THUNDERSYNCRL: LOSSLESS ACCELERATION OF AGENTIC REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Language models are moving beyond generating answers to pursuing long-horizon goals in interactive environments. Post-training these agents requires long, heterogeneous trajectories, and synchronous systems leave learner engines idle until rollout and verification finish. To squeeze out these pipeline bubbles, asynchronous training overlaps rollout and learning across updates, but comes at the cost of policy staleness. We introduce *ThunderSyncRL*, which starts gradient computation as soon as all required inputs are fixed, without policy staleness. For group relative policy optimization (GRPO), ThunderSyncRL computes each trajectory's score gradient as soon as the reward for that trajectory arrives, without waiting for the group. For on-policy distillation (OPD), it computes gradients for each completed agentic turn's teacher-scored actions while tool calls run in the sandbox. We prove that gradient streaming produces the same GRPO and OPD updates as batch-synchronous training, without changing either objective. On SWE-bench Verified and Terminal Bench 4.0, we train models to the same performance up to $1.9\times$ faster than synchronous training. With zero policy staleness, ThunderSyncRL also outperforms asynchronous training at a fixed budget by up to 2.47 percentage points.

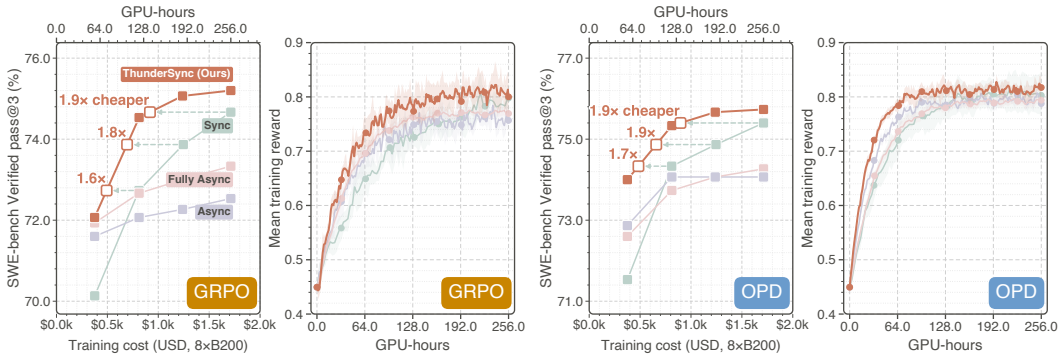

Figure 1: ThunderSyncRL matches Sync's SWE-bench Verified pass@3 at up to $1.9\times$ lower training cost under GRPO (left pair) and OPD (right pair). At a 128-GPU-hour budget, its zero-staleness schedule improves pass@3 by up to 2.47 percentage points over Async and Fully Async, both of which admit stale rollouts.

1 INTRODUCTION

Language models are increasingly trained as agents for long-horizon decision-making in interactive environments, reasoning and using tools in response to feedback. Reinforcement learning (RL) from environment or verifier feedback has emerged as a promising way to improve this behavior (Jin et al., 2025; Feng et al., 2026; Pan et al., 2025; Wei et al., 2025a; Gehring et al., 2025). However, training these agents requires interaction trajectories that are expensive to generate and heterogeneous in duration, with long-tailed completion times (Hu et al., 2026; Chen et al., 2026a; Ren et al., 2026). In batch-synchronous RL, learner engines remain idle until rollout engines finish the batch and all rewards arrive (Fu et al., 2025; Ren et al., 2026). We found that learner GPUs sit idle for 49.5%

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.

of step time in synchronous GRPO and OPD training while they wait for rollout and verification (Figure 3). Prior work reports idle fractions as high as 81% (Ren et al., 2026), and the slowest 10% of rollout requests can take up to half of rollout time (Qin et al., 2026), which itself accounts for 70% to over 90% of training time (Gao et al., 2025; Zhou et al., 2025).

Existing systems reduce these pipeline bubbles, idle gaps between dependent stages, by overlapping computation and coordinating resource allocation (Mei et al., 2025; Sheng et al., 2025; Zhong et al., 2025b; Fu et al., 2025; Han et al., 2026). Asynchronous execution overlaps rollout and learning across updates, but some trajectories come from older policies. This policy staleness can require extra algorithmic machinery, including importance-weighted off-policy correction, filtering, clipping, or explicit staleness control (Espeholt et al., 2018; Fu et al., 2025; Han et al., 2026; Hou et al., 2026). On-policy pipelines overlap execution within a fixed rollout, but complete-group schedules still delay trajectory-local backward passes until all group rewards arrive (Chen et al., 2026b).

We argue that the apparent trade-off between pipeline overlap and zero staleness is artifactual, arising from coarse execution barriers. Although a synchronous update requires the complete batch, its gradient computation can begin earlier. For example, in group relative policy optimization (GRPO) (Shao et al., 2024), group rewards determine how to weight each trajectory's score gradient, but the gradient can be computed before all rewards arrive. We therefore ask: *can learner computation begin as soon as its objective dependencies are satisfied with zero staleness?*

We introduce *ThunderSyncRL*, which streams gradients through dependency-aware scheduling of learner work. During update k, trajectories are sampled from π_{θ_k} and gradients are evaluated at θ_k. For GRPO, a trajectory's advantage seems to need every reward in its group. ThunderSyncRL nevertheless starts that trajectory's backward pass as soon as its own reward arrives, and two running sums of score gradients recover the exact group gradient once the group closes (Section 3.4). For on-policy distillation (OPD) (Agarwal et al., 2024), ThunderSyncRL releases a turn's student backward pass once a frozen teacher scores the turn's action tokens, allowing overlap with tool execution and later turns. After all trajectories, groups, and gradient work finish, ThunderSyncRL combines the gradients, takes one optimizer step, and publishes θ_{k+1}. ThunderSyncRL is therefore asynchronous in execution and synchronous in policy updates.

We use SWE-smith and SETA-Env for post-training (Yang et al., 2025; Shen et al., 2026) and evaluate on SWE-bench Verified and Terminal Bench 4.0 (Jimenez et al., 2024; Merrill et al., 2026). Our scheduling analysis predicts step wall time speedup approaching $2\times$ over synchronous execution when rollout and learner work are balanced and the residual tail and publication cost vanish. With a fixed trajectory batch size, shorter steps let ThunderSyncRL learn from more trajectories within the same GPU-hours, potentially improving end-to-end performance. Figure 1 compares ThunderSyncRL with Sync by training cost: each arrow marks where ThunderSyncRL reaches the pass@3 of a Sync checkpoint, at 1.6 to $1.9\times$ lower cost.

Our main contributions are as follows:

- We introduce ThunderSyncRL and gradient streaming, overlapping rollout and learner computation while preserving a synchronous optimizer step and zero policy staleness.
- We derive exact gradient reductions for GRPO and turn-level OPD, allowing early backward computation while preserving their batch-synchronous updates.
- We evaluate ThunderSyncRL on SWE-bench Verified and Terminal Bench 4.0 under GRPO and OPD. ThunderSyncRL reaches the common pass@3 targets up to 1.9× faster than Sync while retaining zero policy staleness. At 128 GPU-hours, ThunderSyncRL achieves the highest pass@3 in all settings, 1.27 to 2.47 points above Async.

2 Related Work

2.1 Post-training with Reinforcement Learning Towards Agentic Tasks

RL post-training extends single-turn generation to agents that learn through long-horizon interaction with external environments. ArCHer develops hierarchical multi-turn RL, while DigiRL combines offline and online RL for mobile-device control (Zhou et al., 2024; Bai et al., 2024). Web and search agents learn through self-evolving curricula, browser interaction, and outcome rewards (Qi et al.,

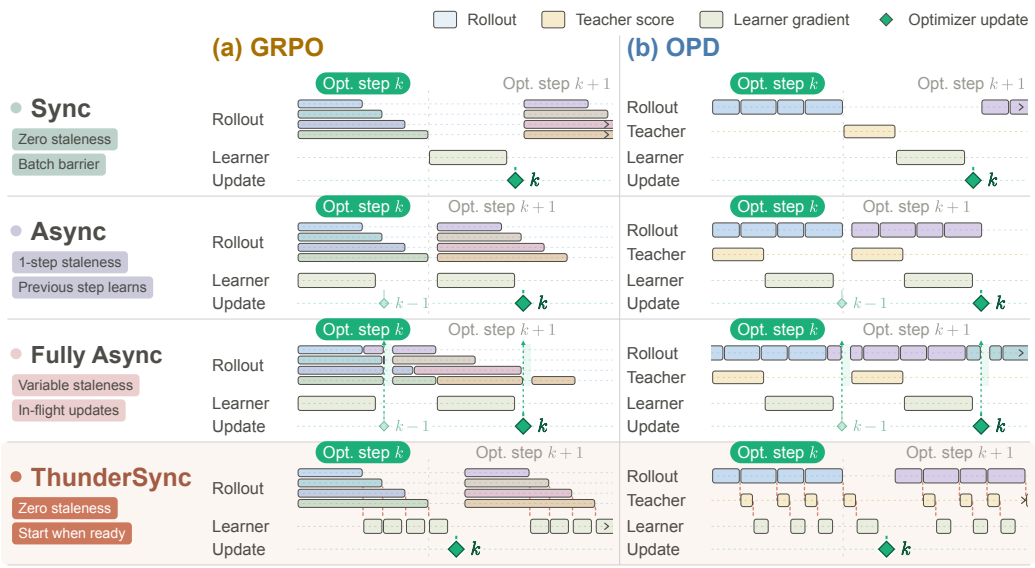

Figure 2: Schematic schedules for GRPO (a) and OPD (b). • **Sync** waits for full batches; • **Async** overlaps adjacent batches with one-step staleness; • **Fully Async** updates rollout weights while trajectories remain in flight. • **ThunderSyncRL** streams ready gradients with one batch-synchronous optimizer step and zero staleness.

2025; Wei et al., 2025b; Jin et al., 2025; Song et al., 2025; Chen et al., 2025; Zheng et al., 2025). Tool-use research studies strategic interpreter use, fine-grained rewards, planning supervision, and dynamic user interaction (Feng et al., 2026; Qian et al., 2025; Li et al., 2025; Zhao et al., 2025). Further work addresses trajectory-level optimization failures, multi-step supervision, and procedural or capability-targeted training environments (Wang et al., 2025; Goldie et al., 2025; Ding et al., 2026; Sullivan et al., 2025; Kang et al., 2026). Software agents learn from executable environments, software-evolution data, and execution feedback (Pan et al., 2025; Wei et al., 2025a; Gehring et al., 2025). Agent Lightning and VerlTool provide reusable infrastructure across heterogeneous agent workflows (Luo et al., 2025; Jiang et al., 2026).

2.2 Optimization of Reinforcement Learning

Distributed RLHF systems coordinate generation, reference and reward inference, and actor and critic training, using model placement, device reallocation, and stage fusion to reduce idle time (Yao et al., 2023; Hu et al., 2025; Mei et al., 2025; Sheng et al., 2025; Zhong et al., 2025b). Building on actor-learner decoupling in distributed RL (Espeholt et al., 2018; Petrenko et al., 2020), LlamaRL, AReaL, AsyncFlow, and StreamRL support disaggregated generation and training, streaming execution, asynchronous weight synchronization, and control of policy staleness (Wu et al., 2025; Fu et al., 2025; Han et al., 2026; Zhong et al., 2025a). For heterogeneous agentic rollouts, PipelineRL and RolloutPipe pipeline rollout and learner execution (Piché et al., 2025; Chen et al., 2026b). JigsawRL, Libra, TideRL, and WAR address fine-grained scheduling, elastic resource allocation, long tails, and inference caching (Hu et al., 2026; Chen et al., 2026a; Ren et al., 2026; Xu et al., 2026). SAO performs asynchronous optimization from individually completed rollouts (Hou et al., 2026). ThunderSyncRL schedules learner computation as each objective source becomes ready, overlapping it with unfinished rollouts while preserving a fixed policy snapshot and a single synchronous optimizer step. The agent, reward, and optimization objective remain unchanged.

3 Methods

We present ThunderSyncRL, a gradient streaming framework for zero-stale agentic post-training (Figure 2). ThunderSyncRL releases learner computation when each source becomes *ready*. Exact gradient reduction and a synchronous optimizer barrier preserve the update semantics of GRPO and on-policy distillation (OPD) while allowing rollout and learning to overlap.

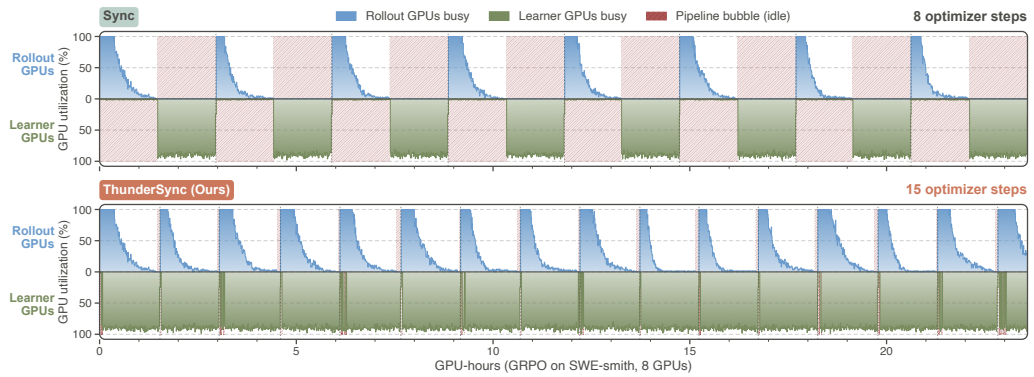

Figure 3: Pipeline bubbles in synchronous agentic RL. GPU utilization during GRPO training on SWE-smith, with four rollout GPUs above the axis and four learner GPUs below. Rollout utilization (busy share of generation slots) drops as long-tail trajectories finish. ▨ Hatched bands mark idle time. ● **Sync** learners wait for the last reward for 49.5% of each step, and rollout GPUs wait while the learner trains. ● **ThunderSyncRL** starts each trajectory's backward pass when its reward arrives and completes 15 optimizer steps in the GPU-hours Sync needs for 8.

3.1 Background

Group relative policy optimization (GRPO). Given a prompt, GRPO samples a group of trajectories and normalizes their rewards within the group to obtain relative advantages. The policy update weights each trajectory's scored action-token log probabilities by its detached advantage.

On-policy distillation (OPD). OPD generates trajectories with the student policy and evaluates the student's action tokens under a frozen teacher. The resulting token-level supervision defines a distillation loss for the action tokens of each agentic turn.

Actor-learner execution protocols. Agentic reinforcement learning alternates between two expensive stages: generating long, tool-using trajectories and optimizing the policy from the resulting training signals. Conventional synchronous training places a batch barrier between these stages, leaving the learner idle until every trajectory and reward is available (Figure 3). Fully asynchronous methods remove this idle time, but typically update the policy while older rollouts remain in flight, introducing policy staleness. ThunderSyncRL follows a single principle: each computation is released at the earliest time when its mathematical dependencies are satisfied.

3.2 Problem Formulation

Objective sources. Consider update k with policy $\pi_{\boldsymbol{\theta}_k}$ and logical rollout batch $\mathcal{B}_k$. Each trajectory is a sequence of agentic turns, where a turn contains a policy-generated action and the resulting environment or tool observation. We call an independently releasable unit of learner computation an *objective source*. Source j is *ready* at time τ_j once all its inputs are fixed.

Release semantics. The ready unit depends on the training objective. For a GRPO group $\mathcal{G}$ of m trajectories, source i is associated with trajectory i and becomes ready when its token sequence, scored-token mask, and reward r_i are fixed. The group statistics remain unavailable until all m rewards arrive. In OPD, the scored action tokens of each turn form a source. Once these tokens and their mask are fixed, the frozen teacher can score them using their realized causal context. The teacher scores release the corresponding student backward pass, which can overlap with tool execution and subsequent turns.

Scheduling objective. Our problem is to release each source at its earliest objective-valid time while preserving the corresponding batch-synchronous update. All trajectories in update k are sampled from $\pi_{\boldsymbol{\theta}_k}$, and all gradients are evaluated at the fixed parameters $\boldsymbol{\theta}_k$. After the logical batch closes and all gradient work finishes, the optimizer takes one step to produce $\boldsymbol{\theta}_{k+1}$. The logical batch and objective remain unchanged; only the timing of eligible learner work differs.

