WHERE DOES STALENESS ACCUMULATE? POOL AWARE EFFECTIVE STALENESS CONTROL FOR ASYNCHRONOUS RL IN LLM POST TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Fully asynchronous reinforcement learning (RL) improves resource utilization in large language model post-training by overlapping rollout generation with policy optimization, but it also introduces policy lag as trajectories are generated and queued while the trainer continues to update. We study how this lag accumulates over a trajectory's lifetime and how it can be controlled without sacrificing the wall-clock benefits of asynchronous execution. We decompose trajectory staleness into Generation Staleness, accumulated before rollout completion, and Waiting Staleness, accumulated after a completed trajectory enters the pool. Motivated by this decomposition, we introduce PACE (Pool-Aware Control of Effective Staleness). PACE converts excess pool occupancy into an adaptive rejection budget and ranks completed trajectories using an effective-staleness score that combines Waiting Staleness with prefix-aware Generation Staleness. This avoids penalizing long or interrupted rollouts solely because they span multiple policy versions. In single-turn mathematical reasoning, PACE improves the six-benchmark average validation accuracy by 18.7% over unfiltered asynchronous RL at the same wall-clock budget and matches synchronous RL performance with 47.1% less GPU time. PACE also improves validation performance in multi-turn tool-integrated reasoning, outperforming both synchronous and unfiltered asynchronous RL. Further experiments with the mixture-of-experts model and an alternative RL algorithm support its applicability across model architectures and training algorithms.

1 INTRODUCTION

Fully asynchronous (Fu et al., 2025) reinforcement learning (RL) for large language models offers a direct system benefit: rollout workers can continue generating trajectories while the trainer updates the policy. This is particularly useful for long-horizon reasoning (Gao et al., 2025; Feng et al., 2026b) and tool-using tasks (He et al., 2026; Zhou et al., 2024; Feng et al., 2026a), where autoregressive generation is expensive and completion times vary widely across prompts and tasks (Zhou et al., 2025). In a synchronous pipeline, optimization waits for the slowest trajectories in each batch, leaving resources idle at different stages of the iteration. Fully asynchronous systems remove this batch barrier and let generation and training proceed at their own pace.

The cost of this decoupling is that training data can become stale as the policy continues to evolve. Existing work addresses the resulting mismatch between behavior and training policies through off-policy correction (Degris et al., 2012; Hou et al., 2026; Li et al., 2025b). Recent methods further improve the use of stale trajectories by controlling importance-weighted updates. M2PO constrains the second moment of importance weights to suppress extreme updates while preserving useful training signals (Zheng et al., 2025b). VCPO combines learning-rate adaptation based on effective sample size with a variance-reducing baseline to stabilize asynchronous training (Huang et al., 2026). These studies show that stale trajectories can remain useful for learning, provided that the resulting off-policy updates are properly controlled. The benefit of overlapping rollout generation and training therefore depends on both system utilization and learning stability.

Complementary to these algorithmic approaches, asynchronous RL systems constrain policy lag to keep training data fresh. AReaL (Fu et al., 2025) limits how far rollout generation can run ahead of

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.

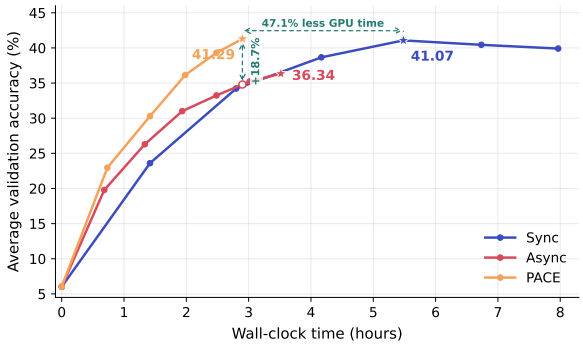


Figure 1: Average validation accuracy across six benchmarks as a function of cumulative logged training-step time. Markers correspond to checkpoints evaluated every 50 training steps, and stars indicate the best average accuracy achieved within the first 300 steps. PACE reaches its peak in 2h54m, earlier than synchronous and asynchronous RL.

training and prioritizes older trajectories when forming training batches. DORA (Hu et al., 2026) maintains a bounded window of policy versions and advances it only after all trajectories from the oldest version have been collected and forwarded to the trainer. These constraints introduce a tradeoff: a tighter lag limit keeps data closer in version to the training policy but can force rollout generation or training to wait, whereas a looser limit allows greater overlap at the cost of staler training data. This tradeoff motivates a complementary perspective: controlling the flow of completed trajectories through the training pool. Selectively filtering surplus completed trajectories can reduce avoidable waiting while preserving asynchronous execution. The challenge is to do so without starving the trainer or discarding useful training signals. This leads to our central question:

Under a fixed allocation of rollout and trainer resources, how can we filter completed trajectories to maintain stable policy improvement while maximizing system utilization?

Answering this question requires deciding *how many* trajectories to reject and *which ones*. We first decompose policy staleness into *Generation Staleness*, accumulated before rollout completion, and *Waiting Staleness*, accumulated in the pool afterward. In our unfiltered runs, persistent pool backlogs accompany elevated Waiting Staleness. We therefore use pool occupancy as feedback for the rejection budget, removing surplus when the pool is overfilled and relaxing rejection when ready data becomes scarce.

With the rejection budget determined by pool occupancy, we next consider which trajectories to remove. Ranking by raw staleness may disproportionately reject harder problems, since longer rollouts are more likely to accumulate generation lag (Fu et al., 2025). This can bias training toward shorter, easier examples. We introduce the Prefix Forgetting Score, which measures changes in prefix token log-probabilities, to refine the staleness signal.

We combine these components in PACE (Pool-Aware Control of Effective Staleness), as illustrated in Figure 2. Pool occupancy sets an adaptive rejection budget, while an effective-staleness score ranks trajectories under that budget. The score retains Waiting Staleness in full and weights Generation Staleness by relative prefix drift. All filtering takes place at the completed-trajectory pool, where the trainer continues drawing trajectories until it has formed a full training batch. We evaluate PACE against synchronous and asynchronous RL on six mathematical reasoning benchmarks. As shown in Figure 1, PACE reaches its peak average validation accuracy earlier than both baselines, at 2h54m of logged training-step time.

Our contributions are:

1. We characterize how staleness accumulates in fully asynchronous RL. Our Stale-k experiments show that larger admitted policy lag slows early optimization, and we decompose trajectory staleness into Generation and Waiting Staleness.
2. We identify persistent pool backlogs as a source of avoidable Waiting Staleness and establish pool occupancy as a natural signal for adaptive trajectory rejection.

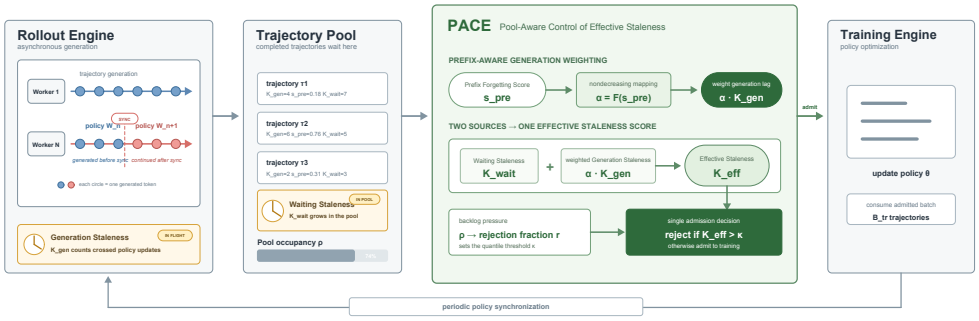


Figure 2: Overview of PACE. The gray-shaded modules depict the standard asynchronous RL pipeline, while the green-shaded modules highlight our proposed components. PACE uses an effective-staleness score and pool occupancy to adaptively reject trajectories before training.

3. We introduce PACE, which combines a pool-aware rejection budget with prefix-aware trajectory ranking. Across six mathematical reasoning benchmarks, it improves learning over unfiltered asynchronous RL while retaining the time advantage of asynchronous execution.

2 Where Does Staleness Accumulate?

As discussed above, asynchronous RL improves resource utilization but allows trajectories to become stale as training proceeds. Higher throughput therefore does not necessarily translate into faster policy improvement. In this section, we first define trajectory staleness and use controlled Stale-k experiments to examine how increasing policy lag affects early optimization. We then decompose this lag into *Generation Staleness*, accumulated before rollout completion, and *Waiting Staleness*, accumulated in the pool afterward. This decomposition forms the basis of our method.

