AsyncOPD: How Stale Can On-Policy Distillation Be?

Wonjun Kang[*1] **Kevin Galim**[*1] **Seunghyuk Oh**[1] **Minjun Kang**[2] **Sanghyun Park**[2]
Donghoon Kim[1] **Minjae Lee**[1] **Minseo Kim**[1] **Rishabh Tiwari**[3] **Yuchen Zeng**[4]
Hyung Il Koo[1,2] **Kangwook Lee**[5,6]

[1] FuriosaAI [2] Ajou University [3] UC Berkeley [4] Microsoft Research
[5] KRAFTON [6] Ludo Robotics

Code: `https://github.com/furiosa-ai/async-opd`

Abstract

On-policy distillation (OPD) trains a student on its own rollouts guided by teacher feedback and is becoming increasingly important for large language model (LLM) post-training. Like reinforcement learning (RL), however, OPD faces an on-policy systems bottleneck, as rollouts can dominate training time for reasoning workloads. Asynchronous training pipelines can alleviate this bottleneck by decoupling rollout generation from learner updates, but doing so introduces stale-policy data. While prior work has studied stale data in asynchronous RL, its effects in OPD remain underexplored. We present the first systematic study of staleness in asynchronous OPD, focusing on a practical setting where teacher feedback is implemented through local KL losses and full-vocabulary teacher logits are too expensive to store or transfer, necessitating finite teacher-score caches. We first show that KL direction changes the stale-data problem: teacher-weighted forward KL is more robust to stale rollouts, whereas student-weighted reverse KL is vulnerable. Second, for this vulnerable reverse-KL case, we study whether methods designed to stabilize asynchronous RL can mitigate OPD staleness. In our experiments, they do not improve over a simpler OPD-specific surrogate: recomputing the reverse-KL signal under the current student at learner time. Third, we analyze how finite teacher-score caches create a bias-variance tradeoff for sparse and sampled reverse-KL OPD estimators. This motivates multi-sample Monte Carlo (MC), which preserves MC correctability while reducing one-sample variance. Finally, we present and open-source **AsyncOPD**, a fully asynchronous OPD training pipeline built from these estimator choices. Experiments show that AsyncOPD improves training throughput by $1.6\times$ to $3.8\times$ over strict synchronous training while reaching comparable accuracy.

1 Introduction

On-policy distillation (OPD) [1, 6, 20] and reinforcement learning (RL) [26, 28] have become central post-training methods for improving large language model (LLM) reasoning [7], including mathematics [17] and coding [27]. OPD trains a student on its own rollouts using dense token-level feedback from a teacher [12], whereas RL learns from reward feedback on rollouts. OPD provides an effective and efficient route for LLM post-training, especially for smaller student models [24]. Recent work shows that OPD is not limited to distilling large teachers into small students: it also supports on-policy self-distillation [32] and multi-teacher distillation from domain-specialized teachers comparable in size to the student [2, 21].

[*]Equal Contribution. Emails: `{kangwj1995, kevin.galim}@furiosa.ai`.

Accepted at ICML 2026 Workshop on Decision-Making from Offline Datasets to Online Adaptation: Black-Box Optimization to Reinforcement Learning.

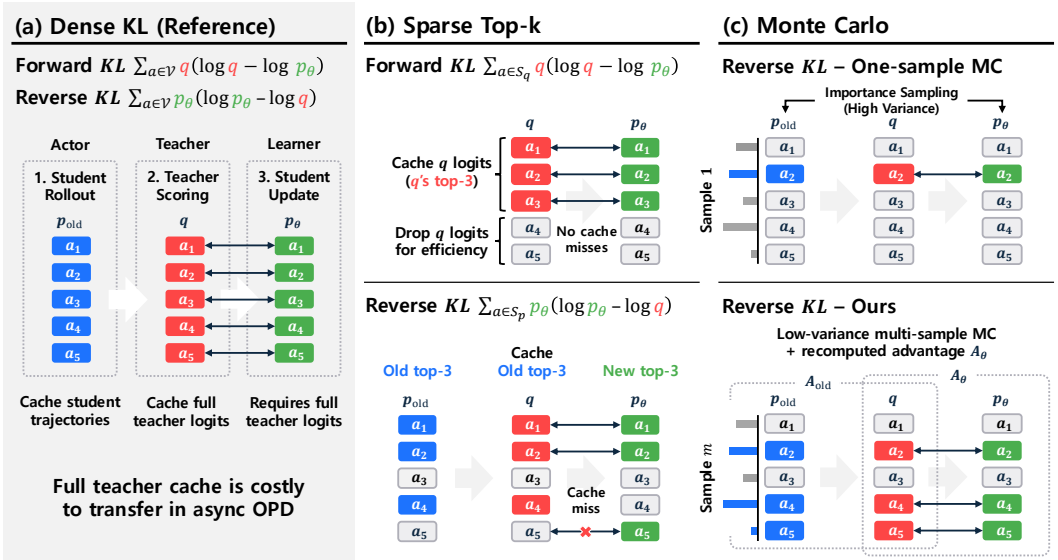


Figure 1: Estimator design for asynchronous OPD. (a) Dense KL is the full-vocabulary reference, but full teacher-logit caches are costly to store or transfer in asynchronous OPD. (b) Sparse top-k exposes a support mismatch under staleness: forward KL is teacher-supported, but reverse KL is student-supported and may require actions outside the cached teacher-scored support. (c) One-sample Monte Carlo is correctable in expectation by importance sampling, but has high variance; our estimator recomputes A_θ at learner time and uses multi-sample MC to reduce variance.

OPD and RL inherit an on-policy systems bottleneck: each learner update must wait for fresh rollouts from the model being trained [5]. For reasoning tasks, these rollouts are long and expensive, so synchronous training often waits on generation rather than updating the model, leaving learners underutilized. Asynchronous RL [4, 14] relieves this bottleneck by decoupling rollout generation from learner updates: rollout workers keep generating data while the learner updates on earlier rollouts, improving training efficiency and hardware utilization [18, 23, 34]. A similar pipeline can be applied to OPD by running student rollout, teacher scoring, and learner updates in parallel [19].

However, asynchronous execution introduces stale-policy data, and learning from such data can degrade model quality [3]. This creates a trade-off: more aggressive asynchrony improves training throughput, but it also increases the policy lag between rollout and learning. Prior work on asynchronous RL therefore studies how to stabilize learning from stale-policy data [4, 10, 33]. However, it remains underexplored whether these ideas and stale-data solutions transfer to OPD, because practical implementations of OPD expose a different feedback interface. Teacher feedback is often implemented through local KL losses, which require teacher scores over actions at student-visited prefixes. Since full-vocabulary teacher logits are expensive to store or transfer, especially in an asynchronous pipeline, teacher scores are usually cached only on a finite set of actions (Fig. 1). Once the learner receives a teacher-scored cache, it can recompute current-student log probabilities on cached actions, but it cannot recover teacher scores for actions that were never scored. This raises three questions that structure our study: (i) how asynchronous OPD behaves under staleness, (ii) whether asynchronous RL ideas and stale-data solutions transfer to OPD, and (iii) how finite teacher-score caches shape OPD estimator design.

First, we study how KL direction shapes staleness. Under asynchronous OPD with cached teacher scores, the same stale rollout cache can affect different KL objectives differently. As illustrated in Fig. 1, forward KL is teacher-weighted and is more robust to stale rollouts, whereas reverse KL is student-weighted and becomes vulnerable when current-student actions fall outside the scored cache. We therefore focus on reverse-KL OPD in the remainder of the staleness analysis.

Second, focusing on the reverse-KL case, we ask whether methods designed to stabilize asynchronous RL can also mitigate OPD staleness. This comparison is natural because reverse KL in OPD admits an RL-style policy-gradient surrogate, where the teacher-student log-ratio acts as a token-level advantage. We therefore evaluate PPO-style clipping [16], decoupled PPO [4], and M2PO [33].

In our experiments, they do not improve over a simpler OPD-specific surrogate: recomputing the reverse-KL token-level advantage under the current student at learner time without clipping.

Third, we return to the teacher-cache constraint and study the resulting bias-variance tradeoff for sparse and sampled reverse-KL OPD implementations. Stale student top-k supports provide deterministic coverage but are support-mismatched because they may omit actions required by the current top-k objective, and reweighting inside the stale support cannot recover the missing teacher scores. One-sample Monte Carlo (MC) avoids this fixed-support mismatch through importance-correctable samples from the stale rollout policy, but suffers from high variance. This motivates multi-sample MC, which caches and teacher-scores multiple stale-policy samples at each decoding step, preserving MC correctability while reducing one-sample variance.

