PipelineRL: Limits of Asynchronous Reinforcement Learning for Long-Horizon Trajectories

Shurui Liu
Stanford University
srliu@stanford.edu

Henry Bosch
Stanford University
hbosch@stanford.edu

Extended Abstract

Long-horizon language-model reinforcement learning is bottlenecked by rollout generation. Agentic coding and mathematical reasoning tasks can require thousands of tokens or tool calls before reward is known, so synchronous PPO/GRPO training waits for the slowest rollout before updating. PipelineRL addresses this systems problem by overlapping rollout generation, preprocessing, and learner updates with in-flight weight broadcasts. The price is off-policy staleness: the learner trains on trajectories (partially) generated by older behavior policies. This creates an policy mismatch measured by the importance sampling (IS) ratio

$$\rho_t = \frac{\pi_\theta(a_t \mid s_t)}{\mu(a_t \mid s_t)} = \exp(\log \pi_\theta(a_t \mid s_t) - \log \mu(a_t \mid s_t)).$$

where π is the policy we are updating and μ is the behavior policy. Due to in-flight weight updates, μ may not be generated by fixed set of parameters, but will have its parameters shifted over the course of the rollout.

When lag grows, this ratio can become heavy-tailed and makes training unstable with huge variance. PipelineRL uses a leave-one-out group baseline with clipping, while we use a clipped PPO/GRPO objective for a better stability. The choice of clipping raises a bias-variance trade-off; the advantage is that we avoid overflows with a pre-exponential clamp and prevent excessive policy drift with a post-exponential, one-sided IS clip whose direction depends on the sign of the advantage function, just as in PPO/GRPO.

We implemented PipelineRL (replacing leave-one-out group baseline and post-exp clamp with PPO/GRPO stype clipping) on a 2xA100-80GB system and carried out experiments with Qwen2.5-0.5B-Instruct on GSM8K. We logged diagnostics including the mean and maximum ρ, tail mass, forward/reverse KL approximations, token counts, and a group-level self-normalized importance-sampling proxy for GRPO baseline bias to understand the effect of the off-policy staleness on bias and variance of the training process. The training loss uses a pre-exp log-ratio clamp $\tilde{r}_t = \mathrm{clip}(r_t, -C, C)$ with $C = 5$, while diagnostics still use raw r_t with only a metric-side cap.

Our main accomplishment can be summarized as follows. First, PipelineRL's original post-exp clamp can overflow before the clamp is applied, producing NaNs and persistent rollout failures. We fix this by a GRPO style clipping pre-exp. Second, we found in this horizon and scope, our method of asynchronous RL speeds up the training (raw wall-clock time) while achieving comparable rewards and validation loss. Third, we verified that larger lags will increase the variance of importance sampling ratio. Finally, we conducted a much longer run (ten times as many steps) with the maximum lag, in order to see the limits of our method. While at our scale (fine-tuning Qwen2-0.5B-Instruct) everything was stable, we believe that larger models will introduce greater potential lags due to higher inference throughput requirements for rollouts as well as longer rollouts themselves, so high induced variance of the importance sampling ratio will cause much greater instability.

1 Abstract

Long-Horizon language-model reinforcement learning is bottlenecked by rollout generation. PipelineRL addresses this by overlapping rollout generation, preprocessing, and learner updates with in-flight weight broadcasts. As a result, there is *lag*, i.e. a difference in optimizer step between the behavior policy and the policy we are updating, which means we are performing off-policy RL. This is corrected by importance sampling, but large IS ratios can result in training instability. We investigate this phenomenon, introducing several techniques which mitigate training instability, and study the relationship between the amount of tolerated lag and off-policy metrics like forward KL and IS ratio statistics. Contrary to our expectations, we found that with our adjustments, allowing the maximum possible lag resulted in the fastest learning in terms of wall-clock time.

2 Introduction

In postraining of the agents with reinforcement learning, the rollout generation is long, variable length, and expensive. Coding agents may edit, run tests, inspect errors, and revise repeatedly before a reward is available. Web agents and reasoning agents have similar structure: the expensive part is not only the backward pass, but also generating many long attempts and evaluating them. Synchronous on-policy RL methods such as PPO (Schulman et al., 2017) and GRPO-style training wait for a complete batch of fresh rollouts before updating. Therefore, the trainer GPU kernel has to idle during the wait, which lowers the GPU utilization and slows down the overall training. This is illustrated in the left of Figure 1.