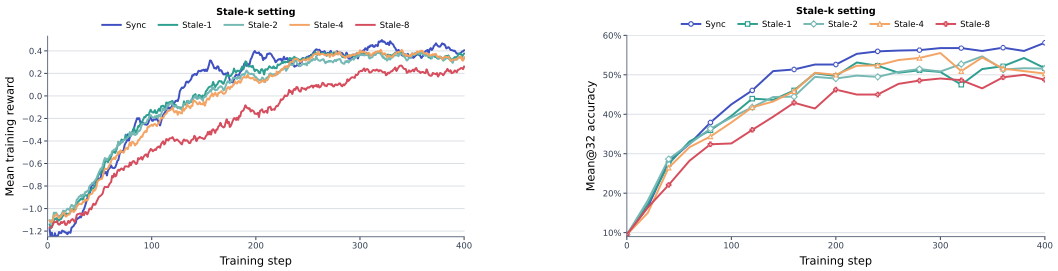
Trajectory Lifecycle and Policy Lag. Let $\tau = (x, y)$ denote a complete trajectory, where $x \sim \mathcal{D}$ is a prompt and $y = (y_1, \ldots, y_T)$ is a response of length $T = |y|$. An autoregressive language model parameterized by θ defines a policy π_θ, with $\pi_\theta(y \mid x) = \prod_{t=1}^{T} \pi_\theta(y_t \mid x, y_{<t})$. A reward model or verifier scores (x, y), and the resulting rewards are used to estimate token-level advantages $\hat{A}_t$.

In asynchronous RL, the trainer maintains a sequence of policy versions $\{\theta_j\}_{j \geq 0}$, whose weights are periodically synchronized to rollout workers. Let $\mu_v := \pi_{\theta_v}$ denote the behavior policy corresponding to trainer version v. Generation proceeds concurrently with training and may span multiple policy versions (Team et al., 2025; Fu et al., 2025; Zhou et al., 2025). For each token y_t, the rollout worker records the policy version index v_t used for generation and the corresponding log-probability $\ell_t^\mu = \log \mu_{v_t}(y_t \mid x, y_{<t})$. Let $c(\tau)$ denote the trainer version when trajectory τ completes generation. The completed trajectory is subsequently buffered until it is consumed at training step j. This step uses the current trainer policy π_{θ_j} and produces θ_{j+1}. When trajectory τ is used for training at step j, its token-level policy lag is $k_t(\tau; j) = j - v_t$. Unless otherwise stated, we define the policy lag of a trajectory as the mean over its response tokens:

$$\bar{k}(\tau; j) = \frac{1}{T} \sum_{t=1}^{T} (j - v_t). \tag{1}$$

This definition accommodates trajectories generated across multiple policy versions and reduces to $j - v$ when all response tokens are generated under a single version v.

Controlled Stale-k Analysis. To isolate the effect of policy staleness in asynchronous RL, we follow the controlled Stale-k setting used in prior work (Zheng et al., 2025b; Fu et al., 2025), where only trajectories with a policy lag of at most k are admitted for training. Synchronous RL serves as the on-policy baseline. For the asynchronous runs, we vary k while keeping the remaining training configuration unchanged. As shown in Figure 3, larger values of k slow early optimization, with the most pronounced delay at $k = 8$. The AIME 2024 validation curves show a similar trend. These results suggest that limiting policy lag can accelerate policy improvement.



(a) DAPO-MATH-17K training reward. (b) AIME 2024 mean@32 accuracy.

Figure 3: Effect of the maximum admitted staleness k on Qwen3-8B during the first 400 training steps. Sync is the on-policy baseline. Stale-k admits trajectories with policy lag at most k; for example, Stale-1 admits trajectories at most one policy update behind the trainer.

Staleness Decomposition. To understand where this lag arises, we examine two stages of a trajectory's lifetime. During generation, a rollout may span several trainer updates or policy-weight synchronization boundaries, producing *Generation Staleness*. After completion, it may continue to age while waiting for training, producing *Waiting Staleness*. Let $c(\tau)$ denote the trainer version when trajectory τ completes generation, and let j denote the version at which it is consumed. Waiting includes any intermediate buffering between completion and consumption. The following lemma decomposes the policy lag defined in Equation 1 without presuming that either component dominates.

Lemma 1 (Staleness decomposition)**.** *Consider a completed trajectory $\tau = (x, y)$ with T response tokens, where token t was sampled under behavior-policy version v_t. Suppose that $v_t \le c(\tau) \le j$ for all t. Then its staleness at training step j decomposes as*

$$\bar{k}(\tau; j) = \underbrace{\frac{1}{T}\sum_{t=1}^{T}(c(\tau) - v_t)}_{K_{\mathrm{gen}}(\tau)} + \underbrace{j - c(\tau)}_{K_{\mathrm{wait}}(\tau; j)}. \tag{2}$$

The proof is provided in Appendix B.1.

Generation Staleness. $K_{\mathrm{gen}}(\tau)$ measures the policy-version lag accumulated before rollout completion. It can become large for long or interrupted trajectories simply because their generation spans multiple trainer updates. Consequently, a large K_{gen} does not necessarily imply that the current rollout policy assigns a substantially different likelihood to the realized prefix.

Waiting Staleness. $K_{\mathrm{wait}}(\tau; j)$ measures the additional lag accumulated after the trajectory has entered the completed-trajectory pool. Every token in the trajectory receives the same waiting component, and this component grows directly with pool residence time.

Taken together, the Stale-k results and Lemma 1 show that controlling aggregate policy lag requires addressing two distinct sources. Waiting Staleness arises after rollout completion and grows with pool residence time. The completed-trajectory backlog therefore provides a natural signal for controlling rejection pressure. Generation Staleness accumulates while a rollout is in flight and depends on response length and synchronization timing. Version age alone is therefore an incomplete signal for ranking trajectories. In the following section, we describe how we use these observations to design an efficient asynchronous RL algorithm.

3 PACE

In fully asynchronous RL, rollout production and trainer consumption can proceed at different rates. When rollout supply is insufficient, the trainer must wait for new data. When production persistently exceeds consumption, surplus trajectories accumulate in the pool. This backlog adds Waiting Staleness without increasing the throughput of an already saturated trainer. We first examine how this backlog develops and then present PACE (Pool-Aware Control of Effective Staleness) to control it. Building on the decomposition in Section 2, we use pool occupancy to determine *how much* data to reject and an effective-staleness score to determine *which* completed trajectories to reject.

3.1 POOL-AWARE CONTROL

In this part, we describe how PACE determines the rejection budget and selects trajectories for training. We begin with the example in Figure 4 to motivate this design. We train Qwen3-8B on mathematical reasoning tasks using the same setup as the Stale-8 experiment and track Generation Staleness and Waiting Staleness over time.

As shown in Figure 4 (left), when rollout production outpaces trainer consumption, completed trajectories accumulate without improving the throughput of an already saturated trainer. Each trainer update adds one unit of Waiting Staleness to every trajectory remaining in the pool, so persistent backlog increases Waiting Staleness while Generation Staleness remains relatively stable. Under stable operating conditions, Little's law relates average pool occupancy to average residence time at a given departure rate (Little & Graves, 2008). This motivates using occupancy above a target pool size to adapt the rejection budget online, removing surplus trajectories while retaining a buffer of ready data for training. Figure 4 (right) illustrates how controlling this backlog reduces Waiting Staleness.

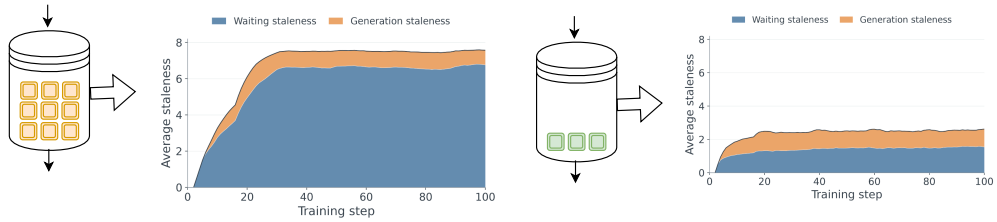


Figure 4: Pool occupancy and staleness. A persistently backlogged pool (left) accumulates Waiting Staleness as completed trajectories wait for training. Removing surplus trajectories keeps the pool smaller and continually replenished (right), reducing this additional waiting.

To determine the rejection rate, we measure the fraction of trajectories above the target pool size. Let N_j denote the number of completed trajectories in the pool at training step j, and let N^{tar} denote the desired buffer size of ready trajectories for training. When $N_j > N^{\text{tar}}$, we treat the $N_j - N^{\text{tar}}$ trajectories above this target as surplus. The instantaneous target rejection rate is then given by

$$r_j = \text{clip}\left(\frac{N_j - N^{\text{tar}}}{\max(N_j, 1)}, 0, 1\right). \tag{3}$$

Thus, $r_j = 0$ when the pool is at or below its target occupancy, and a larger relative surplus leads to a higher target rejection rate. To avoid abrupt changes in the rejection budget as pool occupancy fluctuates, we smooth r_j using an exponential moving average,

$$\widehat{r}_j = \beta \widehat{r}_{j-1} + (1-\beta) r_j, \qquad \beta \in [0, 1). \tag{4}$$

The smoothed budget $\widehat{r}_j$ determines how much data to reject. We next specify which completed trajectories to discard using a score-based filtering rule.