Finally, we instantiate these findings in **AsyncOPD**, a fully asynchronous OPD pipeline that overlaps student rollout, teacher scoring, and learner updates. On Qwen3-Base models, AsyncOPD improves training throughput by $1.6\times$ to $3.8\times$ over strict synchronous training while maintaining comparable accuracy. Our contributions are:

- We provide the first systematic study of staleness in asynchronous OPD through the lens of an OPD-specific teacher-cache constraint.
- We show that KL direction changes the stale-data problem: forward KL is comparatively robust to stale rollouts, whereas reverse KL is vulnerable because it is student-weighted (Section 4).
- We identify that the most effective reverse-KL policy-gradient surrogate uses the advantage recomputed at learner time without clipping, and that advanced asynchronous RL surrogates do not improve over this choice (Section 5).
- We show that stale student top-k supports are support-mismatched, while one-sample MC remains correctable but high-variance; this motivates multi-sample MC (Section 6).
- We present and open-source **AsyncOPD**, a fully asynchronous OPD training pipeline, and demonstrate improved training efficiency while maintaining OPD quality (Section 7).

2 Related Works

On-Policy Distillation On-policy distillation (OPD) trains a student on its own rollouts while using a teacher to provide dense token-level feedback on the visited prefixes [12, 20]. GKD [1] introduced a token-level KL formulation, while MiniLLM [6] studied a sequence-level reverse-KL variant. Li et al. [11] study token-level OPD training dynamics and recipes for unstable configurations. TIP [22] characterizes per-token importance through student entropy and teacher-student divergence. G-OPD [25] interprets token-level OPD as dense KL-constrained RL and extends it with reward scaling. These works clarify OPD as an effective post-training objective, but assume rollouts, teacher scoring, and learner updates stay synchronized.

Asynchronous RL In synchronous RL pipelines, training often waits for the longest rollout in a batch to finish, leaving learner resources idle. Asynchronous RL improves hardware utilization by decoupling rollout generation from learner updates. Async RLHF [14] overlaps generation and learning so that new samples are produced while the learner trains on earlier ones. StreamRL [34] further disaggregates the RLHF pipeline into streaming stages. AReaL [4] fully decouples rollout workers from training workers for continuous asynchronous execution. Laminar [18] uses fine-grained weight synchronization for trajectory-level asynchrony. However, asynchronous RL must learn from stale-policy data. Decoupled PPO [4] stabilizes asynchronous RL training by separating the behavior policy for stale rollouts from the proximal policy that anchors PPO [16] updates. M2PO [33] stabilizes stale updates with second-moment importance-weight constraints, and A-3PO [10] reduces decoupled PPO overhead through staleness-aware interpolation.

Asynchronous OPD VeRL [19] implements step-off OPD schedulers that overlap student rollout, teacher scoring, and learner update by fixing rollout lag to one or two learner steps. These schedulers establish the practical feasibility of asynchronous OPD, but leave open how OPD estimators behave under stale teacher-scored caches. KDFlow [29] improves systems efficiency for LLM distillation by decoupling teacher inference from learner training and transmitting teacher hidden states, but targets synchronous OPD and leaves asynchronous execution as future work. We study this missing asynchronous OPD regime directly and build AsyncOPD from the resulting estimator choices.

3 Preliminaries: On-Policy Distillation

OPD setup At each decoding timestep, we view the visited prefix s as the local state and the next token a as the action. Let $q(a \mid s)$ denote the teacher policy and $p_\theta(a \mid s)$ denote the current student policy. Following prior work on token-level OPD [11, 12, 25], we apply local losses to generated output tokens and analyze the resulting objectives at a fixed prefix state s. OPD can be defined with different divergences; forward and reverse KL are two standard choices [1].

Forward-KL OPD At a fixed prefix s, forward-KL OPD is teacher-weighted:

$$D_F(\theta; s) = \mathrm{KL}(q(\cdot \mid s) \,\|\, p_\theta(\cdot \mid s)) = \textstyle\sum_{a \in \mathcal{V}} q(a \mid s) \left(\log q(a \mid s) - \log p_\theta(a \mid s)\right). \tag{1}$$

At a fixed prefix s, the gradient is $\nabla_\theta D_F(\theta; s) = -\sum_{a \in \mathcal{V}} q(a \mid s) \nabla_\theta \log p_\theta(a \mid s)$.

Reverse-KL OPD At the same prefix, reverse-KL OPD is student-weighted:

$$D_R(\theta; s) = \mathrm{KL}(p_\theta(\cdot \mid s) \,\|\, q(\cdot \mid s)) = -\textstyle\sum_{a \in \mathcal{V}} p_\theta(a \mid s) \left(\log q(a \mid s) - \log p_\theta(a \mid s)\right). \tag{2}$$

Differentiating and using $\mathbb{E}_{a \sim p_\theta(\cdot \mid s)}[\nabla_\theta \log p_\theta(a \mid s)] = \nabla_\theta \sum_a p_\theta(a \mid s) = 0$ gives

$$\begin{aligned} \nabla_\theta D_R(\theta; s) &= \textstyle\sum_{a \in \mathcal{V}} p_\theta(a \mid s) \left(\log p_\theta(a \mid s) - \log q(a \mid s) + 1\right) \nabla_\theta \log p_\theta(a \mid s) \\ &= -\textstyle\sum_{a \in \mathcal{V}} p_\theta(a \mid s) \left(\log q(a \mid s) - \log p_\theta(a \mid s)\right) \nabla_\theta \log p_\theta(a \mid s). \end{aligned} \tag{3}$$

Viewing Eq. (3) as a policy-gradient estimator and $A = \log q(a \mid s) - \log p(a \mid s)$ as the advantage term connects reverse-KL OPD to standard RL training machinery, and practical implementations typically use PPO-style surrogates. Given behavior-policy samples $a \sim p_{\mathrm{beh}}$, define $\rho_\theta(a, s) = p_\theta(a \mid s)/p_{\mathrm{beh}}(a \mid s)$ and $\bar{\rho}_\theta(a, s) = \mathrm{clip}(\rho_\theta(a, s), 1 - \epsilon, 1 + \epsilon)$. The PPO-style local surrogate uses these ratios with a frozen behavior-time signal $A_{\mathrm{beh}}(a, s)$, where $\mathrm{sg}(\cdot)$ denotes stop-gradient:

$$L_{\mathrm{PPO}}(\theta; A_{\mathrm{beh}}) = -\mathbb{E}_{a \sim p_{\mathrm{beh}}} \left[\min\left(\rho_\theta \,\mathrm{sg}(A_{\mathrm{beh}}), \bar{\rho}_\theta \,\mathrm{sg}(A_{\mathrm{beh}})\right)\right]. \tag{4}$$

Sparse and sampled implementations The dense objectives above are full-vocabulary references. Practical OPD instead evaluates local KL losses on finite supports or sampled actions [11, 12], trading computation against support coverage and estimator variance. **Sparse top-k** implementations choose a support $S(s)$ and evaluate the corresponding restricted KL after renormalizing teacher and student distributions on $S(s)$. **Monte Carlo (MC)** implementations draw actions from a proposal distribution and estimate the corresponding local gradient; for reverse KL, this yields the student-sampled policy-gradient estimator. Details are in Section A.

4 Forward- and Reverse-KL OPD Under Staleness

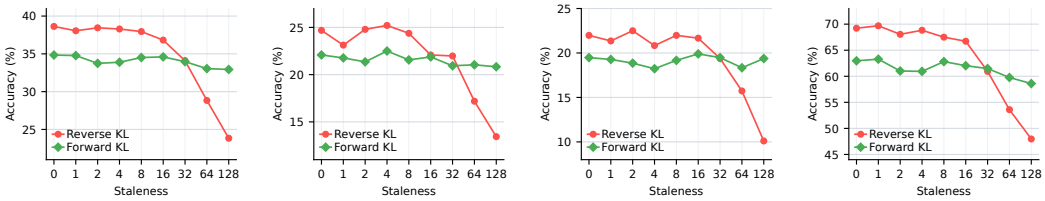
Asynchronous OPD has both prefix-level and action-level staleness. Once a rollout is generated, its visited prefixes are fixed, so an action-level estimator cannot change which states the learner sees. We therefore focus on the action-level staleness that estimator design can directly address.

4.1 Asynchronous OPD Setup

Asynchronous OPD is a cached-data pipeline: rollout first selects prefixes and actions, teacher scoring then annotates those actions, and the learner updates the student later. Unlike synchronous OPD, these stages are separated in time, so the visited prefixes, action cache, teacher scores, and update policy may be tied to different student versions. Figure 1 summarizes this cached-teacher setting and the estimator contrasts induced by the three-stage cache pipeline.

Teacher-cache constraint Full-vocabulary teacher logits allow dense KL computation, but caching and transferring them is prohibitively expensive, especially in an asynchronous pipeline. We therefore focus on sparse top-k supports and MC samples as the sparse and sampled cases.

Stage 1: Student rollout A rollout actor samples trajectories from a stale student p_{old}, which fixes the visited prefixes s. At each prefix, it stores cached actions $C_{\mathrm{old}}(s)$ together with their rollout-time log probabilities under p_{old}, such as a sampled token or a top-k support.



(a) Average (b) AIME24 (c) AIME25 (d) AMC

Figure 2: Accuracy comparison under staleness for forward- and reverse-KL OPD. Reverse KL starts higher at zero staleness but degrades faster as staleness grows; forward KL is flatter across the sweep.

Stage 2: Teacher scoring Let $C_{\text{score}}(s)$ denote the teacher-scored cache; it may come from the rollout cache $C_{\text{old}}(s)$ or be selected by the teacher at scoring time. Once teacher scoring is complete, teacher logits are available only on $C_{\text{score}}(s)$.

