AGENTIC OPTIMIZATION FOR ASYNCHRONOUS RL POST-TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning (RL) post-training is computationally expensive, motivating asynchronous training for improved efficiency. Asynchronous RL, however, exposes a large, complex, and coupled configuration space across training, rollout, and resource orchestration. If configured improperly, asynchronous training will lose the efficiency advantages while introducing policy staleness. In this paper, we formulate a joint objective over model quality and training efficiency for asynchronous RL training and task LLM-based agents with iteratively optimizing it through hypothesis formulation, experimentation and evaluation. We evaluate both open- and closed-source agents across reasoning, multimodal and agentic post-training tasks using VeRL, a widely adopted open-source RL infrastructure. Our agents improve model quality and training throughput over human reference configurations while reducing per-task training GPU-hours by up to 50%, and distill transferable insights that accelerate optimization on a new task. Analysis of optimization trajectories provides insights into configuring asynchronous RL training and suggests directions for efficient RL post-training.

1 INTRODUCTION

Reinforcement learning (RL) has become crucial to LLM post-training for improving instruction following, reasoning and task-specific capabilities Ouyang et al. (2022); Shao et al. (2024); Yu et al. (2025); Guo et al. (2026). During RL post-training, a pretrained LLM generates multiple responses, or policy rollouts. Each rollout then receives a reward signal, for instance from human preferences or verifiable outcomes Ouyang et al. (2022); Shao et al. (2024). The policy rollouts and their corresponding rewards are then used to update the LLM's weights to produce higher-reward responses. This is also known as *synchronous* training, as policy rollout, grading and training are executed sequentially. The stages during synchronous training, however, have different mechanisms: policy rollout requires step-wise autoregressive decoding or interaction with agentic environments Feng et al. (2026), while policy training processes completed rollouts in parallel through batched forward and backward propagation. Despite the mechanism differences, synchronous training uses the same pool of GPUs for all stages, potentially leading to overprovisioned resources or prolonged training time. RL post-training is therefore known to be computationally expensive and time-consuming.

Asynchronous RL training has recently been adopted to decouple policy training and rollout, placing them on separate pools of GPUs and executing the two stages concurrently Fu et al. (2025); Piché et al. (2026); Wu et al. (2025); Noukhovitch et al. (2025). Existing infrastructures (e.g., VeRL, NemoRL) often expose a large number of hyperparameters, and if not configured properly, can silently degrade into synchronous training, for instance when the speeds of policy rollout and training are mismatched. Yet, practical guidance on how to configure asynchronous RL training remains limited. Though hyperparameter optimization (HPO) methods can be applied to configure ML experiments, existing methods generally search over only a small, predefined set of hyperparameters, and none focuses on asynchronous RL training Eimer et al. (2023); Chen et al. (2026); Liu et al. (2025). Moreover, these methods primarily focus on model quality, without considering training computation budget and efficiency, which are crucial for asynchronous RL training.

In this paper, we study not only *how* to set hyperparameters for asynchronous RL training, but also *what* hyperparameters contribute to improved efficiency and quality, providing insights into configuring asynchronous RL. We formulate a joint optimization objective for LLM-based agents

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.

to discover configurations that result in higher training efficiency, lower training cost and better model quality. LLM-based agents are tasked with iteratively optimizing for the objective within training resource budgets through hypothesis formulation, experimentation and evaluation. Since an RL infrastructure serves tasks of different natures (e.g., reasoning, agentic), we investigate whether agent-discovered optimization insights are task-agnostic and reflect the properties of the shared infrastructure. We hence distill each agent's experiment trajectory into an optimization prior Shinn et al. (2023) that seeds the optimization on a different task, and investigate the effect of the prior.

Across a range of reasoning, multimodal and agentic RL tasks using VeRL Sheng et al. (2025), an open-source infrastructure, both closed-source Claude Code with Claude Opus 5 (*Claude*) and open-source OpenCode with Qwen3.8-27B (*Qwen*) discover configurations that reduce per-task training GPU-hours by up to 50%, while improving model quality and throughput over released reference configurations. Moreover, we find the optimization prior distilled from one task accelerates the optimization of another. We provide detailed analysis of the agent optimization trajectories, and identify what configurations contribute to improved training efficiency and model quality.

Our contributions are summarized as follows:

- We formulate a joint optimization objective over model quality and training efficiency for asynchronous RL training. LLM-based agents are tasked with iteratively optimizing for the joint objective, without a predefined configuration search space.
- We show that released reference configurations for asynchronous RL training are suboptimal: across reasoning, multimodal and agentic tasks, agent-discovered configurations reduce training GPU-hours by 4.5–49.5% and raise throughput by up to 2× without regressing model quality.
- Optimization priors discovered by closed-source agents reflect task-agnostic insights for configuring asynchronous RL training: seeded with a prior, closed-source agents consistently discover qualifying configurations sooner. Open-source agents, however, do not show such consistent behavior.
- We provide practical insights into configuring and designing asynchronous RL training infrastructures, which allow for more efficient and effective training, saving computational resources and training time.

2 Background

2.1 RL Post-Training

RL post-training primarily consists of three stages: policy rollout, reward calculation and policy training. The stages have different mechanisms and each stage benefits from a different optimization strategy. First, during policy rollout, the LLM autoregressively generates multiple responses. As this stage requires step-wise decoding, it is inherently sequential and often results in low GPU utilization. Existing inference engines use techniques such as paged attention and continuous batching to speed up inference Kwon et al. (2023); Zheng et al. (2024). Second, reward calculation evaluates the generated responses, for instance, through human feedback, verifiable outcomes or responses from LLM-judges Yu et al. (2025); Shao et al. (2024); Ouyang et al. (2022); Noukhovitch et al. (2025); Lee et al. (2024). During this stage, the interaction with external environments or APIs can potentially cause latencies and bottleneck the overall training. Lastly, the policy utilizes the rollouts and their corresponding rewards to update its weights via gradient descent. This stage is highly parallelizable but requires accommodating, for instance, model parameters, gradients, and optimizer states for large models. Existing training engines hence use techniques such as parallelism or fused kernels to improve training efficiency Shoeybi et al. (2019); Narayanan et al. (2021); Rajbhandari et al. (2020); Zhao et al. (2023); Hsu et al. (2024). Despite the mechanism differences, synchronous on-policy training often uses the same pool of GPUs to execute these stages sequentially.

2.2 Asynchronous RL Training

Asynchronous RL training infrastructures, as illustrated in Figure 1, are proposed to decouple these stages using different pools of resources Fu et al. (2025); Piché et al. (2026); Wu et al. (2025);

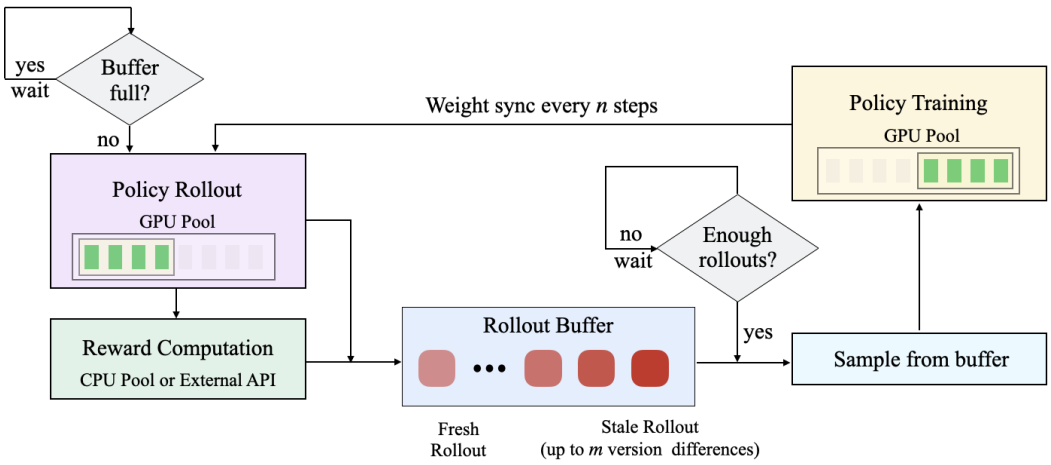

Figure 1: Schematic of asynchronous RL post-training. Rollout and training are executed concurrently using separate pools of GPU resources.