PipelineRL (Piché et al., 2025) is a natural system design to improve the training efficiency by overlapping part of rollout generation and trainer update step. More concretely, instead of serializing rollout, preprocessing, and training, it overlaps them: actors generate under their most recent weights, completed samples flow through queues, the learner trains as data arrives, and new weights are broadcast back to inference workers, as illustrated in the right of Figure 1. This is also the direction highlighted by Cursor Composer 2 for agentic coding RL (Chan et al., 2026). However, asynchrony changes the estimator. A token may be generated under a behavior policy μ and later trained under a learner policy π_θ that has already taken several optimizer steps. The relevant mismatch is

$$r_t = \log \pi_\theta(a_t \mid s_t) - \log \mu(a_t \mid s_t), \qquad \rho_t = \exp(r_t). \tag{1}$$

Importance sampling reweighting eliminates off-policy bias, but high-variance ratios make the training unstable. Clipping and truncation control this variance but introduce bias. In LLM RL systems, there is an additional numerical issue: if $\exp(r_t)$ is evaluated in mixed precision before clipping, sufficiently stale or otherwise mismatched samples can create infinite values, NaN gradients, and corrupted weights. This is observed in our experiments with original PipelineRL codebase (see Figure 3).

By replacing post-exp clamping with pre-exp PPO/GRPO-style clamping, we fixed the instablity failure and built a reproducible setup for running PipelineRL on a single 2xA100-80GB container with Redis streams, vLLM rollout actors, and the learner colocated. We observed that this speeds up the training significantly (See Figure 4) without downgrading the validation loss or validation rewards. Moreover, we investigate in more detail the following question:

> How does policy-version lag in PipelineRL affect importance-ratio tails, KL drift, reward stability, gradient health, and the GRPO group baseline?

This is important for potential scaling to industrial level long horizon tasks and large models. This also help us understand how to properly design a GRPO group baseline in this asynchronous RL setting. We carried out experiments and observed that the stale lag increases maximum-ratio tails and KL drift, while short-run reward and the online group-baseline proxy remain comparatively flat, which means on this small model and the GSM8k task, the larger lags will not influence group-baseline but still potentially increase the variance in long run. The results are summarized in Figure 6.

3 Related Work

Long-horizon and agentic RL. Cursor Composer 2 (Chan et al., 2026) is the main motivating paper for this project: it emphasizes large-scale RL for coding agents and highlights asynchronous

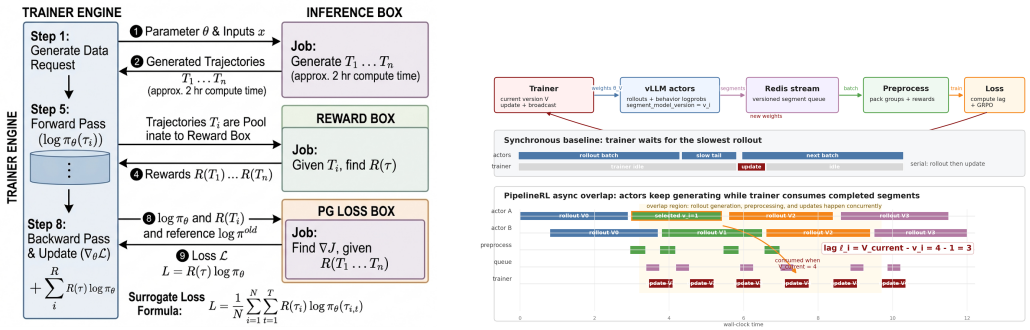

Figure 1: Left: Asynchronous long-horizon RL has multiple engines for the different tasks (Trainer, Inference, Reward and Policy Gradient (PG) Loss). Right: PipelineRL overlaps rollout generation, preprocessing, and learner updates, which improves utilization but creates stale-policy samples.

RL as a way to avoid idle learner hardware during long agent rollouts. Kwa et al. (2026) provide complementary motivation by measuring model ability on increasingly long software tasks, showing why training systems must eventually handle very long, uneven episodes.

PipelineRL and asynchronous reinforcement leraning. PipelineRL (Piché et al., 2025) brings pipelined execution to LLM RL by overlapping generation and learner updates while broadcasting new weights to actors. Older deep RL systems such as IMPALA (Espeholt et al., 2018) also study distributed actors whose samples are stale by the time they reach the learner; V-trace is an importance-weighted correction for that setting. LLM sequence RL differs because trajectories are long token sequences, rewards are often sparse and group-normalized, and numerical precision issues can interact with off-policy ratios.

LLM RL objectives and stability. PPO (Schulman et al., 2017) clips policy ratios to stabilize policy-gradient updates. GRPO, popularized in mathematical-reasoning RL systems such as DeepSeekMath (Shao et al., 2024), replaces a value model with group-relative baselines. DAPO (Yu et al., 2026) identifies instability from sampling/training mismatch and entropy collapse in GRPO-style training, while UloRL (Du et al., 2025) studies ultra-long reasoning outputs and segmentation. These works motivate our focus on importance sampling ratio control and group-relative baselines in PipelineRL.