Stage 3: Student update By learner update time, the student has moved to the current policy p_θ. The learner can recompute $\log p_\theta(a \mid s)$ for $a \in C_{\text{score}}(s)$, but the current local OPD objective may place mass on actions outside this teacher-scored cache, such as the current student top-k support or current student-sampled actions. Thus the learner can update current student probabilities on cached actions, but cannot recover teacher signals for missing actions without additional teacher access.

Experimental setup Unless otherwise stated, we train a Qwen3-4B-Base student using a Qwen3-30B-A3B-Instruct-2507 teacher [24]. The training data is DeepMath [8], filtered to 57,630 math problems with difficulty level at least 6, and we report final-checkpoint Avg@32 accuracy on AIME24 [30], AIME25 [31], and AMC [13]. Experimental details are provided in Section B; dataset and metric details are in Section C.

4.2 Forward KL vs. Reverse KL Under Staleness

The KL direction fixes the action weighting: forward KL weights actions by the teacher q, whereas reverse KL weights them by the student p_θ. With cached teacher scores, this weighting difference becomes a support-ownership difference (Fig. 1). Under a scored-cache restriction, this makes forward KL less exposed to stale student action choices: it does not need to convert stale student-sampled actions into a current-student expectation. Reverse KL instead depends on student-weighted action terms, so the same asynchronous cache creates a different action-level staleness problem.

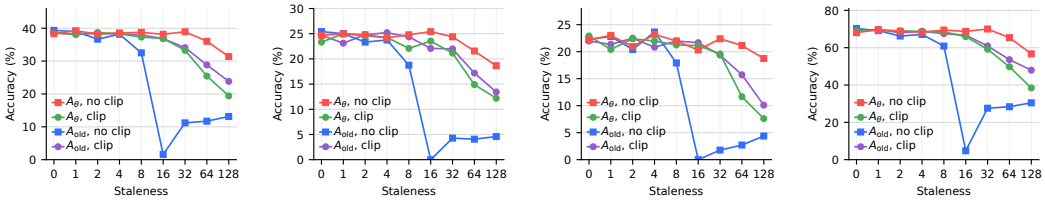
Experimental results Figure 2 compares representative practical OPD implementations from prior work: sparse top-k forward KL [19] and PPO-style reverse-KL surrogates [12, 25]. Reverse KL starts higher at zero staleness, but as staleness increases it drops faster and is eventually overtaken by forward KL. We therefore focus the rest of the staleness analysis on how to make reverse-KL OPD robust under larger rollout staleness.

Finding 1. Forward KL is teacher-weighted and robust to rollout staleness, whereas reverse KL is student-weighted and vulnerable to rollout staleness.

Two axes of reverse-KL staleness The cache analysis above suggests a possible mechanism for this gap: because reverse KL is weighted by the current student, stale teacher-scored caches may fail to cover actions needed by the current reverse-KL objective. In addition, reverse-KL policy-gradient updates can be instantiated with multiple stale-data surrogates, including PPO-style and asynchronous-RL variants. We therefore split the reverse-KL analysis into a **policy-gradient surrogate axis**, studied in Section 5, and a **cached-support axis**, studied in Section 6.

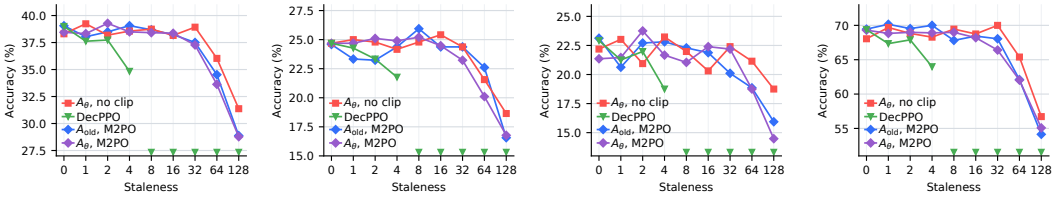
5 Reverse-KL: Policy-Gradient Surrogates Under Staleness

Reverse-KL OPD admits several policy-gradient surrogate choices under stale rollouts. This section compares which choices remain effective under staleness.



(a) Average (b) AIME24 (c) AIME25 (d) AMC

Figure 3: Accuracy comparison under staleness for the advantage-and-clipping ablation. Recomputing A_θ at learner time and avoiding clipping gives the most stable performance across the sweep, while clipping mainly helps the frozen A_{old} baseline.



(a) Average (b) AIME24 (c) AIME25 (d) AMC

Figure 4: Accuracy comparison under staleness for advanced asynchronous RL surrogates. Decoupled PPO [4] and M2PO [33] do not consistently improve over the simpler OPD/IS surrogate that recomputes A_θ without clipping; Decoupled PPO is clipped for readability because of low accuracy.

5.1 Policy-Gradient Surrogate Choices

PPO-style objective In the PPO-style surrogate in Eq. (4), the advantage is computed under the behavior policy and then held fixed during the learner update. In stale reverse-KL OPD, the behavior policy is the rollout student, so a mechanical PPO-style adaptation sets $p_{\text{beh}} = p_{\text{old}}$ and uses the rollout-time reverse-KL advantage $A_{\text{old}}(a, s) = \log q(a \mid s) - \log p_{\text{old}}(a \mid s)$ as A_{beh}, together with the clipped old-to-current ratio. The unclipped variant simply drops the clipped term.

Exact importance-sampling identity In contrast, rewriting the reverse-KL objective (Eq. (2)) by importance sampling suggests a different surrogate choice. With the current reverse-KL advantage $A_\theta(a, s) = \log q(a \mid s) - \log p_\theta(a \mid s)$, and assuming p_{old} has support wherever p_θ does, the current reverse-KL objective admits the exact old-to-current importance-sampling (IS) identity

$$D_R(\theta; s) = -\mathbb{E}_{a\sim p_\theta}\left[A_\theta(a, s)\right] = -\mathbb{E}_{a\sim p_{\text{old}}}\left[\rho_\theta(a, s) A_\theta(a, s)\right]. \tag{5}$$

For the policy-gradient update, the advantage is used as a stop-gradient weight; the derivative of the omitted A_θ term cancels by the score-function identity, as in Eq. (3). Thus the IS view points to the opposite surrogate choice from the mechanical PPO adaptation: recompute A_θ under the current student and use the old-to-current ratio without clipping, with A_θ treated as a stop-gradient advantage.

A two-by-two surrogate ablation The PPO-style adaptation and the OPD/IS identity suggest different surrogate choices. We therefore ablate the advantage (A_{old} versus A_θ) and whether to clip the ratio, with $\text{sg}(\cdot)$ denoting stop-gradient:

$$L_{\text{old}}^{\text{clip}}(\theta) = -\mathbb{E}_{a\sim p_{\text{old}}}\left[\min\left(\rho_\theta\, \text{sg}(A_{\text{old}}), \bar{\rho}_\theta\, \text{sg}(A_{\text{old}})\right)\right], \quad L_{\text{old}}^{\text{noclip}}(\theta) = -\mathbb{E}_{a\sim p_{\text{old}}}\left[\rho_\theta\, \text{sg}(A_{\text{old}})\right], \tag{6}$$

$$L_{\theta}^{\text{clip}}(\theta) = -\mathbb{E}_{a\sim p_{\text{old}}}\left[\min\left(\rho_\theta\, \text{sg}(A_\theta), \bar{\rho}_\theta\, \text{sg}(A_\theta)\right)\right], \quad L_{\theta}^{\text{noclip}}(\theta) = -\mathbb{E}_{a\sim p_{\text{old}}}\left[\rho_\theta\, \text{sg}(A_\theta)\right]. \tag{7}$$

Here $L_{\text{old}}^{\text{clip}}$ is the PPO-style adaptation, while $L_{\theta}^{\text{noclip}}$ is the OPD/IS surrogate.

Advanced asynchronous RL surrogates Decoupled PPO [4] and M2PO [33] are asynchronous RL surrogates designed to improve robustness to stale-policy updates. We evaluate whether these previously unstudied asynchronous RL surrogates also help OPD under staleness.

5.2 Experimental Results

Fig. 3 compares the four combinations of A_{old} versus A_θ and clipping versus no clipping. The best variant is the OPD/IS choice: A_θ without clipping. The PPO-style baseline, A_{old} with clipping, remains a strong stale-surrogate baseline. Clipping helps A_{old} by limiting stale, large-ratio updates, but hurts A_θ: recomputing A_θ already reduces the high-percentile ρ_θ tail at staleness 64 (Fig. 5), so clipping removes useful signal. Likewise, Fig. 4 shows that advanced asynchronous RL surrogates such as decoupled PPO and M2PO do not outperform A_θ without clipping, which becomes our reference surrogate below.

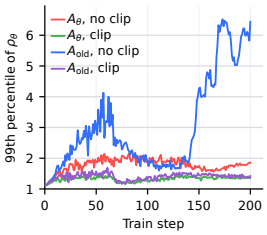


Figure 5: A_θ reduces the p99 ρ_θ tail under no clip.

