Adaptive Scheduling Pipeline for Multi-Instance Asynchronous Reinforcement Learning

Salah Chikhi* École polytechnique
Luis Zuin* Huawei Technologies
Entong Li **Li Zeng†** Shenzhen Research Institute of Big Data

Abstract

We present the Adaptive Scheduling Pipeline (ASP), an offline rollout scheduler for multi-instance asynchronous reinforcement learning (RL) of large language models (LLMs). ASP jointly reduces three sources of rollout underutilization—padding bubbles from intra-minibatch length heterogeneity, skewness bubbles from long-tail outputs, and sorting bubbles from fluctuations in training-queue occupancy induced by output-length-sorted dispatch—through a single offline preprocessing pass. This enables efficient multi-instance rollout that overcomes the early scaling plateau of single-instance inference while reducing fluctuations in the supply of generated samples to the trainer. We integrate the scheduler into an asynchronous architecture that decouples rollout from training while keeping policy lag observable and bounded. Within our asynchronous architecture, ASP's scheduling reduces mean iteration time by 10.9%. The complete system improves end-to-end throughput by up to $3.22\times$ and reduces iteration time by up to 76.8% relative to MindSpeed-RL, while reducing iteration time by up to 62.5% relative to verl. It also scales more robustly with additional generation resources than these baselines and shows no visible convergence degradation in the measured bounded-lag regime. ASP has since been used in an internal telecom-domain RL post-training workflow, reducing the reported training period from more than three months to two months and producing a domain-enhanced model for knowledge answering and tool calling.

1 Introduction

Reinforcement learning (RL) has become a central component of large language model (LLM) post-training, improving alignment and reasoning by training on generated responses and reward signals [1, 2]. However, RL post-training also creates a demanding systems problem: each policy update requires rollout generation, scoring, log-probability computation, and backpropagation. As models produce longer outputs, rollout/generation often becomes the dominant bottleneck, accounting for a large fraction of training time in modern RL with Verifiable Rewards (RLVR) systems [3, 4].

Large-scale RL for LLMs is commonly executed in a *hybrid* loop: for each batch, the system first performs inference and then runs a training step on the resulting samples. This sequential organization is simple and keeps rollouts close to the current policy, but it has several drawbacks. First, it creates idle time: while the trainer updates, no new samples are generated. Second, it forces repeated mode switches and possible re-sharding on the same machines. Third, it couples two stages with different system profiles: backpropagation is typically compute-bound and benefits from synchronized parallelism, whereas autoregressive generation is memory-bandwidth-bound and often reaches an early scaling plateau when more resources are added to a single inference instance [5].

*Equal contribution.
†Corresponding author: zengli@sribd.cn.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

Asynchronous and disaggregated pipelines address part of this problem by allowing rollout and training to proceed concurrently [5, 6, 7]. In this producer–consumer setting, generation instances produce samples that the trainer consumes for policy updates, with a queue buffering completed samples until they are needed. However, separation alone does not overcome the early scaling plateau of single-instance rollout. Scaling rollout across multiple generation instances exposes a local–global scheduling conflict: decoding minibatches should use each instance efficiently despite variable response lengths, while their dispatch order should maintain a steady supply of samples to the trainer over time.

We identify three sources of underutilization in this pipeline. The first is *padding bubbles*: prompts in the same inference minibatch produce responses of different lengths, so shorter sequences finish early while the generation actor remains occupied by the longest generations. The second is *skewness bubbles*, following StreamRL [5]: LLM output lengths are often heavy-tailed, and a few long generations can collapse the effective active batch size near the end of decoding. Sorting prompts by predicted output length helps reduce both local inefficiencies, but can create a global imbalance between generation and training. If short minibatches are dispatched first and progressively longer minibatches later, the sample arrival rate decreases over time: the queue may initially build up and then drain as longer minibatches dominate. Queue buildup makes samples wait through additional policy updates, increasing policy lag; if the queue later empties, the trainer must wait for new samples. We call this temporal imbalance caused by sorted dispatch *sorting bubbles*.

We propose the **Adaptive Scheduling Pipeline (ASP)**, an offline rollout scheduler for multi-instance asynchronous RL. ASP first uses predicted output lengths and dynamic programming to construct length-homogeneous decoding minibatches, improving local rollout efficiency. It then reorders these minibatches into consecutive batch groups with balanced aggregate predicted load, smoothing the supply of samples to the trainer over time. This single offline preprocessing pass uses predicted lengths to jointly target padding, skewness, and sorting bubbles without adding online scheduling overhead during rollout.

ASP is integrated into a multi-instance asynchronous RL pipeline designed to keep policy lag bounded. A Training Unit (TU) performs reward computation, log-probability evaluation, and policy updates; one or more Generation Units (GUs) host multiple generation actors that perform rollouts in parallel; and a Control Unit (CU) dispatches scheduled minibatches and forwards completed samples through a controlled Training Queue. The queue absorbs short-term variability and provides backpressure, while periodic weight refresh keeps the rollout policy close to the trainer. This design preserves the main benefits of disaggregation—continuous rollout, multi-instance scaling, and independent training/generation configurations—while keeping policy lag observable and bounded.

ASP targets a complementary scheduling layer relative to prior asynchronous systems. StreamRL [5] uses an output-length ranker primarily for inter-instance routing, assigning the top $\alpha\%$ longest predicted samples to dedicated instances and prioritizing long requests to reduce generation makespan. It does not optimize which prompts share a decoding minibatch or how the resulting minibatches are ordered to stabilize trainer arrivals. AsyncFlow [6] instead focuses on asynchronous dataflow across RL stages, scheduling ready samples across tasks and data-parallel consumers with consumer-side token balancing; its mechanisms do not construct predicted-length decoding minibatches or balance their dispatch order. Other recent systems address complementary aspects of asynchrony: AReaL [8] controls staleness through rollout admission and interrupted generation, whereas Laminar [9] reactively handles in-flight long-tail trajectories. ASP acts before rollout, jointly choosing predicted-length minibatch composition and balancing consecutive batch groups so that local reductions in padding and skewness do not create unstable trainer arrivals. Thus, ASP is complementary to disaggregation, routing, and staleness-control mechanisms rather than an alternative to them.

We implement ASP inside MindSpeed-RL [10] with Group Relative Policy Optimization (GRPO) [11] on Ascend 910B3 Neural Processing Units (NPUs), and evaluate it on the Qwen3 Mixture-of-Experts (MoE) and dense models [12]. Our contributions are:

1. We identify a local–global scheduling conflict in multi-instance rollout and propose ASP, an offline scheduler that combines predicted-length minibatch construction with balanced batch-group ordering to reduce padding, skewness, and sorting bubbles.

2. We isolate ASP's scheduling contribution through staged scheduling ablations and predictor-quality analysis, showing that naive length sorting improves local efficiency but destabilizes trainer arrivals, while balanced ordering preserves the local gain and restores queue stability.

3. We integrate ASP into a bounded-lag asynchronous RL pipeline and evaluate it on Qwen3 MoE and dense models [12] against matched hybrid and optimized asynchronous/disaggregated baselines.

2 Background and related work

2.1 RL post-training pipelines

Large-scale RL post-training for LLMs is commonly implemented as a hybrid loop: a batch of prompts is first sent to rollout/generation, the generated samples are scored, and the resulting data are then consumed by the trainer for log-probability evaluation and policy update. This sequential organization is simple, but it couples two stages with different system profiles. This coupling can leave accelerators idle, force repeated mode switches between inference and training, and prevent each stage from using its most suitable parallelism strategy.

Instead, asynchronous pipelines overlap rollout with training, while disaggregation allows the two stages to use separate resources and configurations [5, 6, 7]. This requires coordinating the delivery of generated samples to the trainer and the refresh of model weights used for generation. With multiple generation instances, efficient execution also depends on allocating resources between stages and scheduling variable-length rollouts to limit idle time [5, 6].

2.2 Existing asynchronous systems and remaining gaps

Prior asynchronous RL systems address complementary layers of rollout execution. StreamRL [5] improves disaggregated RL training through stream generation to reduce pipeline bubbles and an output-length ranker to mitigate skewness bubbles. It routes the top $\alpha\%$ of predicted long-tail prompts to dedicated machines, improving batching efficiency. While closely related to our motivation, StreamRL primarily uses length prediction for inter-instance routing, without explicitly optimizing decoding-minibatch composition. Moreover, α is fixed, and its dispatching policy does not explicitly balance the predicted load of consecutive batch groups to smooth sample arrivals at the trainer. In contrast, ASP uses predicted lengths offline to construct length-homogeneous minibatches and reorder them into balanced batch groups, jointly addressing padding, skewness, and sorting bubbles.

AsyncFlow [6] uses TransferQueue for asynchronous dataflow across RL stages and tasks, with consumer-side token balancing. Its padding optimizations target data storage and transmission rather than decoding minibatch composition or dispatch ordering. Other systems address different bottlenecks: AReaL [8] controls staleness through generation admission and interrupted decoding across weight updates; Laminar [9] reactively relocates in-flight long-tail trajectories. ASP instead builds its schedule before rollout, without interrupting or migrating work that has already been dispatched. These mechanisms are complementary rather than mutually exclusive.

verl [13, 14] is an RL framework for LLM post-training supporting disaggregated rollout and training. It notably supports training-side remove-padding and dynamic batching, as well as rollout-side continuous batching. However, unlike ASP, it does not explicitly optimize decoding-minibatch composition or batch-group ordering to stabilize trainer arrivals. This makes it a natural asynchronous baseline. A more detailed comparison with commonly used RL frameworks is provided in Appendix A. Table 1 summarizes this design space.

Table 1: Comparison of ASP with existing asynchronous RL systems.

Method	Async	Multi-inst. rollout	Padding-aware rollout scheduling	Sorting-bubble mitigation
Hybrid RL	No	No	No	No
StreamRL [5]	Yes	Yes	Partial	No
AsyncFlow [6]	Yes	Config.-dep.	No	No
AReaL [8]	Yes	Yes	No	No
Laminar [9]	Yes	Yes	No	No
verl [13]	Yes	Config.-dep.	No	No
ASP	Yes	Yes	Yes	Yes

3 Offline scheduling for multi-instance rollout

In ASP, scheduling is not only a throughput optimization for rollout. It is also the mechanism that makes multi-instance asynchronous generation more predictable: the same strategy that improves per-minibatch utilization also smooths the producer–consumer dynamics seen by the Training Queue over time. This section describes the scheduling problem and the three-step offline preprocessing pipeline used by ASP. Table 2 summarizes the role of each step before we describe the steps in detail.

Table 2: ASP scheduling pipeline. Each offline step targets a different source of rollout inefficiency or queue instability.

Step	Mechanism	Main effect
1	Output-length prediction	Orders prompts by expected rollout length
2	Dynamic programming minibatch construction	Reduces intra-minibatch length heterogeneity
3	Balanced batch-group ordering	Smooths queue arrivals over time

We use the term *sample* for one prompt–response trajectory, *minibatch* for the set of prompts decoded in parallel by one generation actor, and *global batch* for the set of prompts consumed in one training iteration. Each prompt is sampled several times, and all its samples are generated in the same minibatch and consumed in the same global batch; batch sizes are therefore counted in prompts. During rollout, each generation actor repeatedly receives a minibatch, generates the corresponding responses, and returns the completed samples to the Training Queue only after the entire minibatch finishes. The scheduler decides how prompts are grouped into minibatches and how those minibatches are ordered before dispatch. Its objective is twofold: reduce local underutilization during generation, and make the arrival rate of generated samples at the Training Queue balanced over time.

3.1 Bubble sources

Rollout efficiency is limited by three sources of underutilization. The first is *padding bubbles*: prompts in the same minibatch generate responses of different lengths, so shorter sequences finish earlier while the actor remains occupied by the longest ones. As a result, useful token generation can occupy only a small fraction of the token slots implied by the minibatch size and longest response.

The second is *skewness bubbles*, following StreamRL [5]: LLM response lengths are often heavy-tailed, and a few very long generations can collapse the effective active batch size near the end of decoding. Sorting prompts by predicted output length helps reduce both padding and skewness within a minibatch, because prompts with similar expected lengths are decoded together.

The third source is specific to asynchronous rollout. We call it *sorting bubbles*. If the dataset is simply sorted by predicted length, early minibatches tend to be short and later minibatches tend to be long. This improves local length homogeneity, but can destabilize the global pipeline: generation may initially overproduce samples and fill the Training Queue, then slow down as longer minibatches dominate, causing the queue to drain. Thus, naive length sorting can trade local efficiency for temporal instability. ASP addresses this by separating two goals: first forming length-homogeneous minibatches, then reordering those minibatches into balanced batch groups.

3.2 Step 1: output-length prediction

ASP first predicts the output length of each prompt and sorts the dataset by the predicted values. Note that exact output length predictions are neither required nor attainable: response lengths vary across generations for the same prompt and change as the model is updated during RL training. The predictor only needs to be roughly accurate, producing values close enough to the actual lengths and an ordering that separates prompts whose expected response lengths differ substantially.

In our implementation, we collect prompt–length pairs by running inference with the target model on a small subset of prompts, then fine-tune Qwen 2.5 1.5B [15] to predict output lengths (Appendix F.4). We measure its ranking quality with Kendall's τ [16], and Section 5.3 evaluates scheduling across prediction qualities. The predictor remains fixed throughout training, a choice supported by the limited response-length drift in the main `Qwen3-30B-A3B` run, where mean length changes by only 1.6% (Appendix F.7).

Our use of predictions differs from StreamRL's output-length ranker. StreamRL uses predicted lengths to route long-tail samples to dedicated instances and prioritize longer requests. As we explain in the

following steps, ASP uses predicted lengths offline, once before training, to construct minibatches and balance consecutive batch groups.

3.3 Step 2: length-homogeneous minibatch construction

After sorting prompts by predicted length, ASP constructs inference minibatches. With a fixed minibatch size, large gaps in predicted output lengths may fall within a minibatch, forcing prompts of substantially different output lengths to be decoded together. To mitigate this, we allow minibatch sizes to vary slightly between a base size b and $b + \kappa$, where κ is typically small.