Distillation. Our milestone considered combining PipelineRL with co-evolving policy distillation (Gu et al., 2026) and classical knowledge distillation (Hinton et al., 2015). The final project did not implement this multi-branch algorithm since the stablity is the principal bottleneck and this may be a future work once we have compute for larger models and longer runs.

4 Method

4.1 PipelineRL Setting

PipelineRL separates the RL system into vLLM actors, preprocessing workers, Redis streams, and a finetuning learner. Actors generate rollouts with the latest weights they have received, then push token ids, old log-probabilities, rewards, group ids, and a model version into the queue. The learner trains on batches that may contain segments generated under several previous model versions. We define the policy *version* to be the optimizer step (so version 0 is the base model, version 1 is after one step, etc). If the learner's latest broadcasted version is v_{cur} and a segment was generated under version v_i, we define segment lag as

$$\ell_i = v_{\text{cur}} - v_i, \tag{2}$$

as illustrated in Figure 2. Namely, lag is measured in learner weight-broadcast versions rather than wall-clock seconds.

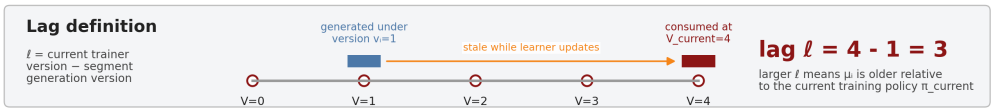

Figure 2: Lag definition: a packed training segment is labeled by the difference between the learner's current weight version and the actor weight version that generated the segment.

For each prompt group g with K sampled rollouts and scalar rewards R_i, GRPO-style preprocessing constructs a leave-one-out advantage

$$A_i = \frac{R_i - \frac{1}{K-1}\sum_{j \in g, j \neq i} R_j}{\mathrm{std}(\{R_j : j \in g\}) + 10^{-4}}, \tag{3}$$

when standard-deviation normalization is enabled. The implementation also uses group normalization so that long groups do not dominate the loss solely because they contain more tokens.

4.2 Loss and Importance Ratios

In the loss engine, the code computes the raw log-ratio

$$r_t = \log \pi_\theta(a_t \mid s_t) - \log \mu(a_t \mid s_t), \tag{4}$$

where μ is the behavior policy that generated the token. For stable training we use a pre-exp clamp

$$\tilde{r}_t = \mathrm{clip}(r_t, -C, C), \qquad \rho_t^{\mathrm{loss}} = \exp(\tilde{r}_t), \tag{5}$$

with $C = 1$ in the implementation. The PPO-style GRPO loss is

$$\mathcal{L}_{\mathrm{PPO}}(\theta) = -\sum_t m_t w_t \min\left(\rho_t^{\mathrm{loss}} A_t,\ \mathrm{clip}(\rho_t^{\mathrm{loss}}, 1-\epsilon_{\mathrm{low}}, 1+\epsilon_{\mathrm{high}}) A_t\right), \tag{6}$$

where m_t is the token mask and w_t is the group-normalization weight. Our runs use $\epsilon_{\mathrm{low}} = \epsilon_{\mathrm{high}} = 0.1$ in Figure 4.

The distinction between ρ_t and ρ_t^{loss} is important. The training objective must be bounded to avoid numerical failure, but the scientific question is how staleness changes the raw mismatch. Therefore the lag diagnostics use the unclamped raw r_t from Equation 4.

4.3 Baseline-Bias Diagnostic

The original hypothesis was that stale rollouts might bias the GRPO group baseline. For a group g, the plain baseline is

$$b_{\mathrm{plain}}(g) = \frac{1}{|g|}\sum_{i \in g} R_i. \tag{7}$$

A self-normalized importance-sampling baseline would instead be

$$b_{\mathrm{SNIS}}(g) = \frac{\sum_{i \in g} \bar{\rho}_i R_i}{\sum_{i \in g} \bar{\rho}_i}, \qquad \bar{\rho}_i = \exp\left(\frac{1}{T_i}\sum_{t \in i} r_t\right). \tag{8}$$

We log $\Delta b_g = |b_{\mathrm{plain}}(g) - b_{\mathrm{SNIS}}(g)|$ by the group's mean lag. This is only a proxy because it uses the segments in the current training batch and a per-segment geometric-mean ratio.