Let $M(\tau; j)$ be a scalar score assigned to a completed trajectory τ at training step j, with larger values indicating higher rejection priority. We will define its construction in Section 3.2. To implement the rejection budget, we target the upper $\widehat{r}_j$ fraction of the score distribution. Using a sliding window $\mathcal{W}_j^M$ of recently observed scores, PACE estimates the corresponding cutoff as $\kappa_j^M = \text{Quantile}_{1-\widehat{r}_j}\left(\mathcal{W}_j^M\right)$. A completed trajectory is rejected when $M(\tau; j) > \kappa_j^M$, and the trainer continues drawing trajectories until a full training batch is formed. This rule preferentially discards trajectories with higher scores while targeting the rejection fraction set by the pool-aware budget.

3.2 STALENESS-BASED TRAJECTORY SCORING

In this subsection, we design the trajectory score $M(\tau; j)$ used by the admission rule. We first introduce the Prefix Forgetting Score to measure policy drift on generated prefixes, then define a weighted staleness metric that accommodates different treatments of Waiting and Generation Staleness.

Prefix Forgetting Score. We address Waiting Staleness by controlling pool backlog. For Generation Staleness, we instead examine how much the policy has changed on the generated prefix. Generation Staleness records the policy-version lag accumulated before rollout completion, but long or interrupted rollouts may span several trainer updates while the updated rollout policy still assigns similar likelihoods to their prefixes. Generation age alone therefore provides an incomplete basis for deciding which trajectories to reject. We use prefix rescoring to measure the change in likelihood on the realized prefix and calibrate the contribution of Generation Staleness.

Let $\xi = (x, y_{1:m})$ denote a realized prefix generated before a policy synchronization, and let u denote the rollout-policy version under which the prefix is rescored. Using the behavior log-probabilities recorded during generation, we define the *Prefix Forgetting Score* as the mean absolute log-ratio between the updated rollout policy and the behavior policies that generated the prefix:

$$s_{\mathrm{pre}}(\xi; u) = \frac{1}{m} \sum_{t=1}^{m} \left| \log \pi_{\theta_u}(y_t \mid x, y_{<t}) - \ell_t^{\mu} \right|. \tag{5}$$

Smaller scores indicate closer agreement with the recorded behavior likelihoods on the realized prefix. This provides a trajectory-specific signal of observed policy drift that complements generation age. We write $s_{\mathrm{pre}}(\tau)$ for the prefix score associated with a completed trajectory.

Trajectory Score. Using the decomposition in Section 2, we define the trajectory staleness score:

$$M(\tau; j) = w_1 K_{\mathrm{wait}}(\tau; j) + w_2(\tau; j) K_{\mathrm{gen}}(\tau), \tag{6}$$

where $w_1 \geq 0$ controls the waiting penalty and $w_2(\tau; j) \geq 0$ controls the generation penalty. The generation weight can depend on the trajectory's prefix score and recent score statistics. Different choices of these weights yield the metrics below, all of which use the same pool-aware rejection budget and quantile-based filtering rule.

Effective Staleness. We use the Prefix Forgetting Score to adjust the generation penalty while retaining the full waiting penalty. Prefix rescoring during generation does not capture subsequent policy changes while the completed trajectory waits in the pool, so we retain Waiting Staleness as a conservative age penalty. Setting $w_1 = 1$ and $w_2(\tau; j) = F_j(s_{\mathrm{pre}}(\tau))$ gives

$$K_{\mathrm{eff}}(\tau; j) = K_{\mathrm{wait}}(\tau; j) + F_j(s_{\mathrm{pre}}(\tau))\, K_{\mathrm{gen}}(\tau), \tag{7}$$

where $F_j : \mathbb{R}_{\geq 0} \to [0, 1]$ is nondecreasing. This weighting discounts generation age more strongly when the observed prefix drift is small. To construct F_j, we measure each prefix score relative to recent observations. Using a sliding window $\mathcal{W}_j^{\mathrm{pre}}$ of recent prefix scores, we compute its empirical percentile rank as

$$q_j(\tau) = \widehat{\mathrm{CDF}}_{\mathcal{W}_j^{\mathrm{pre}}}(s_{\mathrm{pre}}(\tau)) \in [0, 1]. \tag{8}$$

A larger $q_j(\tau)$ indicates greater observed prefix drift relative to other scores in the window. We then apply a smooth, monotone temperature-sharpening map (Berthelot et al., 2019):

$$F_j(s_{\mathrm{pre}}(\tau)) = \phi_\gamma(q_j(\tau)) := \frac{q_j(\tau)^\gamma}{q_j(\tau)^\gamma + (1 - q_j(\tau))^\gamma}, \qquad \gamma \geq 1. \tag{9}$$

For $\gamma > 1$, this mapping pushes weights toward zero below the median and toward one above it, while preserving their ordering. Setting $\gamma = 1$ recovers the linear map $\phi_1(q) = q$. The resulting weights reflect prefix drift relative to recent observations. We report a sensitivity analysis for γ in Appendix D.1.2.

Other Variants. Setting $w_1 = 1$ and $w_2(\tau; j) = 1$ yields PACE (Raw Staleness), which weights Waiting Staleness and Generation Staleness equally. Setting $w_1 = 0$ and $w_2(\tau; j) = F_j(s_{\mathrm{pre}}(\tau))$ yields the prefix-weighted generation-only variant PACE (Generation Staleness), which removes the Waiting Staleness term to examine its role in trajectory ranking. Our algorithm is summarized in Algorithm 1. We then use two remarks to close this section.

Remark 1. The prefix-aware weighting reduces the Generation Staleness penalty for long or interrupted rollouts when their observed prefix drift is small. This design is motivated by the potential bias of raw-staleness filtering toward easier examples, which we examine in Appendix D.

Remark 2. PACE operates at trajectory admission and does not depend on a particular policy-optimization objective. It selects training data using pool occupancy and trajectory metadata, leaving the reward computation, advantage estimation, and policy loss unchanged. This separation allows integration with different single-turn and multi-turn asynchronous RL trainers, provided that they expose the required trajectory metadata and support prefix rescoring.

4 EXPERIMENTS

In this section, we conduct a comprehensive evaluation of our proposed PACE algorithm (Algorithm 1) and compare its performance against the state-of-the-art baseline, M2PO (Zheng et al., 2025b). Our experimental results demonstrate two key advantages of the proposed approach: (1) pool-aware rejection curbs the accumulation of Waiting Staleness by controlling the rollout backlog while maintaining training throughput; and (2) effective-staleness ranking distinguishes generation age from observed prefix-policy drift, enabling more selective trajectory filtering and supporting more stable training and stronger validation performance.

Experimental Setup. Following the experimental setup of DAPO (Yu et al., 2025), we conduct experiments using Qwen3-8B and Qwen3-4B (Yang et al., 2025) as base policies. We compare PACE against vanilla synchronous DAPO, asynchronous DAPO, and M2PO with DAPO as its base trainer. We train on DAPO-MATH-17K and evaluate all methods on six mathematical reasoning benchmarks: AIME 2023, AIME 2024, AIME 2025, BeyondAIME (Seed et al., 2025), BrUMO 2025, and HMMT 2025 (Balunovic et al., 2025). Table 1 reports mean@32 for all six benchmarks. We also evaluate PACE in a multi-turn mathematical reasoning setting based on ReTool. In this setting, the model interleaves natural language reasoning with code execution and incorporates execution feedback into subsequent reasoning. Full implementation details and hyperparameter settings are provided in the Appendix.

Training Stability and Performance. Figure 5 compares the validation trajectories on AIME 2024 and BeyondAIME over the first 300 training steps. All methods improve rapidly during early training, but their performance diverges as optimization progresses. On both benchmarks, PACE sustains improvements and consistently achieves higher accuracy than unfiltered Async after the initial evaluation. Its performance approaches that of Sync on BeyondAIME and exceeds Sync on AIME 2024 during later training. These gains persist across multiple evaluations, showing that the advantage of PACE extends beyond a single best-performing checkpoint. The complete validation trajectories for the single-turn and multi-turn settings are provided in Appendix C.

Table 1 further compares the results at the evaluation with the highest six-benchmark average within 300 updates. For single-turn mathematical reasoning with Qwen3-8B, PACE achieves an average accuracy of 41.29%, improving over Async by 4.95 percentage points and matching the performance of Sync at 41.07%. The improvement is larger in the multi-turn tool-integrated reasoning setting: PACE achieves 47.91% average accuracy, compared with 29.24% for Async, 39.07% for M2PO, and 45.06% for Sync. Together, these results demonstrate sustained gains over unfiltered asynchronous training and strong aggregate performance across both settings.

Table 1: Mean@32 validation accuracy (%) at the evaluation with the highest six-benchmark average within 300 training steps. Each row reports all six benchmarks at the same evaluation; Avg. is their unweighted mean. Green values indicate percentage-point improvements over Async. Bold values indicate the best result among asynchronous methods within each setting.