Finding 2. The most effective reverse-KL correction is to recompute A_θ at learner time without clipping; asynchronous RL surrogates such as decoupled PPO and M2PO do not improve over it.

6 Reverse-KL: Cached Supports Under Staleness

Having fixed A_θ without clipping as the reference surrogate, we now ask which cached actions provide the teacher scores needed to evaluate it, and how to improve this cached-support estimator. This cached-support axis is specific to OPD because teacher scoring is local and expensive: the teacher cache determines which actions have teacher scores available to the learner.

Sparse top-k: stale-support biased Although sparse top-k is biased relative to the dense reverse-KL objective, it is a practical low-variance approximation on the current student support $S_\theta(s) = \mathrm{TopK}(p_\theta(\cdot \mid s), k)$. Under asynchronous rollout reuse, however, teacher scores are cached on the rollout-time support $S_{\text{old}}(s) = \mathrm{TopK}(p_{\text{old}}(\cdot \mid s), k)$, which may miss actions in the current support $S_\theta(s)$. Reweighting within S_{old} cannot recover these missing teacher scores, so stale sparse top-k remains a support-mismatched approximation, not an exact correction of the current top-k objective.

One-sample MC: correctable but high variance Sampled-token MC instead caches an action drawn from a behavior distribution: $a \sim p_{\text{old}}(\cdot \mid s)$ together with $\log p_{\text{old}}(a \mid s)$. When the behavior policy covers the current policy support, exact old-to-current IS gives an unbiased estimator of the current reverse-KL fixed-prefix gradient. Thus one-sample MC is action-level correctable in expectation, but the resulting IS estimator can have high variance. This proposal-sampling structure is the key contrast with stale top-k, whose actions come from a deterministic stale support.

6.1 Proposed Solution: Multi-Sample MC

Multi-sample MC: correctable with reduced variance We propose multi-sample MC (Fig. 6), which, at each decoding timestep of a student rollout, draws multiple local next-token samples from the behavior policy without rolling them out into additional trajectories. It reduces one-sample MC variance by caching these local samples and averaging their IS-corrected gradients.

Multi-sample MC is especially natural in asynchronous OPD. In RL for LLM post-training, branching a prefix into multiple actions is expensive because each branch typically requires a full continuation before the reward or advantage can be evaluated. In synchronous OPD, sparse top-k already provides a low-variance approximation and one-sample MC provides an unbiased sampled gradient estimator, so there is little motivation to cache multiple sampled actions per prefix. Under asynchronous OPD, this tradeoff changes: sparse top-k becomes the stale fixed-support approximation analyzed above, and one-sample MC remains correctable but high-variance, making multi-sample MC a natural cached-support estimator for asynchronous OPD.

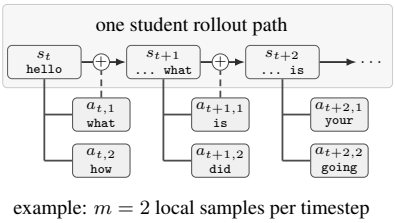
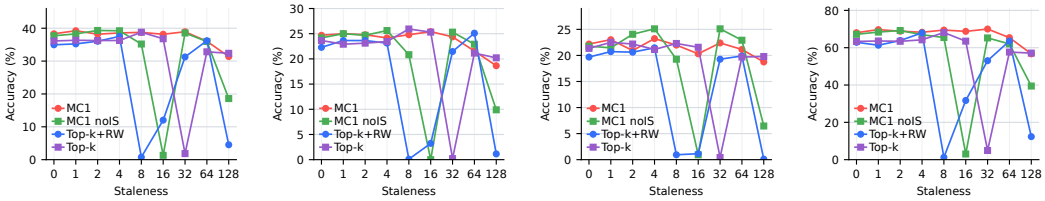


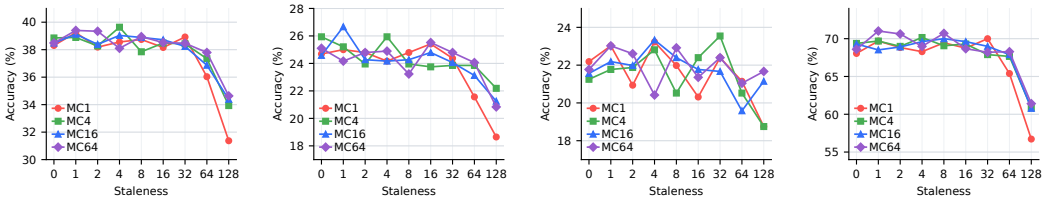
Figure 6: Multi-sample MC ($m = 2$).

Concretely, at each visited timestep t with prefix s_t, rollout samples $a_{t,1}, \ldots, a_{t,m} \sim p_{\text{old}}(\cdot \mid s_t)$ and caches their rollout log probabilities and teacher scores. For notational simplicity, write $s = s_t$ and $a_i = a_{t,i}$ below. At learner time, we recompute $A_\theta(a_i, s)$ and use the averaged unclipped old-



(a) Average (b) AIME24 (c) AIME25 (d) AMC

Figure 7: Accuracy comparison under staleness for sampled MC versus stale top-k. Top-k+RW denotes reweighting on the stale top-k support. Old-to-current IS corrects sampled MC in expectation, whereas reweighting cannot repair the missing teacher scores induced by stale top-k supports.



(a) Average (b) AIME24 (c) AIME25 (d) AMC

Figure 8: Accuracy comparison under staleness for multi-sample MC. Increasing the number of samples improves large-staleness behavior.

to-current IS surrogate $\widehat{L}_m^{\mathrm{MC}}(\theta; s) = -\frac{1}{m}\sum_{i=1}^{m} \rho_\theta(a_i, s)\,\mathrm{sg}(A_\theta(a_i, s))$. By linearity, the gradient has the same expectation as the one-sample MC estimator; averaging independent behavior-policy samples reduces the Monte Carlo variance. We measure this variance reduction in Section E.

6.2 Experimental Results

Sparse top-k vs. one-sample MC Figure 7 compares one-sample MC and sparse top-k, with and without old-to-current reweighting. For one-sample MC, IS substantially improves robustness as staleness increases. For sparse top-k, the same reweighting does not improve performance, since it cannot recover missing current-support actions. As a result, one-sample MC with IS is the strongest of the four methods, consistent with the support-correctability analysis above. We include an additional ablation disentangling MC sample count from IS in Section F.

One-sample MC vs. multi-sample MC Fig. 8 shows that multi-sample MC improves one-sample MC at large staleness: $m = 4$ already gives a clear jump, while $m \in \{4, 16, 64\}$ performs similarly.

Finding 3. One-sample MC is more effective than stale sparse top-k; multi-sample MC further improves this estimator by reducing one-sample variance while preserving MC correctability.

7 AsyncOPD: Fully Asynchronous OPD

AsyncOPD is our fully asynchronous OPD system. Following AReaL [4], it overlaps rollout, teacher scoring, and learner updates.

Scheduler The step-off scheduler family was originally implemented in VeRL [19]: a k-step-off run fixes rollout lag to k learner updates, but still waits for complete rollout batches. AsyncOPD streams examples instead: workers pause only for weight sync, preserve in-flight prefixes, teacher scoring consumes completed items, and the learner updates once a scored batch is ready (Fig. 9).

Experimental setup The main comparison uses Qwen3-{1.7B,4B,8B}-Base students with the Qwen3-30B-A3B-Instruct-2507 teacher. All runs use the same reverse-KL estimator: current-policy A_θ, no clipping, old-to-current IS, and either MC64 or MC1. We compare strict sync, two-step-off, and AsyncOPD for 100 training iterations on the same 8-GPU node; all AsyncOPD runs use $\tau = 4$. Section G gives GPU allocation, queue-depth, and scheduler details.

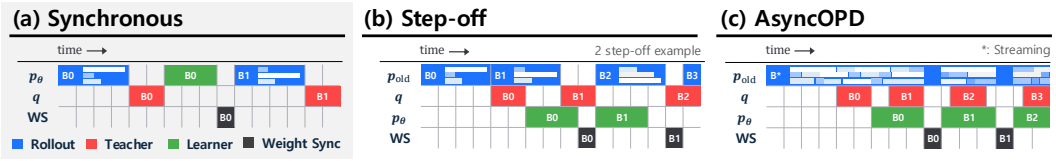


Figure 9: Scheduler comparison for synchronous OPD, step-off scheduling, and AsyncOPD. Synchronous OPD is barriered; step-off scheduling [19] overlaps stages but keeps gated rollout batches, while AsyncOPD streams rollout data to reduce long-tail waiting.

Table 1: AsyncOPD scheduler results for Qwen3-Base models. Train tok/s is training throughput; parentheses show speedup over the matched strict-sync baseline. Overlap is concurrent OPD-stage activity. Avg@32 is final AIME24. AsyncOPD achieves the highest throughput and overlap in all matched settings while maintaining comparable final accuracy.

