TAIL-W4: PREDICTIVE PRECISION SCHEDULING FOR LONG-TAIL RL ROLLOUTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning (RL) post-training can be dominated by rollout, where synchronous execution waits for a few long responses to finish. Weight-only 4-bit quantization (W4) is promising for addressing those small-batch tails because it reduces the cost of loading model weights. However, naïvely applying W4 in RL post-training does not guarantee faster RL steps: quantized sampling can generate more tokens, increasing both decoding and downstream RL work. To address this challenge, we propose TAIL-W4, a rollout precision scheduler that chooses when to switch from BF16 to W4 by jointly predicting response lengths and execution costs. Reserving W4 for the tail can mitigate token inflation, but switching too early risks extra work and switching too late misses acceleration. Fixed schedules require repeated trials to tune and cannot fully adapt to changing training workloads. TAIL-W4 calibrates estimates of token inflation, updates them from training rollouts, and periodically predicts the benefit of switching to W4. On Qwen3.5-4B, TAIL-W4 achieves $1.38\times$ rollout and $1.26\times$ measured RL-step speedup over BF16, versus $1.12\times$ and $1.03\times$ for uniform W4 with the same quantized weights.

1 INTRODUCTION

Reinforcement learning (RL) has become a major paradigm for LLM post-training, improving mathematical reasoning (Shao et al., 2024), code generation (Guo et al., 2025), and search-based question answering (Jin et al., 2025) through exploration and reward feedback. Unlike supervised training on known sequences, RL must first sample responses in a *rollout*: autoregressive decoding for a batch of requests, often with long reasoning chains. In synchronous RL, this phase resembles batched serving and waits for every response to finish. Rollout can dominate step time, with a *long tail* in which only a few requests remain active for many decoding iterations.

Quantization is well suited to this small-batch tail because it reduces traffic of model weights. Each decoding iteration loads model weights to generate few tokens; compressing those weights can reduce memory-bound latency. Prior work accelerates RL with NVFP4 (QeRL (Huang et al., 2026)) and FP8 (Jet-RL (Xi et al., 2026)), with the latter also exploiting low-precision Tensor Core computation. We focus on weight-only W4A16—INT4 weights with 16-bit activations (Lin et al., 2024; Qwen Team, 2024)—using BF16 activations in our setup. Advances such as AutoRound improve low-bit accuracy (Cheng et al., 2024; 2025), while QaRL demonstrates effective W4A16 RL with sequence-level Trust Band Policy Optimization (Gu et al., 2026). We investigate when to use W4A16 within a rollout to improve complete RL-step efficiency.

We find that existing approaches face three core limitations when used for RL with parameter-efficient fine-tuning (PEFT). Kernel execution, adapter overhead, and changes in generated work can limit W4's efficiency in PEFT rollouts. First, quantization formats offer different accuracy, memory, and execution tradeoffs (Frantar et al., 2023; Lin et al., 2024; Dettmers et al., 2023): our tested AWQ path incurs additional processing overhead, while NF4 dequantization can slow rollout (Huang et al., 2026). Second, LoRA adds a BF16 forward branch whose overhead can dilute base-model speedups. Third, 4-bit quantization can retain task accuracy (Frantar et al., 2023; Lin et al., 2024), yet its noise can change sampling behavior (Huang et al., 2026; Lian et al., 2026). We observe longer responses, consistent with prior reasoning-inference results (Lian et al., 2026). We show that the resulting token inflation can erode efficiency throughout the RL step: it keeps more requests active during decoding

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

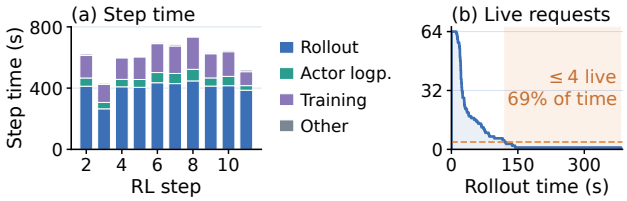

Figure 1: The rollout tail in ten BF16 steps of Qwen3.5-9B on GSM8K ($B = 64$, 24K response cap). (a) Rollout takes 65% of time excluding reference replay. (b) The step with the highest rollout share has at most four live requests for the final 69% of rollout.

and adds downstream log-probability and training work (§3.3). These costs can outweigh the savings from compressed weights.

Reserving W4 for the rollout tail could mitigate token inflation, but choosing when to switch remains difficult. With both weight copies resident, rollout can begin in BF16 and switch the remaining requests to W4. Switching early risks longer responses; switching late leaves less work to accelerate. A fixed rule—switch after K decoding steps or when at most t requests remain—can work well, but selecting it can involve lengthy trials. The same rule also applies across rollouts whose request lifetimes change as training progresses.