Let $l_0 \leq \cdots \leq l_{n-1}$ be the sorted predicted output lengths. A partition of the dataset is defined by endpoints $0 = x_0 < x_1 < \cdots < x_{N_B} = n$, where minibatch i contains prompts $x_i, \ldots, x_{i+1} - 1$. For each minibatch, we define the utilization proxy:

$$R_i = \frac{\sum_{j=x_i}^{x_{i+1}-1} l_j}{(x_{i+1} - x_i)\, l_{x_{i+1}-1}} \tag{1}$$

This ratio compares the predicted useful generated tokens with the total token slots implied by the minibatch size and longest predicted output. Hence, we define $1 - R_i$ to be the estimated padding bubble rate of minibatch i, and

$$\bar{P} = 1 - \frac{1}{N_B} \sum_{i=0}^{N_B-1} R_i \tag{2}$$

is the average predicted padding bubble rate, which should be minimized. With κ small relative to b, N_B varies little over the feasible set, motivating the use of summed utilization as a surrogate objective. In practice, ASP therefore maximizes $\sum_{i=0}^{N_B-1} R_i$ subject to $b \leq x_{i+1} - x_i \leq b + \kappa$. This optimization is solved by dynamic programming in $O(\kappa n)$ time and $O(n)$ space. The full recurrence and pseudocode are given in Appendix E.

Hence, ASP preserves the benefits of output length sorting while reducing within-minibatch output length disparities through flexible minibatch sizing. The result is a sequence of minibatches that are locally efficient for generation, but still ordered from short to long. The next step tackles this temporal imbalance.

3.4 Step 3: balanced batch-group ordering

The final step reorders minibatches to balance the predicted load of consecutive batch groups and stabilize arrivals at the Training Queue. Let B_i be the predicted load of minibatch i, defined as the sum of predicted output lengths in that minibatch. ASP groups minibatches into batch groups of size k, and seeks a permutation σ such that each group has approximately the same total predicted load. With $G = \lceil N_B/k \rceil$ groups and target group load $M = \frac{1}{G} \sum_{i=0}^{N_B-1} B_i$, defined as the average predicted load per group, the objective is

$$\min_{\sigma} \frac{1}{G} \sum_{g=1}^{G} \left(\sum_{i \in \mathcal{G}_g} B_{\sigma(i)} - M \right)^2 \tag{3}$$

where $\mathcal{G}_g$ denotes the indices assigned to group g. This objective directly targets sorting bubbles: instead of dispatching all short minibatches first and all long minibatches later, ASP mixes short and long minibatches so that consecutive groups have similar predicted processing time.

We solve this reordering problem with simulated annealing (SA) [17] in our implementation. After reordering, batch groups may still be shuffled for training randomness, while preserving their internal load balance. The detailed objective, simulated annealing procedure, and hyperparameter choices are deferred to Appendix E.

Overall, ASP performs one offline preprocessing pass: predict lengths, build length-homogeneous minibatches, and reorder them into balanced batch groups. This produces rollout work units that are efficient for individual generation actors and stable for the global Training Queue, which is precisely the scheduling layer missing from naive asynchronous disaggregation.

4 Integration into bounded-lag asynchronous RL

The scheduling pipeline described above is designed to run inside an asynchronous RL system where rollout and training proceed concurrently. This section summarizes the system integration needed to make ASP practical: the architecture that consumes the scheduled minibatches, and the bounded-lag weight-refresh mechanism that keeps asynchronous training close to the current policy.

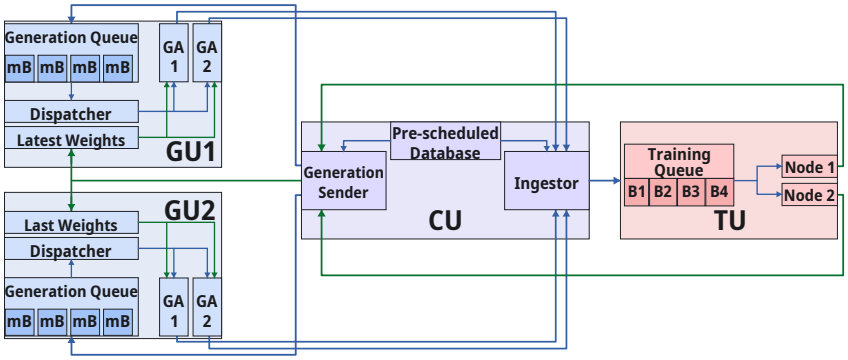


Figure 1: ASP architecture. Control Unit dispatches scheduled minibatches to Generation Units, receives completed samples, and forwards them to Training Queue consumed by Training Unit. GA denotes a generation actor, mB a minibatch, and B a global batch.

4.1 Architecture overview

ASP is integrated into a decoupled RL post-training pipeline with three components: a *Training Unit* (TU), one or more *Generation Units* (GUs), and a *Control Unit* (CU) co-located with the TU. The TU performs reward computation, log-probability evaluation, and policy updates. Each GU hosts multiple generation actors, typically vLLM [18] instances, which repeatedly receive scheduled inference minibatches, generate responses, and return completed samples. The CU coordinates this producer–consumer interaction by dispatching minibatches to the GUs and forwarding completed samples to the TU through a Training Queue. Figure 1 gives the system diagram and more details are given in Appendix B.

This architecture serves two purposes. First, it allows rollout and training to operate continuously instead of alternating as in hybrid pipelines. Second, it keeps the two stages independently configurable: the TU and GUs may use different parallelism layouts, batch sizes, and hardware allocations. This independence is important because backpropagation and autoregressive generation have different system profiles, and a single shared configuration is often suboptimal for at least one of them.

The Training Queue is the main stability mechanism. It buffers short-term fluctuations in generation time and allows the CU to monitor queue occupancy for backpressure control. When generated samples arrive faster than the TU consumes them, the CU can pause the start of new decoding minibatches; actors already decoding finish their current minibatches. Generation resumes as the Training Queue drains. ASP's scheduler reduces the variance of generated-sample arrival times before execution, while the queue provides online stabilization during training. Detailed CU/TU/GU implementation choices and the resource allocation heuristic are deferred to Appendix B.

4.2 Weight refresh and bounded policy lag

Beyond balancing rollout and training rates, asynchronous execution introduces policy lag: samples may be generated by older model weights than those currently being trained, potentially destabilizing learning if the lag becomes too large. ASP therefore uses a bounded-lag weight-refresh path. The TU exports weight snapshots at a configurable interval, set to one training step in all our experiments. Because the TU and GUs may use different parallelism layouts, this snapshot is asynchronously converted into an inference-ready format before being loaded by generation actors. Finally, actors reload weights asynchronously: if an actor is already decoding a minibatch, it finishes that minibatch before reloading, avoiding a global pause of all generation workers.

This design preserves the main benefit of disaggregation—continuous generation—while keeping the lag observable and bounded. For a minibatch that arrives at the Training Queue, the effective

off-policy distance, measured in training steps, can be approximated by

$$d_{\text{off}} \approx Q_{\text{arrival}} + \frac{T_{\text{save}} + T_{\text{convert}} + T_{\text{load}} + T_{\text{gen}}}{T_{\text{train}}} \tag{4}$$

where Q_{arrival} is the Training Queue depth measured in global batches immediately before the minibatch is inserted, T_{save}, T_{convert}, and T_{load} are the snapshot export, conversion, and reload times, T_{gen} is the generation time of a minibatch, and T_{train} is the TU training-step time. Because the queue is centralized, this lag can be estimated directly from queue state and measured system timings.

This bounded-lag regime is compatible with GRPO [11]: prior work suggests that GRPO-style updates are robust to moderate off-policy sampling when policy drift remains small [19]. ASP does not rely on this robustness alone; it explicitly controls the systems sources of lag through scheduling, bounded queueing, and frequent weight refresh. We quantify the resulting policy lag and evaluate convergence under this lag in Section 5.5. A complete off-policy discussion, including the update-rate / version-lag distinction and duty-cycle calculation, is given in Appendix D. The low-level weight-flow implementation is given in Appendix C.

5 Experiments

In this section, we evaluate whether ASP improves the efficiency of RL post-training without destabilizing learning. We study (i) the complete-system performance against MindSpeed-RL and verl; (ii) the contribution of scheduling within a fixed ASP architecture; (iii) scalability with additional generation resources; and (iv) convergence under the measured policy lag.

5.1 Setup

We implement ASP inside MindSpeed-RL using GRPO [11]. Experiments run on Ascend 910B3 nodes, each with 8 NPUs. We evaluate end-to-end performance using the MoE model `Qwen3-30B-A3B` and the dense model `Qwen3-32B` [12]. Scheduling ablations first evaluate padding and load-balance metrics using fixed measured responses from `Qwen3-32B` and `Qwen3-8B`. We then execute real `Qwen3-8B` generation with simulated training to isolate the impact of scheduling on generation efficiency, iteration time, and queue behavior. We compare ASP against two baselines. MindSpeed-RL naturally serves as our hybrid baseline, as ASP is built on top of its infrastructure; it does not decouple rollout and training into a bounded-lag multi-instance pipeline. We also evaluate a verl-based asynchronous and disaggregated baseline [13]. This baseline evaluates the benefit of ASP beyond rollout/training separation: both verl and ASP decouple rollout and training, but only ASP applies the offline scheduling pipeline from Section 3 to make multi-instance rollout locally efficient and globally balanced over time.

We use two datasets. DEEPSCALER [20] contains challenging mathematical problems and is used for throughput, scheduling, and scalability experiments. With an 8192-token output cap on this dataset, the output-length predictors for `Qwen3-30B-A3B`, `Qwen3-32B`, and `Qwen3-8B` reach Kendall's $\tau = 0.64$, 0.44, and 0.58, respectively. MAZE-REASONING-GRPO-V0.1 [21] is used to evaluate convergence on a task less likely to be saturated by the base pre-trained model. We report two maze rewards: *maze integrity*, measuring whether the generated solution is well formed, and *maze correctness*, measuring whether the decoded path solves the maze. Additional dataset details, hyperparameters, node allocations, output-length distributions, and predictor evaluation metrics are deferred to Appendix F.

To evaluate system efficiency, we measure *end-to-end iteration time*, including rollout acquisition, reward/scoring, log-probability computation, policy update, and asynchronous delays visible to the iteration. We define *end-to-end tokens per second* (TPS) as the number of generated tokens per iteration divided by this time and normalized by the total number of NPUs. Mean iteration times used for performance comparisons are arithmetic averages over all measured iterations, including startup.

5.2 End-to-end performance

Table 3 compares complete ASP with MindSpeed-RL and verl using the arithmetic mean of end-to-end iteration times. Across both models, ASP reduces mean iteration time by 59.9–62.5% relative to verl and by 75.1–76.8% relative to MindSpeed-RL. For `Qwen3-30B-A3B`, complete ASP also reaches

58 tokens/s/NPU compared with 18 tokens/s/NPU for MindSpeed-RL, yielding a $3.22\times$ end-to-end throughput improvement.

Table 3: Mean end-to-end iteration time under matched eight-node resources. Reductions are relative to each baseline; lower is better. Configuration details are given in Appendix F.5.

Model	ASP	verl	MindSpeed-RL	ASP reduction	
	Mean iteration time (s)			vs. verl	vs. MindSpeed-RL
`Qwen3-30B-A3B`	**215.2**	536.7	927	59.9%	76.8%
`Qwen3-32B`	**230.7**	615.1	928	62.5%	75.1%

These comparisons measure the complete systems rather than the scheduler in isolation: ASP and the baselines also differ in execution architecture and implementation. Section 5.3 therefore holds the ASP architecture fixed and separately attributes the gains due to scheduling.

5.3 Scheduling ablation

We measure the scheduling pipeline's performance by isolating it within a fixed ASP architecture through five scenarios (C0–C4). C0 uses naïve fixed-size minibatches in the original order; C1 sorts prompts by predicted output length, exposing sorting bubbles; C2 adds dynamic programming to refine minibatch boundaries according to the second step of ASP; C3 randomly shuffles C2 minibatches to test whether randomization alone is sufficient to address sorting bubbles; and C4 applies ASP's complete balanced ordering to measure the additional benefit of explicit load balancing.

Local efficiency and dispatch balance. Table 4(a) evaluates C0–C4 on 39,000 measured `Qwen3-32B` responses. We construct noisy length predictions calibrated to Kendall correlation $\tau = 0.64$; this is a sensitivity-test condition, not the measured accuracy of the 32B predictor. Metrics use measured response lengths and are averaged over three seeds. Intra-minibatch dispersion is the standard deviation (STD) of response lengths within each minibatch, averaged over minibatches. Mean padding rate is obtained by computing $1 - \frac{\text{mean output length}}{\text{maximum output length}}$ for each minibatch and taking their average using true output lengths. Dispatch imbalance is measured by the coefficient of variation (CV), that is the standard deviation divided by the mean, of total response tokens per consecutive batch group. Group-load spread is defined as the ratio of the largest to the smallest batch-group load. Within-minibatch dispersion and padding rate characterize the local length heterogeneity underlying padding and skewness bubbles; group-load CV characterizes the temporal imbalance underlying sorting bubbles.

Table 4: Ablation of scheduling steps. Panel (a) measures schedule properties using fixed response lengths; panel (b) executes real generation with simulated training-step durations and weight-refresh delays. Trend is the Spearman correlation between iteration index and iteration time; values near zero indicate little monotonic drift.

Metric	C0	C1	C2	C3	C4
(a) Qwen3-32B responses; controlled $\tau = 0.64$; three-seed means					
Intra-minibatch length STD (tokens)	2138	1163	1163	1163	1163
Mean padding rate	0.322	0.226	0.224	0.224	0.224
Batch-group load CV	0.103	0.403	0.401	0.162	0.070
Group-load spread (max/min)	1.68	9.88	9.81	2.39	1.39
(b) Qwen3-8B generation with simulated training; measured predictor $\tau = 0.58$					
Mean iteration time (s)	188	185	183	190	178
Iteration-time trend	-0.080	$+0.670$	$+0.510$	$+0.190$	-0.075
Training Queue STD (samples)	41	45	40	36	36
Training Queue maximum (samples)	192	224	176	104	110

C1 shows that sorting improves local efficiency but increases load CV from 0.103 to 0.403, resulting in substantial fluctuations in the workload dispatched over time. The dynamic programming step in C2 slightly refines padding rates (0.226 to 0.224), while leaving this temporal imbalance largely

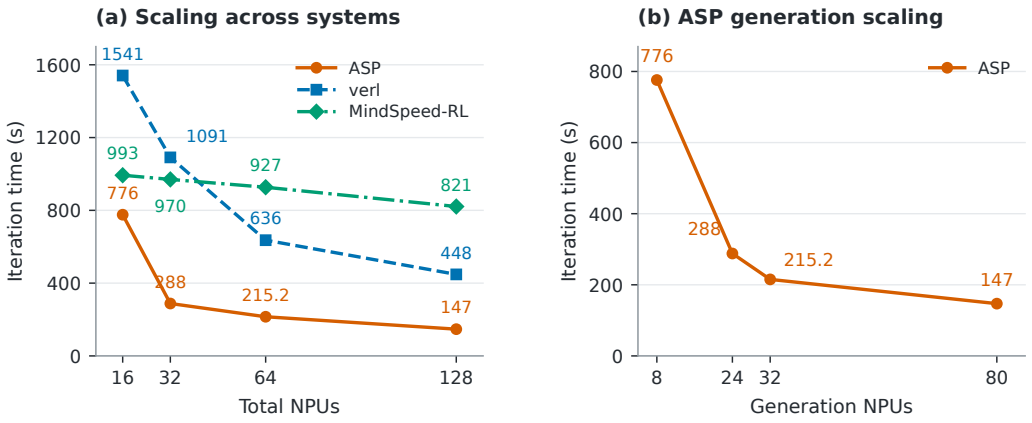


Figure 2: Scalability of `Qwen3-30B-A3B`. (a) Iteration time versus total NPUs for ASP, verl, and MindSpeed-RL. (b) The same ASP results versus generation NPUs. Unlike in Table 3, verl runs without dynamic batch sizing; configurations are detailed in Appendix F.5.

unchanged. Balanced ordering (C4) then reduces the load CV of the same minibatches from 0.401 to 0.070 without changing their padding rate, whereas random shuffling (C3) only reaches 0.162. Thus, balanced ordering preserves the local batching gains while reducing dispatch imbalance below C0 in this condition.