Noukhovitch et al. (2025). Specifically, rollout workers continuously generate responses without waiting for the latest policy update, and store the rollouts in a buffer. Once the buffer contains enough eligible responses, the training workers sample a batch and update the policy. The updated weights are then synchronized back to the rollout workers. This process allows policy rollout and training to be executed concurrently. The asynchronous setup, however, introduces policy staleness, as a rollout may have been generated by an older policy than the one being optimized, potentially causing training instability or collapse Li et al. (2026). Existing methods therefore control staleness by, for instance, constraining the policy-version difference or applying importance sampling Espeholt et al. (2018).

In practice, asynchronous RL training remains costly to configure and the resulting training efficiency and model quality are highly dependent on system configurations Zhong et al. (2025); Fu et al. (2025); Li et al. (2026); Eimer et al. (2023). For instance, if the staleness constraint is too restrictive, the system can effectively degenerate toward synchronous execution, since the rollout buffer might not be filled quickly enough, while loosening the staleness constraint can lead to model collapse. Existing RL infrastructures expose large configuration spaces (see Appendix) Sheng et al. (2025); NVIDIA NeMo Team (2025); Fu et al. (2025), yet existing work provides limited guidance on how these hyperparameters should be configured.

2.3 Asynchronous RL Training Configurations

The configuration space for asynchronous RL training can be broadly categorized as follows: optimization and training settings (e.g., learning rate, total training steps), RL algorithm choices (e.g., Kullback–Leibler (KL) regularization), policy training and rollout settings, and asynchronous execution settings. Policy training and rollout settings can include configurations such as tensor and data parallelism, which shard data or model across distributed GPUs, or dynamic batching, which packs variable-length sequences up to a token limit, improving GPU utilization. Asynchronous execution settings can include, for instance, staleness thresholds or GPU resource splits between rollout and training workers Zhiqiang et al. (2026).

These settings can be coupled: using GPU allocation as an example, the resource split ideally ensures balanced rollout-generation and training-consumption rates, which simultaneously improves training efficiency and reduces policy staleness. The appropriate allocation therefore depends on the relative speeds of rollout and training, which are affected by their batching and parallelism settings. These dependencies make the optimal configuration of asynchronous RL training non-trivial.

3 AGENTIC OPTIMIZATION OF ASYNCHRONOUS RL TRAINING

3.1 JOINT OPTIMIZATION OBJECTIVE OVER MODEL QUALITY AND TRAINING EFFICIENCY

To reduce training cost while maintaining model quality, we formulate a joint quality and efficiency objective for LLM-based agents to optimize. We consider an asynchronous RL infrastructure $\mathcal{I}$ and a training task τ, which specifies a pretrained model, post-training data, reward function, and evaluator. The configuration space is often large (see Appendix), and contains separate settings for policy training, rollout, and asynchronous execution. Executing the training task τ with λ produces post-trained model weights θ_λ and an execution trace r_λ: $(\theta_\lambda, r_\lambda) = \mathrm{Train}(\mathcal{I}, \tau, \lambda)$. The training is then evaluated from two perspectives: training efficiency and model quality. We measure training efficiency in terms of throughput[1] (tokens/GPU-s): $T_\tau(\lambda) = \mathrm{Throughput}(r_\lambda)$ and training cost: $C_\tau(\lambda) = \mathrm{GPUHours}(r_\lambda)$. Additionally, we evaluate model quality in terms of task-specific metrics $Q_\tau(\lambda) = \mathrm{Eval}_\tau(\theta_\lambda)$.

We choose to jointly optimize for these metrics due to the following reasons. Throughput measures the number of tokens processed per GPU-second during training. Higher throughput therefore indicates that more tokens are processed using fewer GPUs or less time per training step. However, even with a quality constraint, throughput alone can be improved without reducing the total training cost. For example, if the trainer intermittently waits for rollouts (see Figure 1), raising the staleness threshold lets the rollout workers run further ahead and keeps the buffer filled. As a result, the trainer stalls less, hence improving throughput. Stale rollouts can cause performance degradation, but training for longer may compensate for the quality degradation at the cost of additional compute. We therefore also constrain the training costs C.

Let λ_0 be the released reference configuration for task τ. Given a quality improvement threshold δ_Q, throughput-improvement threshold δ_T, and training-cost-reduction threshold δ_C, the agent's task is to discover a qualifying configuration λ within its resource budget such that it improves throughput and reduces training cost while maintaining model quality:

$$Q_\tau(\lambda) \geq (1+\delta_Q)Q_\tau(\lambda_0), \qquad T_\tau(\lambda) > (1+\delta_T)T_\tau(\lambda_0), \qquad C_\tau(\lambda) < (1-\delta_C)C_\tau(\lambda_0). \quad (1)$$

3.2 AGENTIC OPTIMIZATION FORMULATION

Algorithm 1 Agentic optimization loop for asynchronous RL training

Require: Asynchronous RL infrastructure $\mathcal{I}$, training task τ, reference configuration λ_0, agent $\mathcal{A}$, agent instruction $\mathcal{P}$, optional optimization prior L, empty experiment history $\mathcal{H}$, optimization GPU-hour limit $G_{\max}$, and wall-time limit $W_{\max}$

$G \leftarrow 0$, $W \leftarrow 0$, and $\lambda^* \leftarrow \emptyset$
Append $(\lambda_0, T_\tau(\lambda_0), C_\tau(\lambda_0), Q_\tau(\lambda_0))$ to $\mathcal{H}$
while $G < G_{\max}$ **and** $W < W_{\max}$ **do**
  $(h_t, \lambda_t, m_t) \leftarrow \mathcal{A}(\mathcal{I}, \tau, \mathcal{P}, L, \mathcal{H})$ ▷ h_t: hypothesis; m_t: PROXY or FULL run
  The agent invokes $(\theta_t, r_t) \leftarrow \mathrm{Train}_{m_t}(\mathcal{I}, \tau, \lambda_t)$
  if $m_t =$ FULL **then**
    Measure $Q_\tau(\lambda_t)$, $T_\tau(\lambda_t)$, $C_\tau(\lambda_t)$
  else if $m_t =$ PROXY **then**
    Measure $T_\tau(\lambda_t)$
  end if
  Append h_t, λ_t, m_t, w_t, $T_\tau(\lambda_t)$, $C_\tau(\lambda_t)$, and $Q_\tau(\lambda_t)$ to $\mathcal{H}$
  if λ_t satisfies Eq. 1 **and** lowers C over λ^* **then**
    $\lambda^* \leftarrow \lambda_t$ ▷ lowest-cost qualifying configuration so far
  end if
  $G \leftarrow G + C_\tau(\lambda_t)$ and $W \leftarrow W + w_t$
end while
$L \leftarrow \mathcal{A}(\mathcal{P}, \mathcal{H})$ ▷ optimization prior for future tasks
return λ^*, $\mathcal{H}$, and L

[1] We measure the median throughput since throughput can be close to zero during training warmup and can fluctuate occasionally. The median therefore better represents the training efficiency.

In Algorithm 1, we formulate how a post-training task is optimized in an agentic optimization loop Yao et al. (2023). Following Section 3.1, we define $\mathcal{A}$ and $\mathcal{P}$ as the agent and the instruction it is given, $G_{\max}$ and $W_{\max}$ as the optimization GPU-hour budget and wall-clock time respectively. Different from the HPO paradigm Eimer et al. (2023); Chen et al. (2026); Liu et al. (2025), $\mathcal{P}$ does not enumerate tunable hyperparameters and candidate values. Instead, the agent is tasked with identifying potential optimization strategies from the RL infrastructure $\mathcal{I}$ to meet the criteria in Eq. 1 within resource limits. Optionally, the agent receives an agent-summarized optimization prior L from a different task, which serves as a compact cross-task memory of potential optimization strategies for asynchronous RL infrastructures. We include this optional prior to test whether the agent can extract transferable and task-agnostic insights, thus revealing common rules of thumb for configuring asynchronous RL training.

