GROUP-AWARE VERIFICATION SCHEDULING FOR REINFORCEMENT LEARNING WITH EXECUTION-BASED REWARDS

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning post-training of LLMs on tasks such as coding computes each reward by executing the rollout's output, for example by applying a generated patch in the task's environment and running its tests. Verification time can vary by an order of magnitude across tasks, and verification is failure-prone. Group-based algorithms such as GRPO make this worse, because a group of rollouts can enter training only when every member has a result, so the learner waits for the slowest member of its slowest group. Existing systems give no priority to the verification jobs that would complete a group. We present LASTCALL, a runtime that schedules verification for group-based reinforcement learning. It bounds each round of verification with a waiting window, after which incomplete groups keep verifying and can join a later batch whole under a one-update staleness bound, and gives each free verifier worker to the group closest to completion. Implemented inside asynchronous veRL, LASTCALL cuts model update time by 13% to 28% across three model scales in our experiments, without lowering the model quality.

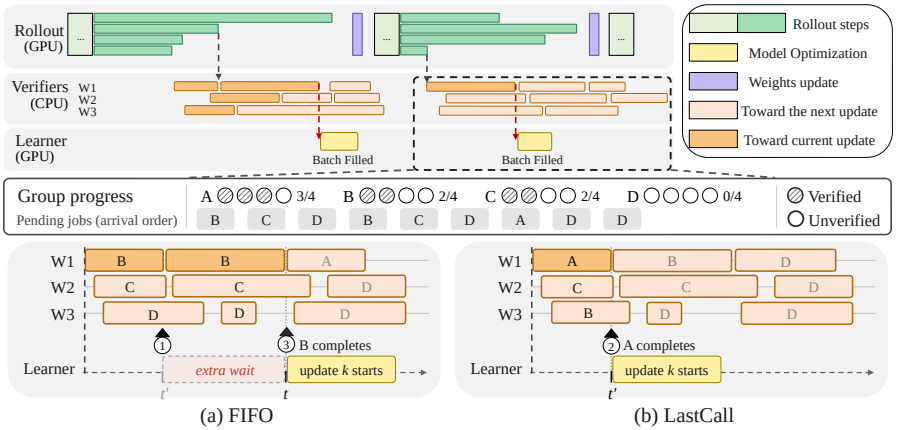
1 INTRODUCTION

After pretraining, a large language model can be improved further through reinforcement learning (RL) (Ouyang et al., 2022; Stiennon et al., 2020; Schulman et al., 2017; Shao et al., 2024). When rewards are computed by executing the result of a *rollout* (the model's attempt at a task), training runs as a three-stage pipeline (Sheng et al., 2025; Fu et al., 2025; Wang et al., 2025a; He et al., 2026) on both GPUs and CPUs. First, actors generate the rollout; the model runs on GPU serving engines, while any tool calls execute on CPUs or external services. Then, verification workers on CPUs check each rollout's result. For example, for a coding task, the verifier applies the generated patch to the repository and runs the project's test suite (Jimenez et al., 2024; Yang et al., 2024; Zhang et al., 2024; Wang et al., 2025b; Xia et al., 2025; Pan et al., 2025). Finally, a GPU learner consumes those outcomes and updates the model.

Efficient RL training, including group-based algorithms like GRPO (Shao et al., 2024), often requires parallelization to derive a large batch of rollouts for exploration (Sheng et al., 2025; Fu et al., 2025; Hu et al., 2025). Earlier RL for language model post-training targets math or QA tasks with verifiable rewards (DeepSeek-AI, 2025; Yu et al., 2025). There, a rollout is a single model call or tool-use response. Verification is often a cheap stateless check, such as comparing the final answer against a reference string. Tasks such as coding instead need their output executed, typically in a container that holds the task's environment. This holds whether the model writes a patch in one call or works as a multi-turn agent with a shell and tools (Pan et al., 2025; Tan et al., 2025; Hou et al., 2026; Jiang et al., 2025; Zhang et al., 2026a). Verification time can vary widely across rollouts and groups (§3). For the coding tasks above, a patch that fails to apply is rejected in seconds, while a patch that applies requires preparing the task's environment and running its tests. In our measurement, the longest completed verification takes over 40 minutes.

Existing RL systems do not treat verification time as a scheduling objective. Asynchronous designs bound rollout staleness (Fu et al., 2025; Li et al., 2026) or sample one rollout per prompt (Hou et al., 2026), and rollout-stage schedulers balance generation across workers (He et al., 2026; Hu et al., 2026). Where these systems order work, it is by staleness or rollout length, not group progress. Asynchronous veRL (Meituan Search, 2026) verifies rollouts in arrival order (FIFO, Figure 1a), so the job that would complete a group can wait behind other jobs, and the update is blocked.

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.



(a) FIFO (b) LastCall

Figure 1: Update k needs only one more complete group, so whichever of A–D completes first fills the batch. (a) First-in-first-out rollout verification in arrival order. At ①, none of A–D is complete, and the learner waits until B completes at t ③. (b) LASTCALL serves the group with the fewest members left to verify, so A completes first ② and update k starts at $t' < t$.

This paper presents LASTCALL (§4), a runtime that schedules RL verification efficiently for group-based algorithms. It bounds each round of verification with a waiting window. Unlike a per-run timeout, the window never cuts a run short or scores it as a failure. When a round ends, groups still incomplete keep verifying and can join a later batch whole, while the actor moves on to the next step. A group can enter the learner only while its rollouts are at most one update behind the current weights (bounded staleness (Ho et al., 2013)). When a verification worker becomes free, LASTCALL prioritizes the verification that would complete a group, so groups complete one after another instead of all late together (Figure 1b). We make three contributions.

- We expose the emerging verification scheduling challenges in RL with execution-based rewards. Group-based algorithms let one slow or failed run block an entire group and delay the batch needed for the next update.
- We formulate verification scheduling for group-based RL as filling a batch of any m complete groups as early as possible. We present LASTCALL, which combines a waiting window that can defer slow groups whole to a later batch with a ranking that gives free workers to the groups closest to completion.
- We implement and evaluate LASTCALL inside veRL's native GRPO path. In asynchronous end-to-end training, LASTCALL achieves up to 1.39× veRL's update throughput.

2 BACKGROUND

2.1 GROUP-BASED REINFORCEMENT LEARNING POST-TRAINING

In RL post-training, the policy model improves by generating rollouts for training tasks and updating on their rewards. Group-based algorithms such as GRPO sample a group of G rollouts for each task and score each rollout with the verifier's reward. Instead of a learned value critic, GRPO estimates each rollout's advantage from its own reward and the rewards of its $G-1$ peers,

$$\hat{A}_i = \frac{r_i - \text{mean}(r_1, \ldots, r_G)}{\text{std}(r_1, \ldots, r_G)},$$

and applies a PPO-style clipped update (Schulman et al., 2017) with this advantage. In our training, the return r_i is the verifier's reward minus $\beta \sum_t \text{KL}_{i,t}$, where $\text{KL}_{i,t}$ is the KL penalty toward a reference policy at token t of rollout i's response and β is its coefficient. The implementation adds a small constant δ to the denominator, which keeps $\hat{A}_i$ defined when all returns in a group are equal. The clipped update maximizes

$$J(\theta) = \mathbb{E}_t\Big[\min\Big(\rho_t(\theta)\hat{A}_i,\ \text{clip}\left(\rho_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_i\Big)\Big], \quad \rho_t(\theta) = \frac{\pi_\theta(o_t \mid q, o_{<t})}{\pi_{\text{old}}(o_t \mid q, o_{<t})},$$

where o_t is the t-th token of a response to task prompt q. Here π_{old} uses the log-probabilities recorded during generation, without recomputing them. Appendix B.1 lists the hyperparameters.

Each member's advantage depends on the rewards of its entire group. A rollout is *resolved* once the verifier has returned a reward. A group is fully generated once all of its rollouts exist, and is *complete* only when every member is resolved. The learner trains on batches of complete groups.

2.2 Existing Large Language Model Reinforcement Learning Runtimes

In large language model (LLM) RL post-training, a verification worker runs the verifier, the program that evaluates a rollout and returns its reward. Each execution is a verification job. In this paper, verification means producing an execution-based reward. The verifier executes the rollout's result in a live environment, e.g., a container, and the outcome determines the reward; for a coding agent, the generated patch from a rollout is applied and the test suite runs.

Existing runtimes track which rollouts are ready for training or bound their staleness, but to the best of our knowledge, none orders verification jobs by group progress (Appendix Table 5).

3 A Measurement Study of Group Verification

We use 400 training tasks from SWE-bench Verified (Jimenez et al., 2024; Chowdhury et al., 2024) as workload. Qwen2.5-7B is used to generate four rollouts per training task, and the verifier applies each patch to the repository and runs the test suite. We measure one veRL (Sheng et al., 2025) training run with no scheduling intervention. It visits every training task once, so it covers 1,600 rollouts in 400 four-member groups over 50 optimizer updates. This study uses eight groups (32 rollouts) per update.

Table 1: Verification results and average time per rollout.