5 Implementation

5.1 Training Pipeline

We also implemented training both locally and remotely on modal. We were able to get four simultaneous 2xA100-80GB pairs to perform four concurrent, non-interacting training runs on an 8xA100-80GB node. The training run with index $i \in \{0, 1, 2, 3\}$ is performed on GPU ranks $\{2i, 2i+1\}$.

The system is a dual-socket, 8-GPU PCIe server with two NUMA-local GPU groups: GPUs 0–3 are attached to NUMA node/socket 0 with CPU cores 0–47, while GPUs 4–7 are attached to NUMA node/socket 1 with CPU cores 48–95. Within each 4-GPU group, communication traverses the local PCIe host bridge (PHB), while communication across the two groups crosses the inter-socket system fabric (SYS). Pairing GPUs within each NUMA node enhances efficiency, so that is what we did.

For each GPU pair, PipelineRL's world placement assigns a vLLM actor to one GPU and the finetune learner to another; a single GPU leaves no learner slot without rewriting the placement code.

Our experiments use Qwen2.5-0.5B-Instruct and GSM8K train/test. Configuration details are summarized in Table 1.

5.2 Lag Metadata

PipelineRL packs multiple rollout segments into one sequence for efficient training. Before our changes, the batch-level model version was a single collapsed value. We added two per-segment tensors during collation to log the model version that generated each packed segment, including sentinel padding segments and a dense local group id so that rollout segments from the same prompt group can be aggregated for the baseline-bias proxy. At the loss site, the learner computes $\ell_i = v_{\text{cur}} - v_i$ for every segment. The lag buckets are fixed to

$$[0,1),\ [1,3),\ [3,8),\ [8,20),\ [20,50),\ [50,200),\ [200,\infty).$$

For every valid token, the function maps its segment lag to a bucket and accumulates:

- token count, mean ρ, variance proxy through ρ^2, and max ρ;
- mean log-ratio and mean absolute log-ratio;
- Schulman-style forward-KL approximation $\rho - r - 1$ and reverse estimate $-r$;
- tail masses $\Pr(\rho > 2)$, $\Pr(\rho > 5)$, and $\Pr(\rho > 10)$.

Derived metrics such as means, fractions, and RMSE values are produced in the finetune loop after per-microbatch aggregation.

5.3 Stability Controls

The milestone experiments showed that PipelineRL's post-exp clamp was insufficient: computing $\exp(r_t)$ before clipping could overflow in mixed precision and corrupt the model weights sent to vLLM. We added non-finite guards and two another controls:

1. **Pre-exp log-ratio clamp.** The loss uses pre-exp clamping so $\rho_t^{\text{loss}} \in [e^{-5}, e^{5}]$ before PPO clipping.
2. **Gradient hard skip.** We use gradient clipping at 0.01 and skip optimizer steps whose pre-step gradient norm exceeds 10^6.

These controls are engineering needed to collect the measurements and avoid the run collapsing.

6 Experiments

6.1 Research Questions

The final experiments address three questions:

1. Does asynchronous RL increase the speed of training without harming the performance metrics?
2. As observed lag increases, do raw importance-ratio tails and KL estimates grow?
3. Does the online SNIS proxy show evidence that the GRPO group baseline becomes more biased as lag grows?

Our configurations and hyperparameters are summarized in Table 1.

Component	Setting
Hardware	Linux Node, 2xA100-80GB, Redis streams
Model	Qwen2.5-0.5B-Instruct
Task	GSM8K train/test, 4 attempts per prompt
Sequence	max sequence length 4096, max generation length 2048, packed segments
Training	PPO-style GRPO, batch size 2, grad accum 8, 500 learner steps
Optimizer	learning rate 10^{-7}, warmup 20, weight decay 0
PPO clip	$\epsilon_{\text{low}} = \epsilon_{\text{high}} = 0.1$
Ratio control	pre-exp log-ratio clamp $C = 5$, diagnostics use raw r_t
Lag sweep labels	`max_lag` $\in \{1, 8, 64, 256\}$
Stability	BF16 DeepSpeed compute with FP32-loaded params, grad clip 0.01

Table 1: Experimental configuration.

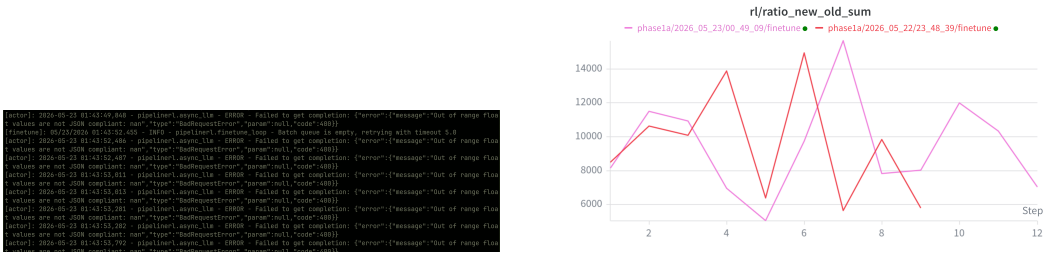

Figure 3: Original post-exp clamp failure. Left: vLLM failures after non-finite weights are broadcast. Right: large new/old ratio variance in early smoke runs.