Impact on iteration time and throughput. Table 4(b) reports results for the five schedules C0–C4 using real `Qwen3-8B` generation through vLLM actors, including ASP's dispatch, queue, and backpressure mechanisms. To isolate scheduling effects, we simulate training-step duration and weight-refresh delay using fixed timings measured in complete ASP runs. Appendix F.6.3 provides the details. Sorted dispatch exhibits positive iteration-time trends (C1: $+0.670$; C2: $+0.510$), whereas C4 shows little monotonic drift (-0.075) and has the lowest mean time, 178 s versus 188 s for C0, a 5.3% reduction. Shuffling matches C4's queue STD and has a slightly lower queue maximum, but a higher mean iteration time (190 s) and trend ($+0.19$).

Separately, we compare complete training runs with and without scheduling on `Qwen3-30B-A3B` (Figure 9, Appendix F.6.1). Enabling scheduling reduces the mean iteration time from 238 to 212 s (10.9%) and increases end-to-end throughput from 47 to 58 tokens/s/NPU (23.4%). Scheduling also reduces the STD of minibatch generation time by 29.4% and Training Queue STD by 15.0%, showing that smoother generation and queue behavior also occur during complete training. First-iteration time also falls from 1042 to 724 s (30.5%).

Sensitivity to prediction quality. Three-seed sweeps on measured `Qwen3-32B` and `Qwen3-8B` responses show that sorting's padding benefit decreases as prediction quality deteriorates and becomes marginally negative by $\tau = 0.25$. Balanced ordering nevertheless keeps group-load CV at or below C0 over the tested oracle-to-random range. Appendix F.6.2 provides the perturbation protocol, measured predictor correlations, and full three-seed results for both models.

5.4 Scalability

We evaluate scalability using `Qwen3-30B-A3B` while varying the number of NPUs. Figure 2 summarizes the main results. As the number of NPUs increases, disaggregated multi-instance systems benefit more than the hybrid baseline in our setting: decoupling enables phase-specific configurations, and extra generation instances increase rollout capacity. ASP achieves lower iteration time than both baselines at every tested scale, reaching 147 s at 128 NPUs.

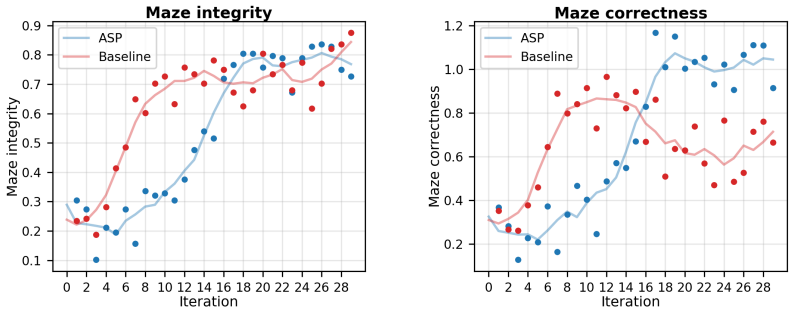
The bottleneck shifts as resources grow. At 16 NPUs, both stages are constrained: the single TU node requires recomputation and parameter offload, and the single GU node makes generation the dominant bottleneck, which explains ASP's smaller margin over the baselines. Adding two GU nodes to the same TU (32 NPUs) yields a nearly balanced allocation (Appendix B.6) and reduces iteration time from 776 s to 288 s, a near-linear $2.69\times$ speedup for three times the generation resources. At 64

NPUs, the four-node TU no longer needs recomputation or offload, and iteration time falls to 215.2 s. At 128 NPUs, training becomes the bottleneck as it approaches its own scaling plateau, so further gains depend on training rather than rollout capacity.

5.5 Convergence under bounded lag

Finally, we verify that asynchronous execution does not visibly harm learning in our bounded-lag regime. On a 4-node run with 1 TU node and 3 GU nodes, the measured timing breakdown gives, using Equation 4, $d_{\text{off}} \approx \bar{Q}_{\text{arrival}} + 3.3$, where $\bar{Q}_{\text{arrival}}$ is the mean queue depth at arrival. At this operating point, the system is rollout-bound and generated samples are dispersed in time, so the Training Queue remains close to empty. The effective off-policy distance is therefore approximately 3.3 trainer steps. The full timing breakdown is reported in Appendix D.

To test whether this staleness affects learning, we run 30 GRPO iterations on MAZE-REASONING-GRPO-V0.1 with `Qwen3-30B-A3B` and a global batch size of 16 prompts. Figure 3 compares ASP with MindSpeed-RL. Both methods reach a similar maze-integrity plateau, with ASP showing a short initial delay caused by generation warmup. Maze correctness follows the same qualitative conclusion: ASP does not show visible degradation under the measured lag.



(a) Maze integrity reward (b) Maze correctness reward

Figure 3: Convergence on MAZE-REASONING-GRPO-V0.1. ASP remains stable under an estimated off-policy distance of approximately 3.3 trainer steps.

These results are consistent with the expected robustness of GRPO to moderate staleness [19]. More importantly, they show that ASP's systems mechanisms—offline scheduling, bounded queueing, and frequent weight refresh—keep the asynchronous pipeline in a small-lag regime while delivering substantial throughput gains.

6 Conclusion

We presented ASP, a scheduling-centered approach for efficient asynchronous RL post-training of LLMs. The main insight is that disaggregating rollout and training is not sufficient by itself: multi-instance rollout must also be scheduled so that generation is locally efficient inside each actor and globally stable at the Training Queue level. ASP addresses this missing layer with an offline bubble-mitigating scheduler that predicts output lengths, constructs length-homogeneous minibatches, and reorders them into balanced batch groups.

On Qwen3 MoE and dense models, ASP reduces iteration time by up to 76.8% versus MindSpeed-RL and 62.5% versus verl, reaching 3.22× the end-to-end throughput of MindSpeed-RL. Ablations confirm scheduling's direct contribution: length-aware minibatching improves local efficiency, balanced ordering preserves these gains and stabilizes trainer arrivals, and complete training runs with and without scheduling show a 10.9% reduction in iteration time within the same architecture. Together with near-linear scaling and no visible convergence degradation at approximately 3.3 trainer steps of lag, these results demonstrate the value of coordinated minibatch construction and dispatch ordering for efficient asynchronous RL post-training.

Acknowledgments and Disclosure of Funding

A significant part of this work was conducted within the Global Technology Service (GTS) team at Huawei Technologies in Dongguan, China, where Salah Chikhi and Luis Zuin were interns, and Entong Li and Li Zeng were full-time employees at the time. We thank the GTS team for its support and collaborative research environment. We also acknowledge the computing infrastructure provided by Huawei Technologies and by the Shenzhen Research Institute of Big Data.

The authors declare no competing interests.

References

[1] Long Ouyang et al. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract.html.

[2] Alex Havrilla, Yuqing Du, Sharath Chandra Raparthy, Christoforos Nalmpantis, Jane Dwivedi-Yu, Maksym Zhuravinskyi, Eric Hambro, Sainbayar Sukhbaatar, and Roberta Raileanu. Teaching large language models to reason with reinforcement learning, 2024. https://arxiv.org/abs/2403.04642.

[3] Xuanming Wen et al. Reinforcement Learning with Verifiable Rewards Implicitly Incentivizes Correct Reasoning in Base LLMs, 2025. https://arxiv.org/abs/2506.14245.

[4] Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Zilin Zhu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, Weikai Fang, Xianyu, Yu Cao, Haotian Xu, and Yiming Liu. OpenRLHF: An Easy-to-use, Scalable and High-performance RLHF Framework, 2025. https://arxiv.org/abs/2405.11143.

[5] Yinmin Zhong, Zili Zhang, Xiaoniu Song, Hanpeng Hu, Chao Jin, Bingyang Wu, Nuo Chen, Yukun Chen, Yu Zhou, Changyi Wan, Hongyu Zhou, Yimin Jiang, Yibo Zhu, and Daxin Jiang. StreamRL: Scalable, Heterogeneous, and Elastic RL for LLMs with Disaggregated Stream Generation, 2025. https://arxiv.org/abs/2504.15930.

[6] Zhenyu Han, Ansheng You, Haibo Wang, Kui Luo, Guang Yang, Wenqi Shi, Menglong Chen, Sicheng Zhang, Zeshun Lan, Chunshi Deng, Huazhong Ji, Wenjie Liu, Yu Huang, Yixiang Zhang, Chenyi Pan, Jing Wang, Xin Huang, Chunsheng Li, and Jianping Wu. AsyncFlow: An Asynchronous Streaming RL Framework for Efficient LLM Post-Training, 2025. https://arxiv.org/abs/2507.01663.

[7] Michael Noukhovitch, Shengyi Huang, Sophie Xhonneux, Arian Hosseini, Rishabh Agarwal, and Aaron Courville. Asynchronous RLHF: Faster and More Efficient Off-Policy RL for Language Models, 2025. https://arxiv.org/abs/2410.18252.

[8] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. AReaL: A Large-Scale Asynchronous Reinforcement Learning System for Language Reasoning, 2026. URL https://arxiv.org/abs/2505.24298.

[9] Guangming Sheng, Yuxuan Tong, Borui Wan, Wang Zhang, Chaobo Jia, Xibin Wu, Yuqi Wu, Xiang Li, Chi Zhang, Yanghua Peng, Haibin Lin, Xin Liu, and Chuan Wu. Laminar: A Scalable Asynchronous RL Post-Training Framework, 2025. URL https://arxiv.org/abs/2510.12633.

[10] Laingjun Feng, Chenyi Pan, Xinjie Guo, Fei Mei, Benzhe Ning, Jianxiang Zhang, Xinyang Liu, Beirong Zhou, Zeng Shu, Chang Liu, et al. MindSpeed RL: Distributed Dataflow for Scalable and Efficient RL Training on Ascend NPU Cluster. *arXiv preprint arXiv:2507.19017*, 2025. https://gitcode.com/Ascend/MindSpeed-RL.

[11] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models, 2024. https://arxiv.org/abs/2402.03300.

[12] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 Technical Report, 2025. URL https://arxiv.org/abs/2505.09388.

[13] ByteDance Seed. verl: Volcano Engine Reinforcement Learning for LLMs, 2026. https://github.com/verl-project/verl.

[14] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A Flexible and Efficient RLHF Framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297, 2025. doi: 10.1145/3689031.3696075. URL https://doi.org/10.1145/3689031.3696075.

[15] Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 Technical Report, 2024. URL https://arxiv.org/abs/2412.15115.

[16] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1-2):81–93, 1938. doi: 10.2307/2332226. https://doi.org/10.2307/2332226.

[17] Scott Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220 (4598):671–680, 1983.

[18] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

[19] Youssef Mroueh, Nicolas Dupuis, Brian Belgodere, Apoorva Nitsure, Mattia Rigotti, Kristjan Greenewald, Jiri Navratil, Jerret Ross, and Jesus Rios. Revisiting Group Relative Policy Optimization: Insights into On-Policy and Off-Policy Training, 2025. https://arxiv.org/abs/2505.22257.

[20] Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Y. Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Li Erran Li, Raluca Ada Popa, and Ion Stoica. DeepScaleR: Surpassing O1-Preview with a 1.5B Model by Scaling RL. Notion blog post, 2025.

[21] Alan Dao and Dinh Bach Vu. AlphaMaze: Enhancing Large Language Models' Spatial Intelligence via GRPO, 2025. https://arxiv.org/abs/2502.14669.

[22] John Schulman et al. Proximal Policy Optimization Algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

[23] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, and Shen Li. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proceedings of the VLDB Endowment*, 16(12):3848–3860, 2023. https://arxiv.org/abs/2304.11277.

[24] Mohammad Shoeybi et al. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv:1909.08053*, 2019.

[25] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems*, volume 37, pages 62557–62583, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.

[26] Philipp Moritz et al. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*, 2018. https://arxiv.org/pdf/1712.05889.

[27] Microsoft DeepSpeed Team. DeepSpeed: Extreme-scale model training for everyone, 2020. https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/.

[28] Thomas Wolf et al. Transformers: State-of-the-Art Natural Language Processing. In *EMNLP: System Demonstrations*, 2020. https://arxiv.org/abs/1910.03771.

[29] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/a85b405ed65c6477a4fe8302b5e06ce7-Abstract-Conference.html.

[30] THUDM. slime: An LLM Post-Training Framework for RL Scaling, 2025. https://github.com/THUDM/slime.

[31] Lasse Espeholt et al. IMPALA: Importance Weighted Actor-Learner Architectures. In *ICML*, 2018. https://arxiv.org/abs/1802.01561.

[32] Shengyi Huang, Jiayi Weng, Rujikorn Charakorn, Min Lin, Zhongwen Xu, and Santiago Ontañón. CLEANBA: A Reproducible and Efficient Distributed Reinforcement Learning Platform, 2023. https://arxiv.org/abs/2310.00036.

[33] Haizhong Zheng, Jiawei Zhao, and Beidi Chen. Prosperity before Collapse: How Far Can Off-Policy RL Reach with Stale Data on LLMs?, 2025. https://arxiv.org/abs/2510.01161.