At each agent iteration t, the agent $\mathcal{A}$ produces a hypothesis h_t after inspecting the reference baseline and the experiment history $\mathcal{H}$. To improve its optimization efficiency, the agent is instructed to choose between two experiment types: $m \in \{\text{PROXY}, \text{FULL}\}$. A proxy experiment trains for fewer steps and reveals only whether λ improves efficiency, while a full experiment trains to completion and verifies whether λ satisfies Eq. 1. After each iteration, the configuration and evaluation results are appended to the experiment history $\mathcal{H}$ as context for subsequent iterations. Each experiment adds its training GPU-hours $C_\tau(\lambda_t)$ and elapsed time w_t to the running totals G and W. Finding a qualifying configuration does not terminate the agent optimization loop: the agent is instructed to keep reducing the training cost while meeting the criteria in Eq. 1 until either $G_{\max}$ or $W_{\max}$ is reached. Finally, the agent summarizes the experiment history $\mathcal{H}$ into an optimization prior L for future tasks.

Algorithm 1 produces an *agent optimization trajectory*, which includes the lowest-cost configuration λ^*, experiment histories $\mathcal{H}$ and an optimization prior L. We report whether qualifying configurations are found within the resource limits ($G_{\max}$ and $W_{\max}$). Additionally, we analyze the optimization trajectories to measure the effect of the optimization prior L, comparing the optimization trajectories with and without L. Finally, we analyze the configuration changes from the reference λ_0 to the agent-discovered λ^* to identify optimization insights for asynchronous RL training.

4 Experimental Setup

4.1 RL Post-Training Tasks

Post-Training Task	Base Qwen2.5 Model	Training data	Eval. data	Eval. Metric (Q)
Reasoning	Math-7B	DAPO-Math	AIME 24–26	MEAN@30
Multimodal	Instruct-7B-VL	Geo3K train	Geo3K test	MEAN@4
Agentic (code sandbox)	Instruct-7B (tool SFT)	DAPO-Math	AIME 24–26	MEAN@30

Table 1: RL post-training tasks optimized by the agents. Each task post-trains a Qwen2.5 model.

We conduct our experiments using three representative RL post-training tasks for the agents to optimize (see Table 1), which include math reasoning from texts, multimodal inputs (i.e., image and text) and multi-turn agentic math reasoning with code sandbox tool calling. These tasks have official implementations in VeRL Sheng et al. (2025) with reference configurations using asynchronous training infrastructures (See Appendix). For these tasks, we re-train each task under its reference configuration to measure the training efficiency metrics defined in Section 3.1. The reasoning, multimodal and agentic tasks run at 3,330, 1,134 and 1,612 tokens/GPU-s and cost 84.1, 45.2 and 135.0 GPU-hours, or 10.5, 5.7 and 16.9 hours on 8 Nvidia H100s.

For the text-based math reasoning task, we post-train Qwen2.5-Math-7B on DAPO-Math-17K Yu et al. (2025) and evaluate the post-trained model on AIME 24–26 Balunovic et al. (2025) using MEAN@30[2]. Each prompt contains a text-only mathematical problem, and the model generates reasoning followed by a final answer. For the multimodal reasoning task, we post-train Qwen2.5-VL-7B-Instruct on Geometry3K Lu et al. (2021), where each prompt contains a geometry figure

[2]MEAN@30 is the accuracy averaged over 30 samples per problem, pooled across AIME 24–26

and a question. The model processes both modalities and generates reasoning and a final answer. The post-trained model is evaluated on the Geometry3K Lu et al. (2021) test split using MEAN@4. Compared with the text-based task, the vision encoder in Qwen2.5-VL-7B-Instruct introduces an additional modality during both policy rollout and training. For the agentic task, we follow the ReTool training recipe Feng et al. (2026), using the same data and evaluation metrics as the text-only reasoning task. This task can invoke coding tool calls to verify intermediate reasoning steps. For instance, the model can call a Python function solver to verify the correctness of an intermediate answer, creating variable-length, multi-turn rollouts. Following the official implementation, we first finetune Qwen2.5-7B-Instruct on tool-use SFT data and then post-train on DAPO- Math-17K Yu et al. (2025). We exclude the SFT stage from the agentic optimization loop and focus only on RL training optimization.

4.2 AGENTIC OPTIMIZATION SETUP

We evaluate both closed- and open-source LLM-based agents and use Claude Code with Claude Opus 5 (*Claude*) and OpenCode with Qwen3.8-27B (*Qwen*) respectively. For both agents, we use the maximum available thinking effort and the default context window length: $1M$ for *Claude* and $256K$ for *Qwen*. To ensure fair evaluation, both agents use the same harness: the agents can only inspect the codebase and launch experiments by changing existing configurations, and are prohibited from modifying the codebase. We enforce these restrictions through filesystem permissions: only the assigned configuration and experiment directories are writable, while unrelated files and directories are read-only or inaccessible. We set the optimization GPU-hour limit in Algorithm 1 to $G_{\max} = 500$ for the reasoning and multimodal tasks, whereas for the agentic task we set $G_{\max} = 1{,}500$. The agentic task receives a larger budget because its reference run consumes 135.0 GPU-hours, against 84.1 and 45.2 for the reasoning and multimodal tasks. All tasks have a six-day wall-clock limit $W_{\max}$. We set the optimization threshold in Eq. 1 with $\delta_Q, \delta_T, \delta_C = 0$ for all tasks. This requires each candidate to improve throughputs and use fewer training GPU-hours, while maintaining model quality than the reference configuration. To test whether L in Algorithm 1 captures transferable insights and reflects properties of the asynchronous RL infrastructure, we pass it in both directions between the agentic task and each of the single-turn tasks (i.e., reasoning, multimodal). In each direction, the target agent receives only the optimization prior L, a `lessons.md` produced by the source agent. Example `lessons.md` can be found in the Appendix.

5 RESULTS

5.1 AGENT OPTIMIZATION RESULTS

Across the three RL post-training tasks, Table 2 reports the post-trained model's quality, throughput, and training cost (Q, T, C) using the lowest-cost qualifying configuration (Eq. 1) discovered by the agents. We also categorize and report the configuration changes compared to the reference.

The results show that released reference configurations are suboptimal. Across the three tasks, agent-discovered configurations reduce training cost by 4.5–49.5% and raise throughput by up to $2\times$, with matching model quality. On the reasoning task, Claude and Qwen reduce the training costs from 84.1 to 42.5 and 79.3 GPU-hours, while increasing the throughput from 3.33K to 4.21K and 3.38K tokens/GPU-s respectively. The multimodal task yields the largest throughput gain: Qwen doubles it from 1.13K to 2.27K tokens/GPU-s while reducing training cost from 45.2 to 31.6 GPU-hours, and Claude reaches 1.82K tokens/GPU-s using 40.6 GPU-hours. On the agentic task the two agents diverge: Claude reduces cost from 135.0 to 102.1 GPU-hours and raises throughput from 1.61K to 2.26K tokens/GPU-s, whereas Qwen finds no qualifying configuration within its resource budget, although later we show that seeded with an optimization prior Qwen is still able to find a qualifying configuration. For all agent optimizations, the model quality is maintained or improved compared to the reference.