Method	AIME23	AIME24	AIME25	BeyondAIME	BrUMO25	HMMT25	Avg.
Single-Turn Mathematical Reasoning (Base: Qwen3-8B)							
Sync	45.00	55.83	41.46	27.34	52.60	24.17	41.07
Async	39.53	51.17	40.57	24.16	40.94	21.65	36.34
M2PO	**48.13**	56.88	40.42	27.31	41.88	18.33	38.82
PACE (Generation Staleness)	42.40	50.21	41.88	25.53	38.49	18.41	36.15
PACE (Raw Staleness)	42.50	53.91	40.42	24.49	50.54	**23.35**	39.20
PACE	43.54 (+ 4.01)	**57.66** (+ 6.49)	**43.44** (+ 2.87)	**27.81** (+ 3.65)	**52.40** (+ 11.46)	22.92 (+ 1.27)	**41.29** (+ 4.95)
Multi-Turn Tool-Integrated Reasoning (Base: Qwen3-4B-Instruct)							
Sync	52.08	59.69	50.83	26.50	48.13	33.13	45.06
Async	39.79	37.50	30.83	16.38	31.67	19.27	29.24
M2PO	49.17	57.29	41.56	22.84	35.94	27.60	39.07
PACE (Generation Staleness)	53.33	56.35	47.40	27.19	49.17	31.67	44.18
PACE (Raw Staleness)	**58.33**	57.81	48.44	27.19	50.94	32.19	45.82
PACE	57.81 (+ 18.02)	**63.02** (+ 25.52)	**52.40** (+ 21.56)	**28.06** (+ 11.69)	**51.77** (+ 20.10)	**34.38** (+ 15.10)	**47.91** (+ 18.67)

Staleness Distribution Diagnostics. We collect statistics over 50 training updates on the single-turn mathematical reasoning task. We record which candidate groups PACE retains or rejects in Figure 6 and compare the groups used for training under PACE and unfiltered Async in Figure 7.

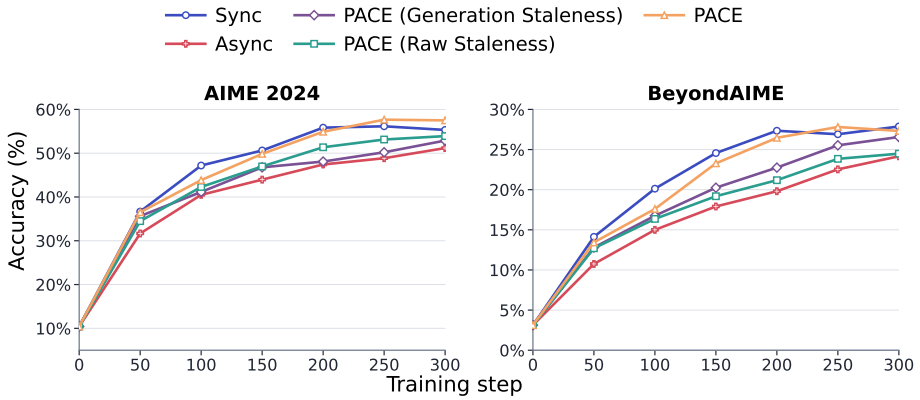
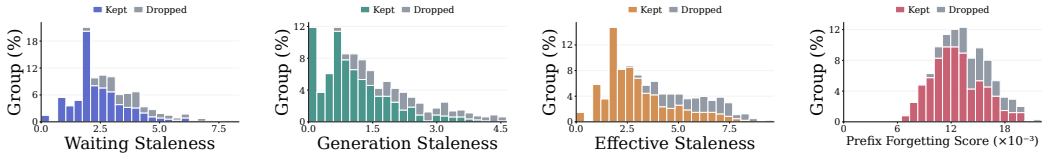
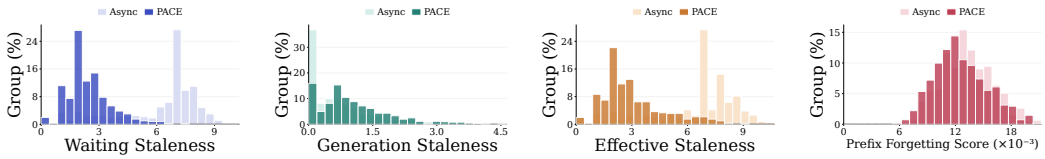


Figure 5: Representative mean@32 validation trajectories over the first 300 training steps. Results for all six benchmarks are provided in Appendix Figure 10.



(a) Waiting staleness. (b) Generation Staleness. (c) Effective staleness. (d) Prefix score.

Figure 6: PACE admission on math rollouts. Histograms show group-level metrics for all candidates encountered during the 50-update diagnostic: metric-specific colors denote retained groups, and gray denotes rejected groups.



(a) Waiting staleness. (b) Generation Staleness. (c) Effective staleness. (d) Prefix score.

Figure 7: Distributions of groups entering training for PACE and unfiltered Async. Within each panel, lighter shading denotes Async; darker shading denotes groups retained by PACE.

Figure 6 shows that groups with larger Waiting, Generation, and Effective Staleness tend to be rejected more often. As candidates are considered for training, PACE filters out high-staleness groups according to the pool-aware rejection budget. Removing surplus data helps keep the pool near its target size, allowing completed trajectories to enter training sooner and lowering the Waiting Staleness of the data consumed by the trainer. Figure 7 compares the groups entering training. With PACE, the Waiting and Effective Staleness distributions shift toward lower values, with fewer groups concentrated at large lags. This matches the goal of controlling pool accumulation so that completed trajectories spend less time waiting for training. Generation Staleness shows no comparable decrease. It accumulates during rollout generation and depends on rollout duration and policy-version progression. A larger generation lag does not necessarily indicate greater prefix-policy mismatch. PACE uses the prefix score to adjust this penalty, allowing trajectories with large generation lag to remain eligible when their observed prefix drift is low relative to recent scores.

Time Cost. Figure 8 compares the average wall-clock time per training step. PACE with raw and effective staleness takes 26.50 s and 28.22 s, respectively, compared with 26.19 s for vanilla Async and 97.01 s for Sync. The small overhead of effective-staleness filtering comes from prefix rescoring, which partially overlaps with rollout-side waiting after synchronization (hatched region). Overall, PACE retains most of the wall-time benefit of asynchronous execution while improving training performance.

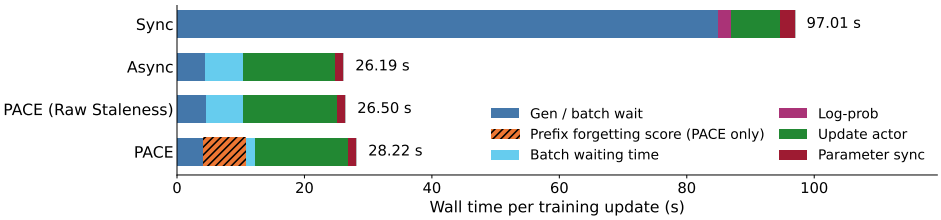


Figure 8: Wall-clock time breakdown per training update. All methods use six GPUs: Sync colocates rollout and training on all six, while asynchronous methods allocate four to rollout and two to training. Bars partition the trainer's elapsed update time into non-overlapping intervals that sum to the total. Hatched regions indicate prefix-scoring activity that overlaps rollout-related waits.

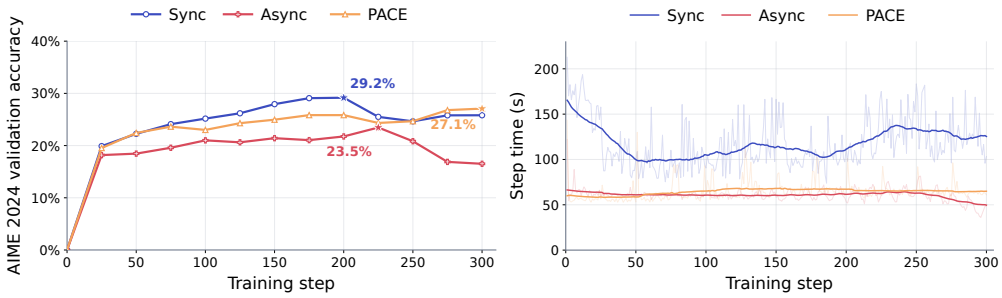


Figure 9: Scaling to Qwen3-30B-A3B-Base. Left: validation accuracy over 300 training iterations; Right: per-iteration training time, excluding validation and checkpoint saving. Timing curves use a 50-iteration moving average, with faint traces showing raw times.

Model Scaling. We further evaluate PACE on Qwen3-30B-A3B-Base (Yang et al., 2025), extending our study from dense 4B/8B backbones to a mixture-of-experts architecture with a larger total parameter count. As shown in Figure 9, PACE achieves higher AIME 2024 accuracy than Async during training, while remaining competitive with Sync. Its per-step training time stays close to Async and substantially below the Sync baseline.