3.3 THUNDERSYNCRL FRAMEWORK

Release rule. ThunderSyncRL releases each source's learner computation at its ready time τ_j, which can precede the completion of its group or of the batch; gradients accumulate at $\boldsymbol{\theta}_k$.

Readiness-driven scheduling. Ready sources enter a work-conserving queue in readiness order; admission never waits for a smaller trajectory index, or for an unrelated group below a per-rank cap (Appendix D). Independent groups may be reduced concurrently, while each group preserves a deterministic reduction order. We include deferred gradient propagation at trajectory or group closure as learner work with its own release time. Index learner work items in service order; let τ_j and p_j denote the release time and service time of item j. On a single learner lane with $c_0 = 0$, completion time satisfies

$$c_j = \max\{\tau_j, c_{j-1}\} + p_j. \tag{1}$$

For n learner work items, let R_k be the time at which the complete rollout batch closes and P_k the common final reduction, optimizer, and publication suffix. The update endpoint is

$$T_k = \max\{R_k, c_n\} + P_k. \tag{2}$$

A batch-synchronous method postpones all learner work until R_k. For GRPO, a group-ready pipeline releases all sources in a group at its final reward. ThunderSyncRL uses each source's ready time for either objective, moving eligible learner work into rollout intervals and reducing the learner tail after the final trajectory.

Zero-stale optimizer barrier. ThunderSyncRL waits for every registered trajectory and group to close and for all deferred gradient propagation to finish. Distributed reduction and objective-specific normalization then produce the batch update direction $\boldsymbol{g}_k$ (Section 3.4). ThunderSyncRL clips this direction and applies exactly one optimizer step,

$$\boldsymbol{\theta}_{k+1} = \mathrm{Opt}\left(\boldsymbol{\theta}_k, \mathrm{clip}(\boldsymbol{g}_k)\right). \tag{3}$$

ThunderSyncRL publishes the resulting adapter or policy parameters to the rollout replicas. Each replica loads the new version, clears policy-dependent state, and acknowledges that it serves $\boldsymbol{\theta}_{k+1}$ before the next update begins. Every rollout and learner source within update k is therefore bound to $\boldsymbol{\theta}_k$, and no trajectory is optimized against a policy that changed while it was in flight. Appendix I gives the full algorithm and the GRPO and OPD implementations.

3.4 OBJECTIVE-SPECIFIC GRADIENT STREAMING

Streaming the GRPO gradient. The main obstacle to streaming the GRPO gradient is that the advantage of one trajectory depends on the rewards of its entire group. Let $\mathcal{G}$ be a group of m trajectories. For trajectory $i \in \mathcal{G}$, define its masked policy score and corresponding source gradient as

$$S_i(\boldsymbol{\theta}) = \sum_{t \in \mathcal{M}_i} w_{i,t} \log \pi_{\boldsymbol{\theta}}(a_{i,t} \mid \boldsymbol{h}_{i,t}), \qquad \boldsymbol{H}_i = \nabla_{\boldsymbol{\theta}} S_i(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_k}, \tag{4}$$

where $\mathcal{M}_i$ is the scored action-token set, $w_{i,t}$ the token weighting, and r_i the trajectory reward. Let

$$\mu_{\mathcal{G}} = \frac{1}{m}\sum_{i \in \mathcal{G}} r_i, \qquad \sigma_{\mathcal{G}} = \sqrt{\frac{1}{m}\sum_{i \in \mathcal{G}} (r_i - \mu_{\mathcal{G}})^2}. \tag{5}$$

With the group-normalized advantages held constant during differentiation, the group update direction is

$$\boldsymbol{g}_{\mathcal{G}} = \frac{1}{m}\sum_{i \in \mathcal{G}} \frac{r_i - \mu_{\mathcal{G}}}{\sigma_{\mathcal{G}} + \epsilon} \boldsymbol{H}_i. \tag{6}$$

Although $\mu_{\mathcal{G}}$ and $\sigma_{\mathcal{G}}$ require group completion, the gradient decomposes exactly as

$$\boldsymbol{g}_{\mathcal{G}} = \frac{1}{m}\frac{\boldsymbol{G}^{(1)}_{\mathcal{G}} - \mu_{\mathcal{G}} \boldsymbol{G}^{(2)}_{\mathcal{G}}}{\sigma_{\mathcal{G}} + \epsilon}, \qquad \boldsymbol{G}^{(1)}_{\mathcal{G}} = \sum_{i \in \mathcal{G}} r_i \boldsymbol{H}_i, \qquad \boldsymbol{G}^{(2)}_{\mathcal{G}} = \sum_{i \in \mathcal{G}} \boldsymbol{H}_i. \tag{7}$$

ThunderSyncRL computes $\boldsymbol{H}_i$ as soon as trajectory i becomes ready and incrementally accumulates $\boldsymbol{G}^{(1)}_{\mathcal{G}}$ and $\boldsymbol{G}^{(2)}_{\mathcal{G}}$. When all rewards and source gradients in the group are available, the scalar-weighted combination in Equation 7 gives $\boldsymbol{g}_{\mathcal{G}}$. Summing the group directions gives $\boldsymbol{g}_k = \sum_{\mathcal{G} \subset \mathcal{B}_k} \boldsymbol{g}_{\mathcal{G}}$ for the optimizer barrier. This preserves the batch-barrier update while allowing trajectory backward passes to begin before group completion.

Streaming the OPD gradient. In OPD, each turn provides a gradient source once the frozen teacher has scored its action tokens. Let $\mathcal{U}_k$ be the set of turns in $\mathcal{B}_k$, and let $\mathcal{M}_j$ contain the scored action tokens of turn j. We normalize the OPD update by the total number of scored tokens in the batch. Define the detached token advantage and the turn's unnormalized update direction as

$$A_{j,t} = \mathrm{sg}[\log \pi_{\mathrm{T}}(a_{j,t} \mid \boldsymbol{h}_{j,t}) - \log \pi_{\boldsymbol{\theta}_k}(a_{j,t} \mid \boldsymbol{h}_{j,t})] ,$$
$$\boldsymbol{g}_j = \sum_{t \in \mathcal{M}_j} A_{j,t} \, \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_{j,t} \mid \boldsymbol{h}_{j,t})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_k} , \tag{8}$$

where π_{T} is the frozen teacher and sg denotes stop-gradient. Each advantage depends only on the sampled token, its realized causal context, and the fixed student and teacher policies. ThunderSyncRL releases the turn-local backward pass as soon as the teacher scores are available, allowing learner computation to overlap with tool execution and subsequent turns. It accumulates unnormalized gradient contributions and completes any deferred gradient propagation through earlier context before final reduction. The batch update direction is

$$\boldsymbol{g}_k = \frac{1}{N_k} \sum_{j \in \mathcal{U}_k} \boldsymbol{g}_j, \qquad N_k = \sum_{j \in \mathcal{U}_k} |\mathcal{M}_j|. \tag{9}$$

The total scored-token count N_k is fixed only when the logical batch closes, so ThunderSyncRL defers this scalar normalization while turn-level backward passes proceed. Dividing the accumulated gradient by N_k before clipping and the single optimizer step gives the same mathematical update as batch-synchronous OPD, with every rollout and learner computation bound to $\boldsymbol{\theta}_k$. Appendix G confirms both equivalences numerically on B200 and H200 GPUs.

4 Experiments

4.1 Setup

Models, datasets and benchmarks.

- **Models:** We use Qwen-3.8-27B (Qwen Team, 2026b;a) for full fine-tuning and the separate LoRA configuration (Appendix H.2). The model comparison also includes GLM-4.7-Flash-31B (Z.ai, 2026; GLM Team et al., 2025). Appendix D specifies the Qwen OPD teacher, which remains fixed across execution schedules.
- **Datasets:** We use SWE-smith (Yang et al., 2025) and SETA-Env (Shen et al., 2026) for post-training, covering repository-level software engineering and broader terminal tasks, respectively. Appendix C details data revisions, filtering, splits, and benchmark separation.
- **Benchmarks:** We evaluate on SWE-bench Verified (Jimenez et al., 2024) and Terminal Bench 4.0 (Merrill et al., 2026), using the same tasks, agent interface, and inference budget across execution schedules. Terminal Bench 4.0 tasks are excluded from post-training. Appendix E specifies success criteria, pass@3 aggregation, and checkpoint comparisons. We also report pass@1 and pass@5 on both benchmarks in Appendix H.1.

Workloads and configurations. Each GRPO update uses 32 prompts with eight trajectories sampled per prompt, totaling 256 trajectories. Each OPD update uses 32 prompts with four independent trajectories sampled per prompt, totaling 128 trajectories. Both objectives allow up to 300 agentic turns and 1,024 generated tokens per turn. The reference configuration uses a single node with eight NVIDIA B200 GPUs; the hardware comparison also includes a node with eight NVIDIA H200 GPUs. Appendix D provides the full optimization and generation settings.

Baselines. Within each objective, we hold the model, workload, optimizer, precision, and hardware allocation fixed and compare ThunderSyncRL against three execution schedules:

- **Sync:** A zero-stale baseline that starts learning after the complete rollout batch is available.
- **Async:** A one-update-stale baseline that overlaps learning with the next rollout batch.
- **Fully Async:** Rollout and learning proceed without a batch-level barrier between them. Policy updates can occur during in-flight trajectories, yielding variable policy lag.

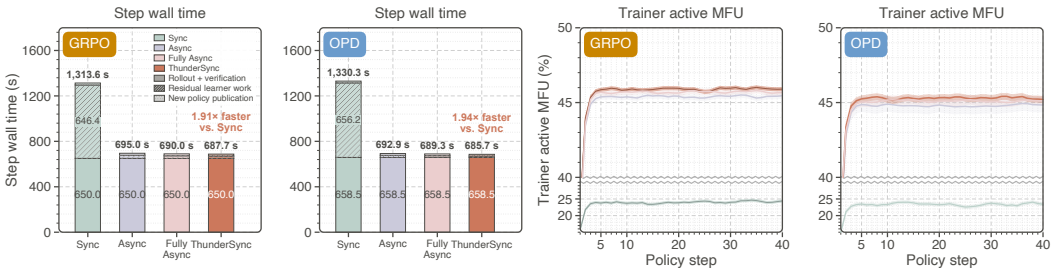

Figure 4: Step wall time (left) and trainer active MFU (right) for GRPO and OPD.

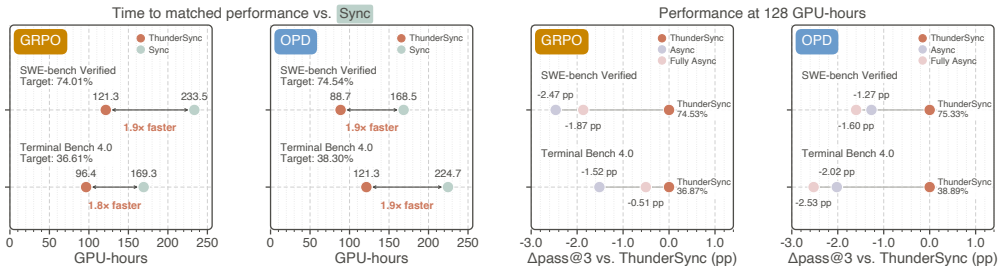

Figure 5: Left: time to matched pass@3. Right: Async and Fully Async relative to ThunderSyncRL at 128 GPU-hours.

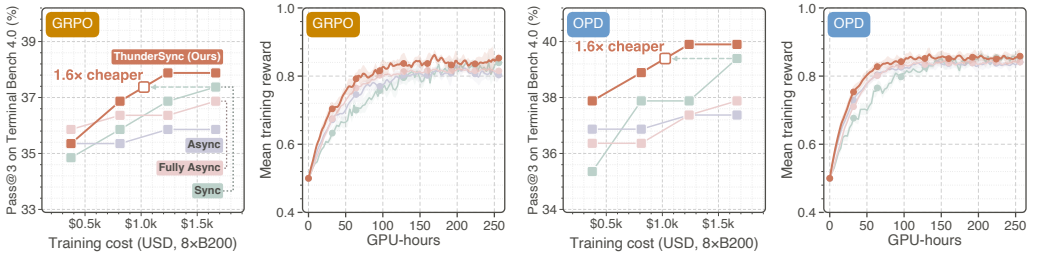

Figure 6: Terminal Bench 4.0 pass@3 vs. training cost and mean training reward vs. GPU-hours. Arrows mark where ThunderSyncRL reaches the score of a Sync checkpoint; labels give the cost ratio.

Sync and Async use the official verl (Sheng et al., 2025) pipeline for GRPO and the SkyRL (Cao et al., 2025) pipeline for OPD. Sync, Async, and ThunderSyncRL use vLLM for rollout, with the same chunked prefill and prefix caching settings. We evaluate with the benchmarks' official harnesses; Appendices D.2 and E describe the rollout pipelines and harnesses.

4.2 Learning efficiency and agent performance

Time to matched performance. ThunderSyncRL reaches the target pass@3 on SWE-bench Verified $1.9\times$ faster than Sync under both GRPO and OPD (Figure 5, left). For example, the GRPO target of 74.01% is first reached at 121.3 GPU-hours with ThunderSyncRL and 233.5 GPU-hours with Sync. The advantage also holds when training on SETA-Env and evaluating on Terminal Bench 4.0, whose tasks are excluded from post-training: ThunderSyncRL reaches the targets $1.8\times$ faster under GRPO and $1.9\times$ faster under OPD (Figure 5, left; curves in Figure 6).

Performance at a fixed training budget. At the same 128-GPU-hour training budget (16 hours on the eight-GPU node), ThunderSyncRL achieves higher pass@3 than Async on both benchmarks (Figure 5, right). On SWE-bench Verified, ThunderSyncRL leads by 2.47 percentage points under GRPO and 1.27 under OPD. On Terminal Bench 4.0, it leads by 1.52 percentage points under GRPO and 2.02 under OPD. Each score comes from the latest completed checkpoint available within the budget; Appendix E details the evaluation protocol.

Step wall time. ThunderSyncRL overlaps learner work with rollout (Figure 4, left). After 650.0 s (GRPO) and 658.5 s (OPD) of rollout and verification, Sync adds 646.4 and 656.2 s of learner work, ThunderSyncRL only 20.5 and 11.6 s. With publication, its steps take 687.7 and 685.7 s, $1.91\times$ and $1.94\times$ faster than Sync and within Async's seed spread, at zero policy lag. Over a run, shorter steps

Table 1: Full fine-tuning efficiency on SWE-smith across models using NVIDIA B200 GPUs. GRPO and OPD are compared separately per configuration; for OPD, the model is the student.

Method	Policy lag (staleness) ↓	GRPO			OPD		
		Avg. trainer active MFU (%) ↑	Speedup (×) ↑	Trainer peak GPU memory (GB) ↓	Avg. trainer active MFU (%) ↑	Speedup (×) ↑	Trainer peak GPU memory (GB) ↓
Qwen-3.8-27B							
Sync	**0**	24.0 ± 0.3	1.00 ± 0.00	$\mathbf{117.0 \pm 0.8}$	23.4 ± 0.4	1.00 ± 0.00	$\mathbf{118.0 \pm 0.9}$
Async	1	45.4 ± 0.2	1.89 ± 0.03	117.2 ± 1.1	44.8 ± 0.5	1.92 ± 0.04	119.3 ± 1.0
Fully Async	Variable	45.7 ± 0.2	1.90 ± 0.03	118.6 ± 1.2	45.1 ± 0.3	1.93 ± 0.04	120.1 ± 1.1
ThunderSyncRL	**0**	$\mathbf{45.9 \pm 0.1}$	$\mathbf{1.91 \pm 0.04}$	122.8 ± 1.3	$\mathbf{45.3 \pm 0.2}$	$\mathbf{1.94 \pm 0.03}$	121.3 ± 1.2
GLM-4.7-Flash-31B							
Sync	**0**	12.2 ± 0.1	1.00 ± 0.00	$\mathbf{124.0 \pm 1.0}$	11.5 ± 0.1	1.00 ± 0.00	$\mathbf{122.1 \pm 0.8}$
Async	1	19.8 ± 0.2	1.62 ± 0.03	$\mathbf{124.0 \pm 1.4}$	18.4 ± 0.2	1.60 ± 0.03	122.5 ± 1.3
Fully Async	Variable	19.9 ± 0.3	1.63 ± 0.04	125.2 ± 1.3	18.5 ± 0.2	1.61 ± 0.03	123.4 ± 1.2
ThunderSyncRL	**0**	$\mathbf{20.0 \pm 0.2}$	$\mathbf{1.64 \pm 0.03}$	129.4 ± 1.9	$\mathbf{18.7 \pm 0.1}$	$\mathbf{1.63 \pm 0.02}$	124.1 ± 1.5

Table 2: Full fine-tuning efficiency on SWE-smith across GPU platforms with Qwen-3.8-27B.

