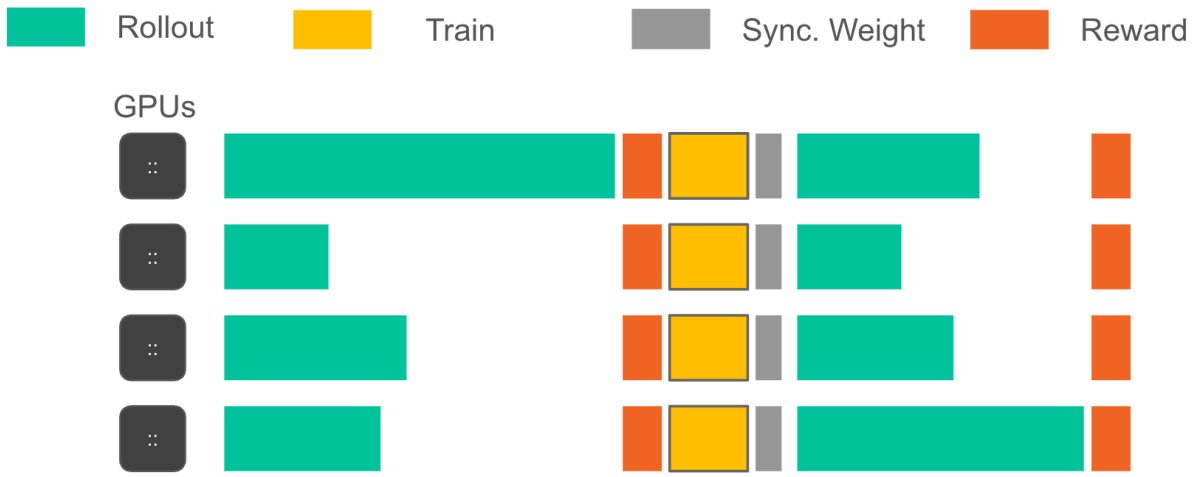
Asynchronous RL Training is Structurally Necessary for LLM Post-Training at Scale

- Zikai Wang wang.zikai1@northeastern.edu
- Northeastern University

Opening

How synchronous RLHF works. A standard RLHF training loop operates in lock-step. First, the policy model generates responses for a batch of prompts across all workers. Once every worker finishes generation, the system computes rewards and advantages and performs a gradient update. Then the cycle repeats. This synchronous design is simple and matches most distributed training setups: a global barrier ensures all workers see the same weights before the next iteration.

Where it breaks down. The problem lies in the generation phase, which consumes 70%-80% of total training time[1, 2]. Response lengths in LLM generation follow a long-tailed distribution [1]. But the synchronous barrier forces all workers to wait for the slowest one. As shown in Figure 1 (adapted from [1]), this leads to a small fraction of prompts generating exceptionally long responses make all other GPUs sit idle. A batch doesn't finish when most workers finish; it finishes when the last worker finishes, and GPU-hours burn with no useful work.



> **Figure 1. Execution workflows of synchronous RL post-training.**
> RollPacker: Mitigating Long-Tail Rollouts for Fast, Synchronous RL Post-Training. *arXiv preprint arXiv:2509.21009* (2025).

Asynchronous RL. Asynchronous architectures sidestep these problems by decoupling generation from training. Rollout workers stream trajectories continuously; Training proceeds

as soon as enough samples accumulate, without waiting for slower rollout workers. The result: 2-10× speedups at scale while matching learning performance [1, 3, 4, 5].

The Variance You Can't Engineer Away

Synchronous training assumes each worker performs roughly the same amount of work, as is usually true in data-parallel supervised learning. But LLM generation violates this assumption.

Response length is determined by the content of generation, for example, the model's choices, the problem's difficulty, the prompt's complexity. The scheduler cannot predict or control which prompts will produce long responses. Worse, it cannot filter them out: the hard problems that require long reasoning chains are exactly the ones you want the model to learn from. The variance is intrinsic to the task.

This creates a structural mismatch with synchronous training. In synchronous mode, effective throughput equals total work divided by the maximum worker time because of the synchronization barrier, not the mean. A single long response doesn't just slow down one worker, it drags down the efficiency of entire batch. Every other GPU that finished early sits idle and waiting, eliminating any benefit of scaling out.

Can synchronous systems mitigate this? Recent work has pushed hard on optimizations within the synchronous paradigm. Seer [6] introduces divided rollout, context-aware scheduling, and adaptive grouped speculative decoding. These techniques reduce long-tail latency by 75–93% while maintaining strict on-policy training. The VeRL framework [7] tries to over-sample for each prompt and early stop when it collects enough samples. These systems achieve impressive speedups, but they are fundamentally working against the nature of the workload. The key insight behind these optimizations is that responses to the same prompt often have correlated lengths. This makes it possible to use early-finishing samples to predict which groups will produce stragglers and mitigate intra-group variance. This only works when variance comes primarily from the model's generation process. But what happens when variance comes from external sources [7]—tool calls, API latencies, code execution times, or robot interactions that have no correlation with the prompt?