[34] Python Software Foundation. *concurrent.futures — Launching parallel tasks*, 2024. Available at https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.ThreadPoolExecutor.

[35] Alberto Maria Metelli, Matteo Papini, Nico Montali, and Marcello Restelli. Importance sampling techniques for policy optimization. *Journal of Machine Learning Research*, 21:1–75, 2020. http://jmlr.org/papers/v21/20-124.html.

Appendix Overview

This appendix provides a comprehensive and self-contained presentation of the algorithmic foundations, implementation details, in-depth analysis, and additional experimental results of our method. Appendix A provides a more detailed discussion of related work. In Appendix B, the system architecture is further explained. Appendix C provides the low-level implementation details behind the weight-refresh path summarized in Section 4. The off-policy analysis and scheduling algorithm details are presented in Appendices D and E, respectively. Appendix F includes additional experimental results, benchmark descriptions, and implementation settings. Finally, several limitations and future directions are discussed in Appendix G. Overall, the appendix complements the main paper with additional justification, implementation details, and experimental context.

A Extended related work

This appendix expands the comparison in Section 2. The main paper focuses on ASP's scheduling-centered contribution; here we give additional context on hybrid pipelines, asynchronous/disaggregated RL systems, and off-policy actor–learner training. Table 5 summarizes the main differences between the pipelines related to ASP.

A.1 Hybrid and asynchronous RL pipelines

Large-scale RL post-training systems are commonly organized either as *hybrid* pipelines or as *asynchronous/disaggregated* pipelines. In a hybrid pipeline, the same accelerator pool alternates between rollout, scoring/log-probability computation, and training. This design is simple and keeps rollouts close to the current policy, but it creates idle periods and forces generation and training to share one resource configuration. This is often suboptimal because autoregressive generation and backpropagation have different system profiles: generation is largely memory-bandwidth bound and sensitive to output-length variability, whereas training is dominated by synchronized backpropagation and often benefits from different parallelism layouts.

Asynchronous pipelines decouple rollout from training so that generation can continue while the trainer updates the model. This improves overlap and allows the two stages to use different resources. Yet separation alone does not overcome the early scaling plateau of single-instance rollout. Scaling rollout further requires multiple generation instances, which introduces a scheduling problem: the system must maintain high local utilization inside each inference minibatch while keeping the global producer–consumer dynamics at the Training Queue stable. ASP targets this missing scheduling layer.

A.2 StreamRL

StreamRL [5] is closely related to ASP because it studies disaggregated RL training for LLMs and also identifies rollout as a major bottleneck. It introduces *stream generation* to overlap generation and training, and uses an output-length ranker to mitigate *skewness bubbles*: underutilization caused by long-tail output lengths. Indeed, since decoding benefits from maintaining a large active batch, utilization drops when most sequences in a minibatch finish early and only a few long generations remain. StreamRL addresses this by routing the top $\alpha\%$ of prompts predicted to generate long outputs to a dedicated subset of machines. It then applies longest-processing-time-first scheduling within each instance.

ASP differs from StreamRL in three main ways. First, StreamRL uses length prediction primarily for inter-instance routing: long-tail prompts are separated from the rest and routed to specific nodes, but the construction of each inference minibatch is not optimized with ASP's explicit padding objective. ASP instead uses predicted lengths to construct length-homogeneous minibatches, directly targeting intra-minibatch padding waste. Second, StreamRL's routing depends on a static hyperparameter α that may be hard to tune, whereas ASP formulates minibatch construction and batch-group ordering as optimization problems over the full prompt set. Third, ASP identifies and targets *sorting bubbles*: queue-level instability caused by naive length-sorted dispatch. Sorting prompts by predicted length improves local minibatch efficiency, but it can create temporal imbalance by sending mostly short minibatches first and mostly long minibatches later. ASP separates these goals: it first builds length-

homogeneous minibatches, then reorders them into balanced batch groups to smooth the generation rate seen by the Training Queue.

Thus, ASP is complementary to StreamRL's motivation. StreamRL demonstrates the value of asynchronous rollout and length-aware routing; ASP studies how length information can be used as a scheduling signal to jointly reduce padding, skewness, and queue instability.

A.3 AsyncFlow

AsyncFlow [6] studies asynchronous RL execution through a distributed data-management module, TransferQueue, which overlaps multiple RL stages such as actor rollout, reference computation, reward computation, and actor update. Its main contribution is a flexible system that allows different RL stages to run concurrently.

ASP targets a narrower but complementary problem. Instead of pipelining all RL stages, ASP keeps a single controlled Training Queue between generation and training, and focuses on scheduling rollout work so that multi-instance generation is both locally efficient and globally stable. This choice is motivated by our workload: inside the Training Unit, backpropagation dominates the stage time, while the main end-to-end bottleneck is rollout. Therefore, trainer-side pipelining would provide limited benefit compared with improving rollout throughput, overlap, and queue stability.

This also clarifies the difference between ASP's Training Queue and a more general distributed data-management substrate. ASP's queue is deliberately a narrow coordination interface between generation and training. It is co-located with the Training Unit, exposes queue occupancy for backpressure, and makes the queue contribution to policy lag directly observable. A more general multi-stage queueing substrate provides broader flexibility, but ASP chooses the narrower design because it is sufficient for the scheduling problem studied in this paper.

A.4 AReaL

AReaL [8] is a fully asynchronous RL system that decouples generation from training, allowing the trainer to update the policy as completed trajectories become available. To limit off-policy effects, AReaL controls the admission rate of new generation requests according to a maximum permitted staleness. It also supports interruptible generation: when new policy weights arrive, ongoing decoding is interrupted, the old KV caches are discarded and recomputed under the updated weights, and unfinished trajectories resume. This avoids waiting for the longest generations before refreshing the policy, but can produce trajectories spanning multiple policy versions. AReaL addresses this algorithmic challenge through decoupled PPO, which separates the behavior and proximal policies.

ASP targets a different source of inefficiency. Rather than interrupting decoding, ASP lets each generation actor complete its current minibatch before refreshing weights, avoiding KV-cache recomputation and mixed-version trajectories. More importantly, ASP proactively constructs length-homogeneous decoding minibatches and balances consecutive batch groups, reducing local padding and skewness without inducing unstable trainer arrivals. While AReaL manages asynchronous execution and staleness, it does not explicitly optimize this joint minibatch-composition and dispatch-ordering objective. ASP operates with unmodified GRPO in its measured small-lag regime, supported by its offline scheduling, bounded queueing, and balanced TU/GU resource allocation, without claiming the larger-lag robustness studied by AReaL.

A.5 Laminar

Laminar [9] addresses long-tail generation inefficiency through trajectory-level asynchrony, allowing trajectories to be generated and consumed independently. It introduces hierarchical relay workers that distribute model weights asynchronously, avoiding global synchronization stalls between training and generation. To mitigate underutilization when only a few long trajectories remain, Laminar further proposes a dynamic repacking mechanism: it monitors KV-cache utilization to identify underutilized rollout instances and consolidates their unfinished trajectories onto fewer instances using the same policy version. This frees other instances to refresh their weights and generate new trajectories, improving generation throughput.

Although both Laminar and ASP target inefficiencies caused by heterogeneous output lengths, they operate at different stages. Laminar reactively redistributes ongoing trajectories after underutilization

emerges, whereas ASP proactively organizes prompts before generation using predicted output lengths. ASP first constructs length-homogeneous minibatches to reduce padding and skewness, then reorders them into balanced batch groups to prevent the sorting bubbles created by naive length sorting. Laminar does not explicitly optimize this initial minibatch composition or the temporal balance of trainer arrivals. Conversely, ASP does not migrate in-flight trajectories, leaving reactive repacking as a potentially complementary mechanism for handling residual prediction errors. This combination remains unevaluated.

A.6 Commonly used RL frameworks

Beyond research systems that explicitly study asynchronous RL pipelines, ASP is also related to widely used RL post-training frameworks. Reinforcement learning training for LLMs usually requires a complex pipeline that combines large-scale response generation, reward computation, policy optimization, and distributed training. Instead of implementing each component manually, practitioners often rely on frameworks that provide integrated support for these stages.

Many of these frameworks are actively maintained and regularly updated, making them suitable for incorporating recent algorithmic and systems-level advances. Consequently, they have become widely used in both academic research and industrial LLM post-training. In this section, we briefly introduce several representative RL frameworks.

verl. verl [13] is an open-source RL post-training framework for large language models based on the HybridFlow design [14]. It provides abstractions for expressing RL training workflows that involve multiple model roles, such as actors, critics, reference models, reward models, and rollout engines. Its design separates high-level control logic from distributed computation, making it easier to construct complex RL dataflows such as PPO [22] and GRPO [11]. verl also integrates with common large-scale training and inference backends, including FSDP [23], Megatron-LM [24], vLLM [18], and SGLang [25].

In the matched-resource comparison of Table 3, verl uses training-side remove-padding and dynamic batch sizing for actor/reference passes, together with vLLM continuous rollout batching. Those optimizations do not themselves select which prompts form an externally returned decoding minibatch or balance the aggregate load of consecutive batch groups on the generation side. ASP's additional contribution resides in predicted-length composition and balanced batch-group ordering for generation efficiency and stability.

OpenRLHF. OpenRLHF [4] is an open-source framework designed to simplify large scale RLHF and RL-based post-training of language models. It builds on commonly used distributed training and inference systems, including Ray [26], vLLM [18], DeepSpeed [27], and Hugging Face Transformers [28]. The framework provides implementations of several alignment and RL algorithms, such as PPO [22], DPO [29], and related policy-gradient variants. Its main advantage is ease of use: it offers a relatively simple interface for launching distributed RLHF experiments while still supporting large models and efficient rollout generation. This makes OpenRLHF a practical choice for researchers and practitioners who want to prototype or scale post-training methods without building the full infrastructure from scratch.

slime. slime [30] is an LLM post-training framework focused on reinforcement learning at scale. Its core design connects Megatron-LM [24] for distributed model training with SGLang [25] for efficient rollout generation. This combination allows the framework to support high-throughput generation while preserving compatibility with large-scale training strategies such as tensor, pipeline, and data parallelism. In addition to standard RL training workflows, slime emphasizes flexible data-generation pipelines, which can be useful for agentic, reasoning-oriented, or task-specific RL settings. Compared with more general-purpose RLHF frameworks, slime is particularly oriented toward large-scale RL systems where tight integration between the training backend and the inference backend is important.

A.7 Distributed actor–learner RL and policy lag

ASP is also related to distributed actor–learner systems such as IMPALA [31] and later asynchronous RL implementations such as CleanBA [32]. These systems decouple acting from learning, improving throughput but introducing policy staleness: samples may be generated by older behavior policies

Table 5: Extended comparison of asynchronous RL systems and their scheduling mechanisms.

Feature	Hybrid RL	StreamRL	AsyncFlow	AReaL	Laminar	ASP
Rollout/training separation	No	Yes	Yes	Yes	Yes	Yes
Multi-instance rollout	No	Yes	Config.-dep.	Yes	Yes	Yes
Independent training/generation configurations	No	Yes	Yes	Yes	Yes	Yes
Predicted-length minibatch construction	No	No	No	No	No	Yes
Long-tail rollout mitigation	No	Yes	No	Yes	Yes	Yes
Explicit minibatch optimization for rollout padding	No	No	No	No	No	Yes
Balanced batch-group ordering	No	No	No	No	No	Yes
Weight-update decoding interruption	No	No	No	Yes	No	No
In-flight trajectory repacking	No	No	No	No	Yes	No

than the learner's current policy. This actor–learner lag is typically handled through importance weighting, clipping, or by keeping the lag small enough in practice.

LLM post-training by RL has similar concerns, but the system bottlenecks are different. Rollouts are expensive, output lengths are heavy-tailed, and weight refresh may require converting trainer checkpoints into inference-ready layouts. Recent work on asynchronous RLVR reports wall-clock speedups under moderate staleness [7], while M2PO studies robustness under farther off-policy regimes by controlling the second moment of importance weights [33]. ASP is complementary to these algorithmic directions: it does not rely only on algorithmic robustness, but also controls the systems sources of lag through bounded queueing, scheduling, and frequent weight refresh.

B System architecture details

This appendix provides implementation details omitted from Section 4. The main paper describes the architecture at the level needed to understand ASP's scheduling and bounded-lag integration; here we summarize the concrete components used in our prototype.

B.1 ASP at a glance

ASP is integrated into an asynchronous RL pipeline with three main components: a *Training Unit*, one or more *Generation Units*, and a *Control Unit*. The TU performs reward computation, log-probability evaluation, and policy updates. Each GU hosts multiple generation actors, typically vLLM instances [18], that repeatedly decode inference minibatches. The CU coordinates dispatch, receives completed samples, and forwards them to the Training Queue.

The architecture is designed around three invariants. First, generation and training operate continuously whenever possible, instead of alternating as in a hybrid loop. Second, the Training Queue remains bounded through backpressure. Third, weight refresh is asynchronous: actors finish their current minibatch before loading a new snapshot, avoiding a global pause of all generation workers.

B.2 Control Unit

The CU is a CPU-side process co-located with the TU. It is the coordination hub of the system: it dispatches scheduled minibatches to GUs, receives completed responses, forwards samples to the Training Queue, and notifies GUs when new weight snapshots are available. Keeping the CU close to the TU makes the Training Queue state directly observable and simplifies backpressure.

The CU contains three logical controllers.

Generation Sender. The Generation Sender sends orchestration messages, prompt minibatches, and weight-snapshot notifications to the GUs. Prompts are dispatched according to ASP's scheduled order. In particular, the sender treats the scheduler's batch groups as dispatch units, so that consecutive groups have approximately balanced predicted generation load. The sender also maintains the metadata needed to merge returned responses with their original prompts before forwarding samples to the Training Queue.

Database. The Database stores prompts that remain to be sent, tracks which minibatches have been issued, and records failed sends for retry. Since ASP performs scheduling offline, the Database does not solve a new optimization problem during training; it supplies minibatches in the precomputed order as the Generation Sender replenishes the per-GU input queues.

Ingestor. The Ingestor receives completed minibatches from GUs and passes them to the Training Queue. Separating ingestion from the training loop keeps network and coordination work off the critical trainer path. It also allows the CU to record per-GU send/receive events, which are useful for measuring queue stability and debugging generation imbalance.

B.3 Per-GU queue targets and backpressure

To keep generation actors fed while retaining flexibility to balance work across GUs, the CU maintains a small per-GU input-queue target, typically 5–10 minibatches in our implementation. The lower bound hides network latency and avoids starving generation actors. The upper bound limits work committed to each GU, leaving more minibatches available for dispatch to other GUs as they become available.