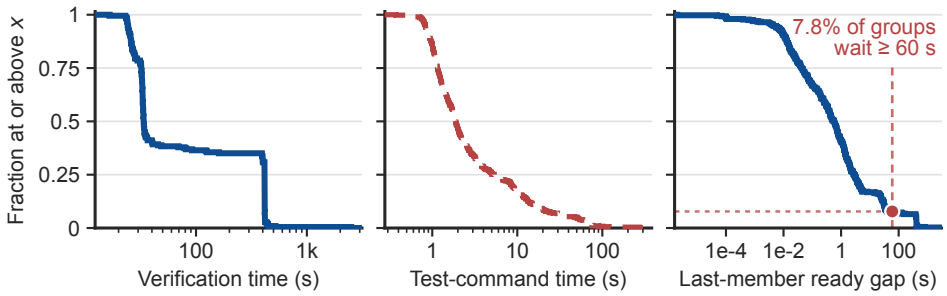
Result	Rollouts	Avg. time (s)
Patch fails to apply	1,083 (67.69%)	0.50
Tests complete	511 (31.94%)	167.26
Tests timed out	2 (0.12%)	3,000.30
Parsing failures	4 (0.25%)	N/A
Overall	1,600 (100.00%)	57.65

Table 1 reports the average verification time per rollout for each outcome. A patch that fails to apply is rejected in seconds (0.50s on average). A patch that applies has its environment prepared and its tests run, and occupies a worker for 167.26s on average. Over two thirds of rollouts fail to apply the patch, but they only consume under 1% of total recorded verification time. In contrast, verification jobs (31.9%) that complete tests consume 93%. A stronger model than the one we measured (Qwen2.5-7B) can produce more patches that apply, which will raise verifier demand. Verification also fails without returning a result. In our end-to-end runs (§6), 4.6%–5.0% of first executions return no usable result, mostly because of result parsing, sandbox I/O, or environment preparation (Appendix Table 9).

Verification time spans an order of magnitude across tasks, and a slow task makes a slow group. Figure 2(a) plots the complementary cumulative distribution function (CCDF) of per-task median verification times for rollouts whose tests complete (214 tasks). Each group contains four rollouts for the same task, evaluated with the same tests. At a time x the curve gives the fraction of tasks whose median is at least x seconds, on a logarithmic axis. Although the median is 33.7s, the 90th percentile is $12\times$ larger at 416.2s. The longest completed verification takes over 40 minutes.

In Figure 2(b), we profile the test running time of all 500 SWE-bench Verified tasks using their gold patches, i.e., patches supposed to pass all the tests. We find that the slowest 49 of the 489 tasks whose gold patches pass account for 66.6% of total test running time (more details in Appendix B.5).

Figure 2(a) times a full verification job, from environment preparation through testing, the interval that occupies a worker and that the learner waits for. The test command alone, run with the gold patch in a prepared environment (Figure 2(b)), takes far less time than a full job in training. The long tail in Figure 2(a) comes from environment preparation, which takes several minutes for 60 of the 214 tasks. Without environment preparation, the gold-patch tests still have a heavy tail, so the tail comes from the tasks and environment preparation stretches it. Caching environments shortens these jobs, and the gain from scheduling then depends on how long the learner still waits.



(a) Training verification. (b) Gold-patch tests. (c) Last-member gap.

Figure 2: CCDFs of (a) per-task median verification time in training, (b) test running time with gold patches, and (c) the interval between each group's penultimate and final results.

We also evaluate three other agentic workloads in Table 2. These verification workloads compute rewards by checking the resulting environment state. Their verification times are short and tightly clustered. This paper therefore focuses on coding. We expect heavier verification to become common as agents take on longer and more complicated tasks.

A group becomes usable only when its last member resolves, and one late member can delay its whole group. In group-based RL, the learner consumes complete groups, not rollouts, and updates only when a full batch of them is ready. If a member of a group is still in verification, the whole group is useless to the learner until that member resolves.

Table 2: Verification time across different workloads. Quantiles are over per-task medians.

Workload	# Tasks (# rollouts)	P50 (s)	P90 (s)	P90/P50
Coding (Jimenez et al., 2024)	214 (511)	33.70	416.20	12.35×
MiniWoB++ (Liu et al., 2018)	100 (400)	4.94	8.69	1.76×
AppWorld (Trivedi et al., 2024)	150 (600)	3.46	3.59	1.04×
τ^3 retail (Barres et al., 2026)	114 (456)	0.59	0.78	1.33×

Figure 2(c) plots the CCDF of the last-member gap of groups in our training measurement, the time between a group's penultimate and final results. About 7.8% of groups wait at least a minute for their final member. Under contention, a scheduler that ignores groups can give a free worker to a job whose group still needs several more members to finish, while a nearly complete group waits, e.g., with only the last member to be resolved. The same holds one level up, since a batch that waits for every group waits for its slowest task. In the measured run, the final group of one batch completed over 747s after the penultimate one.

4 LastCall

Section 3 identifies two problems in the verification stage of group-based RL training. First, verification time varies by an order of magnitude across groups, and a learner that waits for every group waits for its slowest task. Second, within a group, one unresolved member makes the whole group useless, so under contention the job that receives a free worker should be one that is more likely to complete a group. LastCall answers the first with a *waiting window* per verification round. A *waiting window* limits how long a verification round waits (§4.1). A configured limit caps the number of new verification jobs that can start in that window. LastCall answers the second by ranking pending jobs by how close their group is to completion (§4.2).

Problem. A verification round has N generated groups of size G, W verification workers, and a learner that needs $m \leq N$ complete groups. Each rollout needs one non-preemptive verification job whose duration is unknown until it ends. Whenever a worker is free, the scheduler chooses which pending job to start, aiming to minimize the batch-ready time T, the first time the buffer holds m complete groups counting any carried over from earlier rounds, and, secondarily, the verification work spent to reach it. The special case $m = N$, $G = 1$ is multiprocessor makespan minimization, so the problem is NP-hard even with known durations (Garey & Johnson, 1979).

LASTCALL instead exploits two features of the problem. Groups are interchangeable, since only the first m complete groups matter, and a group's number of unresolved members $u_g(t)$ is known exactly even when its remaining time is not, so the scheduler tracks each group's progress and selects jobs by it. Complete groups enter a buffer. With $B(t)$ the buffered groups that are unused and at most one update stale, the runtime hands the learner a batch as soon as $|B(t)| \geq m$, and the learner performs its usual group-based update. Algorithm 1 gives the selection for one free worker and is called once per free worker. This section and the algorithm describe synchronous training, in which rollout generation and learner updates do not overlap, and Appendix C presents the asynchronous flow and its integration with veRL.

4.1 BOUNDING THE WAIT WITH A WAITING WINDOW

Algorithm 1 Selecting the next verification job.

$B(t)$: complete, unused groups within the staleness limit;
m: number of groups per batch; g: the group of job j;
$u_g(t)$: number of unresolved members of g;
candidate: a pending job of the current round's groups;
$\widehat{d}_j$: mean duration of completed verifications in g;
c: the round cutoff; n: jobs started in the current window, bounded by the configured start limit.
Called when a worker is free.

if synchronous and the round is closed **then**
  return
if synchronous and $|B(t)| \geq m$ **then**
  return
for all candidate jobs j in increasing $u_g(t)$, breaking ties by FIFO **do**
  Read the current time t.
  if $t < c$ and n is below its cap **then**
    if $\widehat{d}_j$ is unavailable **or** $\widehat{d}_j \leq c - t$ **then**
      Start j on the free worker.
      $n \leftarrow n + 1$.
      return
Leave the jobs pending.

LASTCALL verifies rollouts one generation step at a time and bounds how long each step's verification can hold the learner. We call the verification of one step's rollouts a round. A round starts once the actor has returned the step's rollouts and they have been grouped by task. At that moment a waiting window opens. The window has a fixed length, and the time at which it ends is the cutoff c. While $t < c$ and the buffer holds fewer than m complete groups, the scheduler starts verification jobs for the round's groups (§4.2). The round ends in one of two ways. If the buffer reaches m complete groups before the cutoff, a batch is handed to the learner at once, and model update starts. Otherwise the round ends at the cutoff.

In the second case, complete groups stay in the buffer, while incomplete groups are deferred. The runtime stops waiting for them in this round, but their already-running verification jobs continue. Later results can complete these groups, while the actor generates more rollouts with the current weights because too few complete groups are available for an update. We enforce a one-update staleness bound. At learner use, a group generated more than one update behind the current policy is discarded (§4.3).

4.2 CHOOSING WHICH JOB TO START

LASTCALL ranks pending jobs by how close their group is to completion. Let $r_g(t)$ be the number of resolved members in a group of size G, so that

$$u_g(t) = G - r_g(t), \quad \Delta(t) = \max\{m - |B(t)|, 0\}$$

are the number of members group g still lacks and the number of groups the next batch still needs. In synchronous training, while $\Delta(t) > 0$, the scheduler examines pending jobs of the current round's groups in increasing $u_g(t)$, and it starts none once the batch is full. Jobs are served in FIFO order only when they have the same $u_g(t)$. Completing a group reduces $\Delta(t)$ by one, and when $\Delta(t) = 1$ the last missing result of one group fills the batch. Asynchronous training supplies successive batches while generation continues, so the scheduler applies the same ordering without the batch check, and each group's own window bounds its starts (Appendix C.1). Groups can be verified in parallel. When a worker becomes free, the scheduler considers jobs from groups with fewer missing results first.

For a pending verification job j, $\widehat{d}_j$ is the mean elapsed duration of completed verifications for other rollouts in the same group. These rollouts share a task, but their patches can take different times to verify. Before the group's first verification completes, the scheduler skips the time check below and retains the other conditions in Algorithm 1. Each completion updates the $\widehat{d}_j$ for subsequent starts.

When an estimate is available, starting a pending job at time $t < c$ requires

$$\widehat{d}_j \leq c - t \tag{1}$$

Equation 1 checks whether the estimated duration fits the remaining window. A job that fails this check stays pending, and the scheduler considers the next candidate.

4.3 From Results to a Learner Batch

A rollout that fails the task receives reward 0. If verification finishes without a usable result, e.g., when the verifier's container is killed or its output cannot be parsed, the rollout stays unresolved. A verification that exceeds the time limit instead receives reward 0 and is not retried. The verification queue permits at most one retry of the same rollout under the same verification conditions. A retry follows the existing window and start limits and does not open a new window.

At learner use, a group generated at policy version v_g must satisfy $0 \leq v - v_g \leq 1$, where v is the learner's version before the update. Synchronous LastCall trains on groups up to one update stale. A group generated under weights θ_k can be trained on by the update that produces θ_{k+1} or by the one that produces θ_{k+2}, and a group that misses both is discarded. For a group used one update after generation, the PPO ratio (§2.1) compares token probabilities under the current policy with those recorded under the policy that generated it.