Method	Policy lag (staleness) ↓	GRPO			OPD		
		Avg. trainer active MFU (%) ↑	Speedup (×) ↑	Trainer peak GPU memory (GB) ↓	Avg. trainer active MFU (%) ↑	Speedup (×) ↑	Trainer peak GPU memory (GB) ↓
NVIDIA B200 × 8							
Sync	**0**	24.0 ± 0.3	1.00 ± 0.00	$\mathbf{117.0 \pm 0.8}$	23.4 ± 0.4	1.00 ± 0.00	$\mathbf{118.0 \pm 0.9}$
Async	1	45.4 ± 0.2	1.89 ± 0.03	117.2 ± 1.1	44.8 ± 0.5	1.92 ± 0.04	119.3 ± 1.0
Fully Async	Variable	45.7 ± 0.2	1.90 ± 0.03	118.6 ± 1.2	45.1 ± 0.3	1.93 ± 0.04	120.1 ± 1.1
ThunderSyncRL	**0**	$\mathbf{45.9 \pm 0.1}$	$\mathbf{1.91 \pm 0.04}$	122.8 ± 1.3	$\mathbf{45.3 \pm 0.2}$	$\mathbf{1.94 \pm 0.03}$	121.3 ± 1.2
NVIDIA H200 × 8							
Sync	**0**	31.3 ± 0.4	1.00 ± 0.00	$\mathbf{117.1 \pm 0.9}$	29.8 ± 0.4	1.00 ± 0.00	118.5 ± 1.0
Async	1	50.4 ± 0.5	1.61 ± 0.04	117.5 ± 1.2	49.2 ± 0.4	1.65 ± 0.03	$\mathbf{118.2 \pm 1.1}$
Fully Async	Variable	50.7 ± 0.4	1.62 ± 0.03	118.9 ± 1.3	49.5 ± 0.5	1.66 ± 0.04	119.0 ± 1.1
ThunderSyncRL	**0**	$\mathbf{51.0 \pm 0.3}$	$\mathbf{1.63 \pm 0.03}$	121.9 ± 1.4	$\mathbf{49.8 \pm 0.4}$	$\mathbf{1.67 \pm 0.03}$	120.4 ± 1.3

fit more trajectory batches into the same GPU-hours, which is why ThunderSyncRL reaches each pass@3 level at lower training cost (Figures 1 and 6).

4.3 Efficiency across models and hardware

The speedup follows how the stages balance on each model and GPU (Section 4.4). On B200, Qwen-3.8-27B is near balance and reaches $1.91\times$ and $1.94\times$ over Sync under GRPO and OPD. GLM-4.7-Flash-31B activates about 3B parameters, so its learner work is short, rollout bounds the step, and the speedup is $1.64\times$ and $1.63\times$ (Table 1). On H200, the same Qwen workload becomes learner-bound and reaches $1.63\times$ and $1.67\times$ (Table 2). Each comparison fixes the workload and GPU allocation across schedules, using Sync on the same hardware as the baseline.

Trainer active MFU divides learner FLOPs by the complete update wall time and by trainer GPU capacity. The window covers rollout waiting, environment execution, teacher scoring, learner work, and policy publication, counting overlap once, so overlapping learner work with rollout raises the metric by reducing trainer idle time. ThunderSyncRL has the highest average active MFU in every configuration, from 18.7% for GLM to 51.0% for Qwen on H200. H200's Sync MFU exceeds B200's because the learner, not rollout, bounds its step, so Sync's learner idles less. Figure 4 (right) shows the Qwen/B200 curves stabilizing after startup; Appendix B defines the FLOP accounting.

ThunderSyncRL's peak GPU memory exceeds Sync's in every configuration, at most from 117.0 to 122.8 GB per learner GPU (Qwen, GRPO). These peaks exclude the pinned host RAM holding the FP32 accumulator and GRPO group statistics (Appendix D); rollout and teacher inference use separate GPUs. Table 7 reports LoRA (Appendix H.2).

4.4 Workload scaling and overlap efficiency

We show how ThunderSyncRL approaches its $2\times$ speedup bound across stage balance, trajectory heterogeneity, batch size, and horizon; Appendix F derives this bound and the release laws. Every comparison in this section holds the workload and GPU allocation fixed between Sync and ThunderSyncRL, so no speedup comes from a change in workload or allocation.

Computation balance. Equal stage times offer the largest saving: complete overlap halves their combined duration, giving $2\times$ speedup when final reduction, optimizer, and publication overhead is negligible. If one stage dominates, it limits the saving. Appendix F.6 sweeps the ratio of rollout time to learner time at full capacity; one denotes equal times. We hold this balance below.

Trajectory heterogeneity. Agentic trajectories vary in length, and Sync's learner waits for the slowest (Figure 3). ThunderSyncRL starts a trajectory's backward pass once its reward arrives, so the learner works while rollout continues. In a separate scaling setup, over 150 measured steps per

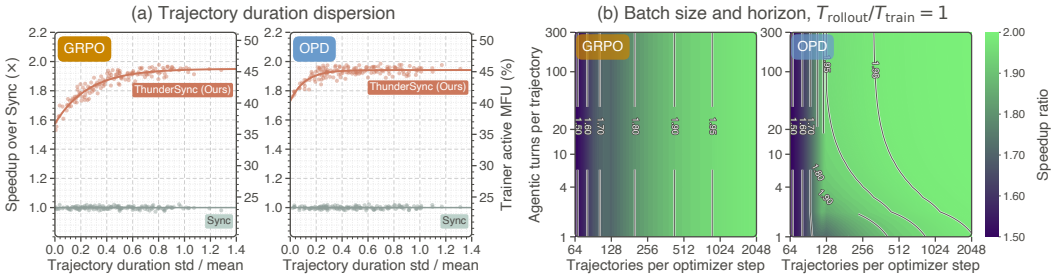

Figure 7: (a) Speedup and trainer active MFU of measured steps versus trajectory duration dispersion, with least-squares fits. (b) Predicted speedup at $T_{\text{rollout}}/T_{\text{train}} = 1$.

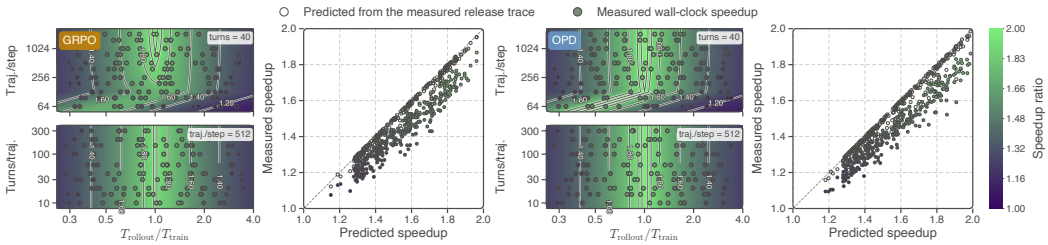

Figure 8: Speedup over Sync. Contours show the analytical prediction across batch size (upper) and horizon (lower). Scatter panels pool both sweeps and plot this prediction against predictions from measured release traces (open circles) and measured wall-clock speedups (filled circles).

objective at balance (Figure 7a), GRPO's speedup rises from $1.58\times$ at near-equal durations, where only rollout waves overlap, to $1.95\times$ as the spread grows. OPD releases work per turn and stays at 1.73 to $1.94\times$.

Batch and horizon. Figure 7b varies trajectories per optimizer step and agentic turns per trajectory. Rollout capacity is fixed at 128 trajectories, so larger batches run in successive waves. ThunderSyncRL processes ready work from earlier waves while later waves run. At balance, only the final wave's learner work remains after rollout ends, forming the *learner tail*. As the batch grows, the tail and policy publication take a smaller share of the update, so the predicted speedup approaches $2\times$.

With the completion-time distribution held fixed, GRPO's predicted speedup does not change with turn count, because a trajectory's work is released only when its reward arrives. For OPD, more turns spread releases and shorten the tail, with diminishing gains limited by deferred work (Appendix F.2).

Prediction and overhead. For Figure 8, we measured 588 paired configurations across both objectives, varying batch size at 40 turns and horizon at 512 trajectories while sweeping the rollout-to-learner time ratio. These scaling experiments use a separate setup from the learning-curve runs; Appendix F.5 specifies allocation and concurrency. Predictions from measured release traces differ from analytical predictions by 1.6% on average and at most 10.6%; measured wall-clock speedups average 6.5% below the trace predictions. These gaps describe release-model error and the remaining runtime overhead, respectively (Appendix F.5).

5 Conclusion

We presented ThunderSyncRL, which overlaps rollout and learning with zero off-policy staleness. It streams ready gradients at a fixed policy snapshot and takes one optimizer step after the batch closes, preserving the synchronous GRPO and OPD updates. Gradient sufficient statistics let GRPO backward passes start before all group rewards arrive, and teacher-scored turns let OPD learning proceed during tool execution. Step wall time speedup can approach $2\times$ when rollout and learner work are balanced and the learner tail and update overhead are negligible. Agent training at frontier scale moves only as fast as experiments turn into evidence. By recovering idle learner time without changing the objective, ThunderSyncRL shortens that loop, so the same compute answers more research questions. Further discussion, including limitations, appears in Appendix A.

AI USE STATEMENT

We used generative AI tools to find related work, translate parts of the manuscript, polish its language, organize existing code, and check consistency between the manuscript and implementation. The authors developed the research ideas, designed and refined the experiments, made implementation decisions, wrote the substantive experimental code, ran the experiments, and drew the scientific conclusions without generative AI assistance. Other required disclosure tasks do not apply to this work. We reviewed all AI-assisted work, including manuscript edits, and checked code organization and manuscript consistency against our implementation and experimental procedures. We take responsibility for the final content, including AI-assisted text, claims, and artifacts.

ETHICS STATEMENT

This work studies the efficiency of agentic post-training using existing software-engineering and terminal-task datasets and benchmarks. Faster post-training can lower computational costs, but it can also make it easier to train agents that execute harmful code or misuse tools. Our evaluation measures task performance and training efficiency; it does not establish that the resulting agents are safe to deploy. Deployment requires controls on tool permissions and execution environments, together with assessment of misuse and data-related risks.

REPRODUCIBILITY STATEMENT

Section 3.4 gives the objective definitions and gradient reductions, and Appendix F provides the scheduling analysis and its assumptions. Appendix C documents data revisions, filtering, development splits, and benchmark separation. Appendix D specifies optimization, generation, model, and hardware settings. Appendices E and B describe the evaluation and profiling protocols, including checkpoint selection and metric definitions. Appendix I provides the algorithm and implementation pseudocode for GRPO and OPD, including shared-prefix backward and the synchronous update barrier.

REFERENCES

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *The Twelfth International Conference on Learning Representations*, 2024.

Hao Bai, Yifei Zhou, Mert Cemri, Jiayi Pan, Alane Suhr, Sergey Levine, and Aviral Kumar. DigiRL: Training in-the-wild device-control agents with autonomous reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 12461–12495, 2024.

Shiyi Cao, Sumanth Hegde, Dacheng Li, Tyler Griggs, Shu Liu, Eric Tang, Jiayi Pan, Xingyao Wang, Akshay Malik, Graham Neubig, Kourosh Hakhamaneshi, Richard Liaw, Philipp Moritz, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. SkyRL-v0: Train real-world long-horizon agents via reinforcement learning, 2025.

Kaiwen Chen, Xin Tan, Jingzong Li, Zhi Zhou, Cen Li, Jiang Liu, Meng Jie, Jiazhi Jiang, and Hong Xu. Libra: Efficient resource management for agentic RL post-training. *arXiv preprint arXiv:2606.03077*, 2026a.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Mingyang Chen, Linzhuang Sun, Tianpeng Li, Haoze Sun, Yijie Zhou, Chenzheng Zhu, Haofen Wang, Jeff Z. Pan, Wen Zhang, Huajun Chen, Fan Yang, Zenan Zhou, and Weipeng Chen. Re-Search: Learning to reason with search for LLMs via reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 85287–85307, 2025.

Rongjian Chen, Jianmin Hu, Kejiang Ye, and Minxian Xu. RolloutPipe: Overlapping pipelined rollout and training in disaggregated on-policy LLM reinforcement learning. *arXiv preprint arXiv:2606.26997*, 2026b.

Yifeng Ding, Hung Le, Songyang Han, Kangrui Ruan, Zhenghui Jin, Varun Kumar, Zijian Wang, and Anoop Deoras. Empowering multi-turn tool-integrated agentic reasoning with group turn policy optimization. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, pp. 42409–42423, 2026.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymyr 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*, pp. 1407–1416, 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. In *International Conference on Learning Representations*, pp. 37909–37926, 2026.

Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, et al. Areal: A large-scale asynchronous reinforcement learning system for language reasoning. In *Advances in Neural Information Processing Systems*, pp. 36256–36282, 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: Mitigating long-tail rollouts for fast, synchronous RL post-training. *arXiv preprint arXiv:2509.21009*, 2025.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In *Forty-second International Conference on Machine Learning*, 2025.

GLM Team, Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. GLM-4.5: Agentic, reasoning, and coding (ARC) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

Anna Goldie, Azalia Mirhoseini, Hao Zhou, Irene Cai, and Christopher D Manning. Synthetic data generation and multi-step reinforcement learning for reasoning and tool use. In *Second Conference on Language Modeling*, 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 preprint arXiv:2507.01663*, 2026.

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

Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, Wenkai Fang, et al. Openrlhf: A ray-based easy-to-use, scalable and high-performance rlhf framework. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 656–666, 2025.

Zhengding Hu, Hehua Ouyang, Chang Chen, Zaifeng Pan, Yue Guan, Zhongkai Yu, Zhen Wang, Steven Swanson, and Yufei Ding. JigsawRL: Assembling RL pipelines for efficient LLM post-training. *arXiv preprint arXiv:2604.23838*, 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. *Transactions on Machine Learning Research*, 2026.

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 *International Conference on Learning Representations*, pp. 54107–54157, 2024.

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

Hangoo Kang, Tarun Suresh, Jon Saad-Falcon, and Azalia Mirhoseini. TRACE: Capability-targeted agentic training. *arXiv preprint arXiv:2604.05336*, 2026.

Zhiwei Li, Yong Hu, and Wenqing Wang. Encouraging good processes without the need for good answers: Reinforcement learning for LLM agent planning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 1654–1666, 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.

Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. ReaL: Efficient RLHF training of large language models with parameter reallocation. In *Proceedings of Machine Learning and Systems*, 2025.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten,

Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, pp. 40903–40986, 2026.

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 *Forty-second International Conference on Machine Learning*, 2025.

Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav Sukhatme, and Vladlen Koltun. Sample factory: Egocentric 3D control from pixels at 100000 FPS with asynchronous reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning*, pp. 7652–7662, 2020.

Alexandre Piché, Ehsan Kamalloo, Rafael Pardinas, Xiaoyin Chen, and Dzmitry Bahdanau. PipelineRL: Faster on-policy reinforcement learning for long sequence generation. *arXiv preprint arXiv:2509.19128*, 2025.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Jiadai Sun, Xinyue Yang, Yu Yang, Shuntian Yao, Wei Xu, Jie Tang, and Yuxiao Dong. WebRL: Training LLM web agents via self-evolving online curriculum reinforcement learning. In *International Conference on Learning Representations*, pp. 79791–79821, 2025.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tur, Gokhan Tur, and Heng Ji. Toolrl: Reward is all tool learning needs. In *Advances in Neural Information Processing Systems*, pp. 105523–105553, 2025.

Ruoyu Qin, Weiran He, Weixiao Huang, Yangkun Zhang, Yikai Zhao, Bo Pang, Xinran Xu, Yingdi Shan, Yongwei Wu, and Mingxing Zhang. Seer: Online context learning for fast synchronous LLM reinforcement learning. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 883–901, 2026.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork. `https://qwen.ai/blog?id=qwen3.8`, August 2026a.

Qwen Team. Qwen3.8-27B. `https://huggingface.co/Qwen/Qwen3.8-27B`, August 2026b. Hugging Face model card.