The Training Queue provides the main backpressure signal. When completed samples accumulate beyond a configured threshold, the CU signals GUs to pause the start of new decoding minibatches, including those already in their input queues. Generation resumes when queue occupancy falls sufficiently. This limits the buildup of completed samples and the policy lag associated with waiting for training. In the rollout-bound configurations used in our experiments, the Training Queue usually remains close to empty, but this mechanism also controls queue growth when generation temporarily outpaces training.

B.4 Training Unit and Training Queue

The TU follows the MindSpeed-RL training structure: it performs reward computation, reference/log-probability computation, and policy updates using GRPO [11]. Unlike systems that pipeline several trainer sub-stages, ASP does not pipeline the internal TU computation. In our workload, backpropagation dominates the TU stage time, and the main end-to-end bottleneck is rollout. Therefore, trainer-side pipelining would provide limited benefit compared with improving rollout throughput and overlap.

The Training Queue is implemented as a separate Ray actor [26]. It stores samples in a lock-protected queue and assembles them into global batches for training, as illustrated in Figure 4. It records queue length, sample arrival events, and per-GU metadata. These counters are used both for backpressure and for estimating the effective off-policy distance.

B.5 Generation Units and generation actors

A GU hosts multiple generation actors. Each actor stores a copy of the inference model and repeatedly executes the same lifecycle: receive one minibatch, generate responses, return the completed samples, and then either accept another minibatch or reload weights if a newer snapshot is available. Actors are independent, so multiple minibatches can be decoded concurrently across a GU.

Each GU also has an inference-side control process that receives minibatches from the CU and assigns them to idle actors. Before assigning a new minibatch, it checks whether the CU has paused generation through its backpressure signal. While the pause is active, queued minibatches remain pending and no new decoding starts; actors already decoding finish their current minibatches.

B.6 Resource allocation heuristic

ASP uses a simple static resource-allocation heuristic. The goal is to choose the number of TU and GU nodes so that rollout throughput is close to training consumption, while keeping generation as a mild bottleneck when possible. This prevents unbounded Training Queue growth and keeps generation resources highly utilized.

The procedure is:

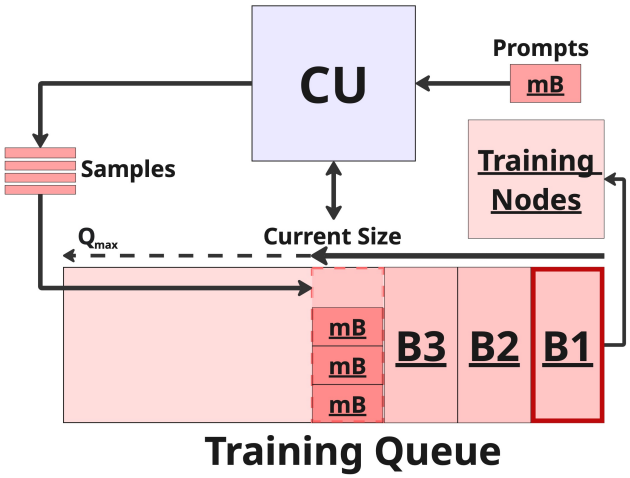


Figure 4: Training Queue. Completed inference minibatches accumulate as prompt–response samples and are consumed by the Training Unit as global training batches. mB denotes a minibatch, B a global batch, and $Q_{\max}$ the queue-occupancy threshold for triggering backpressure.

1. Determine the minimum number of nodes needed by the TU for the target model and global batch size, then measure the TU training-step time.
2. Choose a generation actor configuration on the smallest convenient hardware slice, such as one node or one NPU group, and measure the time to generate one inference minibatch.
3. Replicate generation actors across GUs until aggregate rollout throughput approximately matches TU consumption. We prefer the GU side to be a mild bottleneck, because this keeps the Training Queue bounded and avoids frequent pause/resume behavior.
4. If additional nodes remain, scale the TU and repeat the balancing step. This alternating expansion maintains a near-balanced operating point.

This heuristic is intentionally simple. More elaborate dynamic allocation schemes are possible, but static allocation worked well in our setting because ASP's scheduler reduces generation-time variance before execution and the Training Queue absorbs short-term fluctuations online.

C Weight-flow implementation

This appendix provides the low-level implementation details behind the weight-refresh path summarized in Section 4. The purpose of this path is to preserve independent training/generation specialization without re-coupling the two sides at every update.

C.1 Overview

In a disaggregated RL pipeline, generation actors must periodically receive newer trainer weights. In ASP, this is nontrivial because the TU and GUs may use different parallelism layouts. A trainer checkpoint therefore cannot always be broadcast directly to inference actors. Instead, ASP uses a three-stage weight-flow path:

1. export a trainer snapshot from the TU;
2. convert the snapshot into the layout expected by GU inference actors;
3. reload the converted weights inside each generation actor.

The first two stages are mostly asynchronous relative to generation. The third stage may stall an actor, so ASP uses deferred reload: an actor finishes its current minibatch before loading the new snapshot.

C.2 Trainer-side export

When the TU emits a new snapshot, each training worker prepares a CPU parameter dictionary containing its local shard in a vLLM-compatible format. This synchronous preparation is small relative to a full training iteration. The TU then launches an asynchronous task that writes these local shards to shared storage, preserving the training-side parallelism layout.

After all training nodes finish saving, the CU notifies the corresponding GU that a new snapshot is available. In our prototype, this notification is sent through a lightweight HTTP request containing the file-system locations of the exported checkpoint. To bound storage growth, older snapshots are deleted progressively; in our implementation, we retain only a small recent window.

C.3 Generation-side conversion

Upon notification, the GU converts the training snapshot into an inference-ready layout. Concretely, the conversion merges training tensor-parallel ranks and expert shards when needed, reconstructs the model expected by inference, and repartitions it according to the GU's tensor, expert, and data parallel configuration. This conversion is performed once per snapshot at the GU level, rather than separately inside every generation actor.

This stage allows the TU and GUs to use different parallelism layouts. For example, training may prefer a layout optimized for backpropagation, while generation may prefer smaller tensor-parallel groups replicated across multiple actors. Without conversion, the two sides would need to share a common layout, weakening the benefit of disaggregation.

C.4 Reload methods evaluated

We evaluated several ways to load new weights into vLLM on Ascend NPUs. Two straightforward approaches were too slow in our environment. First, calling vLLM's `load_weights` endpoint took roughly 200 s for Qwen3-30B-A3B. Second, restarting the entire vLLM process after each iteration had a similar cost. These approaches were disproportionate relative to minibatch generation time and would significantly reduce the duty cycle of generation actors.

The bottleneck came primarily from software and I/O limitations rather than raw device bandwidth. In particular, our environment did not provide a fast `safetensors` loading path or IPC-handle-based zero-copy sharing between processes. As a result, repeated deserialization and serialized host-to-device (H2D) transfers became dominant.

C.5 Parallelized host-to-device loading

Our most effective implementation parallelizes loading inside each vLLM worker. We use a `ThreadPoolExecutor` [34] to drive multiple independent device streams in parallel, each issuing H2D `copy_` transfers. Small tensors are grouped together to reduce launch overhead, while large tensors are assigned to separate streams to reduce head-of-line blocking. This improves overlap between file reads, tensor staging, and device copies.

This logic is implemented inside vLLM's `NPUWorker`. With this design, end-to-end reload time dropped from roughly 200 s to about 60–75 s for Qwen3-30B-A3B in our prototype. The exact value depends on the model layout, file system behavior, and concurrent load on the node. In the timing breakdown used for our off-policy analysis, the measured reload time was $T_{\text{load}} = 73.9\,\text{s}$.

C.6 Deferred actor reload

A naive synchronous update would ask every generation actor to stop and reload as soon as a snapshot is available. This creates a global pause and can trigger herd effects: many actors finish, reload, and resume at the same time, increasing CPU and I/O pressure. ASP instead uses deferred reload. If an actor is busy decoding a minibatch, it finishes that minibatch first. It then reloads the latest available snapshot before accepting new work.

Deferred reload preserves execution order within each actor and avoids interrupting in-flight generation. It adds at most one minibatch of actor-local staleness. In practice, this tradeoff is favorable

because it preserves continuous generation and avoids a system-wide synchronization point. The corresponding generation duty-cycle estimate is given in Appendix D.6.

C.7 Alternative optimizations

Further reductions in reload overhead are possible. One practical option is to dedicate one generation actor as a staging actor: it loads the latest converted snapshot and then propagates weights device-to-device (D2D) to sibling actors as they finish their current minibatch. This would replace repeated host-mediated loading with peer-to-peer transfer and could shrink the visible reload window. Similar ideas appear in OpenRLHF [4].

Another optimization is to remove the shared-file-system dependency. Although shared storage is convenient in a single-cluster prototype, it can become a tail-latency bottleneck and is less suitable for cross-cluster deployments. A direct TU-to-GU transfer API, or an exchange based on object storage, would reduce disk dependence and make heterogeneous or geographically separated deployments more practical.

D Off-policy analysis

This appendix expands the bounded-lag discussion from Section 4. The main paper uses a compact approximation of the off-policy distance; here we make explicit the two forms of off-policy behavior induced by asynchronous rollout, the importance-ratio interpretation, the multi-instance offset, and the measured timing breakdown used in our experiments.

D.1 Two off-policy regimes

In asynchronous RL post-training, samples may be generated with slightly older model weights than the weights currently used by the trainer. The relevant question is therefore not whether the system is perfectly on-policy, but whether this lag remains small enough to preserve stable learning. Let π_t denote the trainer policy after iteration t. We use the term *off-policy distance* d_{off} for the lag, measured in trainer steps, between the behavior policy used to generate a sample and the current trainer policy. This notion is closely related to staleness or version lag in distributed actor–learner systems [31, 32, 33].

We distinguish two regimes according to the main source of lag.

Type I: update-rate off-policy. The generation weights are refreshed once every $\Delta > 1$ training iterations, so the lag increases between refreshes and drops when a newer snapshot is loaded. Larger refresh intervals therefore increase policy lag and may impair convergence.

Type II: version-lag off-policy. Even when snapshots are produced after every training iteration, snapshot export, conversion, reload, generation, and queueing introduce delay. This lag reflects the time needed for a snapshot to produce samples that reach training, rather than an intentionally reduced refresh rate. It can remain approximately constant under steady operating conditions, but varies with execution times and queue occupancy in an asynchronous pipeline.

ASP primarily targets the second regime: generation actors load the latest available snapshot after finishing their current minibatch. Training Queue backpressure limits the additional lag caused by completed samples waiting for training. Under this combination of weight refresh and queue control, the measured off-policy distance remains small, and we observe no visible convergence degradation in our experiments. This is consistent with prior work showing that off-policy GRPO can remain effective when the sampling policy stays close to the current policy [19].

D.2 Off-policy distance

After the Training Unit completes iteration t and emits a snapshot, the Generation Units eventually load this snapshot and produce samples with it. These samples enter the Training Queue and are consumed at a later trainer iteration, as illustrated in Figure 5. The resulting distance includes snapshot export, snapshot conversion, actor reload, generation time, and queueing delay.

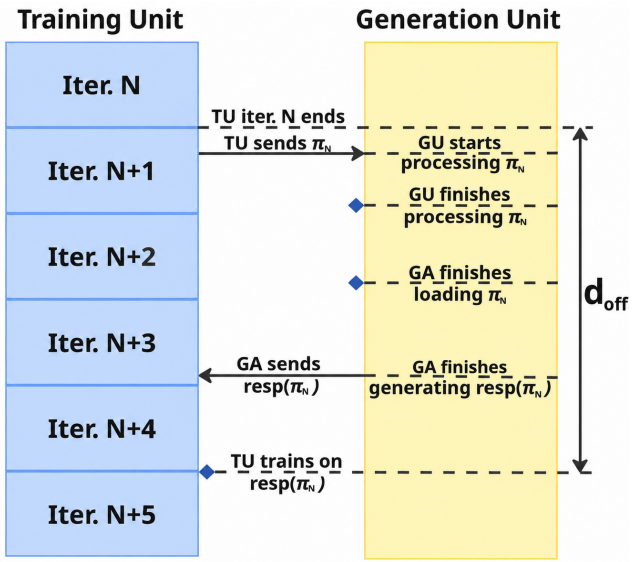


Figure 5: Off-policy distance in ASP. Samples generated with an older snapshot are consumed by a later trainer iteration; their lag, measured in trainer steps, reflects snapshot delay, generation time, and queue depth. GA denotes a generation actor.

Operationally, for a minibatch arriving at the Training Queue, we approximate the instantaneous off-policy distance as

$$d_{\text{off}} \approx Q_{\text{arrival}} + \frac{T_{\text{save}} + T_{\text{convert}} + T_{\text{load}} + T_{\text{gen}}}{T_{\text{train}}},$$

where Q_{arrival} is the Training Queue depth measured in global batches immediately before the minibatch is inserted, T_{save} is the trainer-side snapshot export time, T_{convert} is the generation-side conversion time, T_{load} is the actor reload time, T_{gen} is the time to generate one inference minibatch, and T_{train} is the Training Unit step time.

A practical advantage of ASP's single controlled Training Queue is that the queue contribution to this distance is directly observable at runtime. The remaining terms are system timings that can be measured for each model and deployment configuration. This makes the version-lag contribution easier to monitor than in systems where coordination state is spread across several task-specific queues or controllers.

D.3 Importance-ratio interpretation

In the version-lag regime, the behavior policy for a consumed sample is $\pi_{t-d_{\text{off}}}$ while the update is applied to π_t. For an autoregressive output $y = (y_1, \ldots, y_m)$ generated from prompt x, the per-token importance ratio is

$$r_j = \frac{\pi_t(y_j \mid x, y_{<j})}{\pi_{t-d_{\text{off}}}(y_j \mid x, y_{<j})}.$$

The corresponding sequence-level importance weight is

$$r(y \mid x) = \prod_{j=1}^{m} r_j.$$

When the cumulative drift $\mathrm{KL}(\pi_t \| \pi_{t-d_{\text{off}}})$ is small, these ratios remain concentrated near one. PPO-style clipping [22] and GRPO-style updates [11] further reduce the effect of moderate ratio deviations. ASP complements these update rules by limiting policy lag through frequent weight refresh and Training Queue backpressure.