Post-hoc categorization of the configuration changes explains where these improvements come from and what each agent optimized for. Starting from the RL algorithm, both agents disable KL regularization on the multimodal task. The KL term penalizes deviation from the reference policy, but computing it requires an additional reference-model forward pass. Removing it therefore reduces the computation per step. Next, the rollout stage can be optimized by adjusting tensor parallelism (TP),

			Outcome			RL algo.		Policy Rollout				Policy training					Asynchrony	
Task	Agent	Prior L	Q (%)	T (K/GPU-s)	C (GPU-h)	KL	GRPO	TP	Conc.	DB (K)	Resp. (K)	lr (10^{-6})	DB (K)	FSDP	Liger	Samples (K)	GPU split	Stale
Reason.	Ref	–	18.0	3.33	84.1	off	16	1	16	10.2	8.2	1.0	20.5	2	off	819.2	4:4	0.1
	CC	–	18.1	4.21	42.5	off	16	1	**28**	**16.4**	8.2	**3.0**	**30.7**	**4**	**on**	**491.5**	4:4	0.1
	Qw	–	18.5	3.38	79.3	off	16	1	16	10.2	8.2	1.0	**28.7**	**4**	off	819.2	4:4	**0.4**
	CC	Ag	18.1	4.15	60.8	off	16	1	**32**	10.2	8.2	**3.0**	**41.0**	**5**	off	**691.2**	**3:5**	0.1
	Qw	Ag	18.7	3.36	80.3	off	16	1	16	**8.2**	**6.1**	1.0	20.5	2	off	819.2	4:4	0.1
Multim.	Ref	–	54.9	1.13	45.2	on	4	1	16	32.8	2.0	1.0	5.1	4	off	204.8	4:4	0.1
	CC	–	57.6	1.82	40.6	**off**	**16**	1	16	32.8	2.0	1.0	**24.6**	4	off	**327.7**	4:4	0.1
	Qw	–	56.2	2.27	31.6	**off**	4	1	**48**	32.8	2.0	**3.0**	**10.2**	**6**	off	**289.0**	**2:6**	0.1
	CC	Ag	57.7	1.81	36.9	**off**	4	1	**32**	32.8	2.0	**2.0**	5.1	**5**	off	**288.0**	**3:5**	0.1
	Qw	Ag	56.1	1.68	32.3	**off**	4	1	16	32.8	2.0	**1.5**	**8.2**	4	off	204.8	4:4	0.1
Agentic	Ref	–	27.2	1.61	135.0	off	16	4	16	18.4	16.4	1.0	18.4	4	off	102.4	4:4	0.5
	CC	–	29.5	2.26	102.1	off	16	**2**	16	18.4	16.4	**2.0**	18.4	4	off	102.4	4:4	0.5
	Qw	–	*no qualifying configuration within budget*															
	CC	Re	28.3	2.61	121.3	off	16	**1**	16	18.4	16.4	**2.0**	18.4	4	**on**	**81.9**	4:4	0.5
	CC	Mm	27.3	2.89	94.4	off	16	**1**	16	18.4	16.4	**2.0**	18.4	4	off	**122.9**	4:4	**1**
	Qw	Re	32.2	2.97	91.7	off	16	**1**	16	18.4	**10.2**	**2.0**	18.4	4	off	**122.9**	4:4	0.5
	Qw	Mm	*no qualifying configuration within budget*															

Table 2: Agentic optimization results and the configuration changes. Values in **black bold** are changed by the agent and grey values are unchanged from the reference configuration (*Ref*). *CC*: Claude Code with Claude Opus 5; *Qw*: OpenCode with Qwen3.8-27B; Q, T, C: model quality, throughput, and training costs; Conc.: rollout concurrency; DB: dynamic-batch token limit; Resp.: maximum response length; Samples: total training samples; Stale: policy staleness threshold.

rollout concurrency (Conc.), dynamic batching token limits (DB), and maximum response sequence length (Resp.). Specifically, TP shards the model across multiple GPUs, which reduces per-device memory usage at the cost of additional tensor communication. Since a single Nvidia H100 is able to accommodate a 7B model without memory constraints, both Claude and Qwen reduce TP on the agentic task to reduce tensor communication. Rollout concurrency and the dynamic-batch token limit both affect GPU utilization, as they increase the number of tokens the rollout engine decodes in a single forward pass. On the two shorter-response tasks (i.e. reasoning and multimodal), both agents raise rollout concurrency, while the dynamic-batch token limit and the maximum response length are changed only on the reasoning task.

Policy training is optimized through the learning rate (lr), the training dynamic-batch token limit (DB), FSDP size, and fused kernels (Liger). A common optimization strategy is to increase both the training dynamic-batch token limit and FSDP while using Liger kernels. Similar to policy rollout, increasing the training dynamic-batch token limit improves GPU utilization, but requires additional GPU memory usage. To accommodate this, the FSDP size is increased to shard the model across more GPUs and Liger kernels are enabled to reduce the memory held by intermediate activations. Moreover, the learning rate is changed most often, raised in 9 of the 12 configurations in Table 2 across all tasks. This suggests the base model is under-saturated on these tasks and increasing the learning rate hence speeds up convergence, reducing the overall training time.

Finally, the asynchronous execution is optimized through the rollout–training GPU split and the staleness threshold (Stale) in Table 2. By default, the reference configuration allocates equal GPUs to rollout and training in all tasks. In reality, policy training is compute-bound, and allocating more devices to it alleviates the bottleneck. Meanwhile the rollout side is oversubscribed. For instance, in the multimodal task, the default four rollout H100s are more than its 4 responses (GRPO=4) per prompt require. To better utilize the rollout GPUs, Claude increases the GRPO group size from 4 to 16. Meanwhile, Qwen moves 2 GPUs from rollout to training workers (GPU split 2:6). This results in the lowest training cost and highest throughput in the multimodal task among all configurations, reducing the training GPU-hours from 45.2 to 31.6, while roughly doubling the throughput. To better balance policy training and rollout, the staleness threshold is also raised in two configurations in Table 2, as it lets the rollout stage run further ahead without being bottlenecked by the trainer.

5.2 Cross-Task Transfer of Optimization Priors

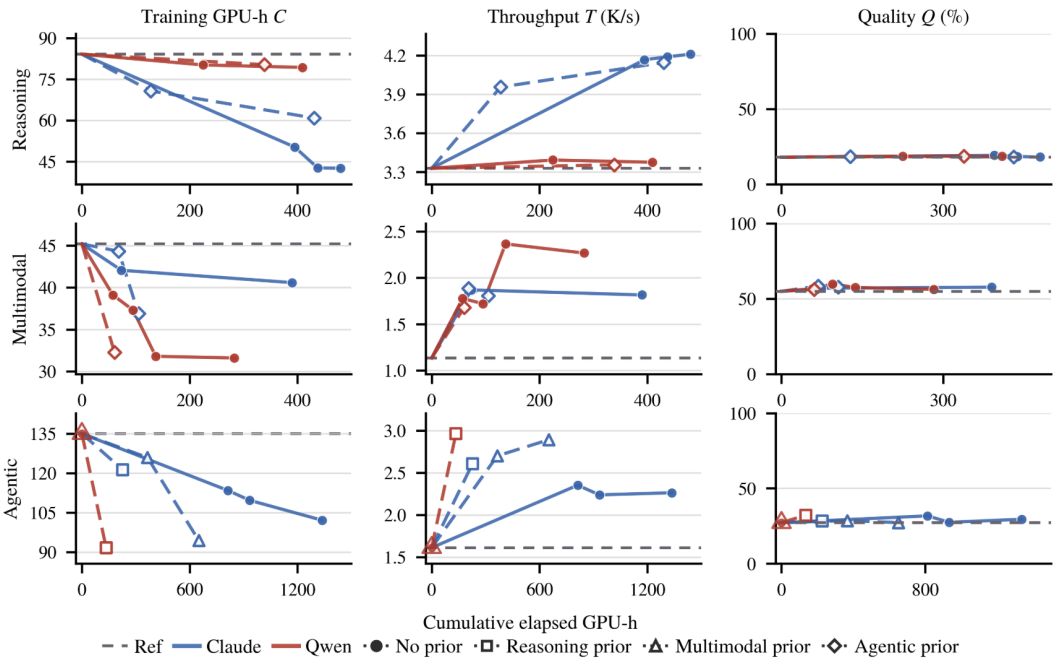

Figure 2: Agentic optimization trajectories on the reasoning, multimodal and agentic tasks as a function of cumulative elapsed GPU-h versus training cost, throughout and model quality.

We examine the effect of an optimization prior L distilled from a different task in Figure 2 and Table 2. In Figure 2, we plot when the agents discover qualifying configurations (Eq. 1) against time, measured by the cumulative GPU-hours the agents spent. Filled markers and solid lines denote optimizations without a prior, while open markers and dashed lines denote those seeded with a prior from another task.