5 Implementation

We extend veRL 0.7.1 (Sheng et al., 2025) with a verification scheduler and a buffer of complete rollout groups. Verification workers, implemented as Ray reward workers, start the pending jobs selected by the scheduler. The workers extract the patch, apply it in a separate execution environment, and run tests there. In synchronous training, our `generate_sequences()` wrapper buffers complete groups and requests additional rollouts until it can return a full batch to veRL. It attaches late results to the saved rollouts. Each group is stored in a file with its rewards and generation log probabilities, while a SQLite index tracks arrival order and selection. The runtime filters groups by policy version before batch assembly and rechecks the selected versions at `_update_actor()`. veRL then performs its native GRPO update. Appendix C.2 describes our integration with veRL's asynchronous rollout and training classes.

6 Evaluation

6.1 Experimental Setup

End-to-end training experiments. We train models on SWE-bench tasks for 200 updates with GRPO at three scales, Qwen2.5-7B, Qwen3-14B, and Qwen3-30B-A3B, on two 8-GPU A100 nodes split between rollouts and updates, with 512 four-member groups per update and up to 256 concurrent verification jobs. Each rollout is a single model call that writes a patch. Its verification applies the patch and runs the task's tests in its container, as for a multi-turn coding agent. Training tasks are a random sample of SWE-bench Full, disjoint from the evaluation tasks. We compare veRL with and without LastCall in synchronous (sync) and asynchronous (async) modes, plus async AReaL (Fu et al., 2025) and OpenRLHF (Hu et al., 2025), and report the mean of three runs per system. AReaL and OpenRLHF keep their own learners but share the verifier, arrival-order verification, and the one-update staleness bound. LastCall uses a 480-second window in our evaluation.

Controlled replay experiments. We also replay a recorded training run to compare scheduling policies under the same workload and verification conditions. Each round's rollouts are released from the recording after their recorded generation time, and a started job holds a verifier worker for its recorded duration before returning its recorded reward. The scheduler and the learner run live on one 8-GPU node, so every policy sees the same rollouts at the same times. The replays therefore measure time and verification work.

Metrics. T_K is the training time to K updates; in end-to-end runs it excludes startup and checkpoint saving. Verification work is the total execution time of verification jobs in worker-seconds, counting

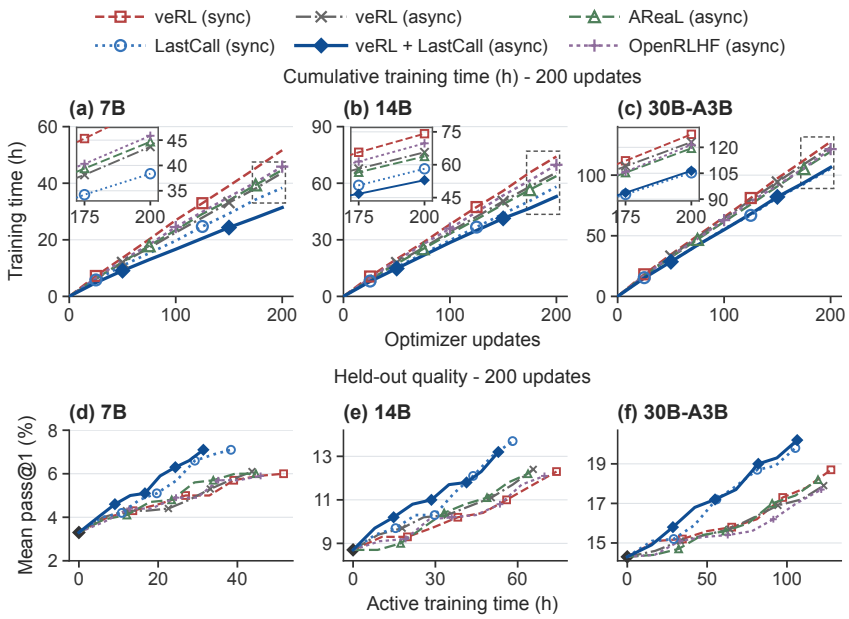


Figure 3: Training time (a–c) and mean pass@1 (d–f) for six systems, averaged over three runs.

every start and retry end to end, and only newly started jobs in replays, where we write it B_K. Quality is greedy pass@1 on 300 held-out SWE-bench Full tasks, measured every 25 updates and averaged over the three runs.

6.2 End-to-End Training

LastCall speeds up training in both sync and async modes. Even though async veRL already overlaps generation with learning, adding LastCall still cuts its time by 13%–28% (Figure 3(a–c)). In sync training, LastCall cuts the time by 17%–26%. The async combination is the fastest of the six systems at 7B and 14B. At 30B-A3B, the two LastCall modes finish about equally fast (105.26 and 106.43 hours), both ahead of the four baselines. In the same mode, LastCall finishes more than 12 hours sooner than veRL at every scale, more than three times the largest run-to-run standard deviation (Appendix B.2).

Overhead of transferring weights. At 14B, all four async systems spend about 300s per update transferring and synchronizing weights, against about 10s in sync training. Once LastCall shortens the wait for batches, weight transfer takes longer per update than the wait itself.

Quality. LastCall's mean pass@1 at update 200 is at or above veRL's in the same mode at every model scale (Figure 3(d–f)). The two systems train on different groups (§6.4), so we read this as quality holding rather than as a gain from scheduling.

6.3 Where the Speedup Comes From

LastCall saves time by making complete batches ready sooner. It also does less verification work. We first find where the 14B end-to-end runs save time. We then separate LastCall's mechanisms, end to end and in controlled replays, by comparing it with LastCall-FIFO for the ordering, veRL_W_WINDOW for the cutoff, and GROUP_W_PRIORITY for the start rules. Finally, we vary the window length and the duration estimates.

Batch supply. Table 3 breaks down the 14B runs. On average, a rollout's result arrives about as soon as with veRL, but full batches are ready 35% sooner in sync training and 25% sooner in async training. Async training overlaps supply with the previous update, so faster supply cuts the learner's wait per update by 53%. This shorter wait accounts for 12.11 of the 12.72 hours that async LastCall saves over async veRL.

Comparing scheduling rules through replays. The two replays differ in how much verification is pending. The *mid-run replay* starts from a recorded point in the run, where almost every rollout the learner uses already has a recorded result. Only 58 of its 12,800 results need a new execution, so it shows how policies spend verification when few jobs are pending. The *fresh replay* starts with all verification pending, so a queue forms as in end-to-end training. In both, the scheduler knows each job's recorded delay. The window and start-rule comparisons use the mid-run replay, and Appendix B.3 lists all replay results. The replays include GROUP_W_PRIORITY, our baseline that favors nearly complete groups like LASTCALL but keeps starting jobs after the batch is full. Its variant GROUP_W_PRIORITY_TIME breaks ties by shorter estimated time.

Table 3: Mean delays in 14B end-to-end runs.

		Result ready (s) ↓	Group complete (s) ↓	Batch supply (s) ↓	Learner wait (s) ↓
Sync	veRL	167	234	975	871
	LASTCALL	166	231	**638**	**562**
Async	veRL	176	267	871	412
	AReaL	172	254	843	388
	OpenRLHF	185	279	924	443
	LASTCALL	175	241	**653**	**194**

At 7B, the window and start rules provide most of the speedup. LASTCALL-FIFO keeps the window and start rules of LASTCALL but schedules jobs in the order they arrive. Trained end to end in async mode at 7B, it needs 35.57 hours against 43.68 for veRL, and LASTCALL needs 31.41. The window and start rules thus provide about two thirds of the saving, and the group-aware ordering the remaining third.

Table 4: LASTCALL vs. LASTCALL-FIFO in the two replays over 25 updates.

		Gen. and supply (s) ↓	Group completion (s) ↓	B_{25} (worker-s) ↓
Mid-run	LASTCALL	7,966	1.15	16,283
	LASTCALL-FIFO	8,014	1.17	16,413
Fresh	LASTCALL	**16,845**	**232.14**	781,254
	LASTCALL-FIFO	19,372	250.71	793,106

The replays show that the ordering helps when jobs queue. In the fresh replay, LASTCALL cuts generation and supply, the replay's counterpart of batch supply, by 13.0% relative to LASTCALL-FIFO and group completion time by 7.4%, with about as much verification work (Table 4). In the mid-run replay, where few jobs wait for a worker, the two orderings perform about the same.

The cutoff helps because groups left behind can join a later batch. In sync training at 14B, LASTCALL's groups complete about as soon as veRL's, yet its batches are ready sooner. At the cutoff, LASTCALL stops waiting for the slowest groups, which keep verifying and can join a later batch. VERL_W_WINDOW has the same cutoff but throws away results that arrive after it, so the work still running at the cutoff is wasted. It also waits out the full window whenever any group is incomplete. As a result, in the mid-run replay it takes twice as long as LASTCALL and does four times the verification work.

The combined start rules cut verification work. End to end, LASTCALL does less verification work than veRL in every same-mode comparison, up to 22.5% less. It starts fewer long jobs (in 14B sync training, 112 against 288 reach veRL's 3,000-second limit), and because retries follow the same window and start limits as first runs (§4.3), it retries at least 40% less often than veRL. The other start rules save work even without the time check. In the mid-run replay, LASTCALL without it still starts under a quarter as many jobs as GROUP_W_PRIORITY with it, and does 65% less work.

One preset window works, even with rough estimates. We set the window to 480s, about 15% above the 90th percentile of per-task verification time in §3 (416.2s), and use it at every scale and mode. At 480s, LASTCALL finishes sooner and does less verification work than both group-priority policies in all five replay comparisons. Given only a fixed mean time per group state and patch outcome, it still finishes at least 11% sooner and does at least 71% less verification work than both.