Student	Scheduler	MC64			MC1		
		Train tok/s ($\times$ sync)	Overlap	Avg@32	Train tok/s ($\times$ sync)	Overlap	Avg@32
Qwen3-1.7B-Base	Strict sync	8.7k (1.00$\times$)	0.81	8.85	8.6k (1.00$\times$)	0.82	**8.65**
	Two-step-off	14.2k (1.64$\times$)	1.49	8.23	18.5k (2.15$\times$)	1.65	8.12
	AsyncOPD (ours)	**23.4k (2.70$\times$)**	**2.13**	**9.38**	**28.1k (3.28$\times$)**	**2.31**	8.44
Qwen3-4B-Base	Strict sync	9.5k (1.00$\times$)	0.81	**25.00**	8.1k (1.00$\times$)	0.82	23.33
	Two-step-off	12.4k (1.30$\times$)	1.64	23.85	12.9k (1.60$\times$)	1.65	**23.54**
	AsyncOPD (ours)	**15.8k (1.66$\times$)**	**2.19**	**25.00**	**16.4k (2.03$\times$)**	**2.27**	23.44
Qwen3-8B-Base	Strict sync	7.5k (1.00$\times$)	0.81	26.56	6.5k (1.00$\times$)	0.84	28.44
	Two-step-off	11.6k (1.55$\times$)	1.76	26.56	9.6k (1.47$\times$)	1.63	**28.85**
	AsyncOPD (ours)	**14.5k (1.94$\times$)**	**2.24**	**28.65**	**10.6k (1.63$\times$)**	**2.17**	27.50

Experimental Results Table 1 reports training throughput, pipeline overlap (average concurrent OPD-stage activity), and final AIME24 Avg@32 for the Qwen3-Base students. AsyncOPD achieves the highest throughput and overlap in every matched comparison. In MC64, it reaches up to $2.7\times$ the strict-sync throughput while achieving the best or tied-best final accuracy. MC1 shows the same trend: AsyncOPD delivers the highest throughput (up to $3.3\times$ strict-sync) and overlap for every student, with competitive final accuracy. Train-time accuracy curves are reported in Section G.

8 Conclusion

We present the first systematic study of staleness in asynchronous on-policy distillation (OPD). Our results show that KL direction shapes the stale-data problem: forward KL remains robust to stale rollouts, whereas reverse KL is more vulnerable because it is student-weighted. In reverse-KL OPD, the most effective policy-gradient surrogate uses the current advantage recomputed at learner time without clipping; advanced asynchronous RL surrogates do not improve over this choice. We also find that stale student top-k supports are support-mismatched, whereas one-sample Monte Carlo (MC) remains correctable but high-variance. This contrast motivates multi-sample MC, which preserves MC correctability while reducing one-sample variance. Finally, we present and open-source **AsyncOPD**, a fully asynchronous OPD training pipeline built from these estimator choices, improving training efficiency while maintaining OPD quality.

Limitations and Future Work We study sparse and Monte Carlo OPD estimators, not dense full-vocabulary KL in the asynchronous setting. Although dense KL avoids cached-support mismatch, it is difficult to implement efficiently when rollout, teacher scoring, and learner updates are decoupled. KDFlow [29] suggests one path by transmitting teacher hidden states and recomputing student logits, but only for synchronous OPD. Extending this approach to asynchronous OPD while handling stale rollouts and preserving throughput is an important future direction. Our experiments are also limited to a single 8-GPU node by available resources, not by the pipeline itself; scaling to larger multi-node clusters remains future work.

Acknowledgments and Disclosure of Funding

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 04-26-03-0081, Energy-Efficient Training–Inference System Optimization for Reinforcement Learning-Based Post-Training). This work was also supported by the "Advanced GPU Utilization Support Program" funded by the Government of the Republic of Korea (Ministry of Science and ICT).

References

[1] R. Agarwal, N. Vieillard, Y. Zhou, P. Stanczyk, S. R. Garea, M. Geist, and O. Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=3zKtaqxLhW`.

[2] DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

[3] F. Devvrit, L. Madaan, R. Tiwari, R. Bansal, S. S. Duvvuri, M. Zaheer, I. S. Dhillon, D. Brandfonbrener, and R. Agarwal. The art of scaling reinforcement learning compute for LLMs. In *The Fourteenth International Conference on Learning Representations*, 2026. URL `https://openreview.net/forum?id=FMjeC9Msws`.

[4] W. Fu, J. Gao, X. Shen, C. Zhu, Z. Mei, C. He, S. Xu, G. Wei, J. Mei, W. JIASHU, T. Yang, B. Yuan, and Y. Wu. AREAL: A large-scale asynchronous reinforcement learning system for language reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL `https://openreview.net/forum?id=X9diEuva9R`.

[5] W. Gao, Y. Zhao, D. An, T. Wu, L. Cao, S. Xiong, J. Huang, W. Wang, S. Yang, W. Su, et al. Rollpacker: Mitigating long-tail rollouts for fast, synchronous rl post-training. *arXiv preprint arXiv:2509.21009*, 2025.

[6] Y. Gu, L. Dong, F. Wei, and M. Huang. MiniLLM: Knowledge distillation of large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=5h0qf7IBZZ`.

[7] D. Guo, D. Yang, H. Zhang, J. Song, P. Wang, Q. Zhu, R. Xu, R. Zhang, S. Ma, X. Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[8] Z. He, T. Liang, J. Xu, Q. Liu, X. Chen, Y. Wang, L. Song, D. Yu, Z. Liang, W. Wang, et al. Deepmath-103k: A large-scale, challenging, decontaminated, and verifiable mathematical dataset for advancing reasoning. *arXiv preprint arXiv:2504.11456*, 2025.

[9] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[10] X. Li, S. Wu, and Z. Shen. A-3po: Accelerating asynchronous llm training with staleness-aware proximal policy approximation. *arXiv preprint arXiv:2512.06547*, 2025.

[11] Y. Li, Y. Zuo, B. He, J. Zhang, C. Xiao, C. Qian, T. Yu, H.-a. Gao, W. Yang, Z. Liu, et al. Rethinking on-policy distillation of large language models: Phenomenology, mechanism, and recipe. *arXiv preprint arXiv:2604.13016*, 2026.

[12] K. Lu and T. M. Lab. On-policy distillation. *Thinking Machines Lab: Connectionism*, 2025. doi: 10.64434/tml.20251026. https://thinkingmachines.ai/blog/on-policy-distillation.

[13] Mathematical Association of America. American Mathematics Competitions – AMC. `https://maa.org/`, 2023. Accessed 2026-04-03.

[14] M. Noukhovitch, S. Huang, S. Xhonneux, A. Hosseini, R. Agarwal, and A. Courville. Faster, more efficient RLHF through off-policy asynchronous learning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=FhTAG591Ve`.

[15] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

[16] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[17] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[18] G. Sheng, Y. Tong, B. Wan, W. Zhang, C. Jia, X. Wu, Y. Wu, X. Li, C. Zhang, Y. Peng, et al. Laminar: A scalable asynchronous rl post-training framework. *arXiv preprint arXiv:2510.12633*, 2025.

[19] G. Sheng, C. Zhang, Z. Ye, X. Wu, W. Zhang, R. Zhang, Y. Peng, H. Lin, and C. Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297, 2025.

[20] M. Song and M. Zheng. A survey of on-policy distillation for large language models. *arXiv preprint arXiv:2604.00626*, 2026.

[21] B. Xiao, B. Xia, B. Yang, B. Gao, B. Shen, C. Zhang, C. He, C. Lou, F. Luo, G. Wang, et al. Mimo-v2-flash technical report. *arXiv preprint arXiv:2601.02780*, 2026.

[22] Y. Xu, H. Sang, Z. Zhou, R. He, Z. Wang, and A. Geramifard. Tip: Token importance in on-policy distillation. *arXiv preprint arXiv:2604.14084*, 2026.

[23] R. Yan, Y. Jiang, T. Wu, J. Gao, Z. Mei, W. Fu, H. Mai, W. Wang, Y. Wu, and B. Yuan. Areal-hex: Accommodating asynchronous rl training over heterogeneous gpus. *arXiv preprint arXiv:2511.00796*, 2025.

[24] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[25] W. Yang, W. Liu, R. Xie, K. Yang, S. Yang, and Y. Lin. Learning beyond teacher: Generalized on-policy distillation with reward extrapolation. *arXiv preprint arXiv:2602.12125*, 2026.

[26] Q. Yu, Z. Zhang, R. Zhu, Y. Yuan, X. Zuo, Y. Yue, W. Dai, T. Fan, G. Liu, L. Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

[27] A. Zeng, X. Lv, Z. Hou, Z. Du, Q. Zheng, B. Chen, D. Yin, C. Ge, C. Huang, C. Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

[28] K. Zhang, Y. Zuo, B. He, Y. Sun, R. Liu, C. Jiang, Y. Fan, K. Tian, G. Jia, P. Li, et al. A survey of reinforcement learning for large reasoning models. *arXiv preprint arXiv:2509.08827*, 2025.

[29] S. Zhang, X. Zhang, T. Zhang, B. Hu, Y. Chen, and J. Xu. Kdflow: A user-friendly and efficient knowledge distillation framework for large language models. *arXiv preprint arXiv:2603.01875*, 2026.