D.4 Multi-instance offset

ASP uses multiple generation actors, which creates a second, smaller source of lag variation. Different actors may briefly hold different snapshots because they reload asynchronously: an actor that is busy decoding finishes its current minibatch before loading the latest snapshot. As a result, samples arriving at the Training Queue may be generated from a small mixture of nearby behavior policies,

$$\{\pi_{t-\delta_1}, \pi_{t-\delta_2}, \ldots\},$$

where the δ_i values differ slightly across actors.

In practice, this actor-specific offset is small in our runs. Indeed, snapshots are frequent, and deferred reload adds at most one minibatch of actor-local staleness. Moreover, generated samples are time-dispersed in the rollout-bound operating point, so the Training Queue usually remains close to empty. We therefore did not apply additional off-policy corrections.

Two lightweight corrections could be useful in larger-lag regimes. First, samples could be bucketed by the snapshot that generated them, with baselines or normalizers computed per snapshot. This would avoid mixing advantages across slightly different behavior policies. Second, minibatches could be treated as draws from a mixture of nearby snapshot policies, using multiple or relative importance sampling to correct the mismatch [35]. We leave these extensions for future work because the measured lag in our experiments is already small.

D.5 Measured timing breakdown

The convergence experiment in Section 5.5 uses a 4-node configuration with 1 Training Unit node and 3 Generation Unit nodes. The measured timing breakdown is

$$T_{\text{save}} = 23.5\text{s}, \quad T_{\text{convert}} = 84.6\text{s}, \quad T_{\text{load}} = 73.9\text{s}.$$

Thus, the snapshot-related delay is

$$T_{\text{delay}} = T_{\text{save}} + T_{\text{convert}} + T_{\text{load}} = 182.0\text{s}.$$

With average minibatch generation time $T_{\text{gen}} = 763.3$s and Training Unit step time $T_{\text{train}} = 287.6$s, we obtain

$$d_{\text{off}} \approx \bar{Q}_{\text{arrival}} + \frac{182.0 + 763.3}{287.6} \approx \bar{Q}_{\text{arrival}} + 3.3.$$

At this operating point, the system is rollout-bound and the Training Queue remains close to empty, so $\bar{Q}_{\text{arrival}} \approx 0$ and $d_{\text{off}} \approx 3.3$ trainer steps. This is the value reported in the main text.

D.6 Generation duty cycle

The same timing breakdown also estimates how much time each generation actor spends doing useful inference. As the trainer-side save and generation-side conversion are asynchronous relative to ongoing generation, the main actor-visible stall is the reload step. Therefore, a simple per-actor duty-cycle estimate is

$$\eta \approx \frac{T_{\text{gen}}}{T_{\text{gen}} + T_{\text{load}}}.$$

Using $T_{\text{load}} = 73.9$s and $T_{\text{gen}} = 763.3$s gives

$$\eta \approx \frac{763.3}{763.3 + 73.9} \approx 0.912.$$

Thus, generation actors spend about 91% of their time performing inference in this configuration. This supports the design choice of deferred reload: weight refresh remains visible, but it does not dominate the generation duty cycle.

E Scheduling algorithm details

This appendix gives the full optimization objectives and pseudocode for the offline scheduling pipeline summarized in Section 3. ASP performs three steps before training: predict output lengths, construct length-homogeneous minibatches, and reorder minibatches into balanced batch groups.

E.1 Length prediction

The scheduler uses predicted output lengths to order and group prompts. In our implementation, we collect a small set of prompt–length pairs by running the model to be trained on prompts from the training dataset, and fine-tune Qwen 2.5 1.5B [15] to predict the corresponding output length. The predictor is kept fixed throughout training.

Ranking quality is particularly important because sorting aims to bring prompts with similar output lengths together. We report the predictor's Kendall rank correlation τ [16] in Appendix F.4 and evaluate the scheduler's sensitivity to prediction quality in Appendix F.6.2.

E.2 Dynamic programming for minibatch construction

Let $l_0 \leq \cdots \leq l_{n-1}$ be the predicted output lengths after sorting the dataset. The goal is to partition the sorted list into minibatches whose sizes are between b and $b + \kappa$, where b is the base minibatch size and κ is a small allowed size variation. This flexibility allows the scheduler to adjust minibatch boundaries to reduce predicted padding, as illustrated in Figure 6. A partition is represented by endpoints

$$0 = x_0 < x_1 < \cdots < x_{N_B} = n,$$

where minibatch i contains prompts with indices $x_i, \ldots, x_{i+1} - 1$. For a candidate minibatch, define

$$R_i = \frac{\sum_{j=x_i}^{x_{i+1}-1} l_j}{(x_{i+1} - x_i)\, l_{x_{i+1}-1}}.$$

This ratio compares the predicted useful generated tokens with the total token slots implied by the minibatch size and longest predicted output. Thus, $1 - R_i$ estimates the minibatch padding bubble rate. ASP aims to minimize the estimated average padding bubble rate, that is:

$$\bar{P} = 1 - \frac{1}{N_B} \sum_{i=0}^{N_B-1} R_i.$$

ASP uses the sum of minibatch utilizations as an additive surrogate for average utilization:

$$\begin{aligned} \textbf{maximize} \quad & \sum_{i=0}^{N_B-1} R_i \\ \textbf{subject to} \quad & 0 = x_0 < x_1 < \cdots < x_{N_B} = n, \\ & b \leq x_{i+1} - x_i \leq b + \kappa \quad \text{for all } i. \end{aligned}$$

Algorithm 1 summarizes the dynamic program used to solve this additive objective. Let $S[i] = \sum_{j=0}^{i-1} l_j$ be the prefix sum of predicted lengths. For any candidate minibatch starting at j and ending before i, its utilization is

$$R(j, i) = \frac{S[i] - S[j]}{(i - j) l_{i-1}}.$$

The dynamic program is

$$\mathrm{dp}[i] = \max_{j \in \{\max(0, i-(b+\kappa)), \ldots, i-b\}} \left(\mathrm{dp}[j] + R(j, i)\right),$$

with $\mathrm{dp}[0] = 0$, unreachable states set to $-\infty$, and an empty maximization yielding $-\infty$. Backpointers recover the optimal endpoints. Since each state checks at most $\kappa + 1$ candidate minibatch sizes, the complexity is $O(\kappa n)$ time and $O(n)$ space. For fixed small κ, this is linear in the dataset size.

E.3 Balanced batch-group ordering

After the dynamic program, the minibatches are locally efficient but still ordered from short to long. Dispatching them in this order would create sorting bubbles: early groups would be fast, later groups slow, and the Training Queue could oscillate. ASP therefore reorders minibatches into balanced batch groups.

Algorithm 1 Dynamic programming for length-homogeneous minibatches

Input: sorted predicted lengths $l[0..n-1]$, base size b, variation κ
Compute prefix sums $S[0..n]$, with $S[0] \leftarrow 0$ and $S[i+1] \leftarrow S[i] + l[i]$
Initialize $\mathrm{dp}[0] \leftarrow 0$ and $\mathrm{dp}[i] \leftarrow -\infty$ for $i > 0$
Initialize $\mathrm{prev}[0..n] \leftarrow -1$
for $i = 1$ **to** n **do**
    for $j = \max(0, i - (b + \kappa))$ **to** $i - b$ **do**
        $R \leftarrow \dfrac{S[i] - S[j]}{(i-j)l[i-1]}$
        if $\mathrm{dp}[j] + R > \mathrm{dp}[i]$ **then**
            $\mathrm{dp}[i] \leftarrow \mathrm{dp}[j] + R$
            $\mathrm{prev}[i] \leftarrow j$
Recover endpoints by backtracking from n through prev
Output: minibatch endpoints and predicted padding rate

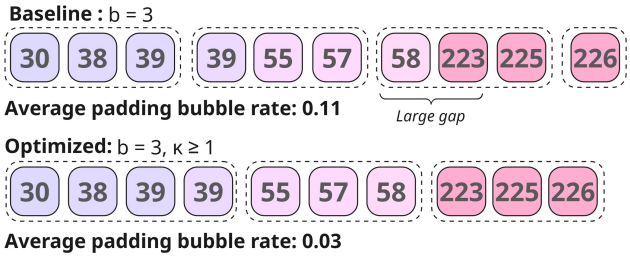


Figure 6: Minibatch construction. Allowing small variations in minibatch size can avoid bad boundaries in the sorted length list and reduce predicted padding bubbles.

Let B_i be the predicted load of minibatch i, defined as the sum of its predicted output lengths. Let k be the number of minibatches per batch group, and let $G = \lceil N_B/k \rceil$ be the number of groups. The target group load is

$$M = \frac{1}{G} \sum_{i=1}^{N_B} B_i.$$

ASP seeks a permutation σ minimizing the variance of predicted load across consecutive groups:

$$\min_{\sigma \in \mathfrak{S}_{N_B}} \frac{1}{G} \sum_{g=1}^{G} \left(\sum_{r=(g-1)k+1}^{\min(gk, N_B)} B_{\sigma(r)} - M \right)^2 .$$

This objective smooths the predicted generation rate seen by the Training Queue. Instead of sending all short minibatches first and all long minibatches later, ASP mixes minibatches so that each consecutive group has approximately the same predicted load, as illustrated in Figure 7.

We optimize this objective using simulated annealing [17], as detailed in Algorithm 2. The method starts from an initial permutation and repeatedly proposes swaps between two minibatches. A swap is accepted if it improves the objective, or with a temperature-dependent probability if it worsens the objective. This allows the search to escape poor local minima.

E.4 Hyperparameters and runtime

Hyperparameters. In our experiments, κ is kept small, typically in $\{1, 2, 3\}$, to prevent excessively large inference minibatches while still allowing the dynamic program to avoid poor boundaries. The batch-group size k controls the time scale over which generation load is smoothed. Larger values of k reduce group-load variance more aggressively, but can also allow larger within-group queue fluctuations. We therefore use intermediate values, typically between 4 and 10. Unless otherwise specified, all scheduled runs use base minibatch size $b = 32$, $\kappa = 2$, and $k = 6$, and Algorithm 2 runs for $n_{\max} = 2 \times 10^6$ iterations with temperature constant $C = 1$.

Algorithm 2 Simulated annealing for balanced batch-group ordering

Input: minibatch loads $B_1, \ldots, B_{N_B}$, group size k, iterations $n_{\max}$, constant C
Initialize permutation σ_0
Define objective $V(\sigma)$ as the batch-group load variance
for $n = 0$ **to** $n_{\max} - 1$ **do**
    Sample two distinct indices i, j uniformly at random
    Let σ' be the permutation obtained by swapping $\sigma_n(i)$ and $\sigma_n(j)$
    Set temperature $T_n \leftarrow C/\log(n+2)$
    Sample $U \sim \mathrm{Uniform}(0,1)$
    if $U < \min\left(1, \exp\left((V(\sigma_n) - V(\sigma'))/T_n\right)\right)$ **then**
        $\sigma_{n+1} \leftarrow \sigma'$
    else
        $\sigma_{n+1} \leftarrow \sigma_n$
Output: optimized permutation $\sigma_{n_{\max}}$

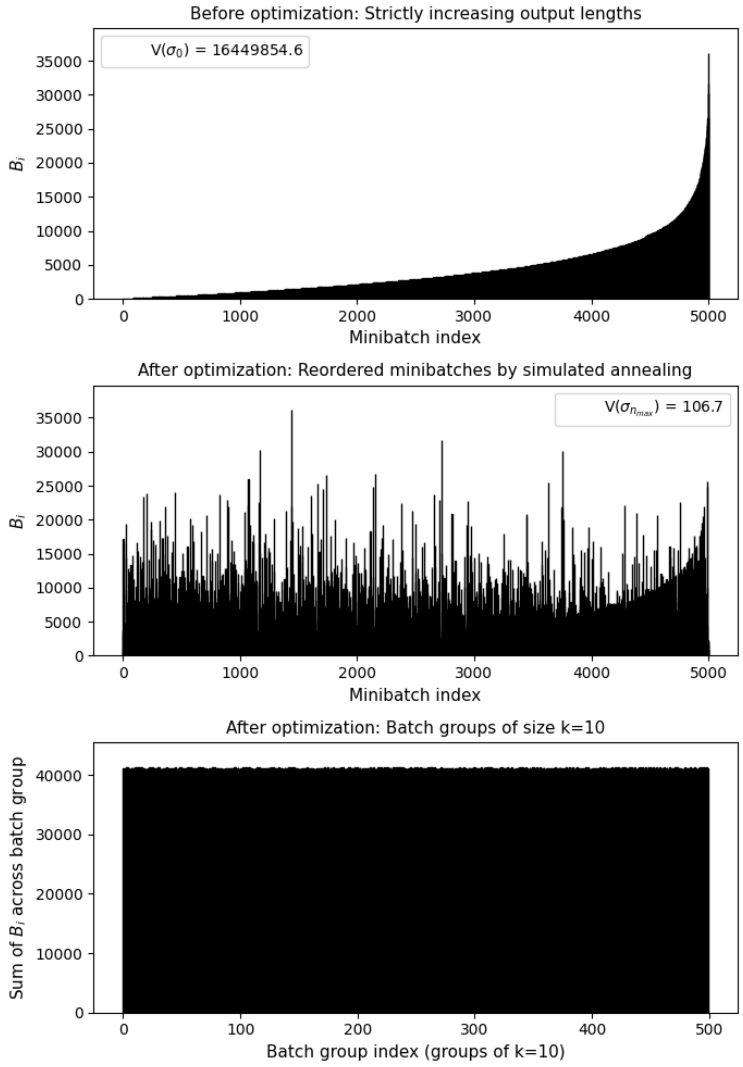


Figure 7: Balanced batch-group ordering. Predicted minibatch loads before (top) and after (middle) simulated annealing, and the resulting group loads for $k = 10$ (bottom). Reordering preserves minibatch composition while balancing the load across consecutive groups.

Table 6: SA scaling results across four output-length distributions (lognormal, exponential, Pareto, bimodal; individual lengths clipped at 8,192 tokens), $k = 10$. Variance reduction: $\big(1 - V(\sigma_{n_{\max}})/V(\sigma_0)\big) \times 100\%$. **Left**: fixed $N_B = 100{,}000$, varying $n_{\max}$. Wall time is averaged across distributions (variation $< 3\%$). **Right**: fixed $n_{\max} = 2 \times 10^6$, varying N_B.