Yanyu Ren, Xizheng Wang, Xiao Liu, Bowen Lv, Hanchen Zhang, Shudan Zhang, Hanyu Lai, Shuai Wang, Li Chen, Dan Li, and Jie Tang. TideRL: Boosting agentic RL goodput with readiness-aware scheduling. *arXiv preprint arXiv:2608.10402*, 2026.

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.

Qijia Shen, Zhiqi Huang, Vamsidhar Kamanuru, Aznaur Aliev, Jay Rainton, Ahmed Awelkair, Zhichen Zeng, Jiajun Li, Shi Dong, Yueming Yuan, Boyuan Ma, Qizheng Zhang, Jiwei Fu, Yuzhen Mao, Wendong Fan, Ping Nie, Philip Torr, Bernard Ghanem, Changran Hu, Jonathan Lingjie Li, Urmish Thakker, and Guohao Li. SETA: Scaling environments for terminal agents. *arXiv preprint arXiv:2607.10891*, 2026.

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*, pp. 1279–1297, 2025.

Huatong Song, Jinhao Jiang, Yingqian Min, Jie Chen, Zhipeng Chen, Wayne Xin Zhao, Lei Fang, and Ji-Rong Wen. R1-Searcher: Incentivizing the search capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2503.05592*, 2025.

Michael Sullivan, Mareike Hartmann, and Alexander Koller. Procedural environment generation for tool-use agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 18544–18562, 2025.

Franklin Wang and Sumanth Hegde. Accelerating direct preference optimization with prefix sharing. In *NeurIPS 2024 Workshop on Fine-Tuning in Modern Machine Learning: Principles and Scalability*, 2024.

Jinghui Wang, Shaojie Wang, Yinghan Cui, Xuxing Chen, Chao Wang, Liang Huang, Can Tang, Xiaojiang Zhang, Junyi Peng, Li Wan, Haotian Zhang, and Bin Chen. Tree training: Accelerating agentic LLMs training via shared prefix reuse. *arXiv preprint arXiv:2511.00413*, 2026.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, Eli Gottlieb, Yiping Lu, Kyunghyun Cho, Jiajun Wu, Fei-Fei Li, Lijuan Wang, Yejin Choi, and Manling Li. RAGEN: Understanding self-evolution in LLM agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025a.

Zhepei Wei, Wenlin Yao, Yao Liu, Weizhi Zhang, Qin Lu, Liang Qiu, Changlong Yu, Puyang Xu, Chao Zhang, Bing Yin, Hyokun Yun, and Lihong Li. WebAgent-R1: Training web agents via end-to-end multi-turn reinforcement learning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 7909–7928, 2025b.

Bo Wu, Sid Wang, Yunhao Tang, Jia Ding, Eryk Helenowski, Liang Tan, Tengyu Xu, Tushar Gowda, Zhengxing Chen, Chen Zhu, Xiaocheng Tang, Yundi Qian, Beibei Zhu, and Rui Hou. LlamaRL: A distributed asynchronous reinforcement learning framework for efficient large-scale LLM training. *arXiv preprint arXiv:2505.24034*, 2025.

Ryan Xu, Atlas Zhao, David Bao, and Frank Du. WAR: Workload-aware rollouts for synchronous agentic reinforcement learning. *arXiv preprint arXiv:2607.17299*, 2026.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems*, pp. 107402–107452, 2025.

Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, Zhongzhu Zhou, Michael Wyatt, Molly Smith, Lev Kurilenko, Heyang Qin, Masahiro Tanaka, Shuai Che, Shuaiwen Leon Song, and Yuxiong He. DeepSpeed-Chat: Easy, fast and affordable RLHF training of ChatGPT-like models at all scales. *arXiv preprint arXiv:2308.01320*, 2023.

Z.ai. GLM-4.7-Flash. `https://huggingface.co/zai-org/GLM-4.7-Flash`, January 2026. Hugging Face model card.

Weikang Zhao, Xili Wang, Chengdi Ma, Lingbin Kong, Zhaohua Yang, Mingxiang Tuo, Xiaowei Shi, Yitao Zhai, and Xunliang Cai. MUA-RL: Multi-turn user-interacting agent reinforcement learning for agentic tool use. *arXiv preprint arXiv:2508.18669*, 2025.

Yuxiang Zheng, Dayuan Fu, Xiangkun Hu, Xiaojie Cai, Lyumanshan Ye, Pengrui Lu, and Pengfei Liu. DeepResearcher: Scaling deep research via reinforcement learning in real-world environments. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 414–431, 2025.

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 preprint arXiv:2504.15930*, 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 *22nd USENIX Symposium on Networked Systems Design and Implementation*, pp. 489–503, 2025b.

Yifei Zhou, Andrea Zanette, Jiayi Pan, Sergey Levine, and Aviral Kumar. ArCHer: Training language model agents via hierarchical multi-turn RL. In *Forty-first International Conference on Machine Learning*, 2024.

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 preprint arXiv:2509.18521*, 2025.

A Limitation and further discussion

ThunderSyncRL rests on one premise: learner work should begin as soon as the objective's dependencies allow. How much this saves is therefore set as much by the workload as by the scheduler. Objectives whose gradient inputs resolve early, such as turn-level OPD, leave more learner work to overlap with rollout than objectives that wait on group-level signals, and the benefit is largest when rollout and learner work take comparable time. Starting work early also means keeping partial gradient state until each group or batch closes, which exchanges trainer memory for wall time. We regard these dependencies as a design space. Finer readiness units and admission that weighs memory against remaining service would move more learner work into the rollout window, and identifying readiness units for other objectives would extend exact gradient streaming beyond GRPO and OPD. Scaling the same principle to multi-node training and a broader set of agentic workloads is a natural next step.

B Efficiency measurement

B.1 Trainer active MFU

Throughout this paper, trainer active MFU denotes the learner's model FLOPs divided by the complete update wall time and the peak capacity of the trainer GPUs. We use this label for the step-averaged trainer metric, whose time window includes rollout waiting and trainer idle intervals.

For update k, let F_k^{train} be the model FLOPs required by the learner, T_k^{step} its complete update wall time, and P_d the dense-BF16 peak rate used for trainer GPU d. We define

$$\mathrm{MFU}_k^{\mathrm{active}} = \frac{F_k^{\mathrm{train}}}{T_k^{\mathrm{step}} \sum_{d \in \mathcal{D}_{\mathrm{train}}} P_d}. \tag{10}$$

The capacity denominator includes the trainer GPUs, with the same peak-rate convention for all methods within a hardware configuration. Figure 4 and Tables 1, 2, and 7 use this definition.

The denominator uses the same wall-clock intervals as step wall time below. For Sync and ThunderSyncRL, an interval begins with the update's rollout and ends when the rollout replicas acknowledge the published policy. It includes environment and reward processing, teacher scoring, all learner work, final reduction, optimizer execution, and publication. Waiting for a source to become ready remains included. For Async, we use complete steady-state update cycles between successive policy-publication acknowledgements, counting concurrent stages once. For ThunderSyncRL, deferred context gradients and shared-prefix backward must finish within the measured update.

The FLOPs numerator follows the model architecture and the contexts processed by the learner. Context tokens contribute computation even when their loss mask is zero. For LoRA, the count includes forward computation and backward propagation through frozen weights, as well as adapter gradients. Activation-checkpoint recomputation contributes to elapsed time without being credited as additional model FLOPs. Rollout generation and teacher scoring contribute to elapsed time, but their FLOPs and GPU capacity are excluded from this trainer-side metric.

Operator accounting. For a matrix product with dimensions $(m \times k)(k \times n)$, the counter credits $2mkn$ FLOPs. A linear layer processing T tokens with input/output widths $d_{\mathrm{in}}, d_{\mathrm{out}}$ therefore costs $2Td_{\mathrm{in}}d_{\mathrm{out}}$ per forward pass. When both input and weight gradients are required, backward adds two such products. A frozen LoRA base omits the weight-gradient product but retains the input-gradient product wherever gradients propagate through it. The rank-r adapter's forward and backward products are counted separately.

The ledger sums executed attention projections, query-key and attention-value products, dense and recurrent blocks, and vocabulary projections using their actual tensor shapes. For GLM, routed-expert products use the number of tokens dispatched to each expert; shared experts and routing computation are included separately. Neither total stored parameters nor the nominal active-parameter count alone is an adequate FLOP counter. Padding that reaches a kernel counts as executed work. Each record keeps useful model work separate from checkpoint recomputation, so the latter increases elapsed time without inflating the credited numerator.

B.2 AVERAGE TRAINER ACTIVE MFU

For a reporting window of N completed updates after warmup, average trainer active MFU is

$$\overline{\mathrm{MFU}}^{\mathrm{active}} = \frac{1}{N}\sum_{k=1}^{N} \mathrm{MFU}_k^{\mathrm{active}}. \tag{11}$$

Each term uses the learner FLOPs and complete wall time of the same update cycle. With fixed learner work and trainer capacity, shorter updates increase trainer active MFU. Overlapping learner computation with rollout can therefore raise this metric without changing kernel throughput. Within one configuration, all schedules process the same learner FLOPs, so the MFU ratio to Sync tracks the step wall time speedup; across models and hardware, learner FLOPs differ.

Profiling window and aggregation. We completed 40 updates for each of seeds 42, 43, and 44, discarded updates 1–4 as startup, and profiled updates 5–40 ($N = 36$). Startup includes initial compilation and allocator/cache growth; it is distinct from learning-rate warmup. All schedules retain the same number of complete steady-state intervals, with extra pipeline startup/drain work recorded outside the Async measurement window. The wall-time denominator uses a common host timebase for complete update cycles. Gaps with no admitted learner work remain included; we do not replace this window with a union of active kernel intervals.

CUDA events recorded device service without synchronizing after every turn. We collected detailed operator counts in a separate untimed trace pass with the same workload and tensor shapes. Within each seed, MFU is averaged over the 36 updates and speedup is the ratio of mean step times. We report the mean and, where shown, sample standard deviation across the three seed-level summaries, rather than treating 108 correlated updates as independent runs.

B.3 STEP WALL TIME SPEEDUP

Step wall time is the elapsed wall-clock time per completed training update. The interval includes rollout, environment and reward processing, learner work, final reduction, and policy publication. It ends when the rollout replicas acknowledge the updated policy. For N completed updates, let $\bar{T}^{\mathrm{step}}_{\mathrm{method}}$ be elapsed time divided by N. Speedup for each objective and configuration is

$$\mathrm{Speedup}_{\mathrm{method}} = \frac{\bar{T}^{\mathrm{step}}_{\mathrm{Sync}}}{\bar{T}^{\mathrm{step}}_{\mathrm{method}}}, \tag{12}$$

where both times use the same workload, hardware allocation, and number of completed updates after warmup. This is a ratio of mean step wall times, not a mean of per-step ratios or a ratio of kernel execution times. For Async, the window covers completed steady-state update cycles with the declared one-update policy lag; overlapping stages contribute once to elapsed time.

B.4 HARDWARE ALLOCATION AND MEMORY

Each eight-GPU configuration assigns four GPUs to the learner. GRPO uses the remaining four for rollout; OPD uses two for rollout and two for the teacher. ThunderSyncRL replicates parameters, shards optimizer state, and accumulates gradients on the host. Appendix D gives the precision and execution settings. The dense-BF16 peak rates used in the MFU denominator are 2,250 TFLOP/s per B200 GPU and 989.5 TFLOP/s per H200 GPU.

We report two kinds of memory. GPU memory is the HBM on each learner GPU; host memory is the CPU RAM of the node. Tables 1 and 2 report trainer peak GPU memory per GPU. The reported peak is the largest among the learner GPUs during an update, including gradient accumulation, distributed reduction, and optimizer execution. It covers resident model and optimizer state, activations, gradient buffers, and temporary storage on those GPUs. Rollout KV caches and teacher memory belong to their separate GPU pools. The serving memory fraction of 0.65 controls the rollout executors' memory budget.

We reset each learner process's CUDA peak-memory counters at the start of the profiled update, after the persistent model and optimizer state is resident, and read them after all backward, reduction, and

optimizer work has completed. We report `torch.cuda.max_memory_allocated()` as the tensor allocation peak, taking the maximum across learner ranks and profiled updates within each seed, and convert bytes to decimal GB (10^9 bytes). We record reserved allocator memory and device-wide memory from NVML separately, since they include different allocations.[1] Deferred-state buffers that stay on the learner GPUs count toward ThunderSyncRL's GPU peak. The FP32 batch accumulator and the GRPO group statistics sit in pinned host memory, so this counter does not include them; Appendix D bounds their host-memory size. A model-state size or a serving-memory limit is not a measured peak. We use the same collection procedure for the full fine-tuning and LoRA comparisons.

C Training data and benchmark separation

C.1 Data sources and versions

The software-engineering workload uses SWE-smith, and the terminal workload uses SETA-Env. These are separate post-training workloads, evaluated on SWE-bench Verified and Terminal Bench 4.0, respectively. Table 3 fixes the latest public revisions inspected on September 6, 2026. Revisions stay fixed throughout a comparison, including benchmark task definitions and evaluation resources.

Table 3: Pinned data sources. Sizes describe the inspected source pools before the filtering and development split in Table 4.

Collection	Role	Tasks	Revision or release
SWE-smith Python	Post-training	50,908	`77cab9055d42`
SETA-Env	Post-training	4,567	`3c3bc8975b05`
SWE-bench Verified	Evaluation	500	`78f471bf655a`
Terminal Bench 4.0	Evaluation	66	`v4.0.0` / `452bf305c6da`

SWE-smith. SWE-smith constructs executable repository-repair tasks by introducing bugs into working codebases and identifying tests that fail after the change (Yang et al., 2025). Its construction methods include language-model rewrites, procedural edits, combined bugs, and pull-request mirroring. The original paper reports 50,137 instances from 128 repositories. The selected, maintained Python release contains 50,908 instances from 131 repositories.[2] Each record supplies a problem statement, a repository environment, a bug-inducing patch, and test identifiers. The environment setup must apply this patch before agent interaction; its direction is the opposite of the solution patch in SWE-bench. Existing repository tests can be visible during SWE-smith training, so training reward and performance on held-out repair tasks measure different things.

SETA-Env. SETA builds terminal environments from community questions and answers, Kaggle notebooks, and natural-language shell-command examples. SETA-Synth creates tasks with executable environments and verifiers; SETA-Evol changes their difficulty or context (Shen et al., 2026). The inspected release has 3,255 synthesized and 1,312 evolved task directories, matching the 4,567 tasks reported in its paper and dataset card.[3] Each task includes instructions, an environment definition, a reference solution, and executable tests. SETA's construction procedure checks a no-op solution and an oracle solution, then audits systematic rollout failures for task-design flaws. These upstream checks describe dataset construction; they do not replace validation of the environments used in ThunderSyncRL runs.

C.2 Filtering and development splits

The static SWE-smith filter requires a nonempty instance identifier, problem statement, repository identifier, and environment image, and retains one record per instance identifier. This removes 11,437 records, all with empty problem statements. The remaining 39,471 instances cover 131

[1] The CUDA memory APIs and the distinction between allocated and reserved memory are documented in the PyTorch memory-management notes.

[2] The selected source is the maintained SWE-smith Python release. The older monolithic `SWE-bench/SWE-smith` release has different contents and directs users to the language-specific collections.

[3] The source is `camel-ai/SETA-Env`.

repositories. Before comparing repository identities, the filter maps SWE-smith mirror names back to their original owner/repository names and normalizes case. None of these repositories matches the 12 repositories in the pinned SWE-bench Verified set.

For SETA, the static filter requires `task.toml`, instructions, an environment Dockerfile, a test entrypoint, and a reference-solution entrypoint. All 4,567 task directories pass this check. A source-family identifier removes the evolution suffix from an evolved task's identifier. Tasks in the same source family stay in one split, including evolved tasks whose original is absent from the release.

The development split contains 5% of groups, rounded up: repositories for SWE-smith and source families for SETA. Groups are ordered by the SHA-256 digest of the normalized group identifier; the first groups form the development set. The remaining groups form the training set. This produces Table 4. Task lists and their digests define the split independently of model performance or trajectory completion time. This five-percent group split is a ThunderSyncRL protocol choice, not an official development split supplied by either dataset.

Table 4: Statically constructed cohorts for the declared protocol, before prompt-length checks, container validation, and semantic-overlap review.

