchor, a candidate exceeding the edit budget, and a prompt too short to match. Table 5 pools these counts over steps 1–200; percentages use rejected attempts as the denominator, unlike the mean per-step rejection rates in Table 4. As tolerance increases, fewer rejections are caused by exceeding the edit-distance budget, and missing anchors become the dominant cause. An anchor may be missing because its tokens changed or its position moved outside the local search window.

Table 5: Logged rejection conditions. Each cell gives its count and share of all rejections in that run; the categories are mutually exclusive.

τ	Total rejected	Distance exceeded	Anchor not found	Prompt too short
0	215,383	214,472 (99.58%)	910 (0.42%)	1 ($<$ 0.01%)
20	3,488	1,959 (56.16%)	1,528 (43.81%)	1 (0.03%)
128	1,816	19 (1.05%)	1,796 (98.90%)	1 (0.06%)

Repeat variation. We compare the primary asynchronous unpacked run and two repetitions (Table 6). Model, optimizer, batch size, and GPU allocation are the same. All three timing comparisons end after 200 training steps. For accuracy, we compare AIME mean@8 at step 200.

Table 6: Asynchronous unpacked training and two repetitions over 200 training steps. AIME is mean@8 at step 200.

Run	Steps	Wall-clock	Tokens/step	Response/turn	AIME
Primary	200	19.18 h	4.21M	676	0.335
Repeat 1	200	16.57 h	3.74M	574	0.319
Repeat 2	200	18.81 h	4.16M	706	0.348

Relative to the same packed run, the three unpacked runs give end-to-end packing speedups of 2.20×, 1.90×, and 2.16×, and token reductions of 2.98×, 2.64×, and 2.94×. Packing's savings therefore persist across repetitions. Relative to synchronous unpacked training, the corresponding asynchronous speedups are 1.14×, 1.32×, and 1.17×.

To describe learning-curve variation, we also compare the eleven shared checkpoints at steps 0, 20, …, 200. Mean absolute AIME differences are 0.0258 between the primary run and Repeat 1, 0.0174 between the primary run and Repeat 2, and 0.0227 between the two repeats. These are descriptive run-to-run differences, not confidence intervals or an equivalence test.

G Agent integration and learning curves

G.1 External adapters

Each adapter implements `process_task_async(ctx, config)`, receiving task content, routing information, and a request identifier, and returning an answer, reward, and metadata. The 122-line math adapter passes tasks through the existing agent interface. The 270-line kernel adapter also coordinates the external evaluator. Most of ZeroClaw's 334 adapter lines handle HTTP/WebSocket communication with its gateway. These external adapters preserve each agent's existing loop and tools. The adapter counts in Table 1 cover the complete adapter files: non-blank, non-comment physical lines, including docstrings and diagnostic code.

Table 7: Math-task integration LOC after excluding documentation, optional diagnostics, and unused code. Complete adapter-file counts in Table 1 use a separate scope.

	TACO	VeRL ReTool
Agent adapter / loop	43	0 (built-in)
Tool implementation	0 (reused)	59
Tool configuration	0	23
Reward definition	13	11
Dataset preparation	44	34
Module setup	7	8
Total	**107**	**135**

Table 7 instead counts task-specific integration code selected from the training path, including the active reward, dataset preparation, required imports, and tool configuration. Both sides exclude blank lines, comments, docstrings, diagnostic output, disabled debug branches, and unused code. Existing agent implementations, shared framework infrastructure, and experiment launch scripts are outside this scope; error handling and resource cleanup are retained.

For TACO, this leaves 60 of the math adapter's 122 lines, plus 47 lines for data preparation and its imports, totaling 107. VeRL's total is 112 Python lines plus 23 tool-configuration lines, or 135. These are filtered subsets of the original source, counted without rewriting or combining lines; they are not complete source-file lengths or counts of the full training system.

G.2 Learning curves

Using these adapters, both additional agents improve their validation rewards without changes to agent source code (Figures 5 and 6). Pale lines and dots show measured checkpoints; darker lines show an exponential moving average (EMA), with spans of five checkpoints for the kernel agent and three for ZeroClaw. All reported rewards use the measured values. For the kernel agent, evaluation requests 12 rollouts on each of 18 validation problems. Mean reward increases from 0.338 before training to 0.704 at step 300; worst-of-12 validation accuracy also rises from 0.056 to 0.222.

For ZeroClaw, we evaluate 70 training steps, with validation every ten steps. Each checkpoint evaluates 20 tasks with one rollout per task. Mean validation reward rises from 0.50 before training to 0.85 at step 20, remains between 0.75 and 0.80 thereafter, and reaches 0.80 at step 70.

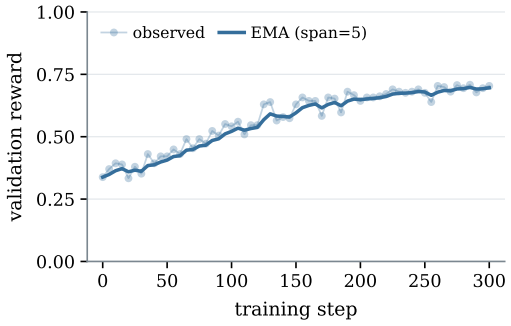

Figure 5: Kernel-agent validation reward over 300 training steps. Pale points: evaluation every five steps; dark line: EMA (span 5).

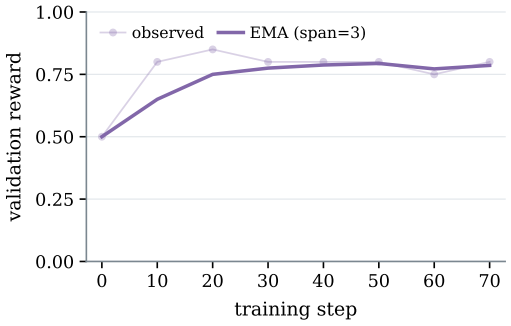

Figure 6: ZeroClaw validation reward over 70 training steps. Pale points: evaluation every ten steps; dark line: EMA (span 3).

H Limitations

Deployment. The worker-identifier mechanism requires a preserved routing field and a dedicated service instance that executes one rollout at a time. Services that rewrite all available routing fields

need another way to propagate rollout identity. The draining barrier also inherits the tail latency of admitted service tasks.

Context approximation. With nonzero tolerance, packing can change policy probabilities, importance ratios, and gradients by replacing replayed history with the recorded token sequence (Appendix C). Any merge that permits token differences must account for this context change; a bounded edit distance alone does not bound its effect on learning.

Optimization and future work. The accuracy gap between packed and unpacked training warrants further study. The recovery after increasing optimizer update count suggests that optimization granularity is a major contributor on our math agent. Future work will compare recorded rollout probabilities with trainer-recomputed old probabilities under matched optimization settings, to assess the effects of context approximation and training–inference mismatch across more agents.