For Claude, agents seeded with a prior consistently discover first qualifying configurations earlier than those without a prior across all three tasks. On the reasoning task, Claude's first qualifying configuration is discovered after 127.8 cumulative GPU-hours with an optimization prior, against 394.8 without one, as shown in Figure 2. The multimodal task shows the same improvement: Claude discovers a qualifying configuration after 68.2 GPU-hours when seeded with a prior versus 73.4 cumulative GPU-hours without a prior. The improvement is the greatest on the agentic task, where Claude with the reasoning prior reaches a qualifying configuration after 226.9 GPU-hours and the multimodal prior after 365.1, against 812.7 without a prior. This indicates that the optimization prior can be used to accelerate optimization on a different task.

Qwen, however, does not show the consistent acceleration behavior with optimization priors, although a prior occasionally enables it to find a qualifying configuration where the no-prior search finds none. For the agentic task, Qwen without an optimization prior or with the multimodal prior both fail to find qualifying configurations within the resource budget. When seeded with the reasoning prior, Qwen discovers a qualifying configuration after 135.0 GPU-hours, and achieves the lowest training cost among all configurations in Table 2.

5.3 Practical Guidance for Configuring Asynchronous RL Training

Policy training and rollout should first be optimized individually, removing potential bottlenecks. Although the decoupled design enables policy training and rollout to run on different engines, both can be improved in the same 3 aspects by reducing communication, increasing utilization, or reducing computation. Table 3 summarizes the settings the agents changed in Section 5.2, counted across all qualifying optimization runs. Policy training can be optimized, for instance, when the

Mechanism	Policy training	Policy rollout	Rollout–training balance
Communication ↓	Model offloading (4) Reshard after forward (4)	Tensor parallelism (7)	–
Utilization ↑	Dynamic-batch tokens (14) FSDP size (7) Liger kernels (4)	Rollout concurrency (11) Dynamic-batch tokens (4)	GPU allocation (6) Staleness threshold (3)
Computation ↓	Learning rate (14) KL regularization (9)	Max response length (2)	–

Table 3: Configuration changes by mechanism and training stage. Rows are the three mechanisms through which a change can improve training; columns are the three places a change can act. (Grey) counts give how many of the 24 qualifying runs in Figure 2 changed each setting. These settings are coupled, so a count is not a measure of importance and no single direction of change applies across tasks: which setting to move, and which way, depends on where the bottleneck lies. Raising the learning rate reduces the number of updates needed to reach a target quality, rather than the computation per update.

tensor communication is minimized (e.g., skip offloading weights, FSDP) and the utilization is maximized (e.g., dynamic-batch token limit, liger kernels). These factors need to be jointly considered during optimization. For instance, a larger FSDP size reduces the memory usage on each GPU, allowing for larger dynamic-batch token limit at the cost of more GPU communications. Moreover, to improve training efficiency, additional computation can be skipped by removing KL loss computation against the reference model. Increasing the learning rate for under-saturated models can also speed up convergence, thus reducing additional computation. Policy rollout can be optimized similarly, for instance, by balancing the tensor communication and memory usage (e.g., tensor parallelism), and maximizing GPU utilization (e.g., dynamic-batch token limit, concurrent samples during policy rollout). Rollout computation can likewise be reduced, for instance by lowering the maximum generated response length, at the risk of truncating long reasoning traces.

The advantage of asynchronous RL training is maximized when the speed of policy training and rollout is matched. A mismatch can leave the faster one idle, reducing asynchronous advantages. As shown in Figure 1, if training outpaces rollout, the trainer will exhaust the rollout buffer faster and wait for new rollouts to be generated. Conversely, if the rollout worker is faster, the staleness limit can pause generation until the trainer catches up and synchronizes updated policy weights to the rollout worker Zhiqiang et al. (2026). In practice, once each stage has been optimized individually, their speed should be profiled to determine which side is the bottleneck. The balance between policy training and rollout can then be optimized by adjusting the total training resources (i.e., GPU split). As unbalanced training can either be training or rollout-bound, allocating resources from over- to under-provision workers can improve the overall balance. Adjusting the staleness threshold can also improve the overall balance at the cost of potential policy staleness or model performance degradation.

6 Conclusion

We studied whether an AI agent can optimize asynchronous RL training. We formulated this task as a resource-bounded optimization problem over model quality, median throughput, and training GPU-hours, and introduced an agentic protocol in which the agent inspects the infrastructure, proposes hypotheses, conducts proxy and full experiments, and uses the resulting history to guide subsequent decisions. Across three LLM post-training workloads, the agent finds configurations that improve model quality and throughput while reducing training GPU-hours by 4.5–49.5% relative to the official human-configured defaults. These findings provide practical insights into effective and efficient RL training.

REFERENCES

Mislav Balunovic, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. Matharena: Evaluating llms on uncontaminated math competitions. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-0679. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/1d27c01ebd3e3aebe226b44fc970d803-Paper-Datasets_and_Benchmarks_Track.pdf.

Minping Chen, Bowen Xiao, Du Liang, Chuxuan Zeng, and Zeyi Wen. Efficient hyperparameter optimization for LLM reinforcement learning. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2026, San Diego, California, United States, July 2-7, 2026*, pp. 27540–27552. Association for Computational Linguistics, 2026. doi: 10.18653/V1/2026.ACL-LONG.1271. URL https://doi.org/10.18653/v1/2026.acl-long.1271.

Theresa Eimer, Marius Lindauer, and Roberta Raileanu. Hyperparameters in reinforcement learning and how to tune them. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 9104–9149. PMLR, 2023. URL https://proceedings.mlr.press/v202/eimer23a.html.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1407–1416. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/espeholt18a.html.

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 C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 37909–37926, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/4038c9208dfc22644c60ad39c24e5c53-Paper-Conference.pdf.

Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. AREAL: A large-scale asynchronous reinforcement learning system for language reasoning. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/33c00862bfa29ac72ecf630a41e19352-Abstract-Conference.html.

Zixun Guo, Calvin Murdock, Sanjeel Parekh, W. Owen Brimijoin, Simon Dixon, Joshua Reiss, and Ishwarya Ananthabhotla. Sphere: Automatic music upmixing via audio language model post-training with spatial heuristics rewards. *arXiv preprint*, 2026. URL https://arxiv.org/html/2608.30559.

Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, and Yanning Chen. Liger kernel: Efficient triton kernels for LLM training. *CoRR*, abs/2410.10989, 2024. doi: 10.48550/ARXIV.2410.10989. URL https://doi.org/10.48550/arXiv.2410.10989.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model

serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (eds.), *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Harrison Lee, Samrat Phatale, Hassan Mansoor, Thomas Mesnard, Johan Ferret, Kellie Ren Lu, Colton Bishop, Ethan Hall, Victor Carbune, Abhinav Rastogi, and Sushant Prakash. RLAIF vs. RLHF: Scaling reinforcement learning from human feedback with AI feedback. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 26874–26901. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/lee24t.html.

Haoyang Li, Sheng Lin, Fangcheng Fu, Yuming Zhou, Xiaodong Ji, Yanfeng Zhao, Lefeng Wang, Jie Jiang, and Bin Cui. Unleashing efficient asynchronous rl post-training via staleness-constrained rollout coordination. *arXiv preprint arXiv:2601.12784*, 2026.

Siyi Liu, Chen Gao, and Yong Li. Agenthpo: Large language model agent for hyper-parameter optimization. In Beidi Chen, Shijia Liu, Mert Pilanci, Weijie Su, Jeremias Sulam, Yuxiang Wang, and Zhihui Zhu (eds.), *Conference on Parsimony and Learning, Stanford University, USA, 24-27 March 2025*, volume 280 of *Proceedings of Machine Learning Research*, pp. 1146–1169. PMLR, 2025. URL https://proceedings.mlr.press/v280/liu25c.html.

Pan Lu, Ran Gong, Shibiao Jiang, Liang Qiu, Siyuan Huang, Xiaodan Liang, and Song-Chun Zhu. Inter-gps: Interpretable geometry problem solving with formal language and symbolic reasoning. In *The Joint Conference of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (ACL-IJCNLP 2021)*, 2021.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using megatron-lm. In Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (eds.), *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, pp. 58. ACM, 2021. doi: 10.1145/3458817.3476209. URL https://doi.org/10.1145/3458817.3476209.