Stage	SWE-smith Python	SETA-Env
Source tasks	50,908	4,567
Removed by static field/file checks	11,437	0
Eligible tasks	39,471	4,567
Split groups	131 repositories	3,474 source families
Training tasks	37,084	4,333
Development tasks	2,387	234
Training/development groups	124 / 7	3,300 / 174

Executable validation. For SWE-smith, we ran the validation harness on the original repository, then applied the bug-inducing patch and tested it again. Its collection rule requires at least one repair test and one retained passing test; evaluation with the gold repair provides a separate harness sanity check.[4] For SETA, we started a fresh environment for each of the no-op and reference solutions, recorded the verifier output, and required the reference solution to pass without the no-op already solving the task. These checks validate the environment and reward signal, not the trained policy.

The run manifest records the task identifier, source revision, image digest, verifier revision, validation outcome, and any exclusion reason. Prompt-length filtering uses the pinned student tokenizer and the actual serialized system, tool, and task prompt before rollout. A failing task is removed from every schedule's cohort before shuffling; its removal updates the task count and list digest. Table 4 reports the static cohort construction; the execution manifest records subsequent validation exclusions and the final task lists used by each comparison.

A task's difficulty, a zero-variance GRPO group, or a long trajectory does not trigger exclusion. Although campaign-specific SETA launchers can select benchmark tasks or discard uniform-reward groups,[5] our comparison retains those groups and excludes Terminal Bench tasks from training. Within a seed, schedules draw prompts from the same shuffled training list without replacement, reshuffling when it is exhausted. The two datasets are not mixed, and development tasks contribute no training gradients.

C.3 Scope of benchmark separation

Excluding benchmark task identifiers is only one check on data separation. The SWE-smith paper describes using a SWE-bench Verified problem statement as a style demonstration during issue generation (Appendix D of Yang et al., 2025). Repository disjointness therefore does not establish that its construction was independent of benchmark text. For SETA, checks must cover instructions, reference solutions, tests, and source ancestry, including relationships to earlier Terminal Bench releases. The static checks above do not establish semantic decontamination. The overlap review

[4] See the SWE-smith validation and evaluation guide.

[5] See the versioned SETA GLM training launcher.

first matches task IDs, canonical repository or source IDs, and hashes of whitespace-normalized instructions, solutions, and tests. Candidate paraphrases and evolved descendants then require source inspection. A confirmed benchmark derivative excludes its entire source family before splitting; ambiguous cases remain recorded rather than being certified clean. Counts of candidates, confirmed overlaps, and exclusions accompany the final cohort manifest.

The protocol excludes benchmark tasks, solutions, verifier outputs, and identified derivatives from training, prompts, and development selection. Upstream data generation or model pretraining may still expose the student or teacher to benchmark content. Results therefore compare schedules with matched data and models, without establishing that every task was unseen throughout model development.

D Optimization and generation settings

D.1 Logical batches and optimization

Table 5 gives shared optimization settings for Sync, Async, and ThunderSyncRL within each objective. The main configuration updates the student's language-model parameters. The text-only agent interface does not use Qwen's vision components. The LoRA configuration is separate from full fine-tuning.

Table 5: Declared optimization settings. Each column is a separate comparison among execution schedules. The learning rate for the separate LoRA configuration is 10^{-5}.

Setting	GRPO	OPD
Prompts per optimizer update	32	32
Trajectories per prompt	8	4 independent samples
Trajectories per optimizer update	256	128
Optimizer	AdamW	AdamW
Full fine-tuning learning rate	10^{-6}	10^{-6}
Learning-rate schedule / warmup	Constant / none	Constant / none
Adam (β_1, β_2) / ϵ	$(0.9, 0.999)$ / 10^{-8}	$(0.9, 0.999)$ / 10^{-8}
Weight decay	0	0
Global gradient-norm clipping	1.0	1.0
Optimizer steps per logical batch	1	1
Additional PPO epochs	0	0
Reference-policy KL / entropy bonus	0 / 0	0 / 0
Loss-bearing tokens	Generated assistant tokens	Generated assistant tokens
Reduction	Mean over trajectories	Global scored-token mean
Advantage stabilization ϵ	10^{-6}	Not applicable
Student forward precision	BF16	BF16
Gradient accumulation / Adam state	FP32 (host) / BF16	FP32 (host) / BF16
Training seeds	42, 43, 44	42, 43, 44

The full fine-tuning learning rate, Adam moments, clipping threshold, and temperature-one sampling follow conventional GRPO settings, including the corresponding TRL defaults. They are fixed across schedules rather than selected separately for each method.[6] The constant learning rate, trajectory-mean GRPO reduction, and single logical update are explicit ThunderSyncRL choices; adopting an entire framework's default configuration would not preserve the objective defined here. We save the resolved configuration with each run's model and data revisions. For GRPO, let L_i be trajectory i's scored-token count. Setting $w_{i,t} = 1/(32L_i)$ in Section 3.4 makes the sum of group directions a mean over the 256 trajectory-normalized directions. Rewards are standardized within each eight-trajectory group using the population standard deviation. Uniform-reward groups contribute zero policy gradient and retain their place in the batch and timing record. The group-normalized advantages are held constant during differentiation, and each logical batch produces one optimizer step.

[6] See the TRL GRPO configuration. The table does not list all TRL, SWE-smith, or SETA defaults.

OPD applies the sampled-token log-probability difference in Equation 8, with both distributions evaluated at temperature one. Four trajectories per prompt provide independent samples; OPD has no group-relative reward normalization. Accumulation uses the global scored-token denominator in Equation 9. Both objectives normalize and reduce gradients across ranks before global clipping and the single optimizer step. Prompt and observation tokens supply causal context without direct loss.

For the Qwen LoRA configuration, the specified rank is 32, scaling parameter $\alpha = 64$, dropout is zero, and the base weights remain frozen. The targets are `q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, `down_proj`, `in_proj_qkv`, `in_proj_z`, `in_proj_b`, `in_proj_a`, and `out_proj`. Batch sizes, generation limits, and evaluation settings remain those of the corresponding objective.

D.2 Agent interaction and generation

Rollout pipelines. We use verl's official Agent Loop for GRPO and SkyRL's official multi-turn generation pipeline, through `SkyRLGymGenerator` and SkyRL-Gym, for OPD. Both use vLLM as the rollout engine. The agent loop alternates model generation with shell-tool execution in the task environment and retains the resulting history for the next turn. Independent trajectories run concurrently, so a tool call waiting for its result does not block generation for other trajectories.[7] The same agent interface and vLLM serving settings apply to Sync, Async, and ThunderSyncRL within each objective.

Table 6: Generation settings shared across schedules. Training and evaluation use the same per-attempt token and turn limits.

Setting	Value
Maximum agentic turns per trajectory	300
Maximum generated tokens per turn	1,024
Maximum total generated tokens per trajectory	16,384
Maximum initial prompt tokens	16,384
Maximum total context tokens	131,072
Maximum retained observation per tool call	4,096 tokens (head and tail equally)
Sampling temperature / top-p / top-k	1.0 / 1.0 / disabled
Repetition / presence / frequency penalties	1.0 / 0 / 0
Thinking mode	Enabled; reasoning tokens count toward the limits
Tool calls per agent turn	At most one
Ordinary tool-command timeout	120 seconds
Student rollout weights / KV cache	BF16 / BF16
Student serving memory fraction	0.65
Maximum active sequences per rollout replica	16
Chunked prefill / prefix caching	Enabled / enabled
Speculative decoding	Disabled
Evaluation attempts per task and checkpoint	3 (5 in Appendix H.1)

Shared serving optimizations. All schedules enable vLLM's chunked prefill and automatic prefix caching, and disable speculative decoding. Chunked prefill batches prompt chunks with decode requests. Prefix caching reuses matching cached prefixes, including retained conversation history across turns when the cache is available on the serving replica. Policy publication invalidates policy-dependent caches before requests use the new weights. These serving optimizations are common to the comparison; ThunderSyncRL changes when ready learner work is admitted.[8]

The limits in Table 6 are independent ceilings. In particular, 300 turns do not imply an allowance of $300 \times 1{,}024$ generated tokens once the trajectory-level limit is reached. Initial prompts exceeding their limit are recorded before training and excluded consistently across schedules. During a trajectory, reaching the generation or context limit ends the attempt; previous turns are not removed to create additional context capacity. The agent retains the first and last 2,048 tokens of longer tool outputs, with the same truncation rule in every arm. An unparsable action consumes its turn and returns a fixed parser-error observation. Each independent attempt starts from a fresh task environment.

[7] See the verl Agent Loop and SkyRL multi-turn rollout documentation.

[8] See vLLM's chunked prefill and automatic prefix caching documentation.

The agent interface supplies the task instruction and a common shell tool, with fixed system and tool descriptions within a model configuration. There is no retrieval from benchmark solutions or evaluator-controlled files. The teacher scores the student's serialized causal history, including its retained observations, rather than a separately rewritten conversation. Qwen reasoning history is preserved in that history. Sampling seeds are assigned by run seed, optimizer update, task identifier, trajectory index, and turn index, independently of arrival order. Matching these seeds does not imply identical trajectories once the policies differ, particularly for Async.

D.3 Teacher and hardware allocation

The teacher for Qwen-3.8-27B is Qwen/Qwen3.8-Flash-Next-FP8, revision `236dfdf28582`; the student is Qwen/Qwen3.8-27B, revision `1d4bf0f2ff60`. The identical tokenizer gives token-level correspondence for OPD. The teacher remains frozen, uses its released FP8 quantization configuration on both GPU types, and supplies log probabilities only for the sampled assistant tokens. Log-probability normalization and score transport use FP32; this does not undo approximation introduced by FP8 teacher weights. Teacher scoring uses the same causal prefix and scoring implementation across schedules, with work becoming eligible after each completed turn.

The hardware configurations are one node with eight B200 GPUs and one with eight H200 GPUs. The declared allocation reserves four GPUs for the student learner. GRPO uses the remaining four for rollout replicas; OPD uses two for rollout replicas and two for the teacher, with teacher tensor parallelism of two. Student replicas each use one GPU. ThunderSyncRL replicates the BF16 student on each of the four learner GPUs, shards the AdamW state across them, and accumulates gradients in FP32 in host memory, with activation checkpointing; this lets each rank release sources independently. Under GRPO, each open group also keeps its two FP32 statistics $\boldsymbol{G}_{\mathcal{G}}^{(1)}$ and $\boldsymbol{G}_{\mathcal{G}}^{(2)}$ in pinned host memory; they are allocated with the group's first source and freed when the group closes. An optional admission cap k defers the backward of a trajectory from a new group while k groups on its rank hold statistics, so only its tokens wait. The statistics and the batch accumulator then occupy at most $4(2k+1)P$ bytes of host memory per rank for P parameters, regardless of the number of prompts per batch. We set $k = 1$, which bounds the statistics and batch accumulator at 1.30 TB for Qwen and 1.49 TB for GLM across the four learner ranks. The Sync and Async baselines use the same GPU allocation and the optimization and precision settings of Table 5. With 16 active sequences per replica, the reference allocation permits up to 64 concurrently scheduled generation sequences for GRPO and 32 for OPD. These limits govern vLLM generation, separately from the number of trajectories in a logical update or waiting on tools. Environment and verifier resource limits and concurrency settings are fixed across schedules. The resolved launch configurations will accompany the code release. The workload-scaling experiments use a separate allocation and trajectory-capacity setup (Appendix F.5).

The Qwen teacher's released tensors occupy approximately 185.5 GB before quantization scales, caches, and runtime buffers. Its two-GPU allocation and scoring procedure remain fixed across schedules.

GLM configuration. The GLM student is `zai-org/GLM-4.7-Flash`, revision `7dd20894a642`, with about 31.22 billion stored parameters and approximately three billion active parameters per token. Its released configuration has 47 decoder layers, 64 routed experts per MoE layer, and four selected experts per token.[9] The GLM LoRA configuration uses $r = 32$, $\alpha = 64$, zero dropout, and the same 10^{-5} learning rate. It targets the attention projections `q_a_proj`, `q_b_proj`, `kv_a_proj_with_mqa`, `kv_b_proj`, and `o_proj`. Routed experts, embeddings, and the output head remain frozen. These targets give 42,063,872 adapter parameters over the 47 decoder layers. The adaptation rank is separate from the architecture's query/KV compression ranks.[10] Comparisons remain within each model's adapter configuration.

For GLM OPD, we use the frozen teacher `zai-org/GLM-5.3-Flash`, FP8 revision `690b705278a3`. Its tokenizer file is byte-identical to the student's. We score the student's unchanged token IDs and causal prefix, retain its reasoning and tool history, and return sampled-token log probabilities in

[9] See the pinned GLM student release.

[10] Projection names and shapes follow the Transformers GLM implementation. MoE expert tensors require separate targeting when adapted; see the PEFT LoRA reference.

FP32.[11] The teacher's tensors require about 328.25 GB before caches and runtime buffers. GLM OPD uses tensor parallelism over two B200 GPUs; the H200 comparison uses Qwen. Before timing, we checked capacity with the 131,072-token context, scoring buffers, and resident quantization state.

Our shared temperature of one differs from the GLM model card's SWE-bench/Terminal Bench setting of 0.7 at top-$p = 1$. Table 6 fixes the other generation limits; the comparison does not reproduce the model card's evaluation budget.

D.4 Comparison controls

Comparisons fix the objective, model, adaptation, initial checkpoint, training list, rewards, optimizer, precision, serving settings, environment resources, and hardware across schedules. GRPO and OPD differ in trajectory counts and supervision costs, so equal updates do not imply equal compute. Sync and ThunderSyncRL use one policy snapshot for rollout and learning. Async follows a *one-step-off* schedule: after startup, each training batch is sampled from a policy one optimizer update behind the current learner. Specifically, trajectories are sampled from $\pi_{\theta_{k-1}}$, while the learner evaluates the same surrogate at θ_k, without an additional behavior-policy importance correction. Holding the surrogate fixed makes this a controlled comparison of execution schedules rather than of staleness corrections such as the decoupled objective of AReaL (Fu et al., 2025). Fully Async follows the fully asynchronous trainers of verl (Sheng et al., 2025) for GRPO and SkyRL (Cao et al., 2025) for OPD, in the style of AReaL (Fu et al., 2025). Rollout and learner GPUs run independently: the learner starts an update once the completed groups for one logical batch are available, while rollout continues. After each update, the new weights reach the rollout engines in flight: generation pauses, the weights are swapped, and unfinished trajectories resume under the new policy. A trajectory can therefore span several policy versions, and one batch can mix trajectories of different ages. A staleness bound limits how far rollout runs ahead of the learner, so the policy lag varies but stays bounded.

For Sync, Async, and ThunderSyncRL, completion order affects admission, not batch membership: every scheduled trajectory and scored turn completes before the update. The shared-prefix learner optimization in Appendix I also applies to all schedules. Equal mathematical updates need not be bitwise identical across kernels.

E Evaluation setup

E.1 Benchmarks and success criteria

SWE-bench Verified. We use the official SWE-bench evaluation harness (`swebench.harness.run_evaluation`) for all 500 tasks in the pinned Verified test set. This human-validated subset of SWE-bench contains real GitHub issues paired with repository states and executable tests (Jimenez et al., 2024). Each attempt receives the issue statement and the repository at its base commit. The evaluator applies the agent's proposed changes and the benchmark's test patch in an isolated environment. Resolution requires all designated `FAIL_TO_PASS` and `PASS_TO_PASS` tests to pass. Missing test results, invalid patches, and agent timeouts count as failures. Gold patches and evaluation-only tests are evaluator inputs excluded from the agent prompt.[12]

Terminal Bench 4.0. We use Harbor, the benchmark's official harness, to evaluate all 66 tasks in release `v4.0.0`, including their pinned instructions, environments, resource allocations, and verifiers. The benchmark tests completion of terminal tasks through changes to the environment (Merrill et al., 2026). Each attempt starts from the task's initial state, and success follows its official verifier. Reference solutions and verifier files remain outside the agent's accessible task files. Tasks requiring GPUs or multiple containers retain those resources in the evaluation configuration; they are not silently excluded.

[11] See the pinned GLM teacher release. GLM-4.7 and GLM-4.5-Air lack 3,491 student vocabulary entries.

[12] See the pinned Verified release and the official evaluation guide.

Version 4.0 changes task definitions and resources relative to 3.0 and sets an eight-hour agent timeout.[13] The protocol retains those task resources and the eight-hour outer timeout, while imposing the shared token, turn, and ordinary tool-command limits in Appendix D.2. Its scores therefore describe ThunderSyncRL's specified agent interface and inference budget. They are not directly interchangeable with leaderboard results from other interfaces or budgets. SETA's original experiments used Terminal Bench 2.0; their scores are not baselines for the version-4.0 comparison here.