Ablation Studies. We examine staleness ranking, model scale, and the sharpening parameter. **Staleness Scores.** Table 1 compares Generation-Staleness, Raw-Staleness, and Effective-Staleness ranking. The effective-staleness score achieves the highest average validation accuracy, supporting the inclusion of Waiting Staleness and prefix-aware weighting of Generation Staleness. **Model Scale.** We compare Async, PACE (Raw Staleness), and PACE on the smaller Qwen3-4B backbone. **Sharpening Parameter.** We vary $\gamma \in \{1, 2, 4\}$ and observe similar validation and training-reward trajectories, suggesting limited sensitivity within the tested range. Results for model scale and γ are provided in Appendix D.1.

5 Conclusion

In this work, we study where staleness accumulates in fully asynchronous RL and decompose policy lag into Generation and Waiting Staleness. Based on this distinction, we propose PACE, which uses pool occupancy to determine how much data to reject and effective staleness to determine which trajectories to reject. The score retains the full waiting penalty and uses prefix drift to adjust the generation penalty, avoiding rejection based on generation age alone. Our main experiments on single-turn and multi-turn mathematical reasoning show improved validation performance over unfiltered asynchronous training while preserving most of its wall-clock benefits. Further experiments with mixture-of-experts models and another RL algorithm support the applicability of PACE across model architectures and training algorithms.

AI USE STATEMENT

The authors used generative ai tools to assist with language editing, manuscript formatting, and reference checking. OpenAI's Codex was used to assist with implementing, debugging, and optimizing the experimental code. The authors reviewed and revised the AI-assisted text and verified the code before incorporating them into the manuscript and experimental implementation. The authors take full responsibility for the final manuscript, the experimental implementation, and the reported results, including all content produced with AI assistance.

REPRODUCIBILITY STATEMENT

To facilitate reproducibility, pseudocode is provided in Algorithm 1. Appendix C.1 documents the experimental settings, including model initialization, training data, optimization hyperparameters, resource allocations, and evaluation protocols. Appendix E specifies the ReTool prompt template, tool interface, execution settings, and reward calculation. Additional validation curves and ablation results are provided in Appendices C and D.

REFERENCES

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

Sanghwan Bae, Jiwoo Hong, Min Young Lee, Hanbyul Kim, JeongYeon Nam, and Donghyun Kwak. Online difficulty filtering for reasoning oriented reinforcement learning. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 700–719, 2026.

Mislav Balunovic, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. Matharena: Evaluating llms on uncontaminated math competitions. *Advances in Neural Information Processing Systems*, 38, 2025.

David Berthelot, Nicholas Carlini, Ian Goodfellow, Nicolas Papernot, Avital Oliver, and Colin Raffel. Mixmatch: A holistic approach to semi-supervised learning. *Advances in neural information processing systems*, 32, 2019.

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

Aili Chen, Aonian Li, Bangwei Gong, Binyang Jiang, Bo Fei, Bo Yang, Boji Shan, Changqing Yu, Chao Wang, Cheng Zhu, et al. Minimax-m1: Scaling test-time compute efficiently with lightning attention. *arXiv preprint arXiv:2506.13585*, 2025.

Thomas Degris, Martha White, and Richard S Sutton. Off-policy actor-critic. *arXiv preprint arXiv:1205.4839*, 2012.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms. In *International Conference on Learning Representations*, volume 2026, pp. 37909–37926, 2026a.

Yukang Feng, Jianwen Sun, Zelai Yang, Jiaxin Ai, Chuanhao Li, Zizhen Li, Fanrui Zhang, Kang He, Rui Ma, Jifan Lin, et al. Longcli-bench: A preliminary benchmark and study for long-horizon agentic programming in command-line interfaces. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 29952–29963, 2026b.

Wei Fu et al. AReaL: A large-scale asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025.

Jiaxuan Gao, Wei Fu, Minyang Xie, Shusheng Xu, Chuyi He, Zhiyu Mei, Banghua Zhu, and Yi Wu. Beyond ten turns: Unlocking long-horizon agentic search with large-scale asynchronous rl. *arXiv preprint arXiv:2508.07976*, 2025.

Tyler Griggs, Sumanth Hegde, Eric Tang, Shu Liu, Shiyi Cao, Dacheng Li, Charlie Ruan, Philipp Moritz, Kourosh Hakhamaneshi, Richard Liaw, Akshay Malik, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Evolving skyrl into a highly-modular rl framework, 2025. Notion Blog.

Bowei He, Minda Hu, Zenan Xu, Hongru Wang, Licheng Zong, Yankai Chen, Chen Ma, Xue Liu, Pluto Zhou, and Irwin King. Search-r2: Enhancing search-integrated reasoning via actor-refiner collaboration. *arXiv preprint arXiv:2602.03647*, 2026.

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

Tianhao Hu, Xiangcheng Liu, Youshao Xiao, Yang Zheng, Xuan Huang, Jinrui Ding, Yufei Zhang, Tao Liang, Hongyu Zang, Quan Chen, et al. Dora: A scalable asynchronous reinforcement learning system for language model training. *arXiv preprint arXiv:2604.26256*, 2026.

Luke J Huang, Zhuoyang Zhang, Qinghao Hu, Shang Yang, and Song Han. Stable asynchrony: Variance-controlled off-policy rl for llms. *arXiv preprint arXiv:2602.17616*, 2026.

Prime Intellect. prime-rl, 2025. URL `https://github.com/PrimeIntellect-ai/prime-rl`.

Chenliang Li, Adel Elmahdy, Alex Boyd, Zhongruo Wang, Alfredo Garcia, Parminder Bhatia, Taha Kass-Hout, Cao Xiao, and Mingyi Hong. St-ppo: Stabilized off-policy proximal policy optimization for multi-turn agents training. *arXiv preprint arXiv:2511.20718*, 2025a.

Chenliang Li, Adel Elmahdy, Alex Boyd, Zhongruo Wang, Siliang Zeng, Alfredo Garcia, Parminder Bhatia, Taha Kass-Hout, Cao Xiao, and Mingyi Hong. Stabilizing off-policy training for long-horizon llm agent via turn-level importance sampling and clipping-triggered normalization. *arXiv preprint arXiv:2511.20718*, 2025b.

Siheng Li, Zhanhui Zhou, Wai Lam, Chao Yang, and Chaochao Lu. Repo: Replay-enhanced policy optimization. *arXiv preprint arXiv:2506.09340*, 2025c.

John DC Little and Stephen C Graves. Little's law. In *Building intuition: insights from basic operations management models and principles*, pp. 81–100. Springer, 2008.

Fanbin Lu, Zhisheng Zhong, Shu Liu, Chi-Wing Fu, and Jiaya Jia. Arpo: End-to-end policy optimization for gui agents with experience replay. *arXiv preprint arXiv:2505.16282*, 2025.

Shubham Parashar, Shurui Gui, Xiner Li, Hongyi Ling, Sushil Vemuri, Blake Olson, Eric Li, Yu Zhang, James Caverlee, Dileep Kalathil, et al. Curriculum reinforcement learning from easy to hard tasks improves llm reasoning. In *International Conference on Learning Representations*, volume 2026, pp. 156944–156972, 2026.

Nicolas Le Roux, Marc G Bellemare, Jonathan Lebensold, Arnaud Bergeron, Joshua Greaves, Alex Fréchette, Carolyne Pelletier, Eric Thibodeau-Laufer, Sándor Toth, and Sam Work. Tapered off-policy reinforce: Stable and efficient reinforcement learning for llms. *arXiv preprint arXiv:2503.14286*, 2025.

ByteDance Seed, Jiaze Chen, Tiantian Fan, Xin Liu, Lingjun Liu, Zhiqi Lin, Mingxuan Wang, Chengyi Wang, Xiangpeng Wei, Wenyuan Xu, et al. Seed1. 5-thinking: Advancing superb reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.13914*, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. Kimi k1. 5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*, 2025.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244, 2025.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, et al. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025a.

Haizhong Zheng, Jiawei Zhao, and Beidi Chen. Prosperity before collapse: How far can off-policy rl reach with stale data on llms? *arXiv preprint arXiv:2510.01161*, 2025b.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

Yuzhen Zhou, Jiajun Li, Yusheng Su, Gowtham Ramesh, Zilin Zhu, Xiang Long, Chenyang Zhao, Jin Pan, Xiaodong Yu, Ze Wang, et al. April: Active partial rollouts in reinforcement learning to tame long-tail generation. *arXiv preprint arXiv:2509.18521*, 2025.