[30] Y. Zhang and T. Math-AI. AIME 2024. `https://huggingface.co/datasets/Maxwell-Jia/AIME_2024`, 2024. Hugging Face dataset; accessed 2026-04-03.

[31] Y. Zhang and T. Math-AI. AIME 2025. `https://huggingface.co/datasets/yentinglin/aime_2025`, 2025. Hugging Face dataset; accessed 2026-04-03.

[32] S. Zhao, Z. Xie, M. Liu, J. Huang, G. Pang, F. Chen, and A. Grover. Self-distilled reasoner: On-policy self-distillation for large language models. *arXiv preprint arXiv:2601.18734*, 2026.

[33] H. Zheng, J. Zhao, and B. Chen. Prosperity before collapse: How far can off-policy RL reach with stale data on LLMs? In *The Fourteenth International Conference on Learning Representations*, 2026. URL `https://openreview.net/forum?id=IIgl5MWelz`.

[34] Y. Zhong, Z. Zhang, X. Song, H. Hu, C. Jin, B. Wu, N. Chen, Y. Chen, Y. Zhou, C. Wan, et al. Streamrl: Scalable, heterogeneous, and elastic rl for llms with disaggregated stream generation. *arXiv preprint arXiv:2504.15930*, 2025.

A Sparse and Monte Carlo Reverse-KL Implementations

A.1 Sparse Top-k Reverse-KL OPD

The dense reverse-KL objective in Eq. (2) sums over the full vocabulary. A sparse top-k implementation instead evaluates reverse KL on a finite student support

$$S_\theta(s) = \mathrm{TopK}(p_\theta(\cdot \mid s), k)\,. \tag{8}$$

For any support S, define the restricted normalizers $Z_p^S(s) = \sum_{u \in S} p_\theta(u \mid s)$ and $Z_q^S(s) = \sum_{u \in S} q(u \mid s)$, and the renormalized distributions

$$\tilde{p}_\theta^S(a \mid s) = \frac{p_\theta(a \mid s)\mathbf{1}[a \in S]}{Z_p^S(s)}, \qquad \tilde{q}^S(a \mid s) = \frac{q(a \mid s)\mathbf{1}[a \in S]}{Z_q^S(s)}. \tag{9}$$

The sparse reverse-KL objective is

$$\begin{aligned} D_R^S(\theta; s) &= \mathrm{KL}\big(\tilde{p}_\theta^S(\cdot \mid s) \,\|\, \tilde{q}^S(\cdot \mid s)\big) \\ &= -\sum\nolimits_{a \in S} \tilde{p}_\theta^S(a \mid s)\,\big(\log \tilde{q}^S(a \mid s) - \log \tilde{p}_\theta^S(a \mid s)\big)\,. \end{aligned} \tag{10}$$

In practice, when $S = S_\theta(s)$, we treat the selected top-k support as fixed during the local update.

A.2 Monte Carlo Reverse-KL OPD

Let $A_\theta(a, s) = \log q(a \mid s) - \log p_\theta(a \mid s)$. From Eq. (3), the dense reverse-KL gradient can be written as

$$\nabla_\theta D_R(\theta; s) = -\mathbb{E}_{a \sim p_\theta(\cdot \mid s)}\left[A_\theta(a, s)\nabla_\theta \log p_\theta(a \mid s)\right]. \tag{11}$$

A one-sample current-policy Monte Carlo estimator is therefore

$$\widehat{g}_{\mathrm{MC}}(s, a) = -A_\theta(a, s)\nabla_\theta \log p_\theta(a \mid s), \qquad a \sim p_\theta(\cdot \mid s). \tag{12}$$

With m independent samples $a_i \sim p_\theta(\cdot \mid s)$, the corresponding multi-sample estimator averages the same local term:

$$\widehat{g}_m(s) = -\frac{1}{m}\sum\nolimits_{i=1}^{m} A_\theta(a_i, s)\nabla_\theta \log p_\theta(a_i \mid s). \tag{13}$$

B Experimental Details

This section details the experimental setup. Unless explicitly stated otherwise, experiments use the common setup in Table 2 and report final-checkpoint Avg@32 accuracy. Our implementation uses vLLM [9] for rollout generation and teacher scoring, PyTorch FSDP [15] for learner training, and runs each experiment on a single 8×B200 node. Individual experiments take roughly 1–12 hours, depending on the setting. Asset URLs, license names, and versions are summarized in Table 4.

Constructing the staleness axis. The main text uses staleness as an experimental control over how old the cached rollout data is when the learner updates on it. In all staleness plots and tables in Sections 4 to 6, staleness is measured in train-batch steps. One train-batch step is one logical rollout batch consumed by the learner for a training iteration. The sweep value k is therefore the target number of train-batch steps by which the consumed cache trails the current learner; equivalently, it is the target cache depth in logical rollout batches. A value $k = 0$ is synchronous: rollout, teacher scoring, training, and weight synchronization occur in strict sequence. For $k > 0$, the run first generates exactly k rollout batches with the initial student snapshot before the first learner update. Training then consumes the oldest available generated batch; after each learner update and weight synchronization, a new rollout batch is generated with the latest student snapshot whenever needed to restore the target cache depth.

This protocol is the operational source of the prefix- and action-level staleness discussed in the main text: the consumed prefixes and cached actions come from an older rollout student, while the update is applied to the current student. For a consumed rollout batch, let t_{roll} be the train-batch index of the

Table 2: Experimental settings.

Setting	Value
Student	Qwen3-4B-Base for main staleness/estimator experiments; AsyncOPD experiments also use Qwen3-{1.7B,4B,8B}-Base and thinking-disabled Qwen3-{1.7B,4B,8B} [24].
Teacher	Qwen3-30B-A3B-Instruct-2507 [24]
Training data	DeepMath [8] filtered to 57,630 math problems with difficulty level at least 6
Prompt / response lengths	2,048 prompt tokens / 16,384 response tokens
Training horizon	200 training iterations; in the common setup each iteration contains four mini-batch optimizer updates
Optimization	batch size 256, mini-batch size 64
Optimizer	AdamW-style optimizer with learning rate 3×10^{-6}, constant schedule, weight decay 0.01
PPO clipping	$\epsilon = 0.2$ for clipped PPO-style surrogates
M2PO budget	0.01
Rollout generation	temperature 1.0, top-$p = 1.0$, no top-k truncation
Evaluation generation	32 samples per problem, temperature 1.0
Implementation	vLLM rollout and teacher scoring; PyTorch FSDP learner training
Hardware	Single 8×B200 node
GPU allocation	One teacher GPU; strict sync time-shares seven GPUs between rollout and training; stale-cache sweeps and AsyncOPD use concurrent rollout and trainer GPU pools

Table 3: Evaluation datasets.

Dataset	Problems
AIME 2024	30
AIME 2025	30
AMC 2023	40

student snapshot used for generation and t_{train} be the train-batch index at learner time. The staleness used in the plots is

$$\Delta_{\text{batch}} = t_{\text{train}} - t_{\text{roll}}.$$

All examples in a logical batch share the same rollout snapshot and therefore share the same Δ_{batch}. Under the controlled cache protocol, Δ_{batch} ramps as $0, 1, 2, \ldots$ while the initial cache is drained and then plateaus at k. Thus a 64-batch target cache depth is plotted as staleness 64. A train-batch step can contain multiple mini-batch optimizer updates; in the common setup, $B = 256$ and $B_{\text{mini}} = 64$, so each train-batch step contains $M = B/B_{\text{mini}} = 4$ optimizer updates. This conversion is useful for implementation accounting, but it is not the staleness axis used in the plots.

We use k as the x-axis because it is the controlled train-batch staleness intervention shared across methods. The sweep covers $k \in \{0, 1, 2, 4, 8, 16, 32, 64, 128\}$ across the forward-KL, reverse-KL / PPO-style, M2PO / DecPPO, top-k, and Monte-Carlo support-size variants; apart from the estimator choice and k, these runs share the common model, data, batch-size, generation, and evaluation settings in Table 2.

C Datasets and Metrics

Training data. We filter the DeepMath dataset [8] to retain 57,630 math problems with difficulty level greater than or equal to 6, and use this filtered subset as the training data.

Evaluation datasets. Table 3 lists the evaluation datasets used: AIME 2024 [30], AIME 2025 [31], and AMC 2023 [13]. AIME24 is evaluated every 20 steps. The remaining datasets are only evaluated for the final checkpoint.

Table 4: Existing assets used in this work, with source URLs, license names, and versions.