Training rewards and evaluation success. For the declared SWE-smith workload, the training reward is one only when all designated repair and regression tests pass, and zero otherwise. For SETA, the selected training reward follows its partial-progress definition: $r = p + 0.5\,\mathbf{1}[p = 1]$, where p is the fraction of passed verifier tests (Shen et al., 2026). The same task-specific reward is used across schedules; OPD records it for monitoring while its gradient uses teacher scores. Terminal Bench evaluation uses official task success rather than SETA's partial reward. Consequently, reward scales should be compared within a workload.

E.2 Attempts, aggregation, and uncertainty

The protocol specifies three independent attempts per task at each evaluated checkpoint, with settings from Table 6. If $y_{ij} \in \{0, 1\}$ indicates success on attempt j for task i, the percentage is

$$\text{pass@3} = \frac{100}{M} \sum_{i=1}^{M} \mathbf{1}\left[\sum_{j=1}^{3} y_{ij} > 0\right], \tag{13}$$

where $M = 500$ for SWE-bench Verified and $M = 66$ for Terminal Bench 4.0. Each task has equal weight. Attempts restart the environment and agent history; a later attempt receives no solution or verifier feedback from an earlier one. Pass@3 is the fraction solved at least once in three attempts. Averaging attempt success estimates pass@1. No fourth attempt is selected using verifier feedback.

We evaluated training seeds 42, 43, and 44 separately, then averaged their scores. Three evaluation attempts from one checkpoint do not constitute three independent training runs.

An infrastructure failure is recorded separately from an agent failure. A retry is allowed only to replace an attempt invalidated by an infrastructure fault, using the same task, checkpoint, and assigned seed; all retries remain in the execution record. A persistently unavailable task leaves the benchmark evaluation incomplete. It is not dropped from the denominator to improve the reported score. Agent mistakes, refusals, and exhausted inference budgets remain failed attempts.

E.3 Checkpoints and GPU-hours comparisons

Matched evaluation and training budgets. Within each objective and model configuration, ThunderSyncRL, Sync, Async, and Fully Async use the same evaluation tasks and settings. Terminal Bench tasks are excluded from post-training. We report held-out pass@3 and compare schedules by the time needed to reach the same benchmark-specific pass@3 target and by pass@3 after the same 128-GPU-hour training budget. Time to matched performance includes the learning progress across updates; it is distinct from the per-step speedup defined in Appendix B.

We requested checkpoint snapshots at one-hour training-time intervals and saved them at the next completed optimizer boundary, recording their publication times. The fixed-budget comparison uses the latest completed checkpoint published no later than 128 GPU-hours. Time to a target is the first evaluated checkpoint attaining that target, with an unreached target reported as unreached within the budget. We fixed the targets before inspecting the comparative runs.

For comparisons across seeds, we formed a common time axis from the union of checkpoint publication times. At each time, each run contributes its latest published, evaluated checkpoint score; no future checkpoint is used. The mean of the three runs defines the displayed step curve. Its first target crossing and its value at 128 GPU-hours use this same rule.

The training clock starts with the first training rollout request, after model and environment initialization, whose time is reported separately. GPU-hours multiply this clock by the eight GPUs of the

[13] See the official release description, versioned source, the 66-task inventory, and the official Harbor invocation.

node. Training cost in US dollars multiplies GPU-hours by Lambda's on-demand price for an eight-GPU B200 instance, $6.69 per GPU-hour ($53.52 per node-hour, excluding tax). It covers rollout, environment interaction, verification, teacher scoring, learner computation, and policy publication, counting overlap once. Teacher scoring contributes to elapsed time but its FLOPs are excluded from the student MFU numerator. Offline benchmark evaluation is timed separately from training. Checkpointing pauses remain in the training clock. Warmup time excluded from steady-state step profiling remains in end-to-end learning curves.

Hyperparameter or prompt changes are selected using development tasks, then frozen for all schedules. Benchmark checkpoints are evaluated on the declared cadence rather than selected retrospectively for a favorable score. Alongside GPU-hours curves, run records must retain optimizer updates, sampled trajectories, and generated tokens, so that a fixed-time comparison can be distinguished from a fixed-work comparison.

Checkpoint records. Each checkpoint record binds the training seed, update number, policy hash, publication time, processed trajectories/tokens, and one success/failure record for each task and attempt. It also stores benchmark, prompt, tokenizer, harness, and container revisions, plus infrastructure retry records. We used these records to compute pass@3 and select checkpoints.

F Scalability theory

F.1 Step time and the learner tail

We compare schedules for one logical batch generated and differentiated at the fixed policy snapshot θ_k. Let R be the time until the last rollout event required by the objective is available. Learner job j becomes ready at time τ_j and takes p_j units of service on the learner. The total learner service is $L = \sum_{j=1}^{n} p_j$. After rollout and learning finish, a common serial suffix P covers final reduction, the optimizer step, policy publication, reload, and rollout-replica acknowledgement.

The comparison fixes the physical jobs and their service times, rollout time, hardware allocation, and suffix across schedules. It assumes enough device memory to admit each source when it becomes ready. In particular, service times do not change with concurrent rollout activity or with the schedule's choice of release times. The release laws below assume $0 \le \tau_j \le R$; work reserved for the final optimizer barrier belongs to P. These assumptions isolate savings from earlier release.

Index jobs in nondecreasing readiness order. On one learner lane, the completion time c_j follows

$$c_0 = 0, \qquad c_j = \max(\tau_j, c_{j-1}) + p_j, \qquad T_{\text{ThunderSyncRL}} = \max(R, c_n) + P. \tag{14}$$

Repeated substitution gives

$$c_n = \max_{1 \le j \le n} \left(\tau_j + \sum_{i=j}^{n} p_i \right). \tag{15}$$

For the induction step, substitute the expression for c_{j-1} into $\max(\tau_j, c_{j-1}) + p_j$: the new job either starts at its own readiness time or extends one of the earlier uninterrupted service intervals. Releasing a job earlier keeps or reduces the endpoint when service times and job order stay fixed. Readiness order also attains the minimum endpoint on one lane: all jobs released at or after τ_j must be processed after that time, giving the lower bound attained in Equation 15.

Define the learner tail in units of rollout time as $\lambda_K = [c_n - R]_+ / R$, where $[x]_+ = \max(x, 0)$. Thus $R\lambda_K$ is the learner time remaining after rollout closes. The two other normalized quantities are $\rho = L/R$, the learner service relative to rollout time, and $\kappa = P/R$, the final suffix relative to rollout time. Sync starts all learner jobs after R, so

$$\begin{aligned} T_{\text{Sync}} &= R + L + P, & T_{\text{ThunderSyncRL}} &= R(1 + \lambda_K) + P, \\ S_{\text{Sync}} &= \frac{T_{\text{Sync}}}{T_{\text{ThunderSyncRL}}} = \frac{1 + \rho + \kappa}{1 + \lambda_K + \kappa}. \end{aligned} \tag{16}$$

Both stages must finish, so every overlap schedule takes at least $\max(R, L) + P$. Thus,

$$S_{\text{Sync}} \le \frac{R + L + P}{\max(R, L) + P} \le 2. \tag{17}$$

Reaching $2\times$ requires equal stage times, a negligible suffix, and no learner tail. Equal stage times maximize the ideal overlap opportunity; the actual peak can shift when a nonzero tail remains.

F.2 How work release determines the tail

Write $u = t/R$ for time as a fraction of the rollout window. Let $F^-(u)$ be the fraction of learner service released strictly before uR:

$$F^-(u) = \frac{1}{L} \sum_{\tau_j < uR} p_j. \tag{18}$$

This fraction weights each job by its service time. Counting ready jobs alone is insufficient when late trajectories also require more learning. At a release time, $L[1 - F^-(u)]$ includes the newly released jobs and all later jobs. Equation 15 therefore gives

$$1 + \lambda_K = \max\left\{1, \sup_{0 \le u \le 1} \left[u + \rho\big(1 - F^-(u)\big)\right]\right\}. \tag{19}$$

The strict-before convention includes the service of jobs released at the same instant. Replacing it with an inclusive fraction would omit those jobs at their release time.

The maximum lag behind uniform service release is $D = \sup_{0 \le u \le 1}[u - F^-(u)]$; any trace satisfies

$$[\rho - 1]_+ \le \lambda_K \le [\rho - 1]_+ + \rho D, \qquad \lambda_K = D \quad \text{when } \rho = 1. \tag{20}$$

To obtain the upper bound, write the expression inside the supremum in Equation 19, minus one, as $(\rho - 1)(1 - u) + \rho[u - F^-(u)]$. Its first term is at most $[\rho - 1]_+$ and its second at most ρD. The lower bound follows from the need to process all L service. At balance, the first term vanishes, so the tail equals the release lag.

GRPO. The plotted model assumes independent trajectory completion times $U \sim \mathrm{Uniform}(0, 1)$ and conditional mean learner service proportional to U^γ, with $\gamma \ge 0$. The fraction of service associated with trajectories completed by u is then

$$H(u) = \frac{\int_0^u t^\gamma \, dt}{\int_0^1 t^\gamma \, dt} = u^{\gamma+1}. \tag{21}$$

A fraction $1 - \beta$ of service is ready at each trajectory's reward; the rest, β, waits for all m trajectories in the group. By independence, all other members finish with probability u^{m-1}, giving

$$F_{\mathrm{GRPO}}(u) = (1 - \beta)u^{\gamma+1} + \beta u^{\gamma+m}. \tag{22}$$

Substituting this continuous release model into Equation 19 gives the plotted tail. If $\rho[\gamma + 1 + \beta(m - 1)] \le 1$, it is zero. Otherwise an interior maximum, when present, satisfies

$$(1 - \beta)(\gamma + 1)u^\gamma + \beta(\gamma + m)u^{\gamma+m-1} = 1/\rho. \tag{23}$$

The endpoint values cover the boundary cases. Turn count does not appear after ρ, m, β, and the normalized completion and service distribution are fixed. This explains the flat GRPO surface under these conditions; changing trajectory length in a real workload can change those inputs.

OPD. For a trajectory completed at U, model a equal-service turns with teacher-scored releases at jU/a, $j = 1, \ldots, a$. Let δ be the fraction of learner service deferred to trajectory completion. Block j is ready by u when $U \le au/j$. Weighting by the same service model U^γ gives

$$F_{\mathrm{OPD}}(u) = \frac{1 - \delta}{a} \sum_{j=1}^{a} \min\left(1, \frac{au}{j}\right)^{\gamma+1} + \delta u^{\gamma+1}. \tag{24}$$

Each turn can release work before its trajectory closes, including for late trajectories with large learner service. When $\delta = 1$, all work waits for trajectory completion. For $\delta = 0$ and $a \to \infty$, the release fraction tends to $u(1 + 1/\gamma) - u^{\gamma+1}/\gamma$ for $\gamma > 0$, and to $u(1 - \log u)$ for $\gamma = 0$, with value zero at $u = 0$. These functions are at least u on $[0, 1]$, so $D = 0$ and the tail reaches its minimum $[\rho - 1]_+$. For finite horizons, deferred work and the actual spacing of teacher scores determine how much of this benefit is realized.

F.3 Batch size and successive rollout waves

Let one full rollout wave take time W, with learner service $\rho_{\mathrm{w}}W$ and normalized tail λ_{w}. Consider w identical full waves launched successively at intervals of W, followed by one final suffix P; write $\kappa_{\mathrm{w}} = P/W$. A job released at normalized time u within wave i has all later waves' service behind it. Its candidate completion time, divided by W, is

$$(i-1) + u + \rho_{\mathrm{w}}[1 - F^{-}(u)] + \rho_{\mathrm{w}}(w - i). \tag{25}$$

Maximizing over i selects the last wave when $\rho_{\mathrm{w}} \leq 1$ and the first when $\rho_{\mathrm{w}} > 1$. After also including rollout completion, the resulting speedup is

$$S_{\mathrm{Sync}}(w) = \frac{w(1+\rho_{\mathrm{w}}) + \kappa_{\mathrm{w}}}{w\max(1,\rho_{\mathrm{w}}) + \kappa_{\mathrm{w}} - [\rho_{\mathrm{w}} - 1]_{+} + \lambda_{\mathrm{w}}}. \tag{26}$$

At balance this becomes $(2w + \kappa_{\mathrm{w}})/(w + \kappa_{\mathrm{w}} + \lambda_{\mathrm{w}})$. The final tail and suffix remain fixed while the overlapped work grows, so the ratio approaches two. Away from balance, the limit is $(1 + \rho_{\mathrm{w}})/\max(1, \rho_{\mathrm{w}})$. These statements assume repeated full waves. Continuous refill and partially occupied waves require either their actual release trace in Equation 19 or an explicit approximation.

F.4 Finite batches and model limitations

Equation 19 is exact for a finite trace under the lane assumptions. The plotted surfaces replace that trace by a continuous release distribution. If two release distributions differ by at most ϵ at every time, their normalized endpoints differ by at most $\rho\epsilon$: their expressions inside the supremum differ pointwise by at most that amount, and taking a supremum and a maximum preserves this bound:

$$|\lambda(F,\rho) - \lambda(G,\rho)| \leq \rho \sup_{u} |F(u) - G(u)|. \tag{27}$$

For example, n equal-service atoms placed at the midpoint quantiles of a continuous distribution have discrepancy at most $1/(2n)$ and tail error at most $\rho/(2n)$. Random finite batches need a bound for their sampling process. At fixed R and L, replacing a random release distribution by its mean is optimistic for completion time: the endpoint is a maximum of affine functions of the release distribution and hence convex, so its expected value is at least the endpoint of the mean distribution.

For statically assigned learner queues, apply the recurrence separately to each rank and take the latest completion. A pooled service rate can miss a slow rank. Memory pressure that delays admission, packing that waits for additional sources, unequal turn costs, teacher delays, or readiness-dependent learner service must also be reflected in the trace. If concurrent rollout changes kernel service or methods execute different physical work, the prediction uses service times measured under those conditions.

F.5 Figure parameters and comparison with measurements

The workload-scaling experiments were run separately to analyze overlap, with a trajectory-capacity setting of $Q = 128$. Their batch sizes, horizons, and GPU allocations are independent of the reference learning-curve configuration in Appendix D; Q is not the reference vLLM replica's active-sequence limit.

Figures 7b, 9, and 10 evaluate Equations 22, 24, and 26. For these analytical sweeps, we fix rollout capacity $Q = 128$ trajectories, GRPO group size $m = 8$, closure fraction $\beta = 0.02$, OPD deferred fraction $\delta = 0.45$, service exponent $\gamma = 0.49$, and suffix $\kappa_{\mathrm{w}} = 0.05$.

Let B denote trajectories per optimizer step and r the rollout time divided by learner time at full capacity. The plotting approximation uses

$$w = \max(1, B/Q), \qquad \rho_{\mathrm{w}} = \frac{\min(1, B/Q)}{r}. \tag{28}$$

Figure 7b fixes $r = 1$. For $B < Q$, the model keeps one rollout window and reduces learner service in proportion to occupancy. For noninteger $B/Q > 1$, it extends the full-wave formula continuously. These approximations extend the full-wave model to partial occupancy. The ratio sweeps vary r; their actual speedup peak need not be exactly at one.

Figure 8 compares the analytical surfaces with measured release traces and wall-clock speedups. We evaluated 588 paired configurations, 294 per objective. At 40 turns, the batch grid is $B \in \{64, 96, 128, 192, 256, 384, 512, 768, 1024, 1536, 2048\}$; at $B = 512$, the horizon grid is $a \in \{10, 15, 20, 30, 40, 60, 100, 150, 200, 300\}$. Each grid point uses seven nominal rollout-to-learner ratios $\{0.35, 0.5, 0.7, 1, 1.4, 2, 2.8\}$ and two repetitions, giving $11 \times 7 \times 2 = 154$ and $10 \times 7 \times 2 = 140$ configurations. The plotted ratio uses the measured stage times for the selected GPU allocation, rather than the ratio of GPU counts. The horizon sweep varies the turn limit up to 300; its turn limit and resulting context lengths are recorded separately from the learning-curve runs.