6.4 WHAT LASTCALL TRAINS ON

Because LASTCALL stops waiting for the slowest groups and retries fewer rollouts, the learner trains on a different mix of groups. We examine long tasks and stale groups in this mix.

LastCall still trains on long tasks. A scheduler that fills batches before the slowest groups complete risks training mostly on fast tasks. At every scale, LastCall trains on at least 93% as many long-task groups as veRL in the same mode, counting a task as long if its mean verification time in separate reference runs exceeds 240s (Appendix Table 13). Of the long-task groups it generates, LastCall uses a smaller share than veRL in sync training and a larger share in async training. About half of the long-task groups it uses enter training through a later batch, one update after they were generated.

LastCall trains on more one-update-stale groups, and quality holds. At 14B, 42% of sync LastCall's training groups and 51% of async LastCall's are one update stale, against 15% for async veRL and 0% for sync veRL. In sync training, they include groups the cutoff left behind, so sync LastCall's gains over veRL come from the scheduler and these stale groups together. In async training, where both systems share the one-update staleness bound, veRL leaves about one in four of its generated groups unused, against one in seven with LastCall. In both modes, quality stays at or above veRL's (§6.2).

7 Related Work

RL post-training runtimes. Recent RL post-training systems decouple rollout generation from learning, control how stale a trajectory can be when the learner uses it, and target long-tail generation (Fu et al., 2025; Li et al., 2026; Gao et al., 2026b; Hu et al., 2026; Zhang et al., 2026a;b; Wang et al., 2025a; He et al., 2026; Zhou et al., 2025; Gao et al., 2026a; Kimi Team, 2025a). DynaRL dynamically reallocates resources among heterogeneous RL-pipeline components as bottlenecks change (Wang et al., 2026a). It operates at the resource-allocation level; our scheduler instead chooses which individual verification job to run so that complete GRPO groups become available to the learner sooner. Verification is a different problem for a scheduler. A test run cannot be paused and resumed like decoding, and a partial run yields no reward. No system we know of schedules which verification job to start from the groups the learner needs.

Coflows and stragglers. A coflow in datacenter networking is a set of flows that is useful only once all of them finish (Chowdhury & Stoica, 2012; Chowdhury et al., 2014; Chowdhury & Stoica, 2015). A GRPO group resembles a coflow with two differences. The learner needs any m complete groups rather than all of them, and a group's remaining member count is known exactly even when its remaining time is not. Straggler mitigation in cluster computing replicates slow tasks on the assumption that the machine is at fault (Dean & Ghemawat, 2004; Dean & Barroso, 2013). A verification that is slow because of its task is slow on any machine, and caching environments shortens verification but still leaves a group waiting for its slowest member, so LastCall bounds that wait.

8 Discussion and Conclusion

This paper treats verification in group-based RL post-training as a scheduling problem, since a group enters training only when its last member resolves and a learner that waits for every group waits for its slowest task. LastCall bounds each round of verification with a waiting window, lets groups that miss it join a later batch under a one-update staleness bound, and gives each free verifier worker to the group closest to completion. Inside veRL, it reduces the time to 200 updates by up to 28% with less verification work, without lowering held-out quality.

We evaluate single-turn rollouts on coding tasks, where verification is long and varied. With multi-turn rollouts, generation takes a larger share of each update, so the gain depends on how long the learner still waits for verification; in 14B synchronous veRL, that wait is 65% of training time. Natural next steps are to measure how the gain changes with the number of verification workers and the verification speed, and to compare against adding workers or shortening the time limit.

REPRODUCIBILITY STATEMENT

We will release the code of LASTCALL, which extends veRL 0.7.1 (Section 5, Appendix C.2), upon acceptance. All training and quality-evaluation tasks come from the public SWE-bench benchmark (Jimenez et al., 2024; Chowdhury et al., 2024). Section 3 also measures verification time on MiniWoB++, AppWorld, and τ^3-Bench retail (Table 2). Section 6.1 and Appendix B.1 give the models, hardware, baselines, replay policies, metrics, and delay definitions. Tables 7 and 8 list the training and generation settings, and Table 6 reports the variation across three runs of each system.

AI USE STATEMENT

We used large language model (LLM) assistants to edit the wording of the paper, to write and modify code of the LASTCALL implementation, and to write the scripts that collect and aggregate the experimental data. The authors designed the system and the experiments, reviewed every AI-assisted change to the text and code, and checked the numbers the scripts produced. The authors take full responsibility for the content of the paper.

REFERENCES

Victor Barres, Ben Shi, Ola Zytek, Soham Ray, Keshav Dhandhania, and Pedram Razavi. τ^3-Bench: Advancing agent benchmarking to knowledge and voice. https://sierra.ai/blog/bench-advancing-agent-benchmarking-to-knowledge-and-voice, Mar. 2026.

Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets'12)*, Oct. 2012.

Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM'15)*, Aug. 2015.

Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with Varys. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM'14)*, Aug. 2014.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench verified. https://openai.com/index/introducing-swe-bench-verified/, Aug. 2024.

Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM (CACM)*, 56 (2):74–80, Feb. 2013.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, Dec. 2004.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. arXiv:2501.12948, Jan. 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. In *Proceedings of the 39th Annual Conference on Neural Information Processing Systems (NeurIPS'25)*, Dec. 2025.

Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, Jiamang Wang, Lin Qu, Bo Zheng, and Wei Wang. RollPacker: Taming long-tail rollouts for RL post-training with tail batching. In *Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'26)*, May 2026.

Wei Gao, Yuheng Zhao, Tianyuan Wu, Shaopan Xiong, Weixun Wang, Dakai An, Lunxi Cao, Dilxat Muhtar, Zichen Liu, Haizhou Zhao, Ju Huang, Siran Yang, Yongbin Li, Wenbo Su, Jiamang Wang, Lin Qu, Bo Zheng, and Wei Wang. RollArt: Disaggregated Multi-Task agentic RL training at scale. In *Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI'26)*, Jul. 2026.

Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

Zhenyu Han. [RFC] A centralized data management module for fine-grained dataflow management across RL tasks. https://github.com/verl-project/verl/discussions/2662, Jul. 2025.

Zhenyu Han, Ansheng You, Haibo Wang, Kui Luo, Guang Yang, Wenqi Shi, Menglong Chen, Sicheng Zhang, Zeshun Lan, Chunshi Deng, Huazhong Ji, Wenjie Liu, Yu Huang, Yixiang Zhang, Chenyi Pan, Jing Wang, Xin Huang, Chunsheng Li, and Jianping Wu. AsyncFlow: An asynchronous streaming RL framework for efficient LLM post-training. arXiv:2507.01663, Jul. 2025.

Jingkai He, Tianjian Li, Erhu Feng, Dong Du, Qian Liu, Tao Liu, Yubin Xia, and Haibo Chen. History doesn't repeat itself but rollouts rhyme: Accelerating reinforcement learning with RhymeRL. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'26)*, Mar. 2026.

Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *Proceedings of the 27th Annual Conference on Neural Information Processing Systems (NeurIPS'13)*, Dec. 2013.

Zhenyu Hou, Yujiang Li, Jie Tang, and Yuxiao Dong. Single-rollout asynchronous optimization for agentic reinforcement learning. arXiv:2607.07508, Jul. 2026.

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. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP'25)*, Nov. 2025.

Tianhao Hu, Xiangcheng Liu, Yuchun Miao, Youshao Xiao, Hongyu Zang, Yang Zheng, Xuan Huang, Jinrui Ding, Yufei Zhang, Yu Yang, Yi-Kai Zhang, Yueqing Sun, Chengcheng Han, Xiandi Ma, Wei Wang, Qi Gu, Yerui Sun, Yuchen Xie, and Xunliang Cai. DORA: A scalable asynchronous reinforcement learning system for language model training. arXiv:2604.26256, Apr. 2026.

Dongfu Jiang, Yi Lu, Zhuofeng Li, Zhiheng Lyu, Ping Nie, Haozhe Wang, Alex Su, Hui Chen, Kai Zou, Chao Du, Tianyu Pang, and Wenhu Chen. VerlTool: Towards holistic agentic reinforcement learning with tool use. arXiv:2509.01055, Sep. 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *Proceedings of the 12th International Conference on Learning Representations (ICLR'24)*, May 2024.

Kimi Team. Kimi k1.5: Scaling reinforcement learning with LLMs. arXiv:2501.12599, Jan. 2025a.

Kimi Team. Kimi K2: Open agentic intelligence. arXiv:2507.20534, Jul. 2025b.

Haoyang Li, Sheng Lin, Fangcheng Fu, Yuming Zhou, Xiaodong Ji, Yanfeng Zhao, Lefeng Wang, Jie Jiang, and Bin Cui. StaleFlow: Staleness-aware data management for mitigating data skewness in fully disaggregated RL post-training. arXiv:2601.12784, Jan. 2026.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'18)*, Apr. 2018.

Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Y. Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Li Erran Li, Raluca Ada Popa, and Ion Stoica. DeepScaleR: Surpassing O1-Preview with a 1.5B model by scaling RL. https://pretty-radio-b75.notion.site/DeepScaleR-Surpassing-O1-Preview-with-a-1-5B-Model-by-Scaling-RL-19681902c1468005bed8ca303013a4e2, 2025.

Meituan Search. Recipe: Fully Async Policy Trainer. https://github.com/verl-project/verl/blob/v0.7.1/docs/advance/fully_async.md, 2026.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Proceedings of the 36th Annual Conference on Neural Information Processing Systems (NeurIPS'22)*, Nov. 2022.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. In *Proceedings of the 42nd International Conference on Machine Learning (ICML'25)*, Jul. 2025.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv:1707.06347, Jul. 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:2402.03300, Feb. 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 20th European Conference on Computer Systems (EuroSys'25)*, Mar. 2025.

Nisan Stiennon, Long Ouyang, Jeff Wu, Daniel M. Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul Christiano. Learning to summarize from human feedback. In *Proceedings of the 34th Annual Conference on Neural Information Processing Systems (NeurIPS'20)*, Dec. 2020.