Category	Asset	URL	License name	Version
Dataset	DeepMath	https://huggingface.co/datasets/zwhe99/DeepMath-103K	MIT	2025
Dataset	AIME 2024	https://huggingface.co/datasets/Maxwell-Jia/AIME_2024	MIT	2024
Dataset	AIME 2025	https://huggingface.co/datasets/yentinglin/aime_2025	Not specified	2025
Dataset	AMC 2023 / American Mathematics Competitions	https://huggingface.co/datasets/math-ai/amc23	Not specified	2023
Model	Qwen3-4B-Base	https://huggingface.co/Qwen/Qwen3-4B-Base	Apache-2.0	Not specified
Model	Qwen3-1.7B-Base	https://huggingface.co/Qwen/Qwen3-1.7B-Base	Apache-2.0	Not specified
Model	Qwen3-8B-Base	https://huggingface.co/Qwen/Qwen3-8B-Base	Apache-2.0	Not specified
Model	Qwen3-4B	https://huggingface.co/Qwen/Qwen3-4B	Apache-2.0	Not specified
Model	Qwen3-1.7B	https://huggingface.co/Qwen/Qwen3-1.7B	Apache-2.0	Not specified
Model	Qwen3-8B	https://huggingface.co/Qwen/Qwen3-8B	Apache-2.0	Not specified
Model	Qwen3-30B-A3B-Instruct-2507	https://huggingface.co/Qwen/Qwen3-30B-A3B-Instruct-2507	Apache-2.0	Qwen3-2507
Software	vLLM	https://github.com/vllm-project/vllm	Apache-2.0	0.16.0
Software	PyTorch / PyTorch FSDP	https://github.com/pytorch/pytorch	BSD-3-Clause	2.9.1

Accuracy metric. Evaluation samples 32 responses per problem. For a dataset D, the reported Avg@32 is the mean per-problem pass rate,

$$\mathrm{Avg@32}(D) = 100 \cdot \frac{1}{|D|} \sum_{i \in D} \frac{c_i}{32}, \tag{14}$$

where c_i is the number of sampled responses judged correct for problem i. Paper tables and plots use Avg@32 unless noted otherwise.

D Existing Asset Licenses

Table 4 lists the reused assets.

E Multi-Sample MC Variance at Large Staleness

We measure how multi-sample MC reduces the variance of the old-to-current IS surrogate at large staleness. At timestep t with prefix s_t, local MC actions $a_{t,1}, \ldots, a_{t,m}$ are sampled iid with replacement from $p_{\mathrm{old}}(\cdot \mid s_t)$ (duplicates are allowed), and the learner evaluates

$$\widehat{L}_m^{\mathrm{MC}}(\theta; s_t) = -\frac{1}{m} \sum_{i=1}^{m} \rho_\theta(a_{t,i}, s_t)\, \mathrm{sg}(A_\theta(a_{t,i}, s_t)), \qquad \rho_\theta(a, s) = \frac{p_\theta(a \mid s)}{p_{\mathrm{old}}(a \mid s)}. \tag{15}$$

Using the Qwen3-4B-Base staleness-128 runs, we report R_m^{local} in the fixed-prefix column and R_m^{seq} in the sequence-level column of Table 5. Both ratios are normalized to the corresponding $m = 1$ estimator within the same old-to-current pair:

$$R_m^{\mathrm{local}} = \frac{\mathbb{E}_{s_t}\left[\mathrm{Var}_{a_{t,1},\ldots,a_{t,m} \sim p_{\mathrm{old}}(\cdot \mid s_t)}\left(\widehat{L}_m^{\mathrm{MC}}(\theta; s_t) \mid s_t\right)\right]}{\mathbb{E}_{s_t}\left[\mathrm{Var}_{a_t \sim p_{\mathrm{old}}(\cdot \mid s_t)}\left(\widehat{L}_1^{\mathrm{MC}}(\theta; s_t) \mid s_t\right)\right]}, \tag{16}$$

$$R_m^{\mathrm{seq}} = \frac{\mathrm{Var}\left[\frac{1}{T}\sum_{t=1}^{T} \widehat{L}_m^{\mathrm{MC}}(\theta; s_t)\right]}{\mathrm{Var}\left[\frac{1}{T}\sum_{t=1}^{T} \widehat{L}_1^{\mathrm{MC}}(\theta; s_t)\right]}. \tag{17}$$

For R_m^{seq}, the generated prefix path $s_{1:T}$ is fixed when computing the variance; the MC samples are local scorer queries at each fixed prefix, not separate rollout branches.

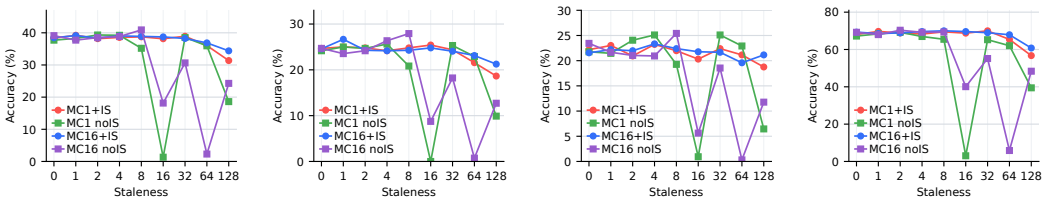
Table 5: MC variance ratios at large staleness. The fixed-prefix column isolates local next-token action-sampling variance; the sequence-level column averages the same estimator over generated timesteps before computing variance.

Diagnostic m	Fixed-prefix ratio R_m^{local}	Sequence-level ratio R_m^{seq}
1	1.000	1.000
4	0.255	0.338
16	0.0588	0.152
64	0.0149	0.112

Table 5 shows that larger m consistently reduces variance. Fixed-prefix ratios closely follow the $1/m$ reference ($m = 64$ leaves 1.49% of the one-sample local variance, versus $1/64 = 1.56\%$). After timestep aggregation, $m = 64$ still leaves only 11.2% of the one-sample sequence-level variance, showing that multi-sample MC reduces variance in practice even though sequence-level averaging makes the reduction less extreme than the local next-token effect.

F Importance-Sampling Ablation

Before treating multi-sampling as an improvement, we separate it from IS. Increasing m changes the Monte Carlo variance of the estimator. It does not define the target distribution. Old-to-current IS is still the mechanism that turns behavior-policy samples into an estimator of the current reverse-KL local gradient. Figure 10 compares MC1 and MC16 with and without IS to test this distinction directly.



(a) Average (b) AIME24 (c) AIME25 (d) AMC

Figure 10: Accuracy comparison under staleness for MC importance-sampling ablations. Increasing the number of samples reduces Monte Carlo variance, but old-to-current IS is still needed to correct stale-policy sampling.

G AsyncOPD Scheduler Details

This appendix gives the implementation details omitted from Section 7. Our scheduler, AsyncOPD, follows the fully asynchronous systems structure of AReaL [4], but the queue contains OPD cache items rather than reward-labeled RL trajectories.

Queue interface. The pipeline has three long-running stages: rollout generation, teacher scoring, and learner training. Rollout workers sample trajectories from their latest synchronized student snapshot. For each visited prefix s, they cache the MC actions, rollout log probabilities under p_{old}, and the rollout student version. The main scheduler comparison uses MC64; the MC1 runs use the same queue interface with one cached action. The teacher scores the cached actions. The learner then recomputes $\log p_\theta(a \mid s)$ and $A_\theta(a, s)$ under the current student, and applies the unclipped old-to-current IS estimator from Section 6.

Weight synchronization and queue capacity. During AsyncOPD weight synchronization, rollout workers pause generation. In the keep-mode used for the scheduler experiments, in-flight requests are not discarded: the already sampled token prefix is kept, the student weights are updated, and

the running-request prefix cache is reset so the engine rebuilds the attention state for that prefix under the new weights before generation resumes. Thus, tokens before the synchronization point are reused rather than regenerated, while later tokens are sampled under the new student snapshot. Each completed sample records the token index at which the active weight version changes. The queue-depth parameter τ is enforced as a capacity bound rather than as a learner-side drop rule. The coordinator creates a semaphore with $(\tau + 1)B$ permits, where B is the effective train batch size. The prompt feeder acquires one permit before submitting a prompt to rollout, and the train dispatcher releases permits only after the corresponding samples have been consumed by a learner update. During weight synchronization, a sync gate prevents the feeder from using newly released permits until rollout workers have received the updated weights. Thus, smaller τ limits the amount of unconsumed rollout work in the pipeline, while larger τ permits a deeper backlog and more overlap. The queues themselves remain FIFO; items are not evicted for being stale.

Training throughput metric. The table reports training throughput. Let n_j be the number of response tokens used by learner update j, and let t_j be the train wall-clock time after that update. Discarding the first five warmup updates, we compute

$$\text{throughput} = \frac{\sum_{j=6}^{J} n_j}{\sum_{j=6}^{J}(t_j - t_{j-1})} = \frac{\sum_{j=6}^{J} n_j}{t_J - t_5}.$$

Speedups are normalized to the strict-sync run with the same student and MC setting.