		Variance reduction (%)			
$n_{\max}$	Wall (s)	Logn.	Exp.	Pareto	Bim.
10^4	< 0.01	30.6	29.8	30.5	29.6
8.9×10^4	< 0.01	93.9	95.6	92.2	95.6
2.7×10^5	< 0.01	98.9	99.9	97.2	99.8
8.0×10^5	0.03	99.4	100.0	97.6	100.0
2.4×10^6	0.09	99.4	100.0	97.8	100.0
1.2×10^7	0.45	99.5	100.0	97.8	100.0
1.1×10^8	4.0	99.5	100.0	97.8	100.0
10^9	35.5	99.5	100.0	97.8	100.0

N_B	Wall (s)
10^2	0.05
896	0.05
4,642	0.08
24,041	0.07
1.2×10^5	0.06
6.4×10^5	0.11
1.9×10^6	0.16
5.8×10^6	0.28
10^7	0.31

Runtime. All three stages of the scheduling pipeline are performed offline, introducing no online scheduling overhead during training. Moreover, the preprocessing cost is negligible compared to the overall training time, even for large datasets.

For step 1, output-length prediction scales linearly with the number of prompts for a fixed predictor. This step is computationally inexpensive, as the predictor is substantially smaller than the model being trained and only needs to generate a few tokens encoding the predicted output length. For step 2, minibatch construction requires sorting the predicted lengths in $O(n \log n)$ time, followed by dynamic programming with $O(\kappa n)$ time complexity, as established in Algorithm 1, which allows for efficient scaling to large datasets.

Finally, in step 3, simulated annealing is highly scalable: each proposed swap affects at most two batch-group loads, allowing the objective to be updated in constant time, independently of N_B. To characterize its runtime, we evaluate SA across four synthetic output-length distributions—lognormal, exponential, Pareto, and bimodal—covering N_B up to 10^7 and $n_{\max}$ up to 10^9 (Table 6). At fixed $N_B = 100{,}000$, runtime scales approximately linearly with the iteration budget, achieving around 30 million iterations per second with less than 3% variation across distributions. On a single CPU core, 2.7×10^5 iterations already achieve over 95% variance reduction across all four distributions in under 0.01 s on average, while even the largest tested budget of 10^9 iterations completes in just 35.5 s. At a fixed budget of 2×10^6 iterations, runtime remains below one second across all tested minibatch counts, reaching 0.31 s at $N_B = 10^7$. Further improvements are possible through stratified initialization, such as snake-deal assignment, but seemed unnecessary given the already low observed runtimes.

F Experimental details

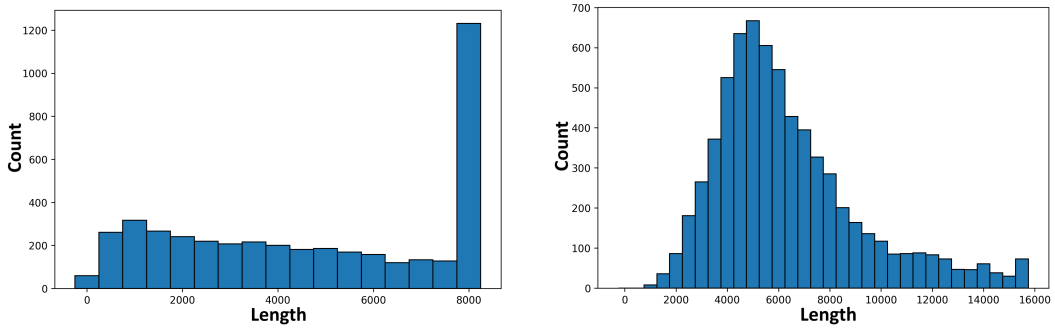
This appendix provides additional details for the experiments in Section 5: hardware, datasets, length prediction, node allocations, and additional plots.

F.1 Hardware and implementation

Hardware. All training and generation experiments are conducted on nodes that are each equipped with 8 Ascend 910B3 NPUs. The simulated annealing runtime benchmarks in Appendix E are conducted on a single CPU core.

ASP implementation. ASP is implemented on top of MindSpeed-RL using GRPO, with vLLM (v0.9.1) as the generation backend. The weight-flow mechanism described in Appendix C handles the conversion of trainer snapshots into inference-compatible weights. The scheduling steps are implemented as per the pseudocodes given in Appendix E.

Baselines. We compare ASP against two baselines. The first is the original MindSpeed-RL hybrid pipeline, which serves as backend framework for ASP. This same-hardware comparison captures the



(a) DEEPSCALER, capped at 8192 tokens.

(b) MAZE-REASONING-GRPO, capped at 16384 tokens.

Figure 8: Output-length distributions for the datasets used in our experiments. The long tails motivate length-aware scheduling.

combined effect of ASP's architectural and implementation changes, while the specific contribution of scheduling is isolated in Section 5.3.

The second baseline is verl (v0.7.1), with vLLM for generation and FSDP [23] for training. We use its one-step off-policy pipeline, which generates the next rollout batch while updating the policy on the current batch.

F.2 Models and datasets

Our main end-to-end experiments evaluate two Qwen3 models: Qwen3-30B-A3B, a Mixture-of-Experts model, and Qwen3-32B, a dense model [12]. We additionally use Qwen3-8B for scheduling ablations and predictor-sensitivity experiments.

We use two datasets. DEEPSCALER [20] contains approximately 41K mathematical problems and is used for throughput, scheduling, and scalability experiments. MAZE-REASONING-GRPO-V0.1 contains more than 180K maze problems and is used for convergence evaluation [21]. We use the maze task because it is less likely to be saturated by the pre-trained model than standard math problems already represented in modern LLM training data.

For the maze benchmark, we report two rewards: *maze integrity*, which measures whether the generated solution is well-formed, and *maze correctness*, which measures whether the decoded path solves the maze. These rewards are used only to compare learning dynamics under ASP and the hybrid baseline.

F.3 Output-length distributions

The scheduling problem is motivated by long and variable output lengths. Figure 8 shows the output-length distributions used in our experiments. For DEEPSCALER, we set the maximum output length to 8192 tokens. A significant fraction of outputs reaches this cap, suggesting that longer contexts may further increase the relative importance of rollout scaling. For MAZE-REASONING-GRPO-V0.1, we use a maximum output length of 16384 tokens.

F.4 Output-length predictor

The output-length predictor is fine-tuned on 4000 prompt–length pairs extracted from the target datasets. The cost of fine-tuning this predictor is negligible relative to the target RL training run. The predictor remains fixed throughout training and is used only during the offline scheduling pass.

We evaluate the predictor using Kendall's rank correlation coefficient τ [16], because ASP's sorting mechanism in the first step relies primarily on relative order rather than exact length values. For `Qwen3-30B-A3B` on DEEPSCALER, the predictor reaches approximately $\tau = 0.64$ with an 8192-token cap and $\tau = 0.55$ with a 16384-token cap. The value $\tau = 0.64$ corresponds to roughly 82%

Table 7: ASP and MindSpeed-RL configurations for Table 3. TP, PP, DP, and EP denote tensor, pipeline, data, and expert parallelism, respectively. All listed trainer layouts use context parallelism degree 1. Generation parallelism is specified per ASP actor or for the MindSpeed-RL rollout layout. A dash indicates no dedicated generation pool; N/A indicates that expert parallelism does not apply.

	`Qwen3-30B-A3B`		`Qwen3-32B`	
Setting	ASP	MindSpeed-RL	ASP	MindSpeed-RL
Workload per iteration				
Prompts	16	16	32	32
Responses per prompt	8	8	8	8
Trajectories	128	128	256	256
Training and generation				
Training NPUs	32	64	32	64
Dedicated generation NPUs	32	—	32	—
Trainer TP / PP / DP	4 / 1 / 8	4 / 1 / 16	8 / 2 / 2	8 / 4 / 2
Trainer EP	4	8	1	1
Dedicated generation actors	16	—	16	—
Generation TP / PP	2 / 1	4 / 1	2 / 1	4 / 1
Generation EP	1	8	N/A	N/A
Measurement (including startup)				
Iterations	120	20	50	50
Mean iteration time (s)	215.2	927	230.7	928

of prompt pairs being ordered consistently with the measured output lengths, which is a substantial improvement over random ordering.

F.5 Configurations for performance and scalability

F.5.1 Matched-resource comparisons

The main comparisons in Table 3 use eight nodes (64 NPUs) per system. ASP and verl each allocate 32 NPUs to training and 32 to generation, whereas the MindSpeed-RL hybrid baseline alternates training and rollout on the same 64-NPU pool. For verl, generation uses tensor parallelism of 8 and vLLM continuous batching. Training uses remove-padding (`use_remove_padding=True`), dynamic batch sizing for actor/reference passes (`use_dynamic_bsz=True`), gradient checkpointing, and CPU offloading of parameters and optimizer states.

For `Qwen3-30B-A3B`, each iteration processes 16 prompts with eight responses per prompt, giving 128 trajectories. For `Qwen3-32B`, it processes 32 prompts with eight responses per prompt, giving 256 trajectories. For each model, verl uses the same global batch size and prompt and response token limits as the corresponding ASP run.

Table 7 details the ASP and MindSpeed-RL layouts and measurement windows.

F.5.2 Scalability

The `Qwen3-30B-A3B` scalability series keeps the workload fixed at 16 prompts and eight responses per prompt (128 trajectories per iteration) for ASP, MindSpeed-RL, and verl at every tested scale. The verl baseline uses the same settings as in the matched-resource comparison, except that dynamic batch sizing is disabled. Table 8 maps total resources to ASP's training/generation split and to the generation-NPU counts used in Figure 2.

F.6 Scheduling ablation details

This appendix provides additional details on the scheduling experiments presented in Section 5.3: the full-training comparison with and without scheduling, the robustness analysis across prediction accuracies, and the evaluation with real generation and simulated training. We report the experimental configurations, methodology, and additional results for each.

Table 8: Resource allocations for the `Qwen3-30B-A3B` scalability series.

Total NPUs	ASP training NPUs	ASP generation NPUs
16	8	8
32	8	24
64	32	32
128	48	80

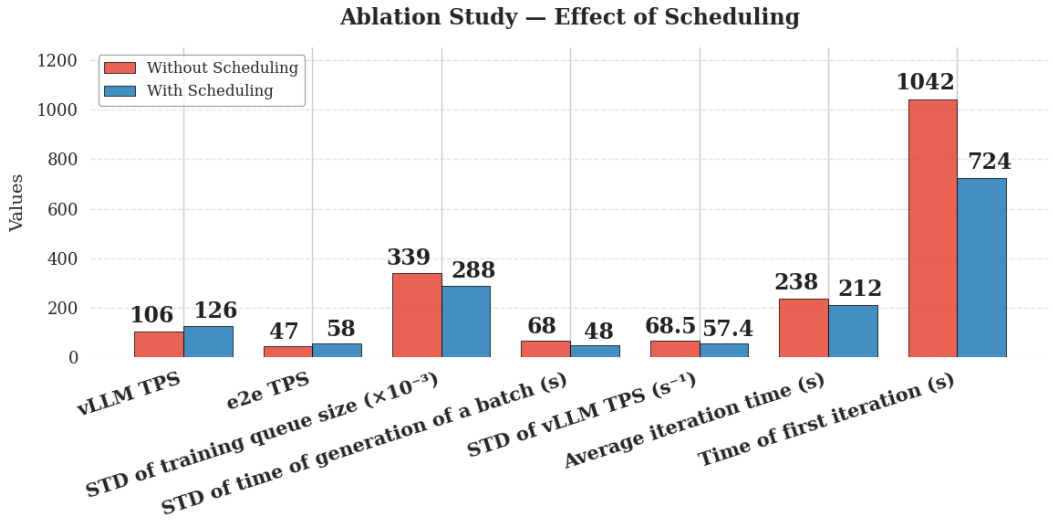


Figure 9: Training with and without scheduling on `Qwen3-30B-A3B` using DEEPSCALER, with 200 iterations per run. vLLM TPS measures generated non-padding tokens per second per generation NPU; end-to-end (e2e) TPS uses full iteration time and all NPUs. Mean iteration time includes startup, which is also reported separately. Queue STD is reported in 10^{-3} global batches.

F.6.1 Training with and without scheduling

Figure 9 compares training with and without scheduling on `Qwen3-30B-A3B` using DEEPSCALER. Both runs use the allocation, trainer/generation parallelism, and 16-prompt workload of the main MoE ASP run in Table 7, with eight responses per prompt. Prompt and response limits are 1024 and 8192 tokens, respectively. Both runs last 200 training iterations and differ only in whether scheduling is enabled. The scheduled run uses the default scheduler settings (Appendix E.4).

Enabling scheduling reduces mean iteration time from 238 to 212 s (10.9%) and increases end-to-end throughput from 47 to 58 tokens/s/NPU (23.4%). Scheduling also reduces the standard deviation (STD) of minibatch generation time from 68 to 48 s (29.4%) and that of Training Queue occupancy by 15.0%, indicating more consistent generation times and queue occupancy.

Considering generation alone, vLLM throughput, measured in generated non-padding tokens per second per generation NPU, increases from 106 to 126 tokens/s/NPU, while its STD decreases from 68.5 to 57.4 tokens/s/NPU. Unlike end-to-end throughput, this metric focuses on generation rather than the full training pipeline. Mean iteration time includes all 200 iterations, including the first (startup) iteration, which is also reported separately and decreases from 1042 to 724 s (30.5%).

F.6.2 Robustness of the scheduler to output length prediction quality

Here, we study the robustness of the scheduling pipeline to the quality of the output length predictions.

Conditions and fixed responses. We use 39,000 measured responses from each of `Qwen3-32B` and `Qwen3-8B`. For each model, a complete inference pass supplies fixed measured lengths ℓ_i. C0 forms fixed-size minibatches in original order; C1 sorts by predicted length before fixed-size batching; C2 applies dynamic programming to refine the sorted minibatch boundaries; C3 randomly shuffles the C2

Table 9: Local schedule metrics across controlled predictor qualities on three seeds. C2–C4 share minibatches and therefore share dispersion and padding rate.

		Dispersion (tokens)			Mean padding rate		
τ	MAE	C0	C1	C2–C4	C0	C1	C2–C4
`Qwen3-32B`							
1.00	0	2138	5	5	0.322	0.003	0.003
0.75	839	2138	820	828	0.322	0.176	0.175
0.64	1203	2138	1163	1163	0.322	0.226	0.224
0.50	1692	2138	1558	1563	0.322	0.273	0.271
0.25	2729	2138	2064	2057	0.322	0.330	0.328
0.00	3826	2138	2267	2253	0.322	0.348	0.347
`Qwen3-8B`							
1.00	0	2066	5	5	0.294	0.003	0.003
0.75	819	2066	783	783	0.294	0.154	0.152
0.64	1186	2066	1123	1123	0.294	0.206	0.204
0.50	1682	2066	1525	1525	0.294	0.250	0.251
0.25	2712	2066	1998	1998	0.294	0.296	0.295
0.00	3879	2066	2190	2190	0.294	0.313	0.313

minibatches; and C4 applies ASP's balanced ordering. C2–C4 have identical minibatch composition. Their local length metrics are therefore identical, while their batch-group loads can differ.