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. https://pretty-radio-b75.notion.site/rLLM-A-Framework-for-Post-Training-Language-Agents-21b81902c146819db63cd98a54ba5f31, 2025.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL'24)*, Aug. 2024.

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:2506.06122, Jun. 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 *Proceedings of the 13th International Conference on Learning Representations (ICLR'25)*, Apr. 2025.

Yuanqing Wang, Hao Lin, Junhao Hu, Chunyang Zhu, Quanlu Zhang, Zhen Guo, Yuchen Zhang, Xu Fu, Si Xu, Bo Dai, Zixiao Huang, Chao Yu, Boxun Li, Guohao Dai, Zhi Yang, and Yu Wang. DynaRL: Flexible and dynamic scheduling of Large-Scale reinforcement learning training. In *Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI'26)*, Jul. 2026.

Zhixin Wang, Jiaming Xu, Tianyi Zhou, Mingjun Zhang, Liming Liu, Jiarui Hu, Dian Yang, Tongyu Wang, Ping Zhang, Jinlong Hou, Siyuan Feng, Yuan Qi, and Yuan Cheng. DistFlow: A fully distributed RL framework for scalable and efficient LLM post-training. In *Proceedings of the 43rd International Conference on Machine Learning (ICML'26)*, Jul. 2026.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering (PACMSE)*, 2 (FSE):801–824, Jun. 2025.

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. In *Proceedings of the 38th Annual Conference on Neural Information Processing Systems (NeurIPS'24)*, Dec. 2024.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru 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, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. In *Proceedings of the 39th Annual Conference on Neural Information Processing Systems (NeurIPS'25)*, Dec. 2025.

Hao Zhang, Mingjie Liu, Shaokun Zhang, Songyang Han, Jian Hu, Zhenghui Jin, Yuchi Zhang, Shizhe Diao, Ximing Lu, Binfeng Xu, Zhiding Yu, Jan Kautz, and Yi Dong. ProRL Agent: Rollout-as-a-service for RL training of multi-turn LLM agents. arXiv:2603.18815, Mar. 2026a.

Liujie Zhang, Benzhe Ning, Rui Yang, Xiaoyan Yu, Jiaxing Li, Lumeng Wu, Jia Liu, Minghao Li, Weihang Chen, Weiqi Hu, and Lei Zhang. Relax: An asynchronous reinforcement learning engine for omni-modal post-training at scale. arXiv:2604.11554, Apr. 2026b.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'24)*, Sep. 2024.

Yinmin Zhong, Zili Zhang, Xiaoniu Song, Hanpeng Hu, Chao Jin, Bingyang Wu, Nuo Chen, Yukun Chen, Yu Zhou, Changyi Wan, Hongyu Zhou, Yimin Jiang, Yibo Zhu, and Daxin Jiang. StreamRL: Scalable, heterogeneous, and elastic RL for LLMs with disaggregated stream generation. arXiv:2504.15930, Apr. 2025a.

Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, and Xin Jin. Optimizing RLHF training for large language models with stage fusion. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*, Apr. 2025.

Yuzhen Zhou, Jiajun Li, Yusheng Su, Gowtham Ramesh, Zilin Zhu, Xiang Long, Chenyang Zhao, Jin Pan, Xiaodong Yu, Ze Wang, Kangrui Du, Jialian Wu, Ximeng Sun, Jiang Liu, Qiaolin Yu, Hao Chen, Zicheng Liu, and Emad Barsoum. APRIL: Active partial rollouts in reinforcement learning to tame long-tail generation. arXiv:2509.18521, Sep. 2025.

A CAPABILITIES OF EXISTING RUNTIMES

Table 5: Runtime capabilities relevant to group-based verification. ✓: supported; ✓*: partial or optional; ×: not established in the cited versions. "Readiness tracking" denotes explicit per-rollout or per-group tracking of training-ready data; ✓*: completed samples stream into a buffer that trains once it holds enough; ×: data reach the trainer only in whole batches. "Staleness control" means admitting rollouts from older policies under an explicit policy-lag bound; ✓*: the bound is implicit or the asynchronous mode is optional. "Verifier sched." means ranking verification jobs by how close their groups are to completion. The system list extends Table 1 of RhymeRL (He et al., 2026); the columns and marks are ours.

System	Readiness tracking	Staleness control	Verifier sched.
Synchronous trainers (Sheng et al., 2025; Zhong et al., 2025b; Luo et al., 2025) (Wang et al., 2026b; Jiang et al., 2025; Tan et al., 2025)	×	×	×
One-step and partial-rollout trainers (Zhong et al., 2025a; He et al., 2026; Kimi Team, 2025a;b)	✓*	✓*	×
Optional-async RLHF framework (Hu et al., 2025)	×	✓*	×
Readiness-aware data paths (Wang et al., 2025a)	✓	×	×
Readiness-aware paths with delayed updates (Han et al., 2025; Han, 2025)	✓	✓*	×
Bounded-staleness async trainers* (Fu et al., 2025; Meituan Search, 2026; Gao et al., 2026b)	✓*	✓	×
Staleness-aware data management (Li et al., 2026)	✓	✓	×
LASTCALL	✓	✓	✓

*Includes veRL 0.7.1's native fully asynchronous trainer.

B ADDITIONAL EVALUATION RESULTS

This appendix gives the setup, detailed results, and verification measurements for Section 6.

B.1 EXPERIMENTAL SETUP AND METRICS

Controlled workload and policies. A controlled run, or replay, replays a recorded training run through the live scheduler and the veRL 0.7.1 learner. Each round's rollouts are released after their recorded generation time, and each verification job returns its recorded outcome after its recorded delay; the learner runs live on the node's GPUs. Policies in a comparison share this workload and the verification resources, so they differ only in which jobs they start and when. In the window experiments, the scheduler knows each job's recorded delay before starting it; the delay covers execution, result transport, and matching the result to its rollout.

Policy definitions. All controlled policies verify the recorded rollouts. A start runs either a queued verification or, for a rollout left without a usable result, its one permitted retry; results that are already usable are accepted directly. The runtime allows at most a configured number of new starts in each waiting window (Section 4.1).

VERL_W_WINDOW serves pending jobs in the order of the rounds that generated them and waits until the cutoff whenever a group of the round has an unresolved member. group's own round.

GROUP_W_PRIORITY recomputes its priorities before each decision. It serves first the pending groups missing one result, then jobs that would complete a group, then any job that can start; ties go by rollout identifier. It keeps starting jobs while the window is open and the start limit allows, even when the batch is full or the estimate exceeds the remaining window. Results that arrive after the

cutoff still complete deferred groups. GROUP_W_PRIORITY_TIME breaks ties by shorter estimated time. Its rules for starting jobs are unchanged.

LASTCALL ranks pending jobs by group progress (Section 4.2). Jobs from groups with fewer unresolved members come first, and ties go by request arrival. While the buffer holds fewer complete groups than a batch needs, it walks this order and starts a job if its estimated time fits the remaining window, the window's start limit allows it, and the rollout is valid and has no other verification running. concurrently on the available workers; the ranking only decides the order in which they start. All policies share the result buffer, and the learner uses groups under the one-update staleness bound of Section 4.3. these rules but ranks pending jobs by arrival alone.

The window sweeps comprise 41 runs, one per configuration. They include a three-policy, five-window comparison on the mid-run workload (15 runs), GROUP_W_PRIORITY and LASTCALL at 33s and 240s on two more recorded workloads (8 runs), and the three group-aware policies at 33s, 240s, and 480s on two further workloads (18 runs). The variation study adds 18 runs, which use two assignments of durations to the replay's verification jobs, each at three variation levels with three policies and a 480-second window.

Every controlled run completes 25 updates of 128 four-member groups, so the learner uses 3,200 groups and 12,800 rollouts. Verification runs with at most 64 concurrent jobs on one 8-GPU node.

End-to-end training. The six systems train Qwen2.5-7B, Qwen3-14B, and Qwen3-30B-A3B with GRPO for 200 updates on 16 A100 GPUs, eight for rollout generation and eight for updates. Each batch holds 512 groups of four attempts at one task, and rewards come from executing the generated patches as verification completes. All settings are matched at each scale. We train each configuration three times and report the mean; Table 6 gives the standard deviations at update 200. The comparison covers synchronous and asynchronous veRL, each with and without LASTCALL, and the native asynchronous paths of AReaL 2.1.0 and OpenRLHF 0.11.1. These two run under the same conditions as the veRL systems, with a policy lag of at most one update, the same verifier with 256 concurrent jobs and the same timeout and retry rules, verification in arrival order, and the same batches and GPU split.

Table 6: End-to-end training results at update 200 over three independent runs (mean ± sample standard deviation; pass@1 in percentage points). LASTCALL-FIFO runs at 7B only. Figure 3 plots the checkpoints in between. Starts count every verification start through update 200, retries included, and work is their total execution time in millions of worker-seconds.