A Related Work

A.1 Off-policy Optimization for LLMs

Off-policy updates can improve sample efficiency, but heavy-tailed importance ratios can increase policy-gradient variance (Li et al., 2025a). Existing methods address this through replay design or control of importance weights. TOPR (Roux et al., 2025) asymmetrically tapers REINFORCE (Williams, 1992) importance weights. ARPO (Lu et al., 2025) combines fresh and replayed samples to reduce zero-advantage GRPO groups, while RePO (Li et al., 2025c) selects past outputs using recency and reward. CISPO (Chen et al., 2025) clips token-level importance weights, while GSPO (Zheng et al., 2025a) clips length-normalized sequence ratios. M2PO (Zheng et al., 2025b) masks outlying tokens until the batch-level second moment of the importance weights falls below a threshold. In asynchronous training, off-policy data also arises without explicit replay, as the trainer advances during rollout generation and subsequent waiting.

A.2 Asynchronous RL Systems

Asynchronous RL systems overlap rollout generation with policy optimization (Hu et al., 2026; nem, 2025; Intellect, 2025; Griggs et al., 2025; Cao et al., 2025; Sheng et al., 2024). APRIL (Zhou et al., 2025) mitigates stragglers by retaining unfinished rollouts and resuming them in later iterations. AReaL (Fu et al., 2025) fully decouples generation from training and pools trajectories whose segments may use different policy versions. For long-horizon GRPO (Shao et al., 2024), SAO (Hou et al., 2026) removes the group-level synchronization barrier and corrects delayed rollouts using clipped token-level importance ratios. PACE addresses trajectory admission in this setting: pool occupancy determines the rejection budget, and effective staleness determines the rejection order.

B Proofs

B.1 Proof of Lemma 1

Proof. Starting from the definition of staleness in Equation 1, we add and subtract the completion index $c(\tau)$ inside each token-level lag:

$$\begin{aligned}
\bar{k}(\tau; j) &= \frac{1}{T}\sum_{t=1}^{T}(j - v_t) \\
&= \frac{1}{T}\sum_{t=1}^{T}[(c(\tau) - v_t) + (j - c(\tau))] \\
&= \frac{1}{T}\sum_{t=1}^{T}(c(\tau) - v_t) + \frac{1}{T}\sum_{t=1}^{T}(j - c(\tau)) \\
&= \frac{1}{T}\sum_{t=1}^{T}(c(\tau) - v_t) + j - c(\tau).
\end{aligned}$$

The first term is $K_{\text{gen}}(\tau)$ and the second is $K_{\text{wait}}(\tau; j)$, which proves Equation 2. Moreover, $v_t \leq c(\tau) \leq j$ implies that both components are nonnegative. □

C Additional Experimental Results

C.1 Experimental Settings

We describe the Qwen3-8B and ReTool runs in Table 1, together with the additional Qwen3-4B and MoE experiments. Tables 2 and 3 separate shared choices from task-specific settings, including their evaluation intervals.

Table 2: Shared settings for dense math, MoE, and ReTool experiments. The clipping parameters apply to the standard policy objective; M2PO uses its own policy loss.

Setting	Value
Training data	DAPO-MATH-17K
Advantage estimator	GRPO with within-group standardization
Peak learning rate	10^{-6}; constant after warmup
Adam momentum coefficients	$(0.9, 0.999)$
Epochs per training batch	1
Loss aggregation	Token mean
Lower / upper clipping widths	0.20 / 0.28
Dual-clip coefficient	10
Gradient-norm clipping	1.0
KL penalty / entropy bonus	Disabled / disabled
Training temperature / top-p	1.0 / 1.0
Maximum prompt length	2,048 tokens
Generation engine	vLLM
Dynamic accuracy-based group sampling	Disabled

Shared optimization protocol. These experiments use group-relative advantages without a learned critic. The standard objective combines asymmetric clipping with a token-mean reduction; dynamic sampling that discards groups with identical rewards is not enabled. Such accuracy-based sampling is distinct from PACE's staleness-based admission. In the dense math and current MoE runs, the policy ratio uses recorded rollout log probabilities as its behavior-policy reference. No additional rollout importance-weighting or rejection-sampling correction is applied; the importance ratio in the policy objective itself remains present.

Table 3: Task-specific settings. ReTool denotes the main GRPO-based runs; the separate REINFORCE++ comparison is described in Appendix D.1.3.Validation periods are given in logged trainer steps; their conversion to optimizer updates is described below.

Setting	Math (4B/8B)	MoE (30B-A3B)	ReTool (4B)
Trainer backend	FSDP	Megatron	FSDP
Optimizer	AdamW	Megatron Adam	AdamW
Weight decay	0.1	0.1	0.01
Warmup (optimizer updates)	10	20	0
Response budget (tokens)	8,192	20,480	8,192
Responses per training prompt	12	16	8
Prompts per optimizer minibatch	12	24	8
Trajectories per optimizer minibatch	144	384	64
Rollout tensor parallelism	1	2	1
Validation responses per problem	32	32	32
Validation temperature / top-p	1.0/0.7	1.0/0.7	1.0/0.6
Validation period	50	25	25

Single-turn math: Qwen3-4B. We initialize from Qwen3-4B and use four GPUs for rollout and two for training. Each trainer update consumes 12 complete prompt groups, followed by weight synchronization. The Sync baseline uses the same disjoint resource layout with a zero-staleness supply setting; Async and PACE use a supply threshold of eight and allow partial rollouts across weight synchronizations. This threshold controls outstanding work rather than imposing a strict upper bound on every trajectory's measured staleness. The configured horizon is 400 updates, with no initial validation. Training uses the DAPO answer verifier, with signed-integer normalization and an overlong-response penalty applied over the final 4,096 tokens of the response budget, with penalty factor one. Evaluation reports unshaped answer accuracy on AIME 2023/2024/2025, BeyondAIME, BrUMO 2025, and HMMT 2025: 250 problems in total.

Single-turn math: Qwen3-8B. We initialize from Qwen3-8B and retain the Qwen3-4B training settings, including the data, reward, optimizer, sequence-length limits, sampling parameters, and admission configuration. The main results use mean@32 The resource allocation spans two nodes, with 12 GPUs for rollout and four for training (12r4t).

Mixture-of-experts: Megatron 12r12t. We initialize from Qwen3-30B-A3B-Base. The asynchronous configuration allocates 12 GPUs to rollout and 12 to training; the matched synchronous configuration colocates generation and training on 24 GPUs. Training uses tensor and expert parallel degrees of two, with pipeline and context parallel degrees of one. Expert-routing replay and hybrid resource switching are disabled. Each logged trainer iteration processes 96 prompt groups in four minibatches of 24 groups. The asynchronous trainer synchronizes weights after these four optimizer updates. The continuous-rollout inventory is bounded at nine training batches (864 prompt groups, including unfinished work). The configured stopping point is 300 trainer iterations, with AIME 2024 and AIME 2025 evaluated every 25 iterations. The plotted run uses mean@32. Training uses the DAPO reward with a 4,096-token overlong buffer and penalty factor one.

Multi-turn reasoning: ReTool. We initialize from Qwen3-4B-Instruct-2507. The selected Sync run uses four GPUs and a data batch of 32 prompts, split into four optimizer minibatches of eight prompts. Async consumes one eight-prompt minibatch per logged step on a 2r2t allocation. The selected PACE variants and M2PO use 4r2t, with the same optimizer minibatch and eight responses per prompt. Accordingly, ReTool Sync evaluations are spaced by 100 optimizer updates, whereas asynchronous evaluations are spaced by 25; the displayed optimizer-step axis accounts for this difference.

The response budget spans the complete multi-turn interaction. Assistant and tool-response turn limits are each set to eight, with one tool call processed at a time. The Hermes tool-calling template, SandboxFusion feedback, and turn-shaped training reward are specified in Appendix E. Tool feedback remains in the context but is masked from the policy loss. Validation reports unshaped mean@32 accuracy on the same six benchmarks as the math evaluation. M2PO retains GRPO advantages and uses its core loss with budget 0.04.

Admission and prefix-scoring parameters. The controllers above use a 512-group score window, a minimum of 32 observations, a five-update throughput window, and a maximum rejection budget of 0.9. These are controller settings, not a fixed realized drop fraction. Full PACE in the dense math and ReTool experiments uses $\gamma = 4$ and mean aggregation within groups. Prefix scores are collected at partial-rollout interruptions, requiring at least 32 eligible tokens; the scoring caps are 1,024 response tokens for math and 2,048 for ReTool. PACE (Raw Staleness) omits prefix scoring and uses version lag directly.

C.2 Pseudocode

Algorithm 1 summarizes the two-stage procedure in Section 3: rollout workers record policy-version and prefix-score metadata, while the trainer applies the pool-aware rejection budget to completed trajectories. The trainer retains data until a full batch is formed, then performs the original policy update and weight synchronization.

C.3 Per-Benchmark Training Dynamics

Figure 10 reports the complete validation trajectories over the first 300 training steps. The full PACE method generally remains above unfiltered Async across the six benchmarks, showing that the improvement in Table 1 is not produced by a single isolated checkpoint or benchmark. The advantage is particularly visible on AIME 2024, BeyondAIME, and BrUMO 2025. HMMT 2025 exhibits greater checkpoint-to-checkpoint variation, where PACE (Raw Staleness) slightly outperforms the full method at the selected checkpoint. Nevertheless, PACE provides the strongest aggregate performance and the most consistent improvement over unfiltered asynchronous RL.