7 Results

7.1 Post-Exp Clamping Can Fail Before the Clamp

The original PipelineRL uses mixed precision BF16 and clamp before taking exponential, i.e. the code computed $r_t = \log \pi_\theta - \log \mu$, exponentiated it, then clamped the resulting ratio. However, this bounds the policy weight only after $\exp(r_t)$ has already been evaluated. In our smoke runs, large stale-policy log-ratios produced out-of-range floating-point values, which led to NaNs and vLLM rollout failures after corrupted weights were broadcast. See Figure 3.

Therefore, we switched to pre-exp clamping and implemented gradient norm restrictions.

7.2 Larger Lag Windows Do Not Immediately Hurt Reward in the Short Run

Figure 4 summarizes the reward and test metrics across sweep labels. In these 500-step runs, larger lag windows do not cause immediate reward collapse. Final-window test reward remains close across all four runs, and the `max_lag` $= 64$ run has the highest tail train reward in this small sweep. In wall-clock time, the `max_lag` $= 256$ run reaches comparable reward way much faster because the learner can consume stale samples instead of waiting for fresher rollout batches.

However, due to limitation of compute, the runs are short and the model is small, and these observations will not apply to larger models or industrial-level long horizon tasks. The takeaway is that small-scale reward is not affected by staleness of asynchronous reinforcement learning and asynchronous reiforcement learning can accelerate the training significantly.

7.3 Trace Staleness in Ratio Tails and KL

The clearest statistical effect is in maximum-ratio tails. Table 3 summarizes importance resampling ratio ρ across different lags. Median mean ρ stays almost exactly one in every nonempty bucket, but maximum-ratio statistics increase with lag. The median per-step max ratio rises from 1.28 at lag 0 to 1.38 at lag 50–199, a 7.4% increase. The p95 max ratio at lag 50–199 is 1.72 and the maximum observed value in that bucket is 7.16. Forward KL also increases: from 3.15×10^{-4} at lag 0 to 3.56×10^{-4} at lag 8–19, a 13% increase. Reverse KL at lag 50–199 is 9.4% above lag 0.