Michael Noukhovitch, Shengyi Huang, Sophie Xhonneux, Arian Hosseini, Rishabh Agarwal, and Aaron C. Courville. Asynchronous RLHF: faster and more efficient off-policy RL for language models. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=FhTAG591Ve.

NVIDIA NeMo Team. NeMo RL: A scalable and efficient post-training library. https://github.com/NVIDIA-NeMo/RL, 2025. GitHub repository.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 27730–27744. Curran Associates, Inc., 2022. doi: 10.52202/068431-2011. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf.

Alexandre Piché, Ehsan Kamalloo, Rafael Pardinas, Xiaoyin Chen, and Dzmitry Bahdanau. Pipelinerl: Faster on-policy reinforcement learning for long sequence generation. *Trans. Mach. Learn. Res.*, 2026, 2026. URL https://openreview.net/forum?id=A35ak14Cyp.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In Christine Cuicchi, Irene Qualters, and William T.

Kramer (eds.), *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, pp. 20. IEEE/ACM, 2020. doi: 10.1109/SC41405.2020.00024. URL https://doi.org/10.1109/SC41405.2020.00024.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300, 2024. doi: 10.48550/ARXIV.2402.03300. URL https://doi.org/10.48550/arXiv.2402.03300.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient RLHF framework. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, pp. 1279–1297. ACM, 2025. doi: 10.1145/3689031.3696075. URL https://doi.org/10.1145/3689031.3696075.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

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. *CoRR*, abs/2505.24034, 2025. doi: 10.48550/ARXIV.2505.24034. URL https://doi.org/10.48550/arXiv.2505.24034.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Bb4VGOWELI.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, juncai liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 113222–113244. Curran Associates, Inc., 2025. doi: 10.52202/085713-3775. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/a4277440d50f1f15d2cb4c14f7e0c0d2-Paper-Conference.pdf.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch FSDP: experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, 2023. doi: 10.14778/3611540.3611569. URL https://www.vldb.org/pvldb/vol16/p3848-huang.pdf.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/724be4472168f31ba1c9ac630f15dec8-Paper-Conference.pdf.

Tan Zhiqiang, Wang Maoxin, Wang Sijie, Yin Yiming, Wang Qiang, Chu Xiaowen, and Shi Shaohuai. Bidirectional resource scheduling for disaggregated and asynchronous rl post-training. *arXiv.2607.09207*, 2026.

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. *CoRR*, abs/2504.15930, 2025. doi: 10.48550/ARXIV.2504.15930. URL https://doi.org/10.48550/arXiv.2504.15930.

7 AI USE STATEMENT

This paper focuses on optimizing asynchronous RL post-training for LLMs using LLM-based agents. As described in Section 3 and Section 4, we use Claude Code with Claude Opus 5 and OpenCode with Qwen3.8-27B as optimization agents. These agents inspect an asynchronous RL codebase, propose and modify configurations, launch and monitor experiments and evaluate checkpoints. We report the results of the agentic optimization and the discovered configurations in Table 2.

We use generative AI assistants (Gemini, Claude and Codex) to improve the grammar, spelling, and phrasing of the authors' original text, and to help format LaTeX tables. We also used Claude Code to assist in implementing the codebase, all of which was manually verified for correctness by the first author.

8 LIMITATIONS

Our evaluation covers three asynchronous RL post-training tasks (i.e., reasoning, multimodal and agentic) and our tasks are all verifiable. The reward calculation is handled by CPU workers and does not consume GPU time (Figure 1). Other grading mechanisms (e.g., external LLM-as-judges) might need additional profiling and optimization. We do not cover other families of RL post-training, such as RL for diffusion models, whose rollout and training stages have different natures than autoregressive transformers. All three tasks post-train models with 7B parameters. Since model size determines how much device memory the model weights, optimizer states and KV cache consume, it affects settings such as parallelism and parameter offloading. Models with different sizes might be optimized differently. Moreover, we also only use GRPO, a widely adopted algorithm in LLM post-training which does not require a separate critic model. Other RL algorithms such as PPO might have different bottlenecks and require different configurations. We further restrict the agents to editing configurations and prohibit changes to the codebase. This isolates what configuration alone can achieve and keeps the comparison against the released reference configurations well defined, but it also potentially restricts what the agents can find.

Finally, each reported configuration comes from a single run, which in total already consumes around $15.5K$ GPU-hours for all experiments in Table 2. We choose not to duplicate experiments due to resource constraints. We also do not compare our approach against conventional hyperparameter optimization methods. We aim to discover insights into asynchronous RL training and identify common principles rather than proposing a better search algorithm.

APPENDIX

9 RELATED WORKS

Existing work has explored automated hyperparameter search through conventional HPO Chen et al. (2026) and, more recently, LLMs and agents Yang et al. (2024); Liu et al. (2025). Few studies focus on HPO for RL: Eimer et al. (2023) studies conventional RL, while Chen et al. (2026) focuses on RL post-training for LLMs. These methods, however, usually pre-define a limited hyperparameter search space Liu et al. (2025); Chen et al. (2026); Eimer et al. (2023) such as learning rate or batch size. LLM-based optimizers such as OPRO Yang et al. (2024) relax this assumption by generating open-ended candidates from natural-language instructions and previous evaluation results.

Async RL systems expose a much larger configuration space Sheng et al. (2025); NVIDIA NeMo Team (2025), yet which parameters affect efficiency and model quality remains under-discovered. Different from existing approaches, we aim to discover both which parameters to tune and how to set their values, without predefining the search space. Moreover, existing work focuses on small-scale ML experiments and optimizes primarily for model quality. RL tasks, however, are resource-intensive Chen et al. (2026), and our hyperparameter optimization objective takes both model performance and the overall training efficiency into account.

10 REFERENCE CONFIGURATIONS

Configuration in VeRL is composed with Hydra. A base configuration includes separate configurations for the actor, reference model, rollout engine, data, model, critic, reward, algorithm and optimizer, and the fully asynchronous trainer layers an `async_training` group on top of it through `fully_async_ppo_trainer.yaml`. In total, the default configuration exposes around 800 settings, which our agents are allowed to modify.[3]

The per-task reference configurations listed override the default configurations. They set around 80 of those settings for the reasoning, multimodal and agentic tasks respectively, and inherit the remaining. Each layer derives from the asynchronous recipe VeRL releases for that task: `dapo_7b_math_fsdp2_4_4.sh` for reasoning, `geo3k_qwen25vl_7b_megatron_4_4.sh` for multimodal, and `dapo_7b_async_retool.sh` for the agentic task.[4]

Reference configuration, reasoning task

```
algorithm:
  adv_estimator: grpo
  use_kl_in_reward: false
  kl_ctrl:
    kl_coef: 0.0

data:
  train_files: ['${oc.env:ASYNCRL_DATA_ROOT}/dapo/dapo-math-17k.parquet']
  val_files: ['${oc.env:ASYNCRL_DATA_ROOT}/dapo/aime-2024.parquet']
  seed: 42
  prompt_key: prompt
  truncation: left
  max_prompt_length: 2048
  max_response_length: 8192
```

[3] Default RL Training Configuration: https://github.com/verl-project/verl/blob/76e14c9a1b2861c61305b68c71b24c2ebc047ec8/verl/trainer/config/_generated_ppo_trainer.yaml. Default Asynchronous Training Configuration: https://github.com/verl-project/verl/blob/76e14c9a1b2861c61305b68c71b24c2ebc047ec8/verl/experimental/fully_async_policy/config/fully_async_ppo_trainer.yaml.