Pipeline overlap metric. Let $\mathcal{S} = \{\text{rollout}, \text{teacher}, \text{train}\}$. For teacher and train, we merge overlapping busy intervals within each stage and compute the merged busy time T_s. Rollout has N_r workers, so we first merge intervals within each worker i and then define the rollout-stage busy time as the worker-normalized average

$$T_{\text{rollout}} = \frac{1}{N_r} \sum_{i=1}^{N_r} T_{\text{rollout},i}.$$

Let T_{wall} be the elapsed train wall-clock interval from the first to the last recorded pipeline-stage interval. We define

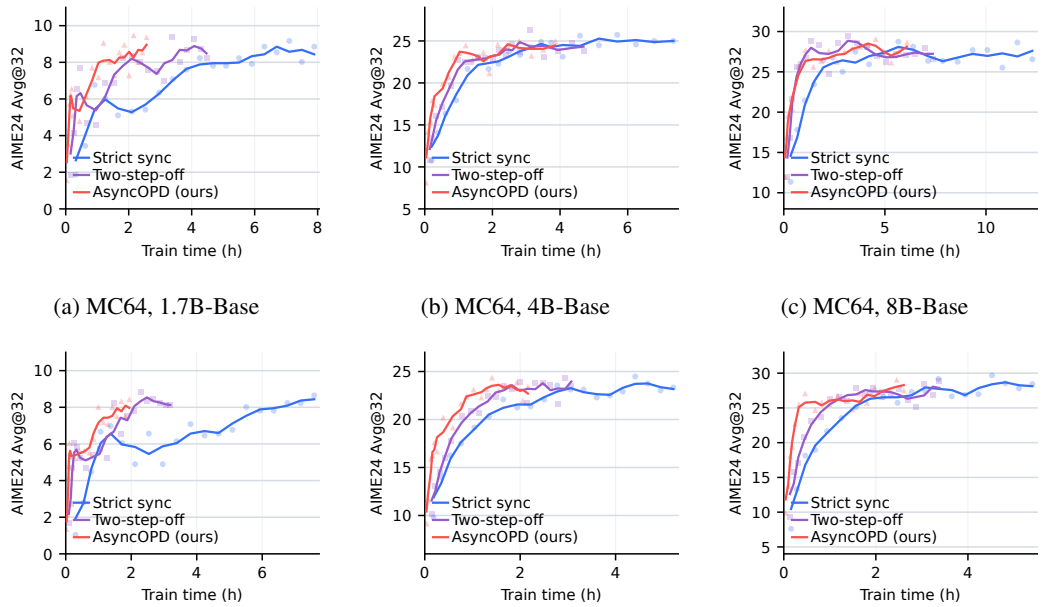
$$\text{overlap} = \frac{\sum_{s \in \mathcal{S}} T_s}{T_{\text{wall}}}.$$

A mostly serial schedule has overlap near 1, and the maximum remains 3: all rollout workers, teacher scoring, and training busy for the full interval.

Hardware and testing protocol. Each scheduler run uses one 8-GPU node. One GPU is reserved for teacher scoring. The remaining seven GPUs are the rollout/training pool. Rollout generation uses data parallelism, and learner training uses PyTorch FSDP. Strict sync runs time-share this pool: all seven GPUs run rollout, then all seven switch to training, and the cycle repeats. The two-step-off and our AsyncOPD runs split the same seven GPUs concurrently: 4 GPUs for rollout workers and 3 GPUs for the FSDP trainer.

For each student size and MC setting, we compare strict sync, two-step-off, and our AsyncOPD scheduler with the same teacher, training data, evaluation metrics, and reverse-KL estimator: current-policy A_θ, no clipping, and old-to-current IS correction. Two-step-off fixes a two-update offset between rollout and the learner update that consumes it, so stale rollout reuse is static and controlled rather than produced by queue timing. We use this offset because it is the fastest static step-off schedule under the 4-rollout/3-trainer split. The OPD pipeline has three serial stages: rollout generation, teacher scoring, and learner training. Therefore, a two-step offset is enough to keep all stages occupied in the gated schedule. Larger offsets only make the consumed data older; they do not create another OPD stage to overlap or remove the step-off batch barrier. We measure final-checkpoint Avg@32 and train wall-clock time over the same training horizon.

Qwen3-Base train-time accuracy. Figure 11 provides the train-time view for Qwen3-Base students. AsyncOPD reaches later checkpoints sooner, so accuracy improves earlier in wall-clock time across student sizes and MC settings.



(a) MC64, 1.7B-Base (b) MC64, 4B-Base (c) MC64, 8B-Base

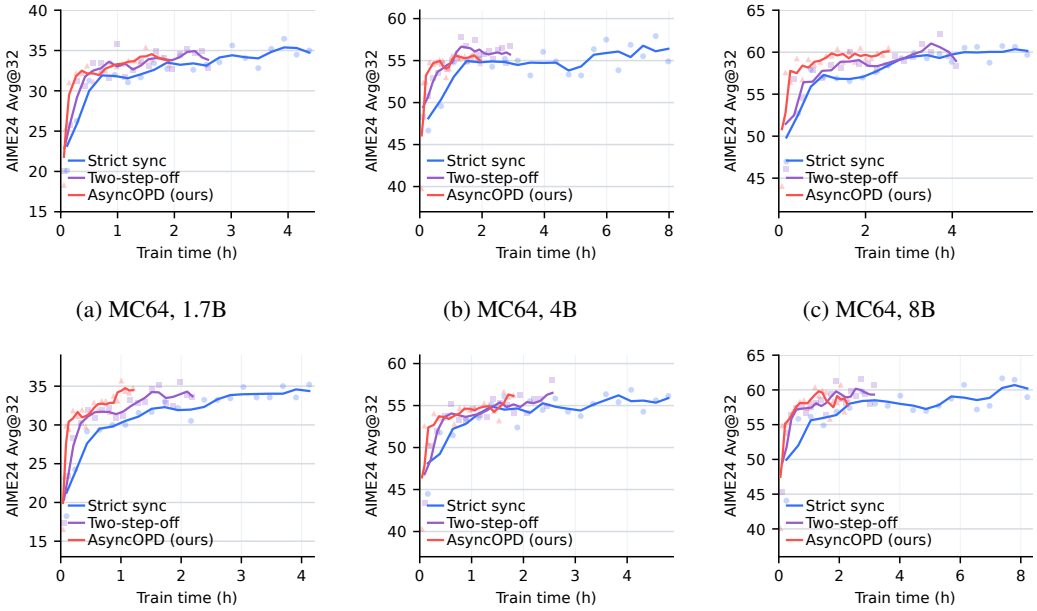
(d) MC1, 1.7B-Base (e) MC1, 4B-Base (f) MC1, 8B-Base

Figure 11: Train-time AIME24 Avg@32 for Qwen3-Base students with MC64 and MC1. Lines are 3-point moving averages; faint markers are raw evaluations; colors denote scheduler. AsyncOPD reaches later checkpoints sooner, so its accuracy improves earlier in wall-clock time.

Table 6: Additional AsyncOPD scheduler results for Qwen3 students with thinking disabled. Train tok/s is training throughput; parentheses show speedup over the matched strict-sync baseline. Overlap is concurrent OPD-stage activity. Avg@32 is final AIME24. AsyncOPD achieves the highest throughput and overlap in all matched settings while maintaining comparable final accuracy.

Student	Scheduler	MC64			MC1		
		Train tok/s (× sync)	Overlap	Avg@32	Train tok/s (× sync)	Overlap	Avg@32
1.7B	Strict sync	11.2k (1.00×)	0.78	**35.00**	11.7k (1.00×)	0.78	**35.21**
	Two-step-off	18.0k (1.61×)	1.58	32.81	20.8k (1.78×)	1.71	33.54
	AsyncOPD (ours)	**24.2k (2.16×)**	**2.07**	33.23	**37.0k (3.17×)**	**2.56**	34.79
4B	Strict sync	4.5k (1.00×)	0.87	54.90	7.4k (1.00×)	0.83	56.15
	Two-step-off	11.5k (2.58×)	1.54	**56.77**	13.0k (1.75×)	1.63	**58.02**
	AsyncOPD (ours)	**17.0k (3.82×)**	**2.15**	54.69	**18.5k (2.49×)**	**2.26**	56.25
8B	Strict sync	5.6k (1.00×)	0.86	59.69	3.9k (1.00×)	0.87	58.96
	Two-step-off	7.5k (1.34×)	1.45	58.33	9.6k (2.45×)	1.61	**60.73**
	AsyncOPD (ours)	**10.1k (1.81×)**	**2.11**	**60.52**	**10.9k (2.78×)**	**2.17**	58.65

Additional Qwen3 AsyncOPD results. For the Qwen3 1.7B, 4B, and 8B student rows, we disable thinking at the tokenizer prompt-formatting level: prompt construction uses the Qwen3 tokenizer's non-thinking chat-template mode before rollout and evaluation. Table 6 and Fig. 12 report this comparison. The systems pattern matches the Qwen3-Base results: AsyncOPD has the highest throughput and overlap, reaching up to 3.8× strict-sync throughput on MC64 and up to 3.2× on MC1. The train-time accuracy plots show the same wall-clock pattern as the main Qwen3-Base results: AsyncOPD reaches later checkpoints sooner, so accuracy improves earlier across student sizes and MC settings.



(a) MC64, 1.7B (b) MC64, 4B (c) MC64, 8B

(d) MC1, 1.7B (e) MC1, 4B (f) MC1, 8B

Figure 12: Train-time AIME24 Avg@32 for Qwen3 1.7B, 4B, and 8B students with thinking disabled, using MC64 and MC1. Lines are 3-point moving averages; faint markers are raw evaluations; colors denote scheduler. AsyncOPD reaches later checkpoints sooner, so its accuracy improves earlier in wall-clock time.