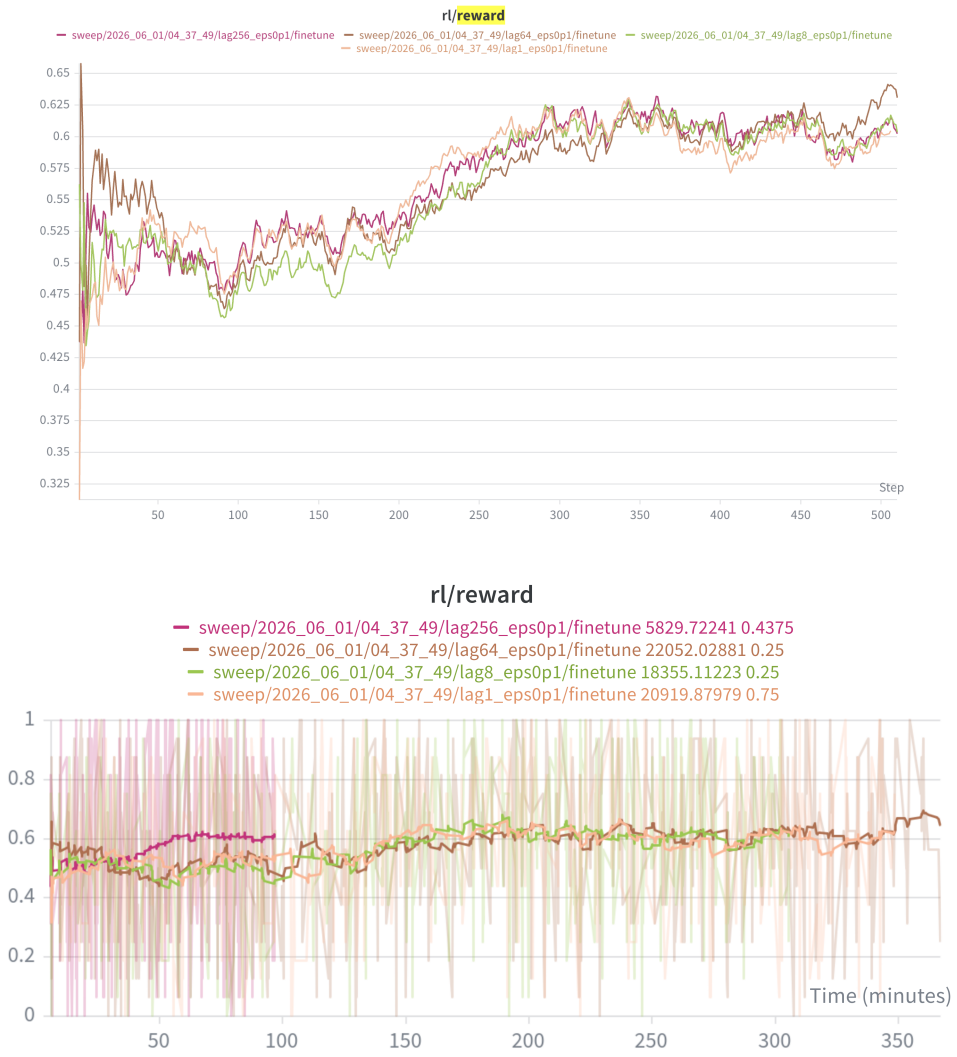

Figure 4: Reward and speed across lag-window sweep labels. Top: final-window test reward remains similar across lag caps in the small clamped GSM8K setup. Bottom: reward plotted against wall-clock time shows the `max_lag` $= 256$ run completing the same training horizon substantially faster than lower-lag runs.

Figure 6 shows the same pattern visually. The effect is not a large mean shift in ρ; it is tail risk and KL drift. This matches the intuition behind clipped policy-gradient methods: most tokens may be close to on-policy while a small number of stale or surprising tokens dominate the variance and numerical risk on a small scope.

7.4 The Online Baseline-Bias Proxy Is Flat

The baseline proxy RMSE is 1.47–1.56×10^{-4} across the nonempty lag buckets, with no monotone increase. Gradient norm also has only a weak Spearman correlation with lag, 0.074, and final-window test success remains between 0.475 and 0.486 across the four sweep labels.

It indicates that on a small scope, the staleness does not introduce high bias on the group-baseline.

max_lag	Early train reward	Tail train reward	Tail test reward
1	0.489	0.575	0.475
8	0.478	0.571	0.478
64	0.520	0.627	0.486
256	0.528	0.586	0.482

Table 2: Reward summary. Values are means from the generated CSV summaries.

Observed lag bucket	Median max ρ	p95 max ρ	Median mean ρ	Baseline RMSE
0	1.284	1.579	0.999973	1.49×10^{-4}
8–19	1.295	1.622	1.000012	1.47×10^{-4}
20–49	1.325	1.637	1.000060	1.56×10^{-4}
50–199	1.379	1.719	1.000015	1.50×10^{-4}

Table 3: Lag-bucket summary. Mean ratios stay close to one, but tail statistics increase with observed lag. Baseline RMSE is the online SNIS proxy and is flat.

7.5 How Close are We to Optimal?

We swept over the PPO/GRPO epsilon with max lag 256, $\epsilon \in \{0.05, 0.1, 0.2, 0.3\}$ at 4000 steps. We found that reward curves continue to go up, and are not showing signs of stopping improvement. Each of these runs took around 24 hours on a pair of A100-80GB GPUs. We can see that the value of ϵ does not significantly affect the outcome. The conclusion from this is that the wall-clock speed improvement from asynchrony continues to work stably when doing a longer training run.

8 Discussion

In this project, we were hoping to find that there would be a limit of asynchrony, over which training stability would break down. While we may expect that to be the case at large scales, we observed in our training setup that decreasing the lag tolerance did not improve training stability at all. Instead, we concluded that maximizing the lag was a free speedup, which reduced wall-clock training time by around 4x. We did observe meaningful differences in forward KL and Importance Sampling metrics as lag increased. Becoming familiar with the relationships between these quantities is instructive for finding the true limits of parallel asynchronous reinforcement learning.

9 Limitations and Future Work

Our experiments use Qwen2.5-0.5B-Instruct and 500 learner steps on GSM8K. These runs are sufficient to expose numerical failure modes and ratio-tail trends, but they do not establish long-run convergence behavior. Larger models, longer horizons, and agentic coding tasks may show stronger stale-rollout effects and a naive GRPO style clipping may not even work anymore. We will do experiments on larger models and harder tasks in the future when compute is sufficient and try out different clipping method or algorithm design (e.g. mutual distillation, as in Gu et al. (2026)) to matigate the influence of the off-policy staleness.