Model	System	Time to 200 updates (h)	pass@1 at update 200 (%)	Starts	Retries	Work (M worker-s)
7B	veRL (sync)	51.66 ± 0.42	6.0 ± 0.33	462,317	19,235	26.8217
	veRL (async)	43.68 ± 0.92	6.1 ± 0.38	503,491	21,075	29.6231
	AReaL (async)	44.67 ± 0.69	6.0 ± 0.33	525,514	14,328	31.1802
	OpenRLHF (async)	45.86 ± 1.48	5.9 ± 0.19	524,962	26,451	30.7749
	LASTCALL (sync)	38.37 ± 0.64	7.1 ± 0.19	485,193	11,342	25.4192
	veRL + LASTCALL (async)	31.41 ± 0.89	7.1 ± 0.38	491,128	10,245	25.8415
	LASTCALL-FIFO (async)	35.57 ± 1.28	6.8 ± 0.38	493,812	10,654	26.0418
14B	veRL (sync)	74.16 ± 0.37	12.3 ± 0.33	446,423	20,805	26.1542
	veRL (async)	65.67 ± 2.80	12.4 ± 0.51	547,975	19,092	32.5081
	AReaL (async)	63.78 ± 2.77	12.2 ± 0.51	508,629	21,334	30.2214
	OpenRLHF (async)	69.77 ± 2.58	12.1 ± 0.38	565,314	24,112	33.5398
	LASTCALL (sync)	58.15 ± 0.42	13.7 ± 0.33	476,289	11,147	24.8117
	veRL + LASTCALL (async)	52.95 ± 2.10	13.2 ± 0.51	482,511	10,147	25.1973
30B-A3B	veRL (sync)	127.54 ± 1.38	18.7 ± 0.67	485,488	19,294	28.3125
	veRL (async)	123.00 ± 3.92	17.9 ± 0.84	522,317	20,911	31.0334
	AReaL (async)	119.43 ± 2.15	18.2 ± 0.69	551,913	18,806	32.8512
	OpenRLHF (async)	121.48 ± 3.58	17.7 ± 0.67	540,926	26,112	31.9756
	LASTCALL (sync)	105.26 ± 1.14	19.8 ± 0.51	495,182	10,847	26.0619
	veRL + LASTCALL (async)	106.43 ± 2.70	20.2 ± 0.69	503,421	10,312	26.4388

Training objective and hyperparameters. The four veRL-based systems use the GRPO configuration in Table 7 at all three model scales. Each rollout's return is its verifier reward minus a per-token KL penalty toward the reference policy, summed over the response, and the group statistics in Section 2.1 standardize these returns, so KL regularization enters only through the return and the separate KL loss is disabled. Beyond the clipped objective in Section 2.1, the implementation applies dual clipping to tokens with negative advantage, capping their per-token loss at $c\,|\hat{A}_i|$. AReaL and OpenRLHF train with their frameworks' native policy objectives, using the same clip range, KL coefficient, and learning rate. The cross-framework comparisons thus include learner differences, while the asynchronous veRL pair differs only in verification scheduling (Appendix C.2). All six systems share the generation and evaluation settings in Table 8.

Table 7: GRPO training configuration of the four veRL-based systems at all three model scales.

Setting	Value
Advantage	Group-standardized return, $\delta = 10^{-6}$
Ratio denominator π_{old}	Rollout log-probabilities recorded during generation
Clip range ϵ (lower / upper)	0.2 / 0.2
Dual-clip bound c (negative advantages)	3.0
KL penalty in the return	Per-token log-ratio to the reference policy, fixed coefficient 0.001
Separate KL loss	Disabled
Entropy coefficient	0
PPO epochs per update	1
Loss aggregation	Mean over tokens

Table 8: Generation and evaluation settings shared by all six systems.

Setting	Value
Maximum prompt length	4K tokens
Maximum response length	28K tokens
Context limit	32K tokens
Sampling temperature	0.2
Top-p	1.0
Rollouts per prompt	4
Qwen3 thinking mode	Disabled; each rollout is one model call
Evaluation decoding	Greedy, one sample per task

Verification-time information. End to end, LASTCALL estimates a job's time as the mean of the completed attempts in its group (§4.2) and uses a 480-second window, set in advance about 15% above the 416.2-second 90th percentile of per-task median verification times (Figure 2(a)), so that tasks whose environment takes several minutes to prepare can finish within one window. Until a group's first attempt completes, the scheduler follows group-progress order without the time check; each completion then updates the mean. The other start conditions follow Algorithm 1 in synchronous training and Appendix C.1 in asynchronous training.

The controlled studies instead supply each job's time. Window runs supply the recorded delay, and variation runs supply a category mean that can differ from the actual duration. When the time check is on, the scheduler compares the supplied time with the time left in the window.

Verification windows and deferred groups. The veRL integration in Section 5 runs each round as follows. Returned rollouts are joined with their task data and grouped, and the window opens at time a, so the cutoff is $c = a + w$ for the window length w (480s). The round ends when a full batch goes to the learner, or at c if the batch is still incomplete. Running verification and result handling continue past the end. If the cutoff comes first, the actor generates more rollouts with the current policy to fill the batch. In a controlled replay, the next round's rollouts are instead released after their recorded generation time. The next window opens when the new rollouts return. In a replay, group state first absorbs the results already available in the recorded workload.

The start limit counts only new starts within the window. One group can receive several starts, and running jobs continue on their own.

Deferred groups keep their running jobs. A result that arrives after the cutoff still counts toward its group, and a group it completes joins the buffer for a later batch. Complete groups stay in arrival order until batch assembly, and the learner uses them only while they meet the policy-version condition of Section 4.3.

Training time and verification work. T_K is the training time to K optimizer updates. We use $K = 200$ in the six-system comparison, excluding startup and checkpoint saving, and $K = 25$ in the controlled runs, which keep their recorded timing, checkpoint and resume included. For a controlled run, the jobs $\mathcal{J}_K$ newly started in the verification rounds up to the 25-update endpoint give

$$B_K = \sum_{j \in \mathcal{J}_K} (e_j - s_j),$$

where s_j and e_j are a job's execution start and end. Each job contributes its full interval, including execution after the endpoint, and overlapping intervals add in worker-seconds. Jobs already running at the start belong to the shared workload and are not counted; a retry adds its own interval. Reductions are $100(1 - X_{\text{policy}}/X_{\text{reference}})$ for $X = T_K$ or B_K.

In the 200-update comparison, verification work instead covers every actual start, first or retry, from environment preparation through testing, for all groups whether used or not (Table 6).

Timing the supply of learner batches. End-to-end delay statistics cover what the learner uses. They take one result-availability value per used rollout, one group-completion value per used group, and one batch-supply and one learner-waiting value per update, zeros included. Verification costs and candidate counts also cover work outside the training batches.

Result-availability delay runs from an attempt's first verification request to the runtime's acceptance of a usable result, including waiting, execution, any permitted retry, and delivery. A failing test counts as a usable result, and a retried attempt keeps its first request as the start.

Group-completion delay runs from the first member's verification request to the acceptance of the group's last missing result, before any buffer waiting.

Batch-supply time runs from the start of collecting groups for a batch until 512 complete, unused groups within the staleness bound are ready. In synchronous training it spans every generation and verification round needed. In asynchronous training it can overlap the previous update.

In replays, *generation-and-supply time* covers releasing each round's recorded rollouts and the verification needed to make complete batches available. Since each round is released after its recorded generation time, differences between policies come from the number of rounds and the verification wait within them.

Learner waiting covers the intervals in which the learner is ready but lacks a full batch and nothing else blocks it; an update whose batch is ready contributes zero. Checkpoint saving, device synchronization, and other causes of blocking are excluded. Cumulative learner waiting sums these intervals over the 200 updates. Because the intervals of different metrics can overlap, total training time is measured separately.

B.2 End-to-End Results by Model and Checkpoint

Six-system, 200-update comparison. Table 6 lists the final training times behind Figure 3(a–c). Asynchronous veRL with LastCall is the fastest system at 7B and 14B. At 30B-A3B, synchronous LastCall finishes 1.17 hours ahead of it, and both LastCall configurations reach 200 updates sooner than AReaL, the fastest baseline there, by 11.86% in synchronous training and 10.89% in asynchronous training.

End-to-end FIFO comparison. Table 6 also lists LastCall-fifo at 7B. This variant keeps every rule of LastCall except the ordering and verifies requests in arrival order. It reaches 200 updates in 35.57 hours, between LastCall's 31.41 and veRL's 43.68, with about as much verification work as LastCall.

Verification work. Relative to veRL in the same mode, total verification work, including retries, falls by 5.1%–7.9% in synchronous training and by 12.8%–22.5% in asynchronous training (Table 6). LASTCALL starts more jobs than veRL in synchronous training and fewer in asynchronous training, and its jobs are shorter on average, 52.1s to 52.6s per start against 58.0s to 59.5s for the baselines. The saving comes mainly from starting fewer long jobs and retrying fewer rollouts that lack a usable result. The ordering changes verification work by only about 1% (§6.3).

Execution paths and retries. In 14B synchronous training, LASTCALL starts more jobs than veRL but does less verification work (Table 9(a)). Runs whose tests complete account for 61.5% of the net saving and timed-out runs for 39.3%. LASTCALL starts slightly more runs whose tests complete, but they are shorter on average, and it times out less often. Both systems keep veRL's native timeout handling, which scores a verification that exceeds 3,000s as a failure with reward 0. LASTCALL reaches that limit 112 times against 288 for veRL, consistent with its time check holding back jobs whose estimate exceeds the remaining window.

A failing test is a usable result, but a completed test can still end without one if its result cannot be parsed or delivered. In both systems, about 5% of first executions leave a rollout without a usable result (Table 9(d)). LASTCALL retries 49.8% of these rollouts and veRL 98.1% (Table 9(b)), because LASTCALL's retries follow the same window and start limits as first runs (§4.3). A rollout left without a retry keeps its group incomplete, so that group is not used for training.

The same panel breaks these rollouts down by cause. Result parsing is the most common; at 7B, the model measured in §3, it affects 1.97% of first executions.

Batch supply and learner waiting. Against the three asynchronous baselines at 14B, LASTCALL cuts mean batch-supply time by 22.5%–29.3% and mean learner waiting by 50.0%–56.2% (Table 10). Single results arrive about as fast on average and later at the median and P95, yet groups finish sooner at the P95 and batches are ready earlier. The intervals follow Appendix B.1 and can overlap in asynchronous training.

Panel (b) of Table 10 splits the 14B training time into disjoint stages. Asynchronous training spends about 300s per update on weight transfer and synchronization, against about 10s in synchronous training, and this stage accounts for most of its extra time outside learner waiting. Within the asynchronous pair, it stays the same while learner waiting falls from 82,414s to 38,822s.