Algorithm 1 PACE: Pool-Aware Control of Effective Staleness

Require: Completed-trajectory pool $\mathcal{Q}$, target occupancy N^{tar}, trainer batch size B_{tr}, smoothing coefficient β, and sharpening parameter γ.

Rollout-side metadata collection

Generate trajectories while recording token-level behavior versions and log probabilities.

When a prefix is rescored after a policy synchronization, attach its Prefix Forgetting Score to the trajectory; do not reject the rollout at this stage.

Insert each completed trajectory and its generation metadata into $\mathcal{Q}$.

Trainer-side admission

for training step $j = 1, 2, \ldots$ **do**

Compute the pool-aware rejection rate r_j and its smoothed value $\widehat{r}_j$.

Compute the empirical percentile $q_j(\tau)$, apply $F_j = \phi_\gamma(q_j(\tau))$, and form $K_{\mathrm{eff}} = K_{\mathrm{wait}} + F_j K_{\mathrm{gen}}$.

Set κ_j^M to the $(1 - \widehat{r}_j)$-quantile of recent effective-staleness scores, with $M = K_{\mathrm{eff}}$.

Draw from $\mathcal{Q}$, reject trajectories with $K_{\mathrm{eff}}(\tau; j) > \kappa_j^M$, and retain the rest until $|\mathcal{B}_j| = B_{\mathrm{tr}}$.

Update the policy using $\mathcal{B}_j$ and periodically synchronize the latest weights to rollout workers.

end for

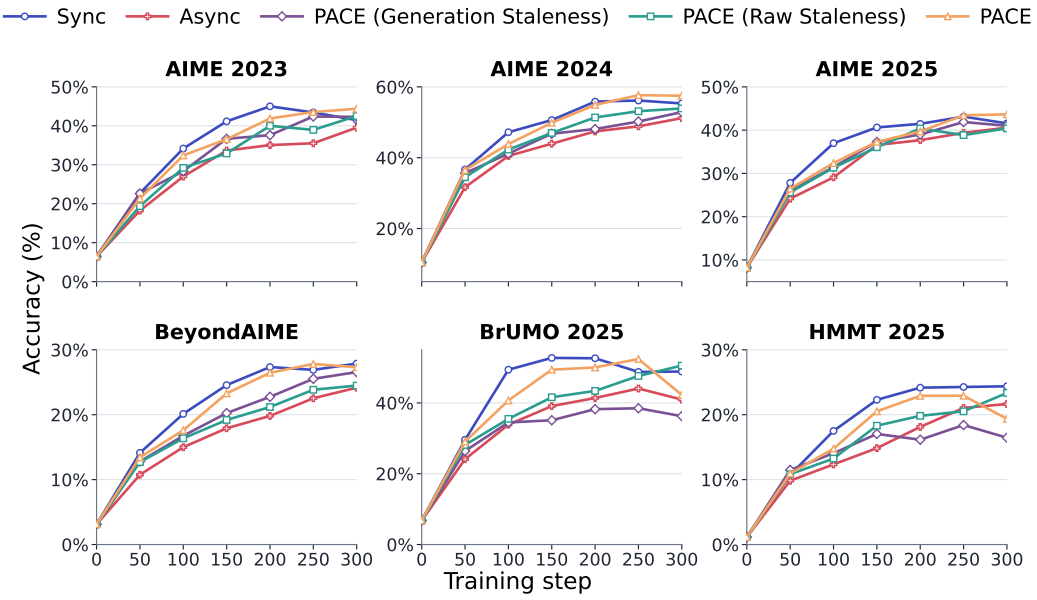


Figure 10: Mean@32 validation performance over the first 300 training steps on six math benchmarks. Sync, Async, PACE (Generation Staleness), PACE (Raw Staleness), and PACE.

C.4 ReTool Validation Curves

Figure 11 provides the per-benchmark trajectories for the ReTool comparison in Table 1. The horizontal axis counts optimizer updates, with four updates per logged Sync step and one per logged Async or PACE step.

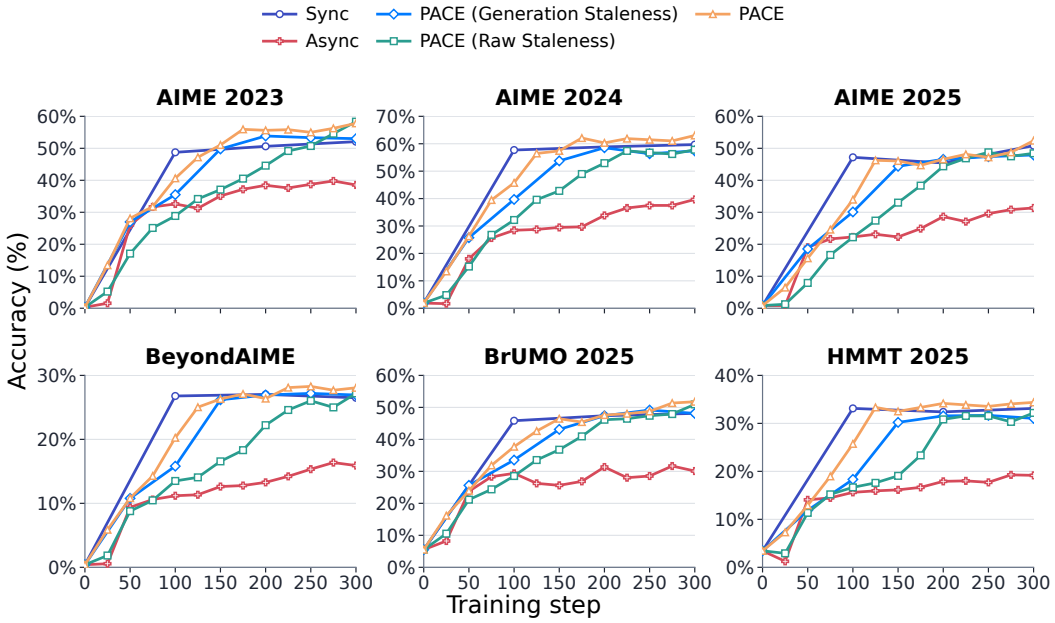


Figure 11: ReTool mean@32 validation accuracy over the first 300 optimizer updates on six benchmarks for Sync, Async, PACE (Generation Staleness), PACE (Raw Staleness), and PACE.

D Selection Bias under Raw-Staleness Filtering

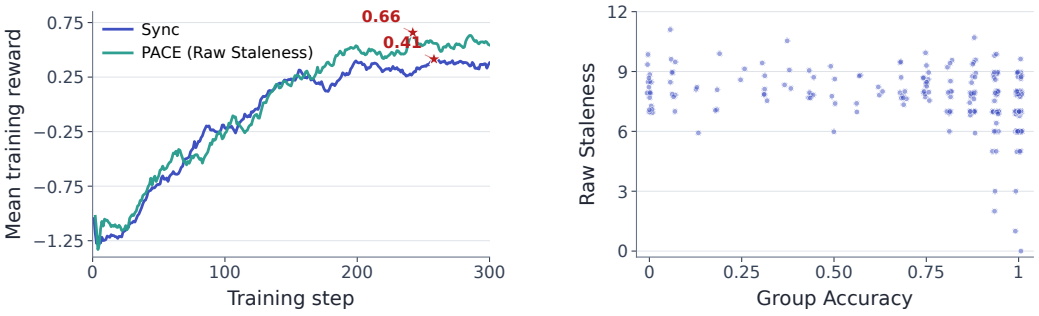
Figure 12 illustrates a potential selection bias introduced by raw-staleness filtering. In the left panel, PACE (Raw Staleness) achieves higher late-stage training reward than Sync under otherwise identical training settings. In the separate fixed-policy ReTool probe shown in the right panel, lower-accuracy prompt groups tend to have larger raw staleness. Since raw-staleness ranking assigns higher rejection priority to larger policy-version gaps, it can preferentially remove harder examples. Together, these observations suggest that the higher training reward may partly reflect a shift toward easier training data. Prior work also suggests that persistently favoring easier examples can limit further policy improvement (Parashar et al., 2026; Bae et al., 2026).

These observations motivate giving greater tolerance to Generation Staleness. Harder problems may require longer or interrupted rollouts and accumulate more generation lag while their prefixes remain compatible with the updated rollout policy. Applying the full generation-age penalty can therefore reject difficult but potentially informative trajectories. PACE discounts this penalty when the Prefix Forgetting Score indicates relatively small policy drift, while retaining the full Waiting Staleness penalty to account for policy updates after rollout completion.

D.1 Ablation Study

D.1.1 Ablation on Model Scale.