10 Conclusion

Asynchronous PipelineRL is the right systems direction for long-horizon sequence RL, but at larger scale it creates bias and numerical-instability problems through stale rollout policies. However, in our experiments on small scope, stale lag under a stable clamped setup first appears as increased maximum-ratio tails and KL drift, not as immediate reward collapse or performance downgrading. Moreover, we observe that asynchronous RL can significantly speed up the training.

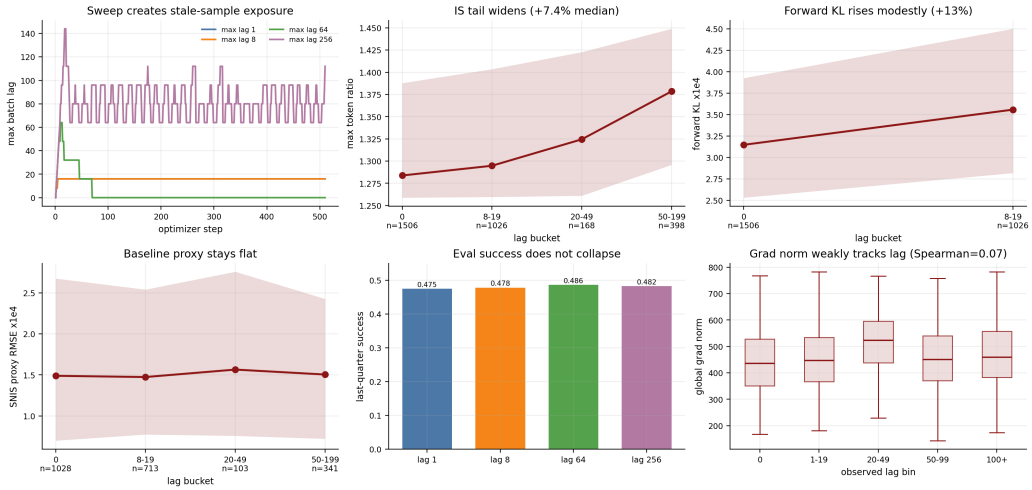

Figure 5: Staleness changes policy-mismatch diagnostics such as max ratio and KL, but reward, grad norm, and the online baseline proxy do not move strongly in this short clamped run.

11 Team Contributions and Research-Process Reflection

11.1 Contributions

- **Shurui Liu**: made the algorithm design (pre-exp GRPO-style clamping) to resolve overflow issue, led the implementation of the codebase based on PipelineRL codebase, did initial experiments on Modal, and did result analysis of experiments and wrote up poster and final report.
- **Henry Bosch**: proposed project direction. Allocated and setup stable compute (8xA100-80B node). Carried out experiments on sweeping hyperparameters: lags (1,8,64,256) at 500 steps and epsilons (0.05,0.1,0.2,0.3) at 4000 steps. Additional analysis of experiments, editing poster and writing up final report.

11.2 Reflection

For comparision, here was our original task breakdown.

> **Henry Bosch**: Henry will setup git repository and github, with github actions. Secure sufficient modal compute from course offering.
> Implement one baseline and evaluation dataset. For owned algorithms, perform training and hyperparameter tuning, iteratively improving.
> In final report, write system technical discussion (e.g. describe the algorithm mathematically, how to take GPU utilization into consideration, and hyperparemeter tuning heuristics).
> **Shurui Liu**: Shurui will implement raw parallel and come up with algorithm design. Shurui will implement training experiment pipleline. In final report, do theoretical analysis (how to use information theory etc. to justify variance control and efficiency/accuracy improvement). Shurui will aslo investigate and do small experiments on system efficiency and compatibility with owned algorithms.

In reflection, the team contributions were not too far off from what we predicted in the proposal. In our initial proposal, we tried to overlap the tasks more. In practice, what turned out to work better was one partner working for a few days on their part, followed by the other partner working on their part, according to people's schedules. This led to more task specialization than we expected.

References

Aaron Chan, Ahmed Shalaby, Alexander Wettig, Aman Sanger, Andrew Zhai, Anurag Ajay, Ashvin Nair, Charlie Snell, Chen Lu, Chen Shen, Emily Jia, Federico Cassano, Hanpeng Liu, Haoyu Chen,