[4] Reasoning: https://github.com/verl-project/verl/blob/76e14c9a1b2861c61305b68c71b24c2ebc047ec8/verl/experimental/fully_async_policy/shell/dapo_7b_math_fsdp2_4_4.sh. Multimodal: https://github.com/verl-project/verl/blob/76e14c9a1b2861c61305b68c71b24c2ebc047ec8/verl/experimental/fully_async_policy/shell/geo3k_qwen25vl_7b_megatron_4_4.sh. Agentic: https://github.com/verl-project/verl/blob/76e14c9a1b2861c61305b68c71b24c2ebc047ec8/verl/experimental/fully_async_policy/shell/dapo_7b_async_retool.sh.

```
  train_batch_size: 0
  gen_batch_size: 1
  return_raw_chat: true

actor_rollout_ref:
  hybrid_engine: false
  model:
    path: Qwen/Qwen2.5-Math-7B
    use_remove_padding: true
  actor:
    optim:
      lr: 1.0e-6
      lr_warmup_steps: 10
      weight_decay: 0.1
    ppo_mini_batch_size: 32
    use_dynamic_bsz: true
    ppo_max_token_len_per_gpu: 20480
    use_kl_loss: false
    kl_loss_coef: 0.0
    clip_ratio_low: 0.2
    clip_ratio_high: 0.28
    clip_ratio_c: 10.0
    entropy_coeff: 0
    grad_clip: 1.0
    loss_agg_mode: token-mean
    ulysses_sequence_parallel_size: 1
    fsdp_config:
      strategy: fsdp2
      fsdp_size: 2
      param_offload: false
      optimizer_offload: false
  rollout:
    name: vllm
    mode: async
    tensor_model_parallel_size: 1
    gpu_memory_utilization: 0.80
    n: 16
    calculate_log_probs: true
    enable_chunked_prefill: true
    max_num_batched_tokens: 10240
    temperature: 1.0
    top_p: 1.0
    top_k: -1
    log_prob_use_dynamic_bsz: true
    log_prob_max_token_len_per_gpu: 30720
    val_kwargs:
      temperature: 1.0
      top_p: 0.7
      top_k: -1
      do_sample: true
      n: 30
  ref:
    log_prob_use_dynamic_bsz: true
    log_prob_max_token_len_per_gpu: 30720
    ulysses_sequence_parallel_size: 1
    fsdp_config:
      param_offload: true

critic:
  strategy: fsdp2

reward:
  reward_manager:
    name: dapo
  reward_kwargs:
    overlong_buffer_cfg:
      enable: true
      len: 4096
      penalty_factor: 1.0
      log: false
    max_resp_len: 8192

async_training:
  staleness_threshold: 0.1
  trigger_parameter_sync_step: 4
  require_batches: 4
  partial_rollout: true

rollout:
  nnodes: 1
  n_gpus_per_node: 4
  total_rollout_steps: 51200

trainer:
  nnodes: 1
  n_gpus_per_node: 4
  balance_batch: true
  logger: ['console', 'tensorboard']
  project_name: dapo_math_async
  experiment_name: qwen25_math_7b_dapo_async_4x4
```

```
  val_before_train: true
  resume_mode: auto
  save_freq: 50
  test_freq: 10
  total_epochs: 10
```

Reference configuration, multimodal task

```
algorithm:
  adv_estimator: grpo
  use_kl_in_reward: false
  kl_ctrl:
    kl_coef: 0.0

data:
  train_files: ['${oc.env:ASYNCRL_DATA_ROOT}/geo3k/train.parquet']
  val_files: ['${oc.env:ASYNCRL_DATA_ROOT}/geo3k/test.parquet']
  image_key: images

  seed: 42
  prompt_key: prompt
  truncation: error
  filter_overlong_prompts: true
  max_prompt_length: 1024
  max_response_length: 2048
  train_batch_size: 0
  gen_batch_size: 1
  return_raw_chat: true

actor_rollout_ref:
  hybrid_engine: false
  model:
    path: Qwen/Qwen2.5-VL-7B-Instruct
    use_remove_padding: true
    enable_gradient_checkpointing: true
    use_fused_kernels: true
  actor:
    optim:
      lr: 1.0e-6
    ppo_mini_batch_size: 128
    use_dynamic_bsz: true
    ppo_max_token_len_per_gpu: 5120
    use_kl_loss: true
    kl_loss_coef: 0.01
    kl_loss_type: low_var_kl
    entropy_coeff: 0
    grad_clip: 1.0
    loss_agg_mode: token-mean
    ulysses_sequence_parallel_size: 1
    fsdp_config:
      fsdp_size: -1
      param_offload: false
      optimizer_offload: false
  rollout:
    name: vllm
    mode: async
    tensor_model_parallel_size: 1
    gpu_memory_utilization: 0.80
    n: 4
    calculate_log_probs: true
    max_model_len: 32768
    max_num_batched_tokens: 32768
    enable_chunked_prefill: false
    free_cache_engine: true
    multi_stage_wake_up: true
    enforce_eager: false
    temperature: 1.0
    top_p: 1.0
    top_k: -1
    log_prob_use_dynamic_bsz: true
    log_prob_max_token_len_per_gpu: 5120
    val_kwargs:
      temperature: 1.0
      top_p: 0.7
      top_k: -1
      do_sample: true
      n: 4
  ref:
    log_prob_use_dynamic_bsz: true
    log_prob_max_token_len_per_gpu: 5120
    ulysses_sequence_parallel_size: 1
    fsdp_config:
      param_offload: true

critic:
```

```
  strategy: fsdp2

reward:
  reward_manager:
    name: naive
async_training:
  staleness_threshold: 0.1
  trigger_parameter_sync_step: 4
  require_batches: 2
  partial_rollout: true

rollout:
  nnodes: 1
  n_gpus_per_node: 4
  total_rollout_steps: 51200

trainer:
  nnodes: 1
  n_gpus_per_node: 4
  balance_batch: true
  logger: ['console', 'tensorboard']
  project_name: geo3k_vl_async
  experiment_name: qwen25_vl_7b_geo3k_async_4x4
  val_before_train: true
  resume_mode: auto
  save_freq: 50
  test_freq: 10
  total_epochs: 200
```

Reference configuration, agentic task

```
algorithm:
  adv_estimator: grpo
  use_kl_in_reward: false
  kl_ctrl:
    kl_coef: 0.0

data:
  train_files: ['${oc.env:ASYNCRL_DATA_ROOT}/retool-rl/dapo-math-17k.parquet']
  val_files: ['${oc.env:ASYNCRL_DATA_ROOT}/retool-rl/aime-2024.parquet']
  prompt_key: prompt
  return_raw_chat: true
  truncation: error
  filter_overlong_prompts: true
  max_prompt_length: 2048
  max_response_length: 16384
  train_batch_size: 0
  gen_batch_size: 1

actor_rollout_ref:
  hybrid_engine: false
  model:
    path: /path/to/checkpoints/multiturn-sft-qwen2.5-7b-retool
    use_remove_padding: true
    enable_gradient_checkpointing: true
  actor:
    optim:
      lr: 1.0e-6
    ppo_mini_batch_size: 16
    use_dynamic_bsz: true
    ppo_max_token_len_per_gpu: 18432
    use_kl_loss: false
    kl_loss_coef: 0.0
    clip_ratio_low: 0.2
    clip_ratio_high: 0.28
    clip_ratio_c: 10.0
    entropy_coeff: 0
    grad_clip: 1.0
    loss_agg_mode: token-mean
    ulysses_sequence_parallel_size: 4
    fsdp_config:
      strategy: fsdp2
      fsdp_size: 4
      param_offload: true
      optimizer_offload: true
  ref:
    log_prob_use_dynamic_bsz: true
    log_prob_max_token_len_per_gpu: 73728
    ulysses_sequence_parallel_size: 4
    fsdp_config:
      param_offload: true
  rollout:
    name: vllm
    mode: async
    tensor_model_parallel_size: 4
```

```
    gpu_memory_utilization: 0.7
    checkpoint_engine:
      update_weights_bucket_megabytes: 256
    n: 16
    calculate_log_probs: true
    enable_chunked_prefill: true
    max_num_batched_tokens: 18432
    temperature: 1.0
    top_p: 1.0
    top_k: -1
    log_prob_use_dynamic_bsz: true
    log_prob_max_token_len_per_gpu: 73728
    val_kwargs:
      temperature: 1.0
      top_p: 0.7
      top_k: -1
      do_sample: true
      n: 30
    multi_turn:
      enable: true
      max_user_turns: 16
      max_assistant_turns: 10
      function_tool_path: ${oc.env:ASYNCRL_CONFIG_DIR}/human_config/multi_turn/tools/code_interpreter
          .py
      format: hermes
      max_tool_response_length: 2048

critic:
  strategy: fsdp2

reward:
  custom_reward_function:
    path: ${oc.env:ASYNCRL_CONFIG_DIR}/human_config/multi_turn/tools/retool_reward.py
    name: compute_score

async_training:
  staleness_threshold: 0.5
  trigger_parameter_sync_step: 4
  require_batches: 1
  partial_rollout: true

rollout:
  nnodes: 1
  n_gpus_per_node: 4
  total_rollout_steps: 16000

trainer:
  nnodes: 1
  n_gpus_per_node: 4
  balance_batch: true
  logger: ['console', 'tensorboard']
  project_name: dapo_math_async_tool
  experiment_name: qwen25_7b_dapo_async_retool_4x4
  val_before_train: true
  log_val_generations: 20
  resume_mode: auto
  save_freq: 50
  test_freq: 10
  total_epochs: 10
```