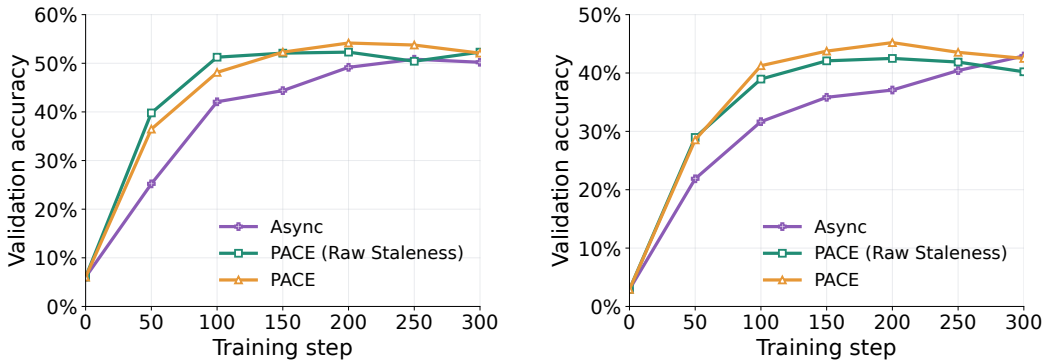
Figure 13 compares Async, PACE (Raw Staleness), and PACE using Qwen3-4B. It reports AIME 2024 and AIME 2025 mean@32 validation accuracy over the first 300 training steps.



(a) Training reward.

(b) Accuracy and raw staleness.

Figure 12: Left: DAPO-MATH-17K training reward for Sync and PACE (Raw Staleness). Right: group accuracy versus raw staleness in a fixed-policy ReTool probe.



(a) AIME 2024.

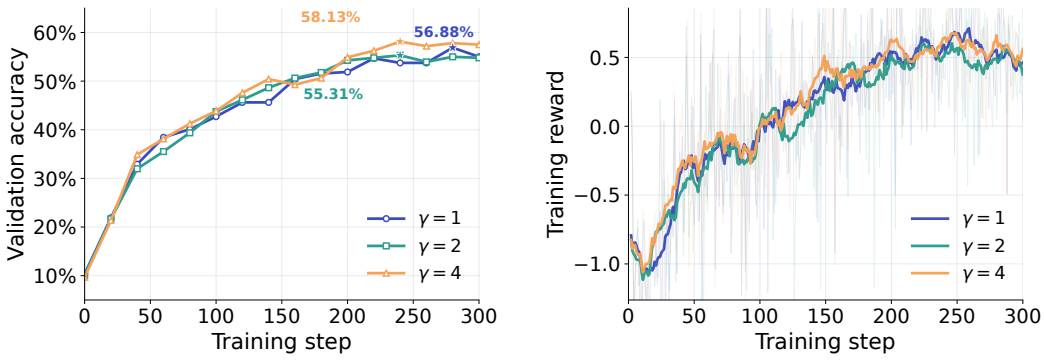
(b) AIME 2025.

Figure 13: Qwen3-4B validation accuracy versus training step on AIME 2024 (left) and AIME 2025 (right) over the first 300 training steps. Curves show the local mean@32 for Async, PACE (Raw Staleness), and PACE. The common measured initial-model evaluation is reused at step zero as a display reference.

D.1.2 Sensitivity to the Parameter γ.

We next vary the sharpening parameter over $\gamma \in \{1, 2, 4\}$. Recall that $\gamma = 1$ recovers the unsharpened quantile weight, whereas larger values increase the polarization between prefixes with low and high forgetting scores. As shown in Figure 14**(a)**, the AIME 2024 validation curves follow similar optimization trajectories. Their best mean@32 accuracies are 56.88%, 55.31%, and 58.13% for $\gamma = 1$, 2, and 4, respectively, giving a maximum difference of only 2.82 percentage points. The corresponding training-reward curves are provided in Appendix Figure 14**(b)**.

These results indicate that PACE is not particularly sensitive to the precise value of γ within the tested range. In particular, changing the polarization strength of the quantile-sharpening function has only a limited effect on the overall optimization dynamics. Because $\gamma = 4$ achieves the highest observed validation accuracy without destabilizing training, we use $\gamma = 4$ as the default in all main experiments.



(a) Validation accuracy.

(b) Training reward.

Figure 14: Sensitivity to $\gamma \in \{1, 2, 4\}$ on Qwen3-8B. Left: AIME 2024 mean@32 validation accuracy. Right: DAPO-MATH-17K training reward over the first 300 training steps.

D.1.3 Other RL Algorithms

To examine whether PACE could be applied beyond GRPO, we compare Async with PACE using REINFORCE++ in the ReTool setting with Qwen3-4B-Instruct-2507. Both runs use the same learning rate, minibatch size, rollout group size, and seed, with an asynchronous staleness threshold of eight. As shown in Figure 15, this variant attains higher validation accuracy on both AIME benchmarks later in training. This comparison supports the applicability of pool-aware admission control beyond GRPO;

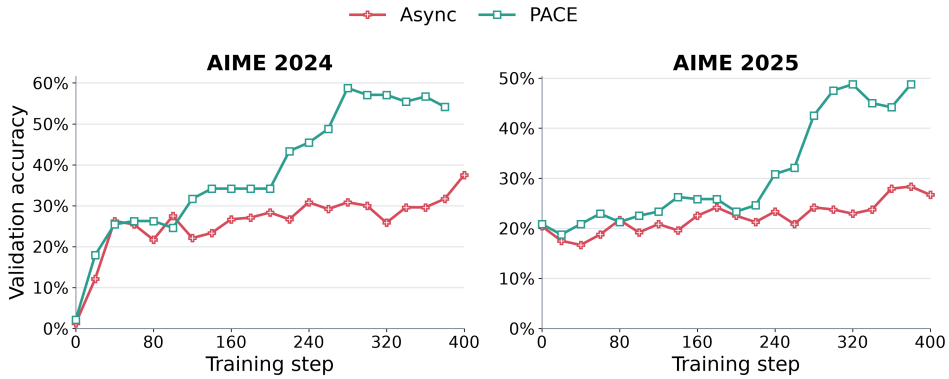


Figure 15: REINFORCE++ on ReTool: AIME 2024 (left) and AIME 2025 (right) validation accuracy versus optimization step.

E ReTool System Instruction and Tool Interface

The training and validation datasets contain a user message, not a separate hand-written system message. Qwen3-4B-Instruct-2507's tokenizer constructs the system instruction below from the registered `code_interpreter` schema. We show the resulting prompt, including the user-message suffix that requests a boxed answer; {`problem`} is the only substituted field. Line wrapping is for readability. The same template is used for Sync, Async, and PACE.

The system instruction defines the code-execution interface and the format of tool-call messages. Execution feedback is returned to the model for subsequent reasoning, and the user-message suffix specifies the boxed-answer format used by the verifier.

Execution and feedback. Each tool request runs a self-contained Python program in a stateless SandboxFusion call, with a 30-second default timeout and a 1024-MB memory limit. The

server permits up to 16 concurrent executions across rollouts. Tool replies are serialized using <tool_response> tags. Replies longer than 256 characters retain their first and last 128 characters, separated by a truncation marker; this limit is in characters, not tokens. Feedback tokens remain in the context but do not contribute to the policy loss.

Reward and validation. The shared reward checks the final boxed answer, assigning 1 when correct. Otherwise it assigns $\min\{-0.6, -1 + 0.05(T - 2)\}$, where T is the logged total turn count (initial user message, assistant turns, and tool-response turns). The reported validation metric instead uses the unshaped binary correctness indicator, averaged over 32 responses per problem. The validation sets contain 30 problems each for AIME 2023, AIME 2024, AIME 2025, BrUMO 2025, and HMMT 2025, and 100 for BeyondAIME.

Table 4: Prompt template and tool interface used for ReTool training and evaluation. The placeholder {problem} is replaced with the input question.

System message

```
<|im_start|>system
# Tools
You may call one or more functions to assist with the user query.
You are provided with function signatures within <tools></tools> XML tags:
<tools>
{
  "type": "function",
  "function": {
    "name": "code_interpreter",
    "description": "Execute a self-contained Python 3 program for mathematical
  calculation and verification. The code value must contain raw executable
  Python source only: do not include Markdown fences, backticks, XML, JSON
  wrappers, shell commands, or explanatory prose. Explicitly print every result
  needed in the response. Each call is stateless, so repeat all required
  imports and definitions. If execution returns an error, correct the code
  before retrying.",
    "parameters": {
      "type": "object",
      "properties": {
        "code": {
          "type": "string",
          "description": "Raw, self-contained Python 3 source with no Markdown
  fencing; use print(...) to return results."
        }
      },
      "required": ["code"]
    }
  }
}
</tools>
For each function call, return a json object with function name and arguments
  within <tool_call></tool_call> XML tags:
<tool_call>
{
  "name": <function-name>,
  "arguments": <args-json-object>
}
</tool_call><|im_end|>
```

User message

```
<|im_start|>user
{problem}
The answer format must be: \boxed{'The final answer goes here.'}<|im_end|>
```

Assistant generation prefix

```
<|im_start|>assistant
```