Each configuration has an analytical prediction s_i, a trace prediction o_i (open marker), and a measured wall-clock speedup f_i (filled marker). We computed o_i by replaying the observed source readiness and service times through Equation 14. The ratio f_i compares Sync and ThunderSyncRL with the same model, logical workload, GPU roles, precision, and generation settings within the cell. These controls hold within a configuration even when the allocation changes between cells.

The reported differences are computed for each paired configuration as

$$e_i = 100\frac{|o_i - s_i|}{s_i}, \qquad d_i = 100\frac{o_i - f_i}{o_i}. \tag{29}$$

Across both objectives and both rows, the mean of e_i is 1.5911% and its maximum is 10.5541%; the mean of d_i is 6.5148%. The main text rounds these to 1.6%, 10.6%, and 6.5%. The first difference measures the release model's agreement with the trace prediction; the second measures the remaining gap to runtime execution. The common optimizer and publication suffix is already present through κ_{w}, so the second difference does not represent the entire publication cost.

Trace collection and calibration. We recorded a manifest for each objective, model, batch, horizon, and integer GPU allocation, including tensor/data parallelism, teacher allocation, environment concurrency, and memory limits. Sync and ThunderSyncRL executed the same logical workload within each cell. The measured R, L, and P then determine the cell's time ratio, so each ratio reflects measured service rates.

Each trace records a source's task, trajectory, group, policy version, learner rank, readiness time, service start/end, scored-token count, and deferred-work completion. GRPO readiness is receipt of the required reward; OPD readiness is receipt of valid teacher scores. Device events measure service, while a shared host clock records arrival and publication acknowledgements. Replay uses both τ_j and p_j in Equation 14, with separate queues for statically assigned ranks. Teacher waiting and deferred backward stay in the trace. We calibrated release-law parameters on separate traces and held them fixed for evaluation. Profiling windows follow Appendix B; missing or infeasible cells are reported explicitly.

F.6 Balance sweeps

Figures 9 and 10 sweep the rollout-to-learner ratio r from 0.3 to 4 with the parameters of Appendix F.5. Both figures are analytical evaluations of Equation 26; they contain no measured points. Figure 9 fixes the batch size and varies the horizon, and Figure 10 fixes the horizon and varies the batch size.

For batches that fill at least one rollout wave ($B \geq Q$), every panel peaks near balance, $r \approx 1$. There, the predicted speedup is $1.84\times$ for GRPO at 256 trajectories per step and $1.98\times$ at 2,048; OPD reaches 1.97 to $1.98\times$ at 256 and $2.00\times$ at 2,048. Away from balance, the larger stage bounds the step, and the speedup falls toward the full-overlap limit $(1 + \rho_{\mathrm{w}})/\max(1, \rho_{\mathrm{w}})$ of Appendix F.3, which is symmetric in $\log r$. For $B \geq Q$, halving or doubling the ratio caps the speedup near $1.5\times$ (1.46 to $1.50\times$), $r = 0.35$ or 2.8 gives 1.34 to $1.36\times$, and $r = 4$ gives 1.24 to $1.25\times$, for both objectives. Below one full wave, the plotting rule scales learner service by B/Q, so balance moves to $r \approx B/Q$: at $B = 64$ the peak sits near $r = 0.5$, with $1.71\times$ for GRPO and $1.95\times$ for OPD. This bends the bright band toward smaller ratios at the bottom of Figure 10.

The horizon matters only near balance. GRPO's surface does not change with turn count, as Appendix F.2 explains. For OPD, more turns spread releases and shorten the tail, but at 256 trajectories per step the speedup at $r = 1$ moves only from $1.967\times$ at 10 turns to $1.975\times$ at 300 turns. Batch

size has the larger effect: more waves amortize the final tail and suffix, which is why the bright band at $r \approx 1$ widens as trajectories per step grow in Figure 10.

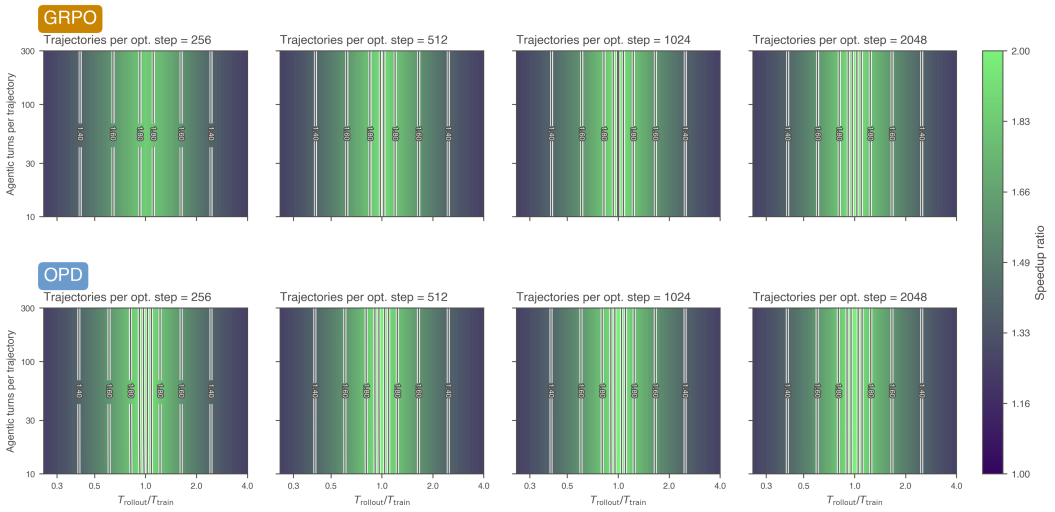

Figure 9: Speedup over Sync as a function of the rollout-to-training time ratio and agentic turns per trajectory, at fixed trajectories per optimizer step (256 to 2048), for GRPO (top) and OPD (bottom).

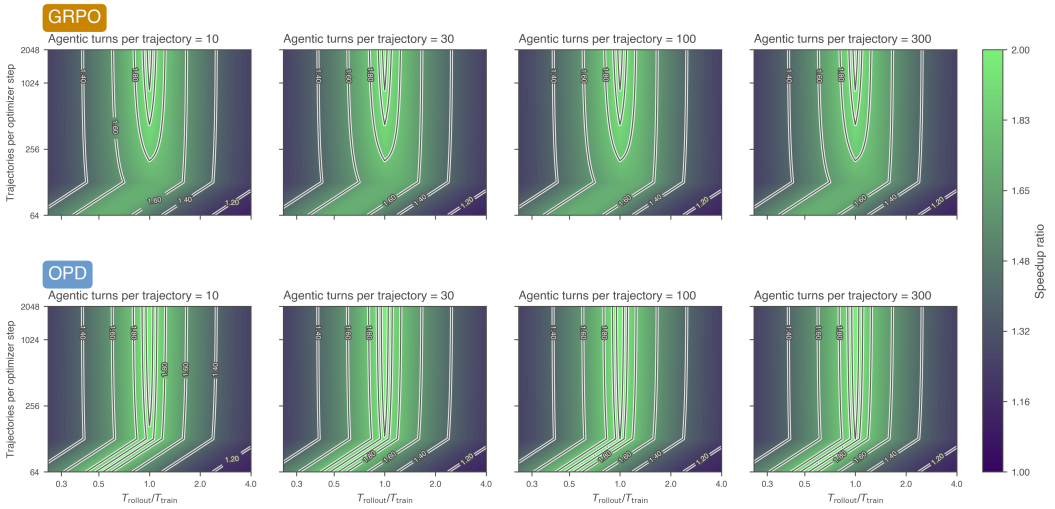

Figure 10: Speedup over Sync as a function of the rollout-to-training time ratio and trajectories per optimizer step, at fixed agentic turns per trajectory (10 to 300), for GRPO (top) and OPD (bottom).

G Empirical update equivalence

We verify the equivalence of Section 3.4 on the B200 and H200 nodes of Appendix D, comparing the updates of Sync and ThunderSyncRL on identical logical batches.

Protocol. For each objective, we record ten consecutive logical batches of a Qwen-3.8-27B Sync run, including tokens, scored-token masks, rewards, and teacher scores, and replay them under Sync and ThunderSyncRL from the same initial parameters θ_0. ThunderSyncRL ingests trajectories and scored turns in their recorded completion order. Replaying fixed batches isolates the learner computation. Both schedules use the optimizer settings of Table 5 and run twice: with the BF16 student forward used in training and with an FP32 forward as the reference. We compare the first

batch direction $\boldsymbol{g}_0$ before clipping, coordinate by coordinate, and after each update k report the parameter gap

$$\Delta_k = \frac{\|\boldsymbol{\theta}_k^{\text{ours}} - \boldsymbol{\theta}_k^{\text{sync}}\|}{\|\boldsymbol{\theta}_k^{\text{sync}} - \boldsymbol{\theta}_0\|}, \tag{30}$$

the distance between the two schedules relative to the distance that training has moved the parameters.

Results. Figures 11 and 12 show the same result on both GPU types. In FP32, the ThunderSyncRL and Sync gradients coincide coordinate by coordinate (panels a and d), with relative errors of 5.1×10^{-7} for GRPO and 1.1×10^{-6} for OPD on B200. In BF16, both schedules stay within a few percent of the FP32 gradient (panels b and e): relative errors are 0.77% for Sync and 0.96% for ThunderSyncRL under GRPO, and 4.2% and 1.2% under OPD, where Sync's BF16 error is larger. Across ten updates, the FP32 parameter gap stays below 6×10^{-5}, nearly three orders of magnitude below the gap between BF16 and FP32 Sync, and the BF16 gap between the schedules follows that same BF16 curve (panels c and f). ThunderSyncRL reproduces the batch-synchronous update at every step within the numerical precision of training.

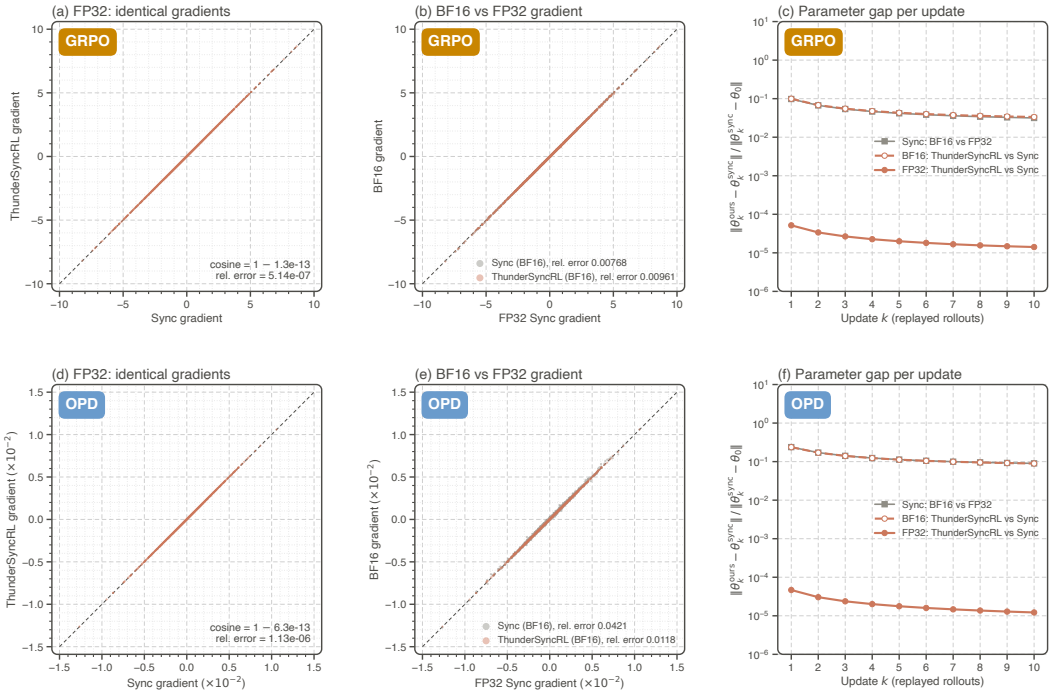

Figure 11: Update equivalence on B200 for GRPO (top row) and OPD (bottom row). (a, d) Each point is one randomly sampled coordinate of the first batch direction $\boldsymbol{g}_0$ with FP32 arithmetic, ThunderSyncRL against Sync. (b, e) Both schedules with BF16 arithmetic against the FP32 Sync direction. (c, f) Parameter gap Δ_k (Equation 30) after each of ten replayed updates, with the gap between BF16 and FP32 Sync as reference. Dashed diagonals mark equality.

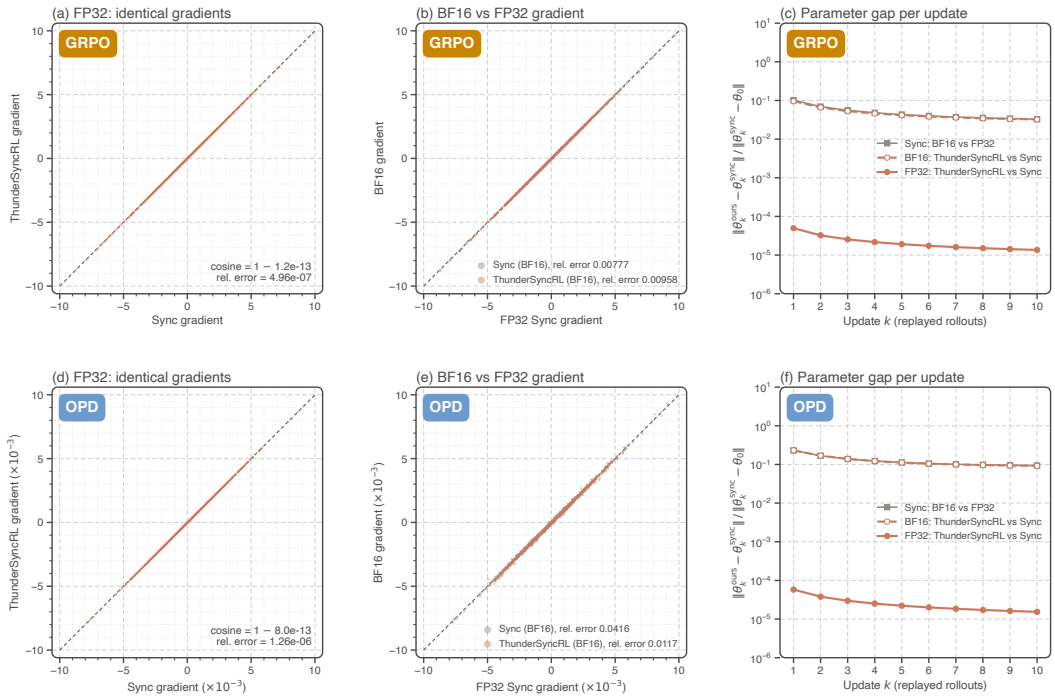

Figure 12: Update equivalence on H200, with panels as in Figure 11.

H Further Experimental Results

H.1 Pass@1 and Pass@5

Pass@k is the fraction of tasks solved at least once in k independent attempts. Following Chen et al. (2021), we estimate it from $n \geq k$ attempts per task: if c_i of the n attempts on task i succeed,

$$\text{pass@}\,k = \frac{100}{M} \sum_{i=1}^{M} \left(1 - \binom{n - c_i}{k} \Big/ \binom{n}{k}\right), \tag{31}$$

which is unbiased for the probability that at least one of k attempts succeeds. Pass@1 is the mean attempt success rate. With $n = k$ the estimator reduces to the empirical rate of tasks solved at least once, the form of pass@3 in Equation 13.

For this appendix, each evaluated checkpoint runs five independent attempts per task ($n = 5$), from which both pass@1 and pass@5 are computed; the other settings follow Table 6, and attempts restart the environment and agent history as in Appendix E. Figures 13 and 14 report pass@1 and pass@5 against training cost for Qwen-3.8-27B and GLM-4.7-Flash-31B under GRPO and OPD, in the form of Figure 1. Pass@1 scores a single attempt; pass@5 credits a task solved in any of the five.