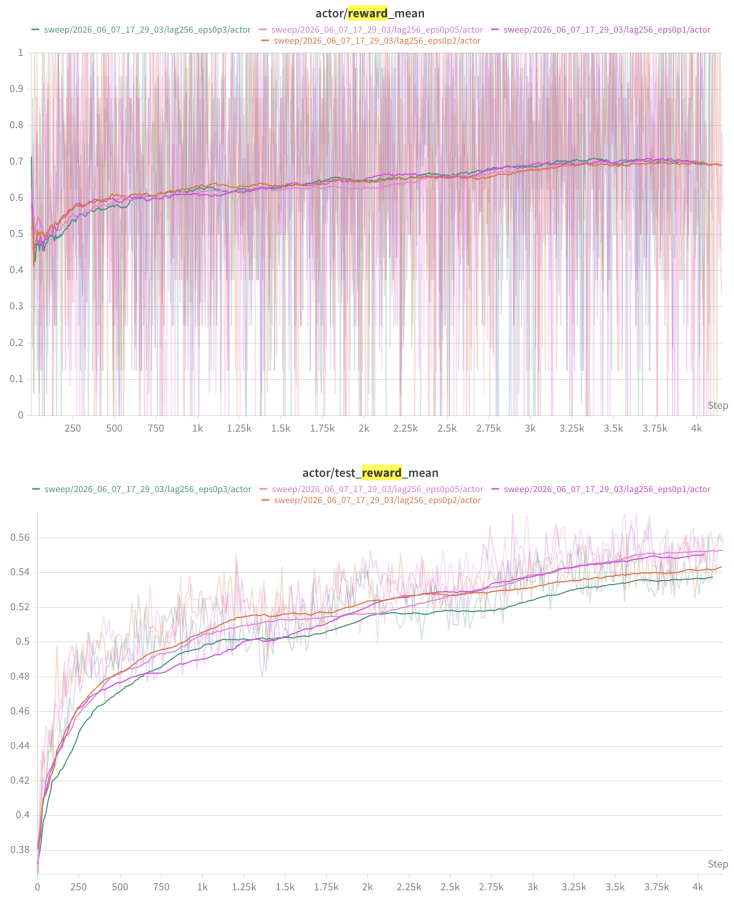

Figure 6: Training/Test reward with 4000 steps.

Henry Wildermuth, Jacob Jackson, Janet Li, Jediah Katz, Jiajun Yao, Joey Hejna, Josh Warner, Julius Vering, Kevin Frans, Lee Danilek, Less Wright, Lujing Cen, Luke Melas-Kyriazi, Michael Truell, Michiel de Jong, Naman Jain, Nate Schmidt, Nathan Wang, Niklas Muennighoff, Oleg Rybkin, Paul Loh, Phillip Kravtsov, Rishabh Yadav, Sahil Shah, Sam Kottler, Alexander M Rush, Shengtong Zhang, Shomil Jain, Sriram Sankar, Stefan Heule, Stuart H. Sul, Sualeh Asif, Victor Rong, Wanqi Zhu, William Lin, Yuchen Wu, Yuri Volkov, Yury Zemlyanskiy, Zack Holbrook, and Zhiyuan Zhang. 2026. Composer 2 Technical Report. arXiv:2603.24477 [cs.SE] https://arxiv.org/abs/2603.24477

Dong Du, Shulin Liu, Tao Yang, Shaohua Chen, and Yang Li. 2025. UloRL:An Ultra-Long Output Reinforcement Learning Approach for Advancing Large Language Models' Reasoning Abilities. arXiv:2507.19766 [cs.CL] https://arxiv.org/abs/2507.19766

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. 2018. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. In *Proceedings of the 35th International Conference on Machine Learning*. https://proceedings.mlr.press/v80/espeholt18a.html

Naibin Gu, Chenxu Yang, Qingyi Si, Chuanyu Qin, Dingyu Yao, Peng Fu, Zheng Lin, Weiping Wang, Nan Duan, and Jiaqi Wang. 2026. Co-Evolving Policy Distillation. arXiv:2604.27083 [cs.LG] https://arxiv.org/abs/2604.27083

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. arXiv:1503.02531 [stat.ML] https://arxiv.org/abs/1503.02531

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, Ryan Bloom, Thomas Broadley, Haoxing Du, Brian Goodrich, Nikola Jurkovic, Luke Harold Miles, Seraphina Nix, Tao Lin, Neev Parikh, David Rein, Lucas Jun Koba Sato, Hjalmar Wijk, Daniel M. Ziegler, Elizabeth Barnes, and Lawrence Chan. 2026. Measuring AI Ability to Complete Long Software Tasks. arXiv:2503.14499 [cs.AI] https://arxiv.org/abs/2503.14499

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

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG] https://arxiv.org/abs/1707.06347

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. arXiv:2402.03300 [cs.CL] https://arxiv.org/abs/2402.03300

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, YuYue, 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. 2026. DAPO: An Open-Source LLM Reinforcement Learning System at Scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=2a36EMSSTp