Quality through 200 updates. Figure 3(d–f) plots the mean pass@1 every 25 updates. Each value averages the three training runs' checkpoints at that update on the same 300 held-out SWE-bench Full tasks, and the figure places each checkpoint at its cumulative training time; Table 6 gives update 200 with its standard deviation.

B.3 CONTROLLED SCHEDULING RESULTS

FIFO comparison. Table 4 compares the two orderings in two replays with a 480-second window; within each replay, they share the workload and the verification resources. In the mid-run replay, 12,649 of the 12,800 results the learner uses already exist and 93 come from jobs running at the start, so only 58 need new executions. In the fresh replay, all verification is pending, and new executions supply all 12,800.

The ordering matters in the fresh replay, where a queue forms. There, LASTCALL cuts generation-and-supply time by 13.04% relative to LASTCALL-FIFO, completes groups 7.41% sooner, and does 1.49% less verification work. In the mid-run replay the three reductions are 0.61%, 1.71%, and 0.79%. The difference lies in how often the scheduler has a choice. At least two jobs compete in 82.4%–84.1% of start decisions in the fresh replay, against 4.2%–4.8% in the mid-run replay. Both replays use the concurrency limit of the other controlled runs; the fresh replay differs only in starting with all verification pending.

Window comparisons. At 480s, the window used end to end, LASTCALL is faster and does less verification work than every other policy, both on the mid-run workload in Table 11 and in all five 480s comparisons of Figure 4, which pairs LASTCALL with each group-priority policy on each workload. Shorter windows trade differently. On the mid-run workload, GROUP_W_PRIORITY fin-

Table 9: Verification execution and retries in the 14B synchronous runs through 200 updates. Panel (a) counts all starts, including retries, by execution outcome. Mean duration is in seconds and verification work is in worker-seconds. Other combines pre-test termination, interruption during testing, and unclassified executions. Panel (b) follows first executions and reports the fraction of those without a usable result that receive a retry; a timed-out verification receives reward 0 and is not retried, so it counts as a usable result. Panel (c) separates first and retry executions by their own execution outcomes. Panel (d) assigns one cause to each first execution that leaves its rollout without a usable result in the synchronous runs, with 7B veRL added; percentages use all first executions, and timeouts are excluded (192, 246, and 103 first executions).

(a) Execution costs

	veRL			LastCall		
Execution outcome	Starts	Mean (s)	Work	Starts	Mean (s)	Work
Patch fails to apply	289,360	0.51	147,861	316,764	0.51	161,852
Tests complete (including valid failures)	150,980	166.04	25,068,247	154,009	157.41	24,243,109
Tests timed out	288	3,000.20	864,058	112	3,000.13	336,014
Other	5,795	12.78	74,071	5,404	13.08	70,693
Total	446,423	58.59	26,154,237	476,289	52.09	24,811,668

(b) Results of first executions and subsequent retries

Metric	veRL	LastCall
First executions	425,618	465,142
First executions without a usable result	21,202	22,394
Fraction without a usable result	4.98%	4.81%
Retries started	20,805	11,147
Fraction of first executions without a usable result retried	98.1%	49.8%
Remaining without a retry start by the measurement endpoint	397	11,247

(c) First and retry executions by outcome

	veRL		LastCall	
Execution outcome	First	Retry	First	Retry
Patch fails to apply	288,129	1,231	314,896	1,868
Tests complete	132,045	18,935	144,761	9,248
Tests timed out	246	42	103	9
Other	5,198	597	5,382	22
Total	425,618	20,805	465,142	11,147

(d) Causes of first executions without a usable result

	7B veRL	14B veRL	14B LastCall
First executions	443,082	425,618	465,142
Result parsing	8,742	10,183	10,437
Sandbox I/O and result delivery	4,128	5,391	5,422
Environment preparation	3,956	2,876	3,428
Other causes	3,495	2,752	3,107
Total unresolved	20,321	21,202	22,394
Fraction unresolved	4.59%	4.98%	4.81%

ishes sooner than LastCall at 1s, 33s, and 90s, and it also does less verification work at 1s and 33s. Across the figure, LastCall does more verification work in all seven 33s comparisons, and at 240s it does less work in all seven but finishes later in three. Short windows favor policies that start few jobs, because most results the learner uses in these replays are already recorded (§6.3). On

Table 10: 14B timing distributions over 200 updates, in seconds. Panel (a) gives mean / P50 / P95. Result-availability delay includes queue waiting and verification execution. The LASTCALL rows use the corresponding synchronous or asynchronous veRL integration. Panel (b) gives means over three runs and splits training time into disjoint stages under the startup and checkpoint exclusions in Appendix B.1; its learner waiting is the cumulative waiting, and Other is the remaining time.

(a) Delays: mean / P50 / P95

System	Result available	Group complete	Batch supply	Learner waiting
Synchronous				
veRL	167 / 67 / 538	234 / 146 / 725	975 / 845 / 1,472	871 / 751 / 1,284
LASTCALL	166 / 82 / 548	231 / 127 / 675	638 / 613 / 955	562 / 491 / 899
Asynchronous				
veRL	176 / 83 / 524	267 / 156 / 755	871 / 771 / 1,352	412 / 375 / 865
AReaL	172 / 81 / 517	254 / 149 / 738	843 / 748 / 1,321	388 / 352 / 824
OpenRLHF	185 / 87 / 551	279 / 163 / 782	924 / 812 / 1,417	443 / 401 / 911
veRL + LASTCALL	175 / 92 / 591	241 / 136 / 665	653 / 578 / 1,032	194 / 164 / 508

(b) Partition of training time through 200 updates

System	Total	Learner waiting	Learner updates	Weight transfer	Other
Synchronous					
veRL	266,976	174,185	16,356	1,984	74,451
LASTCALL	209,340	112,437	16,135	1,975	78,793
Asynchronous					
veRL	236,412	82,414	16,268	59,842	77,888
AReaL	229,608	77,619	16,293	59,633	76,063
OpenRLHF	251,172	88,591	16,259	64,811	81,511
veRL + LASTCALL	190,620	38,822	16,026	59,915	75,857

the mid-run workload, GROUP_W_PRIORITY at 1s even beats LASTCALL at 480s on both metrics; end-to-end training has no such shortcut, because it must produce every result.

Table 11: Three-policy comparison across five windows on the mid-run workload, one run per configuration. All runs complete 25 updates. T_{25} is in seconds; B_{25} sums full execution intervals for newly started jobs, in worker-seconds (Appendix B.1). Panel (b) runs LASTCALL without the time check and GROUP_W_PRIORITY with it on the same workload; Starts counts new starts.

(a) Window sweep

	VERL_W_WINDOW		GROUP_W_PRIORITY		LASTCALL	
Window	T_{25}	B_{25}	T_{25}	B_{25}	T_{25}	B_{25}
1s	9,906.69	354.93	8,133.55	33.95	8,663.67	1,925.66
33s	9,333.49	9,724.93	8,468.07	1,002.85	9,879.13	2,354.03
90s	10,727.09	23,961.53	8,784.84	8,469.67	9,096.67	7,864.03
240s	15,157.05	48,191.41	10,522.91	31,500.63	9,588.10	14,506.46
480s	20,221.65	68,720.78	11,787.16	54,285.84	10,054.79	16,283.37

(b) Time check swapped, 480s window

Policy	T_{25} (s) ↓	B_{25} (worker-s) ↓	Starts
LASTCALL (time check disabled)	11,160.53	**19,238.81**	225
GROUP_W_PRIORITY + time check	11,800.82	55,228.29	952

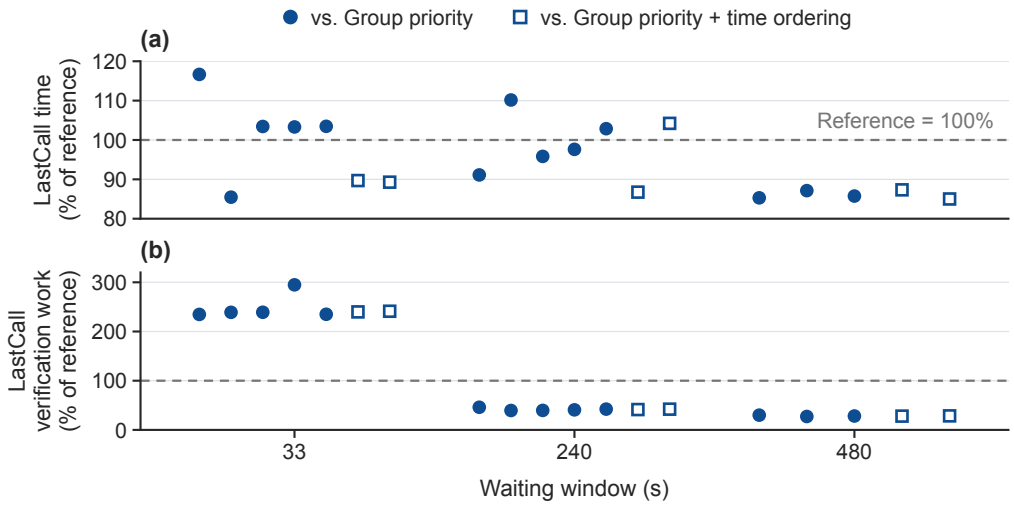


Figure 4: Window sensitivity at 33s, 240s, and 480s over 19 paired comparisons. Points show LASTCALL time and verification work as percentages of the reference (100%); lower is better. Horizontal positions align the same comparison across panels; shapes identify the reference policy.