Scheduler configuration. Both models use the default scheduler settings (Appendix E.4). C0 and C1 use fixed-size minibatches of 32; C2–C4 allow sizes from 32 to 34. The experiment with simulated training in Appendix F.6.3 uses the same scheduler settings.

Prediction perturbations. Holding real output lengths ℓ_i fixed, we construct simulated predictions using

$$\widehat{\ell}_i = \ell_i \exp(\nu\varepsilon_i), \qquad \varepsilon_i \sim \mathcal{N}(0,1),$$

calibrating ν to achieve a target Kendall correlation $\tau(\widehat{\ell}, \ell)$. For each quality level and seed, all conditions receive the same prediction vector. We evaluate schedule metrics on the measured lengths, not on the predictions, and report means over three seeds. The nominal quality levels range from the oracle condition ($\tau = 1$) to the random-quality endpoint ($\tau = 0$). Because generation is stochastic, the oracle uses information from the measured responses and is an upper bound rather than an attainable predictor operating point.

The measured ranker correlations are 0.64 for `Qwen3-30B-A3B`, 0.44 for `Qwen3-32B`, and 0.58 for `Qwen3-8B`.

Metrics. For the jth minibatch, let $\mathcal{I}_j$ denote its set of prompt indices, and let $\bar{\ell}_j$ and $\ell_{\max,j}$ denote its mean and maximum measured response lengths. With N_B minibatches, we report

$$D = \frac{1}{N_B}\sum_{j=1}^{N_B} \mathrm{STD}\{\ell_i : i \in \mathcal{I}_j\}, \qquad P = \frac{1}{N_B}\sum_{j=1}^{N_B}\left(1 - \frac{\bar{\ell}_j}{\ell_{\max,j}}\right).$$

Here, D measures the average length dispersion within minibatches (in tokens), while P denotes the mean padding rate.

To assess load balance across consecutive batch groups, let L_g denote the total number of measured response tokens in group g. We report the coefficient of variation (CV), defined as $\mathrm{STD}(L_g)/\,\mathrm{mean}(L_g)$, and the group-load spread, defined as $\max_g L_g / \min_g L_g$.

Finally, we report prediction mean absolute error (MAE) in tokens alongside Kendall's τ.

Interpretation. Table 9 shows that sorting's local benefit decreases as prediction quality deteriorates and is already marginally negative at $\tau = 0.25$ on both models. The dynamic programming step

Table 10: Batch-group load CV for the same controlled sweeps as Table 9. Entries are three-seed means, evaluated on measured response lengths.

τ	C0	C1	C2	C3	C4
`Qwen3-32B`					
1.00	0.103	0.495	0.492	0.188	0.000
0.75	0.103	0.447	0.445	0.175	0.052
0.64	0.103	0.403	0.401	0.162	0.070
0.50	0.103	0.332	0.330	0.140	0.080
0.25	0.103	0.183	0.183	0.115	0.092
0.00	0.103	0.097	0.111	0.088	0.091
`Qwen3-8B`					
1.00	0.097	0.458	0.455	0.172	0.000
0.75	0.097	0.416	0.414	0.159	0.049
0.64	0.097	0.375	0.374	0.149	0.062
0.50	0.097	0.309	0.308	0.131	0.079
0.25	0.097	0.171	0.172	0.105	0.088
0.00	0.097	0.091	0.094	0.085	0.083

of ASP changes mean padding by at most 0.002 relative to C1 across these sweeps, and does not improve every condition. Its small average effect reflects changes to only a few boundaries among approximately 1,200 minibatches. Individual boundaries can nevertheless be costly: for models such as `Qwen3-30B-A3B`, we observed that misplaced boundaries could delay the generation of one minibatch by up to 12 minutes.

Table 10 shows the complementary ordering effect. For $\tau \geq 0.50$, C1/C2 have approximately three to five times C0's load CV. C4 keeps CV at or below C0 throughout the tested sweep and outperforms random shuffling at every tested level with $\tau \geq 0.25$. For $\tau = 0$, on the 32B model, C3 is slightly lower than C4 (0.088 versus 0.091); both remain below C0. Thus, the results support robust load balance in this sweep.

F.6.3 Real generation with simulated training

As described in Section 5.3, we evaluate C0–C4 using real `Qwen3-8B` generation with modeled training times. All five configurations use the same prompts, predictions, resources, token limits, and architecture. The predictor has a measured Kendall correlation of $\tau = 0.58$.

By keeping training times fixed, we can attribute differences in generation times, sample arrivals, and Training Queue behavior to scheduling alone.

To justify using fixed training times, we measure the variability of training-step durations in an ASP run with one TU node and three GU nodes. Over 19 post-warmup iterations, training-step durations have a coefficient of variation (CV) of only 5.0%, compared with 6.8% for end-to-end iteration times. This limited variability supports using fixed training durations to isolate the effects of rollout scheduling.

C4 achieves the lowest mean iteration time, 178 s versus 188 s for C0 (5.3% lower). The iteration-index/time Spearman correlations are -0.080, $+0.670$, $+0.510$, $+0.190$, and -0.075 for C0–C4, respectively. C4 therefore shows little to no monotonic drift, whereas the sorted C1/C2 schedules become considerably slower over successive iterations. Shuffling matches C4's queue STD (36 samples) and has a slightly lower queue maximum (104 versus 110 samples), but a higher mean time (190 s).

F.7 Main-run timing, queue, and response-length diagnostics

We additionally report runtime diagnostics from the complete `Qwen3-30B-A3B` experiment underlying Table 3. Its 215.2 s mean includes all 120 iterations, including the 809.7 s first iteration. The remaining 119 iterations average 210.2 s, with a fitted linear slope of $+0.044$ s per iteration and 20-iteration means between 197.0 and 221.2 s. These summaries indicate little sustained timing drift over the observed run. Figure 10 shows the complete trace and its post-startup variation.

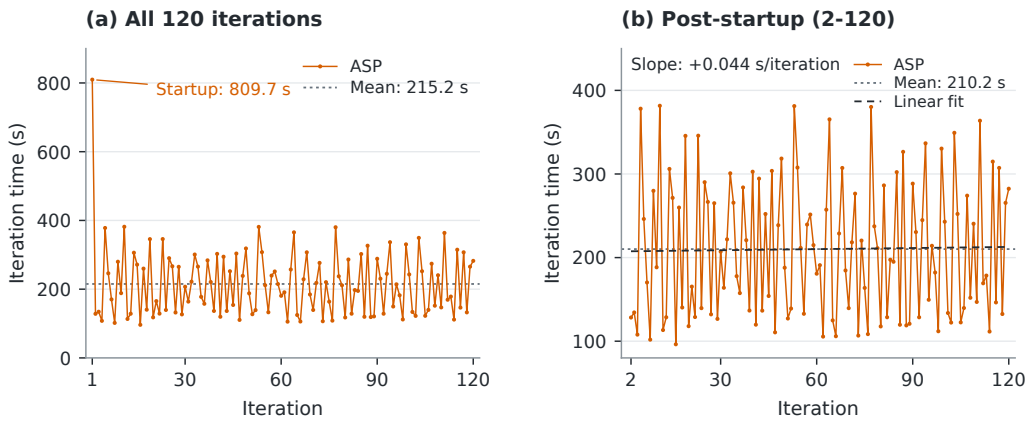


Figure 10: Iteration times for the complete `Qwen3-30B-A3B` run underlying Table 3. (a) All 120 iterations, including the 809.7 s startup; the mean is 215.2 s. (b) Iterations 2–120, averaging 210.2 s; the dashed linear fit has slope $+0.044$ s per iteration.

Across 16 generation actors, recorded prompt counts range from 112 to 140, with a CV of 6.4%. Queue depth measured at arrivals averages 0.39 global batches, with a 95th percentile of 0.75 and a maximum of 1.25 global batches, without progressive accumulation.

Across the 1,992 recorded prompts, mean response length increases from 5,637 tokens at policy versions 1–20 to 5,730 at versions 100–115 (1.6%). A linear trend in mean response length across 103 recorded versions has $R^2 = 0.013$. This supports limited mean-length drift in the observed run. Longer runs, curricula, or domain shifts may require renewed profiling or predictor updates.

F.8 Reproducibility notes

The experiments require access to Ascend 910B3 nodes, MindSpeed-RL, the Qwen3 model checkpoints, the datasets listed above, and the ASP scheduling and weight-flow implementation. The main sources of variability are the sampled prompts, generation stochasticity, scheduling predictor quality, and system load during generation and weight refresh. For reproducibility, we record the model, dataset, output-length cap, node allocation, scheduler configuration (b, κ, k), measured Kendall's τ, generation parameters, and measured timing breakdown for each run.

Appendix F.5 reports the performance and scalability configurations; Appendix F.6.1–F.6.3 give the ablation settings and protocols. Supporting ASP timing and queue diagnostics are reported in Appendix F.7.

G Limitations and discussion

ASP has several limitations, which suggest directions for future improvement. First, our prototype is implemented in the MindSpeed-RL ecosystem on Ascend 910B3 NPUs. This gives a controlled same-hardware comparison against the hybrid pipeline from which ASP is derived, and our ablations isolate the effect of the scheduling mechanism. Direct comparisons with StreamRL [5], AsyncFlow [6], AReaL [8], and Laminar [9] were not conducted, as establishing comparable configurations on our specific Ascend NPU stack would require substantive integration and validation. Such comparisons could confound system differences with implementation quality. We partially address this limitation by adding a verl-based disaggregated baseline in Section 5, as verl is an actively maintained open-source RL post-training framework that supports separated rollout and training configurations [13, 14]. This comparison should be interpreted as measuring the benefit of ASP over a disaggregated implementation without ASP's scheduling layer.

Second, the output-length predictor is fixed during training. This design keeps ASP's scheduling pass simple and avoids online overhead, while still delivering scheduling gains in our runs. Longer

training runs, stronger policy shifts, or curriculum changes may benefit from periodically refreshing the predictor or combining offline scheduling with lightweight online correction.

Third, our main end-to-end evaluations focus on Qwen3-30B-A3B and Qwen3-32B, complemented by scheduling ablations and predictor-sensitivity experiments on Qwen3-8B. Larger models, longer context windows, and workloads with more extreme output-length distributions may amplify the benefits of multi-instance rollout and scheduling, but they should be validated directly. Conversely, short-output workloads may see smaller gains because rollout variability is less dominant.

Fourth, our available NPU resources are limited to 128 Ascend NPUs in our experiments. Thus, our scalability test stops at the 128-NPU point. Although the trend is already clear, larger-scale experiments would provide a stronger evaluation, especially for studying the impact of crossing super-node regions or leaf switches.

Finally, ASP assumes that the prompt set is known before rollout begins, since the scheduler performs an offline preprocessing pass. This matches many RL post-training settings where prompts are sampled from a fixed dataset, but it is less directly applicable to fully online data-generation pipelines. Extending the same ideas to streaming prompts would require an online version of length-aware minibatching and balanced dispatch.

Beyond the controlled experiments reported above, ASP has also been used in an internal telecom-domain RL post-training workflow for `Qwen3-30B-A3B`. In that deployment, the overall training period was reduced from more than three months to about two months, and the resulting domain-enhanced model was used for telecom applications such as knowledge answering and tool calling. Because the workflow, data, and deployment details are confidential, we report this only as qualitative evidence of practical applicability, not as part of the controlled experimental comparison.

H Generative AI assistance

We used generative AI tools to assist with writing, translation, revision, and language polishing; identifying and discussing relevant literature; and code implementation and debugging. These tools were not used for research ideation, methodology design, or conducting or analyzing the experiments. The authors reviewed the AI-assisted material and take full responsibility for the final manuscript, code, and reported results.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes].

   Justification: The abstract and introduction describe ASP's contributions and scope, distinguishing complete-system performance gains from the contribution of scheduling. The experimental claims are supported by Section 5 and Appendix F, with the policy-lag analysis provided in Appendix D.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes].

   Justification: Appendix G discusses the limitations of our evaluation scope and baseline coverage, the fixed output-length predictor, and the requirement that prompts be available before offline scheduling.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A].

Justification: The paper primarily focuses on system design and empirical evaluation rather than novel theoretical guarantees. The proposed scheduling algorithms rely on well-established optimization techniques and theoretical foundations, including dynamic programming and simulated annealing, which are described in detail in Section 3 and Appendix E.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes].

Justification: Section 5, Appendix B, and Appendix F detail the information and settings that are needed for reproducing the experiments.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No].

   Justification: The datasets used in experiments are all open-source and can be downloaded from HuggingFace, which is declared in Appendix F. However, we are not able to provide experiments requiring code as it is being used in confidential industrial scenarios. Based on the Appendix B, Appendix C, Appendix E, and Appendix F, ASP could be fully reproduced step by step.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes].

   Justification: The paper specifies all the training and test details in Appendix F.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No].

Justification: We report arithmetic iteration means for complete-system comparisons, three-seed means for controlled predictor-quality sweeps, and within-run variability diagnostics. We do not report confidence intervals or statistical significance tests for the complete-system comparisons.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes].

   Justification: The paper provides sufficient information on the computer resources in Appendix F.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes].

   Justification: The research in this paper complies with the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.

- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A].

    Justification: This work focuses on improving the computational efficiency of existing LLM reinforcement learning pipelines. It does not introduce new model capabilities, datasets, or applications with direct societal implications. While more efficient LLM training may have indirect positive or negative societal effects, these depend on downstream applications rather than the proposed scheduling method itself. We therefore do not identify any specific societal impacts arising directly from this work.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A].

    Justification: The models and data used in this paper do not pose such risks.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes].

Justification: All third-party models, datasets, and software frameworks used in this work are appropriately cited, with their original sources identified in the references and experimental details. These resources are used in accordance with their applicable licenses and terms of use.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [N/A].

Justification: The paper does not release new assets, as the source code is used in confidential industrial scenarios.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A].

Justification: This paper does not involve crowdsourcing or research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.

- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A].

    Justification: This paper does not involve any human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes].

    Justification: As part of its core design, ASP uses a fine-tuned LLM for output-length prediction, as described in Section 3 and Appendix F in Step 1 of the pipeline. Separately, generative AI assistance with writing, literature exploration, and coding is disclosed in Appendix H. This assistance did not extend to research ideation, methodology design, or conducting or analyzing the experiments.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.