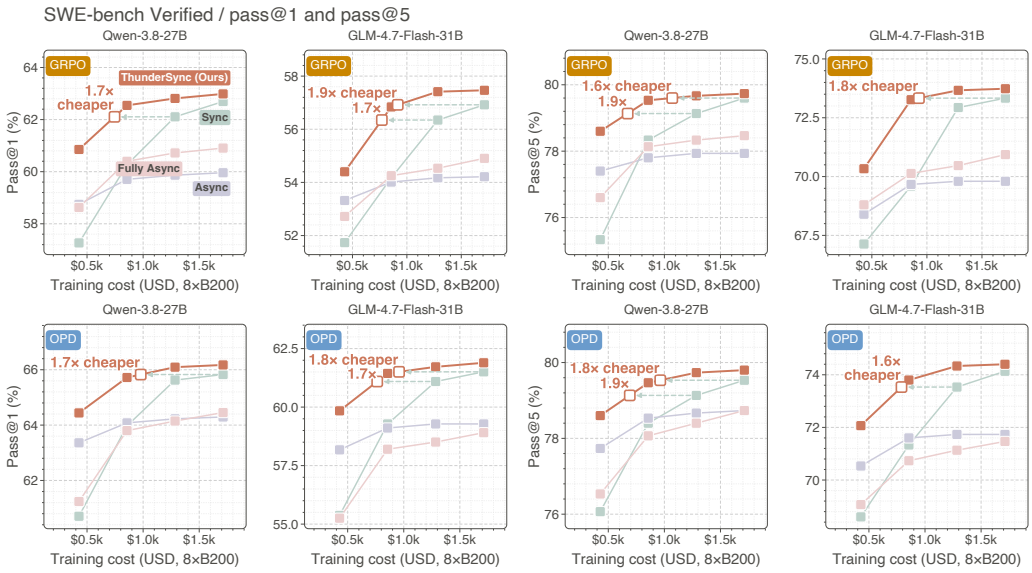

Figure 13: Pass@1 (left two panels) and pass@5 (right two panels) on SWE-bench Verified against training cost for Qwen-3.8-27B and GLM-4.7-Flash-31B under GRPO (top row) and OPD (bottom row). Arrows mark where ThunderSyncRL reaches the score of a Sync checkpoint; labels give the cost ratio.

H.2 Efficiency with LoRA

We profiled the Qwen and GLM LoRA configurations on eight B200 GPUs (Table 7). Each configuration uses four learner GPUs, replicated BF16 frozen weights, FP32 adapter state, and the objective-specific rollout and teacher allocation in Appendix D.

For Sync, Async, and ThunderSyncRL, LoRA retains the efficiency ordering of full fine-tuning. ThunderSyncRL reaches 16.7% to 33.0% trainer active MFU, slightly above Async's 16.5% to 32.6% and above Sync's 10.3% to 19.5%. Its step wall time speedups are $1.69/1.71\times$ for Qwen and $1.64/1.62\times$ for GLM under GRPO/OPD, compared with Async's $1.67/1.69\times$ and $1.62/1.60\times$, respectively. ThunderSyncRL retains zero policy lag in every configuration.

For Qwen, the speedups are smaller than under full fine-tuning; for GLM, whose step is already bound by rollout, they barely change. Freezing base weights removes their parameter-gradient and

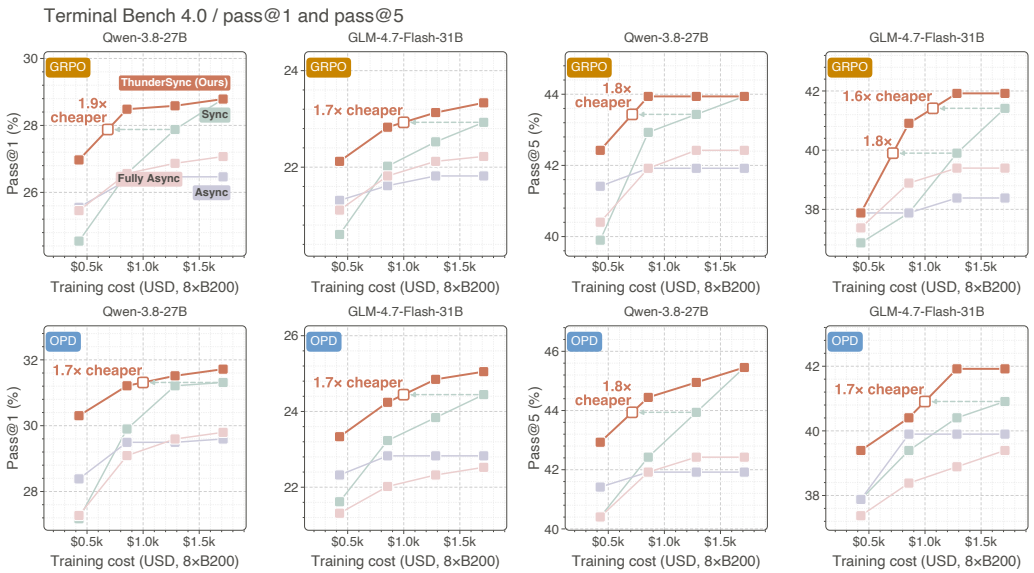

Figure 14: Pass@1 (left two panels) and pass@5 (right two panels) on Terminal Bench 4.0 against training cost for Qwen-3.8-27B and GLM-4.7-Flash-31B under GRPO (top row) and OPD (bottom row). Arrows mark where ThunderSyncRL reaches the score of a Sync checkpoint; labels give the cost ratio.

Table 7: LoRA efficiency on NVIDIA B200 GPUs. Each speedup uses the same model and objective's Sync reference. OPD model names refer to the student.

Method	Policy lag (staleness) ↓	GRPO			OPD		
		Trainer active MFU (%) ↑	Speedup (×) ↑	Peak GPU memory (GB) ↓	Trainer active MFU (%) ↑	Speedup (×) ↑	Peak GPU memory (GB) ↓
Qwen-3.8-27B LoRA							
Sync	0	19.5	1.00	75.4	19.0	1.00	73.4
Async	1	32.6	1.67	76.0	32.1	1.69	74.0
Fully Async	Variable	32.8	1.68	77.1	32.3	1.70	74.8
ThunderSyncRL	0	33.0	1.69	79.2	32.5	1.71	76.0
GLM-4.7-Flash-31B LoRA							
Sync	0	10.9	1.00	75.1	10.3	1.00	73.1
Async	1	17.7	1.62	75.6	16.5	1.60	73.6
Fully Async	Variable	17.8	1.63	76.3	16.6	1.61	74.2
ThunderSyncRL	0	17.9	1.64	77.3	16.7	1.62	75.0

optimizer work, reducing the learner work available to overlap with rollout and teacher scoring. Forward and input-gradient computation through those weights remains necessary. Trainer active MFU counts learner FLOPs over the complete update wall time, including rollout waiting. It therefore reflects both the reduced learner work under LoRA and the idle time remaining after overlap.

For these three schedules, trainer GPU memory is lower with LoRA than with full fine-tuning. ThunderSyncRL's GPU peaks range from 75.0 to 79.2 GB per learner GPU, with an additional 1.9 to 3.8 GB relative to Sync for live gradient statistics and deferred state. The measurements include frozen weights, adapter and optimizer state, activations, and temporary buffers; the separate rollout and teacher pools are excluded. The GLM teacher and adapter targets are specified in Appendix D.3.

I THUNDERSYNCRL ALGORITHM AND IMPLEMENTATION

Algorithm 1 summarizes one ThunderSyncRL update. The rollout and learner threads run concurrently at the fixed policy snapshot $\boldsymbol{\theta}_k$. For GRPO, a ready source contains a trajectory's scored action tokens and reward. For OPD, it contains a turn's scored action tokens and frozen-teacher scores. Completion events release deferred context backward and group reduction once their dependencies are satisfied. The implementations below give the objective-specific normalization and optimizer conventions.

Algorithm 1 ThunderSyncRL: gradient streaming with a synchronous update

Input: Policy $\pi_{\boldsymbol{\theta}_k}$, batch $\mathcal{B}_k$, objective o, frozen teacher π_{T} for OPD
Output: Updated $\boldsymbol{\theta}_{k+1}$, acknowledged by every rollout replica
$\mathcal{Q} \leftarrow \mathrm{Queue}()$; register all trajectories and groups in $\mathcal{B}_k$
Initialize objective accumulators $\mathcal{A}$; keep $\boldsymbol{\theta}_k$ fixed

Thread 1: Rollout and supervision (producer)
for all trajectories $i \in \mathcal{B}_k$ **in parallel do**
    Generate with $\pi_{\boldsymbol{\theta}_k}$; enqueue ready sources and completion events as they occur
end for

Thread 2: Learner (consumer)
while a registered trajectory or group remains unfinished **do**
    $e \leftarrow \mathcal{Q}.\mathrm{pop}()$ *(Wait for the next available event)*
    if e is a GRPO trajectory source i in group $\mathcal{G}$ **then**
        $\boldsymbol{H}_i \leftarrow \nabla_{\boldsymbol{\theta}} S_i(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_k}$
        $\boldsymbol{G}_{\mathcal{G}}^{(1)} \leftarrow \boldsymbol{G}_{\mathcal{G}}^{(1)} + r_i \boldsymbol{H}_i$; $\boldsymbol{G}_{\mathcal{G}}^{(2)} \leftarrow \boldsymbol{G}_{\mathcal{G}}^{(2)} + \boldsymbol{H}_i$
    else if e is an OPD turn source j **then**
        Accumulate $\boldsymbol{g}_j$ and $|\mathcal{M}_j|$ in $\mathcal{A}$ *(Equation 8)*
    end if
    Update completion state; finish eligible group reductions and deferred backward
end while

Synchronous batch barrier
Wait for all rollouts, source gradients, and deferred work in $\mathcal{B}_k$ to finish
$\widehat{\boldsymbol{g}}_k \leftarrow \mathrm{DistributedReduceAndNormalize}(\mathcal{A}, o)$
$\boldsymbol{\theta}_{k+1} \leftarrow \mathrm{Opt}(\boldsymbol{\theta}_k, \mathrm{clip}(\widehat{\boldsymbol{g}}_k))$ *(Exactly one optimizer step)*
Publish $\boldsymbol{\theta}_{k+1}$; wait for every replica to reload, clear caches, and acknowledge
return $\boldsymbol{\theta}_{k+1}$

Shared-prefix backward in all schedules. Sync, Async, and ThunderSyncRL all reuse shared-prefix computation on the learner. Training-time prefix reuse has earlier applications in preference optimization (Wang & Hegde, 2024) and in branching agent trajectories (Wang et al., 2026). At a fixed learner snapshot, we compute the shared prefix once, process each continuation with its causal context, and accumulate its gradient with respect to the prefix boundary states. After all branch contributions are available, one backward pass through the shared prefix propagates their sum, with the objective's weights and normalization preserved. For GRPO, the reward-weighted and unweighted boundary contributions are combined when the group rewards are fixed. For OPD, teacher-scored turns contribute boundary gradients before the final token normalization. This is a common learner optimization in the comparison, distinct from vLLM's inference-time prefix cache. ThunderSyncRL admits eligible branch work earlier; shared-prefix backward remains part of each schedule's measured learner work and update time.

The following Python-style pseudocode describes one logical update of ThunderSyncRL. Rollout workers and the learner use the same fixed policy version throughout the update. Each learner rank processes its assigned groups. Helpers represent model execution, gradient storage, and distributed collectives. `publish_and_wait` publishes the updated parameters and waits for every rollout replica to load them, clear policy-dependent caches, and acknowledge the new version.

I.1 GRPO

Implementation 1 shows reward-ready admission and the decomposition in Equation 7. The source gradient can also be prepared when content is complete and bound to its reward later. Gradient arithmetic includes any captured boundary adjoints; `finish_group_backward` propagates their combined contribution through deferred context. The code applies the sum of group directions, $\boldsymbol{g}_k = \sum_{\mathcal{G}} \boldsymbol{g}_{\mathcal{G}}$, as in Section 3.4; the token weights in Appendix D make this sum the mean over the trajectory-normalized directions.

```python
def thundersync_grpo_step(model, rollout, batch, optimizer, eps, max_norm):
    """One reward-linear GRPO epoch; optimizer has maximize=True."""
    params = trainable_parameters(model)
    version = rollout.policy_version
    groups = register_groups(batch)  # Sizes, rewards, zero G1/G2 buffers
    batch_grad = zeros_like(params)
    optimizer.zero_grad(set_to_none=True)

    # 1. Start learner work while other trajectories are still running.
    for src in rollout.reward_ready_sources(batch, version):
        assert src.policy_version == version
        group = groups[src.group_id]
        logp = rebuild_scored_logprobs(model, src)  # Fixed action mask
        score = (src.token_weights * logp).sum()
        H = backward_with_boundary_capture(score, params)

        # 2. Fold each completed source once into its group statistics.
        group.G1 += src.reward * H
        group.G2 += H
        group.rewards.append(src.reward)
        group.num_reduced += 1

        # 3. Combine only after every reward and source gradient arrives.
        if group.num_reduced == group.size:
            r = torch.tensor(group.rewards, dtype=torch.float64)
            mu, sigma = r.mean(), r.std(unbiased=False)
            update = (group.G1 - mu * group.G2) / (group.size * (sigma + eps))
            batch_grad += finish_group_backward(group, update)
            group.close()

    # 4. Complete the logical batch before any parameter mutation.
    rollout.wait_batch_closed(batch)
    assert all(group.closed for group in groups.values())
    wait_for_all_backward_and_reduction_work()
    global_grad = all_reduce_sum(batch_grad)  # Sum of group directions
    write_parameter_grads(params, global_grad)
    torch.nn.utils.clip_grad_norm_(params, max_norm)
    optimizer.step()  # Exactly one ascent step at the batch barrier
    optimizer.zero_grad(set_to_none=True)

    # 5. The next rollout batch starts only after every replica ACKs.
    rollout.publish_and_wait(model, version + 1)
    return version + 1
```

Implementation 1. ThunderSyncRL GRPO update. Each trajectory releases its score gradient independently; group closure fixes the reward coefficients. The optimizer uses gradient ascent.

I.2 On-policy distillation

Implementation 2 follows the turn-level OPD path. Each teacher score uses the sampled action's realized causal context; scoring may run upstream of the learner, but is shown inline here. The detached student log probabilities come from the same policy snapshot, so the importance ratio equals one while retaining its derivative. The loss gradient is the negative of Equation 8. Turn backward captures adjoints at cut state boundaries; trajectory and group closure propagate the remaining adjoints before the batch update. The final sum divided by the global scored-token count is equivalent to the runner's rescaling followed by a data-parallel average.

```python
def thundersync_opd_step(student, teacher, rollout, batch, optimizer, max_norm):
    """One token-mean OPD update; teacher is frozen, optimizer descends."""
    params = trainable_parameters(student)
    version = rollout.policy_version
    groups = register_groups(batch)
    token_count = 0
    optimizer.zero_grad(set_to_none=True)

    # 1. Ingest turns as rollout workers generate actions and call tools.
    for event in rollout.turn_and_completion_events(batch, version):
        assert event.policy_version == version
        group = groups[event.group_id]
        if event.kind == "turn":
            with torch.no_grad():
                teacher_lp = teacher.score_action_tokens(event)
            logp = append_turn_and_get_scored_logprobs(student, event)
            if logp.numel() == 0:  # Unscored context still updates state
                continue
            old_logp = logp.detach()  # Same fixed student snapshot
            advantage = (teacher_lp - old_logp).detach()
            ratio = torch.exp(logp - old_logp)
            loss = -(ratio * advantage).sum()

            # 2. Backward now; retain adjoints needed by earlier context.
            backward_with_boundary_capture(loss, params)
            token_count += logp.numel()

        elif event.kind == "trajectory_complete":
            # 3. Replay deferred state adjoints through completed turns.
            replay_deferred_state_adjoints(event.trajectory_id)
            group.close_trajectory(event.trajectory_id)
            if group.all_trajectories_closed():
                finish_shared_prefix_backward(group)
                group.close()

    # 4. All turns, replay work, and registered groups must be complete.
    rollout.wait_batch_closed(batch)
    assert all(group.closed for group in groups.values())
    wait_for_all_backward_and_reduction_work()
    local_grad = materialize_parameter_grads(params)
    global_grad = all_reduce_sum(local_grad)
    global_tokens = all_reduce_sum(token_count)
    assert global_tokens > 0
    write_parameter_grads(params, global_grad / global_tokens)
    torch.nn.utils.clip_grad_norm_(params, max_norm)
    optimizer.step()  # Exactly one descent step at the batch barrier
    optimizer.zero_grad(set_to_none=True)

    # 5. Publish the new policy before admitting the next logical batch.
    rollout.publish_and_wait(student, version + 1)
    return version + 1
```

Implementation 2. ThunderSyncRL on-policy distillation update. Teacher-scored turns release backward immediately. Deferred context gradients and global token normalization finish before one optimizer step.