Agentic Workloads Amplify the Problem

The straggler problem is manageable for standard RLHF with single-turn text generation, because response length variance is limited. For agentic workloads, such as training LLMs to use tools, browse the web, or write and execute code, it becomes qualitatively harder. Agent trajectories introduce variance that exceeds text generation. A simple query might resolve in 2 tool calls, while a complex one might require dozens. Code execution might take tens of milliseconds for a trivial script, or seconds for complex computation [9]. External APIs vary from hundreds of milliseconds to several seconds depending on the service [8]. Agent benchmarks report trajectories ranging from tens to hundreds of steps [10].

Crucially, this variance is dominated by external latencies that are both unpredictable and GPU-independent. When a worker is waiting for an API response or code execution result, its GPU sits idle—and in synchronous mode, so do all the other GPUs that finished their

trajectories. The synchronous optimizations that work for text generation (Seer's context-aware scheduling, speculative decoding) don't help here: you can't speculate on what an external API will return.

This is where the async architecture shows its clearest advantage. In async mode, a worker waiting for an API call doesn’t block anyone. Other workers continue generating, and training proceeds on available data. The external latency is absorbed by the individual trajectory, not multiplied across the cluster.

The Async Solution

Asynchronous architectures break the dependency between generation and training. Rollout workers generate trajectories continuously and stream completed ones into a buffer. Training workers pull from this buffer whenever enough samples accumulate. No global barrier. No waiting for stragglers.

The key insight is stragglers now affect only themselves. A worker generating a thousand-token response doesn't block all the others, because they've already moved on to new prompts. Training iteration time depends on how fast the buffer fills on average, not how slow the slowest worker is.

This also enables dynamic load balancing without explicit scheduling. Fast workers naturally take on more rollouts. The system adapts to heterogeneous response lengths automatically.

The concern with async is staleness: trajectories generated by an older policy version might degrade learning. But empirical evidence shows this is manageable. AReaL [3] found minimal degradation with 1-4 mini-batch delays; problems only appeared at 16-64 batch staleness. Larger models are more robust to staleness than smaller ones. Modern frameworks (LlamaRL [4], AReaL [3], Laminar [5]) incorporate importance-weighting corrections that explicitly handle off-policy data.

Conclusion

Synchronous RLHF has a straggler problem. The fact that responses vary in length creates catastrophic resource waste. For agentic workloads with extreme trajectory variance, this becomes economically unworkable.

Asynchronous training solves this by decoupling generation from training. Stragglers no longer block the cluster. Major frameworks are beginning to adopt this design.

As LLM post-training moves toward agentic, tool-integrated, variable-length workloads, synchronous training will increasingly become the exception rather than the default. The straggler problem doesn't go away with better hardware or cleverer scheduling—it's structural. Async is the fix.

Reference

1. Gao, Wei, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang et al. "RollPacker: Mitigating Long-Tail Rollouts for Fast, Synchronous RL Post-Training." *arXiv preprint arXiv:2509.21009* (2025).

2. Hu, Jian, Xibin Wu, Zilin Zhu, Weixun Wang, Dehao Zhang, and Yu Cao. "Openrlhf: An easy-to-use, scalable and high-performance rlhf framework." *arXiv preprint arXiv:2405.11143* (2024).
3. Fu, Wei, et al. "AReaL: A Large-Scale Asynchronous Reinforcement Learning System for Language Reasoning." *arXiv preprint arXiv:2505.24298* (2025).
4. Wu, Bo, Sid Wang, Yunhao Tang, Jia Ding, Eryk Helenowski, Liang Tan, Tengyu Xu et al. "Llamarl: A distributed asynchronous reinforcement learning framework for efficient large-scale llm trainin." *arXiv preprint arXiv:2505.24034* (2025).
5. Sheng, Guangming, Yuxuan Tong, Borui Wan, Wang Zhang, Chaobo Jia, Xibin Wu, Yuqi Wu et al. "Laminar: A Scalable Asynchronous RL Post-Training Framework." *arXiv preprint arXiv:2510.12633* (2025).
6. Qin, Ruoyu, 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." *arXiv preprint arXiv:2511.14617* (2025).
7. Systemic Profiling of Time Consumption in verl Multi-Turn Training https://github.com/zhaochenyang20/Awesome-ML-SYS-Tutorial/blob/main/rlhf/verl/multi-turn/tool_examples/profile_en.md
8. Abhyankar, Reyna, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. "Infercept: Efficient intercept support for augmented large language model inference." *arXiv preprint arXiv:2402.01869* (2024).
9. Majgaonkar, Oorja, Zhiwei Fei, Xiang Li, Federica Sarro, and He Ye. "Understanding Code Agent Behaviour: An Empirical Study of Success and Failure Trajectories." *arXiv preprint arXiv:2511.00197* (2025).
10. Yoran, Ori, Samuel Joseph Amouyal, Chaitanya Malaviya, Ben Bogin, Ofir Press, and Jonathan Berant. "Assistantbench: Can web agents solve realistic and time-consuming tasks?." *arXiv preprint arXiv:2407.15711* (2024).