11 Closed-source Agent Token Usage

Table 4 reports the tokens the closed-source agent consumed over each optimization. The four types are billed separately. Cache reads dominate by two orders of magnitude, reflecting that most of an agent's work is examining the codebase, configuration and experiment histories rather than generating new text; We report the closed-source agent only, as the the open-source agent was run locally-hosted vLLM instances and did not incur any API costs.

12 Example Optimization Prior

An optimization prior L is the `lessons.md` an agent summarizes at the end of its optimization, and only L is passed to the agent optimizing a different task. Below are the example priors Claude and Qwen produced. Note that the advice is largely about *where to look* rather than about which values to set.

Task	Prior L	Input	Output (M)	Cache write (M)	Cache read (M)
Reason.	–	1,917	1.01	3.69	254.7
Reason.	Ag	910	0.76	5.77	163.8
Multim.	–	918	0.67	3.29	156.5
Multim.	Ag	834	0.61	4.34	121.9
Agentic	–	1,350	0.81	4.19	281.4
Agentic	Re	1,426	0.93	8.08	297.1
Agentic	Mm	2,144	0.88	9.48	227.0
Total		9,499	5.68	38.83	1,502.4

Table 4: Tokens consumed by the closed-source agent (Claude Code with Claude Opus 5) over each optimization, by billing type.

The optimization prior produced by the closed-source agent on the reasoning task

```
## Cross-task Optimization Prior

1. Fit `update_actor` and `step` against `total_num_tokens` on the baseline. Ten minutes, no
   GPU, and it caps the whole overhead-amortisation family at ~2.5%.
2. Compute **saturated** rollout capacity in gen tok/s per rollout GPU and compare it to
   demand. Do **not** trust `idle_ratio`. This is the number that decides the GPU split.
3. `grep` the call site of any flag before spending on it. Two of my most attractive ideas
   (`use_prefix_grouper`, `filter_groups`) are config fields nothing reads on this path.
4. Turn on Liger first: one line, numerically neutral, -15.8%.
5. Raise `concurrent_samples_per_replica` early -- 16 caps everything else.
6. Screen on ms/1k-tok, never on proxy throughput.
7. **Start accuracy work first and spend most of the budget there.** Efficiency here is
   a solved, cheap, well-conditioned problem -- I had +24% locked in within about six hours and
   a quarter of the budget. Accuracy is the binding constraint, it cannot be proxied, and its
   optimum is at a *shorter* horizon than the baseline uses.
8. **Sweep the training horizon before sweeping anything else about the optimiser.** It was
   worth more than every other accuracy lever I tried put together: 0.1707 -> 0.1919 for lr 2e-6
   purely by stopping at 70 instead of 100, while also cutting cost 40%.
9. **The concrete config I would start the next episode from** (measured 0.1919 / 4168.0 /
   50.25 GPU-h, all three better than the human baseline):
   ```yaml
   actor_rollout_ref.model.use_liger: true
   actor_rollout_ref.actor.fsdp_config.fsdp_size: 4
   actor_rollout_ref.actor.fsdp_config.reshard_after_forward: false
   actor_rollout_ref.actor.ppo_max_token_len_per_gpu: 30720   # >= max_prompt+max_response
   actor_rollout_ref.actor.optim.lr: 2.0e-6
   async_training.concurrent_samples_per_replica: 28
   actor_rollout_ref.rollout.max_num_batched_tokens: 16384
   actor_rollout_ref.rollout.gpu_memory_utilization: 0.80     # do NOT raise
   rollout.total_rollout_steps: 35840                          # 70 versions, not 100
   trainer.save_freq: 10
   ```
   and the first thing I would try from there, with a budget that allows it, is the horizon
   between 70 and 100 at lr 2e-6 (I have 70 and 100 but nothing between), plus lr 1.5e-6 at
   ~80 -- the accuracy optimum clearly moves later as the LR comes down, and lr 1e-6/100 versions
   (the human baseline) and lr 2e-6/70 are two points on that same ridge.
```

The optimization prior produced by the open-source agent on the multimodal task

```
## Levers that worked (exact effects)
1. **actor.use_kl_loss: false** -- deletes the entire ref-policy phase (~71 s of 281 s/ver, `
      need_reference_policy = use_kl_in_reward OR use_kl_loss`). Quality did NOT degrade: C 0.5654 / f
       0.5984 > baseline 0.5488 over 70 ver. (KL-free instability shows later; at this horizon it's
      net-positive.) `old_log_probs` come free from the vLLM engine in bypass mode, so nothing
      recomputed.
2. **async_training.concurrent_samples_per_replica 16->48** -- was the rollout dispatcher cap (vLLM
      engine headroom: max_num_seqs 1024/engine, KV for ~1800 inflight samples). Math-neutral (GRPO
      group norm over fixed 4-seq groups) but unlocked rollouter utilization; prerequisite for any
      speed lever to transfer.
3. **actor.ppo_max_token_len_per_gpu 5120->10240** -- denser dynamic-bsz micro-batches on trainer
      side: update_actor 159->~120 s/ver (was packing <=5120 tok/gpu/mb with padding slack). No effect
       on rollout.
4. **trainer:rollout GPU split 4:4 -> 6:2** -- best single speed lever: warm ~1720->~2360 t/s/gpu
      (~+37%), wall 4->3.5 h-class. Mechanism: 2 fit-steps->... 6 FSDP shards x bigger toklen shrinks
      update_actor superlinearly; rollouters (2 eng x 512 seqs, conc 96 total) still saturate. **
      Divisibility gotcha**: 4096 seqs/fit-version / (2 reps x n_seqs-per-fit) must hit mini-batch
      alignment -- use ppo_mini_batch_size divisible by trainer world (6 -> 129: 4096/2fits... see g-
      proxy crash 1024%5!=0 with 5-way; h works at 129).
```

```
   - 7:1 is DEAD: single rollouter engine (max_num_seqs 1024, KV) caps; 3:5/2:5 DEAD: rollouter-
       starved trainer again.
5. **optim.lr 1e-6 -> 2e-6** -- pure quality lever: s40 in-run .5672 vs .5369 at 1e-6; zero
     measurable speed cost (microbatch packing dominates). In j, learning curve is ahead of BOTH f
     and h from s20.

## Dead ends (signature -> verdict, do-not-retry)
- **gpu_memory_utilization 0.85** -> OOM in CheckpointEngineWorker NCCL weight-bucket receive on
     rollout GPUs at FIRST param sync (free VRAM drops ~1.2 GB under 0.85 alloc). Ceiling = **0.80**
     on this stack. Killed triple cost 5.68 GPU-h (a4-a6).
- **gradient_checkpointing off** -> OOM in actor update at 85.4 GB PyTorch-alloc / 94.8 GB proc (7B
     VLM + fused kernels + VLA-style visual tower activations). **GC stays on**; reclaim speed via
     toklen batching instead.
- **5-GPU/odd splits**: dynamic-bucket assert `len(seqlen_list) % k_partitions == 0` (g-proxy, 1.698
     GPU-h). Even splits or mini sized to divisor.
```