Verification-time variation. LASTCALL's gains hold when actual durations vary around the estimates the scheduler sees. This study scales how far each verification duration d_i strays from the mean $\mu_{s(i)}$ of its stratum:

$$d_i^{(\alpha)} = \mu_{s(i)} + \alpha\big(d_i - \mu_{s(i)}\big),$$

with $\alpha = 0.10$, 0.55, and 1.00 for Low, Medium, and High. A stratum combines the group state (one unresolved member or several), the source category (fast path or executed tests), and whether the original duration exceeds 480s. The scheduler sees only the mean for the group state and source category, which stays fixed while the replay uses the transformed durations. The transformation keeps the replay's verification jobs and their mean and total duration. The two assignments place the same durations on different jobs, under a 480-second window. Only new starts get transformed durations. LASTCALL improves both metrics in all 12 comparisons (Table 12). Its own verification work is nearly the same in every condition, 15,524 to 15,526 worker-seconds, so the work reductions differ only through the references.

Table 12: LASTCALL reductions under three levels of verification-time variation at a 480-second window. Assignments 1 and 2 distribute the same durations across the replay's verification jobs differently; each reduction is relative to the listed reference.

Variation	Reference	Time reduction (%)		Verification work reduction (%)	
		Assignment 1	Assignment 2	Assignment 1	Assignment 2
Low	GROUP_W_PRIORITY	11.21	12.62	71.88	72.40
	GROUP_W_PRIORITY_TIME	12.71	11.45	71.24	72.68
Medium	GROUP_W_PRIORITY	11.79	13.64	71.89	72.18
	GROUP_W_PRIORITY_TIME	14.33	11.89	71.96	72.40
High	GROUP_W_PRIORITY	12.57	14.73	72.30	72.30
	GROUP_W_PRIORITY_TIME	14.17	12.02	72.30	72.41

Time-check comparison. Without its time check, LASTCALL still starts under a quarter as many jobs as GROUP_W_PRIORITY with one, and does 65% less verification work. Panel (b) of Table 11 swaps the time check between the two policies on the mid-run workload. LASTCALL runs without it and GROUP_W_PRIORITY with it, and both keep their other rules. Running jobs continue past

the cutoff, and each new job is timed to completion. differences, LASTCALL starts jobs within the window only while the batch still needs groups, whereas GROUP_W_PRIORITY also starts them once the batch is full.

Most of the gap in elapsed time comes from releasing rollouts and making results available, which takes 9,037.03s over 29 rounds for LASTCALL and 9,696.90s over 26 rounds for GROUP_W_PRIORITY; the other stages take about as long.

B.4 TRAINING DATA SELECTION

All candidate groups and reward composition. Panel (a) of Table 13 follows every candidate group in the four 14B veRL configurations through update 200. Each candidate is used, discarded by the runtime under the rule at the end of this subsection, or left unused otherwise, and each system uses 102,400 groups. Synchronous LASTCALL generates 9.8% more candidates than synchronous veRL and leaves 12.9% of them unused, against 4.4%. When the cutoff arrives before a batch is full, the actor generates the next step's rollouts. Deferred groups complete only through their running jobs, so some stay unused (Appendix B.1). The end-to-end times include this extra generation, and at 14B synchronous LASTCALL still does 5.1% less verification work and reaches 200 updates 21.6% sooner than synchronous veRL (Table 6). In asynchronous training, LASTCALL generates 11.4% fewer candidates and leaves 14.6% of them unused, against 24.3%.

The last column gives the reward composition. With LASTCALL, the share of used groups with identical member rewards falls by 10.5 points in synchronous and 11.9 points in asynchronous training.

Table 13: All candidate groups and reward composition in the 14B runs through 200 updates. Counts refer to unique groups. The last column gives the percentage of used groups whose member rewards are all equal. Panel (b) follows the long-task candidate groups of the four veRL-based configurations; groups used with one-update lag are included in Used and satisfy $v_{\text{use}} - v_g = 1$ at learner use.

(a) All candidate groups at 14B

System	Candidates	Used	Used with one-update lag	Discarded	Other unused	Equal rewards (%)
Synchronous						
veRL	107,065	102,400	0	1,523	3,142	34.8
LASTCALL	117,586	102,400	42,817	8,314	6,872	24.3
Asynchronous						
veRL	135,295	102,400	15,723	15,211	17,684	34.5
veRL + LASTCALL	119,857	102,400	52,319	9,642	7,815	22.6

(b) Long-task groups at all three scales

Model	System	Candidates	Used	Used with one-update lag	Discarded	Other unused
7B	veRL (sync)	22,749	21,253	0	531	965
	veRL (async)	25,220	19,687	2,948	3,184	2,349
	LASTCALL (sync)	23,394	19,854	8,359	2,012	1,528
	veRL + LASTCALL (async)	23,589	19,546	9,758	2,107	1,936
14B	veRL (sync)	22,585	21,048	0	488	1,049
	veRL (async)	25,346	19,852	3,453	2,943	2,551
	LASTCALL (sync)	23,477	20,447	9,416	1,568	1,462
	veRL + LASTCALL (async)	23,579	20,049	10,513	1,502	2,028
30B-A3B	veRL (sync)	21,076	19,583	0	535	958
	veRL (async)	26,899	21,348	4,271	3,212	2,339
	LASTCALL (sync)	22,511	19,481	9,483	1,498	1,532
	veRL + LASTCALL (async)	24,457	20,947	11,518	1,577	1,933

Long tasks and groups used with one-update lag. A task is long if its mean verification time exceeds 240s in a fixed set of reference measurements, taken independently of any system's schedul-

ing, and a group is long if its task is. The reference times cover execution only, and all systems at a model scale share the same labels.

A used group has one-update lag if

$$v_{\text{use}} - v_g = 1,$$

where v_g is the policy version that generated its rollouts and v_{use} is the learner's version just before the update that consumes it. A complete group can acquire this lag while it waits in the buffer, since several waiting windows can pass between updates.

Long-task groups in end-to-end training. Panel (b) of Table 13 follows the long-task candidate groups of the four veRL-based configurations through 200 updates.

Because candidate counts differ, we compare the share used. Across scales, it is 92.9–93.4% for synchronous veRL versus 84.9–87.1% for synchronous LASTCALL, and 78.1–79.4% for asynchronous veRL versus 82.9–85.6% for veRL + LASTCALL. Section 6.4 divides LASTCALL's used count by that of veRL in the same mode, and reports the share of LASTCALL's used long-task groups that have one-update lag.

Discard rule. A group generated under weights θ_k is discarded once it can no longer be used within the one-update staleness bound of §4.3, that is, when neither the update that produces θ_{k+1} nor the one that produces θ_{k+2} uses it. Under this rule, synchronous LASTCALL uses at least 93% as many long-task groups as synchronous veRL but discards more, while asynchronous LASTCALL discards fewer than asynchronous veRL (Table 13).

B.5 Additional Verification Measurements

Test execution with gold patches. We run the official test command with the gold patch for all 500 SWE-bench Verified tasks from 12 repositories, with network access disabled, four CPU slots per execution, and at most four executions at a time. Only the test command is timed; image preparation, dependency installation, container startup, and patch application happen before it. Each task contributes its final measurement, and quantiles use linear interpolation.

The gold patches pass the official evaluation on 495 tasks, and 489 of these also exit with code zero; these 489 form the distribution in Figure 2(b). The five failures are network errors in the Requests repository, one of them a timeout. For 96 of the 489 tasks, the environment or dependencies needed a fix, applied before timing starts. Table 14 gives both the 489-task and the 495-task distributions.

Table 14: Gold-patch test-command time in seconds. Each task contributes one measurement. The main distribution requires official evaluation success and a zero test-command exit code; the second row includes all tasks meeting the official success criterion.

Tasks	N	Mean	P50	P90	P95	P99	Max
Success, zero exit	489	8.636	1.891	17.466	49.008	78.627	304.544
All successful	495	9.070	1.894	17.869	50.612	102.165	304.544

Test time concentrates in a few tasks. The slowest 49 of the 489 take 66.63% of the total, 33 take 30s or more, 18 take 60s or more, and three take 120s or more. Repositories also differ. Matplotlib's median is 13.9 times Django's, although the longest run, `django__django-10097`, takes 304.544s.

C Asynchronous Execution and Integration

C.1 Verification and Batch Supply

Asynchronous training overlaps rollout generation, verification, and learner updates. Each group opens its own waiting window when its first verification request reaches the verification scheduling service. The window has the same configured duration as in synchronous training, with a cutoff c_g specific to group g. Generation continues under veRL's concurrency, queue, and policy-staleness limits as the runtime supplies successive learner batches.

The scheduler uses the group-progress ordering and within-group duration estimate from §4.2. A pending job with an available estimate must satisfy $\widehat{d}_j \leq c_g - t$ before it starts. Before the group's first completion, the scheduler skips this time check; starts still require time remaining in the group's window, a free worker, and room under the group's configured start limit. Each group's count begins at zero when its window opens and increases with each new verification start. Later completions update the estimate for pending jobs, while running jobs continue.

When group processing returns with a group still incomplete, the rollout component releases generation capacity and the runtime keeps records for deferred groups. Results arriving after the cutoff update their original groups. Once a deferred group is complete and within the one-update staleness bound, it enters the training queue once. The runtime assembles full batches from complete, unused, policy-eligible groups. At learner use, each group satisfies the one-update staleness bound of §4.3. Preparing these batches can overlap the learner's preceding update.

C.2 Integration with veRL

We implement this flow in a separate copy of veRL 0.7.1 by adding a verification scheduling service and modifying its existing asynchronous rollout and training classes. Our `CommonVerifierService` selects arrival-order (FIFO) or LastCall scheduling and matches results to their groups. We modify `FullyAsyncRollouter` to carry group identities, check which groups can enter training after verification, and coordinate stopping. Complete groups reach the learner through veRL's training queue.

We also modify control methods in the existing `FullyAsyncTrainer` class to set the training target, stop training and rollout generation after the requested number of optimizer updates, and control checkpoint saving. Asynchronous veRL, which verifies in arrival order, and LastCall use this same integration, including group-identity and policy-version checks, and differ in verification scheduling. Both retain veRL's generation concurrency, queue, and staleness limits.