To this end, we present TAIL-W4, a scheduler that selects a precision switch by jointly predicting the work it would generate and the time to process that work. W4 continuation measurements account for the BF16 prefix; latency profiles account for the shrinking live batch; and downstream token cost accounts for the later RL stages. The scheduler compares future switch points, updates response-length predictions after each rollout, and revisits its decision as requests finish. These choices address the changing workload that a fixed rule leaves unmodeled. An efficient GPTQ-format W4 path and a shared-adapter LoRA optimization reduce execution overhead under this schedule.

TAIL-W4 translates more of the rollout gain into complete-step savings in our evaluation. On Qwen3.5-4B, it achieves $1.38\times$ rollout and $1.26\times$ measured RL-step speedup over BF16, versus $1.12\times$ and $1.03\times$ for uniform W4 with the same quantized weights. Calibration is a one-time cost, and asynchronous scheduler updates are hidden by GPU training work in our measured runs. On a selected set of hard math problems, TAIL-W4 achieves held-out accuracy comparable to BF16 training at a lower cumulative training cost.

Our main contributions are:

- We characterize the kernel, adapter, and workload obstacles to efficient weight-only quantization in PEFT rollouts (§3).
- We improve the LoRA forward execution path for the common case of a single shared adapter, accelerating both BF16 and W4 rollouts (§3.1).
- We develop and evaluate an adaptive precision scheduler that jointly predicts generated work and execution cost to improve RL-step efficiency (§4–5).

2 BACKGROUND

2.1 RL POST-TRAINING, ROLLOUT, AND ITS LONG TAIL

A synchronous RL step generates responses before processing them for learning (Fu et al., 2025). *Rollout* autoregressively samples responses for B requests; *log-probability computation* evaluates them under the policy; and *training* uses rewards and these probabilities to update the policy (Shao et al., 2024). The latter stages, log-probability computation and training, wait for the last response to finish. Long responses can make rollout a substantial part of step time (Hu et al., 2026). In Figure 1a, our example spends 65% of step time on rollout. In distributed settings, these stages wait for the last response across all rollout ranks, which can amplify the rollout bottleneck (Fu et al., 2025).

In rollout, variation in response lengths creates a long tail of small-batch decoding (Hu et al., 2026). As requests finish, the *live batch* $b \leq B$ shrinks, while a few responses may continue to the *response cap*, the maximum number of output tokens allowed per response. In the step shown in Figure 1b, at

most four requests remain for the final 69% of rollout time. Each iteration then reads model weights to produce only a few tokens, making reduced weight traffic particularly valuable.

2.2 Weight-only quantization

Weight-only quantization compresses model weights while retaining high-precision activations. We use W4A16 (W4 for short): 4-bit integer weights and 16-bit BF16 activations, with an unquantized KV cache. Weight storage is four times smaller than BF16 before quantization metadata. During computation, the quantized weights are dequantized to BF16 on the fly, then multiplied with the BF16 activations using BF16 arithmetic. GPTQ and AWQ report limited accuracy loss with 4-bit weights (Frantar et al., 2023; Lin et al., 2024); QLoRA demonstrates fine-tuning comparable to 16-bit baselines using the NF4 weight format (Dettmers et al., 2023).

2.3 LoRA on a quantized base

LoRA (Hu et al., 2022) reduces trainable parameters by freezing a base matrix W and learning a low-rank update:

$$Y = XW^\top + \frac{\alpha}{r}(XA^\top)U^\top, \tag{1}$$

where $A \in \mathbb{R}^{r\times d_{\text{in}}}$ and $U \in \mathbb{R}^{d_{\text{out}}\times r}$ are trainable, $r \ll \min(d_{\text{in}}, d_{\text{out}})$ is the adapter rank, and α sets its scale. The base and adapter branches process X independently and sum their outputs. QLoRA (Dettmers et al., 2023) combines LoRA with a frozen 4-bit base, propagating gradients through its dequantized weights to train the adapter.

In our RL setup, BF16 and W4 rollout paths share the same trainable BF16 adapter and use different frozen base-weight copies. Adapter updates are copied to the rollout engine after each training step; switching precision changes the base-weight path while retaining the shared adapter.

3 Challenges in Translating W4 Savings to RL

Weight-only quantization (WOQ) can reduce decoding cost while increasing the work performed by an RL step. We first examine kernel and adapter overhead, then identify where W4 benefits the rollout tail (Section 3.1). Even with efficient kernels, quantization can increase the number of sampled tokens (Section 3.2), eroding gains in both rollout and downstream phases (Section 3.3). We then formulate precision switching as a scheduling problem (Section 3.4). Unless stated otherwise, measurements use Qwen3.5-9B with a rank-16 adapter on one A100-80GB.

3.1 Making the QLoRA rollout fast

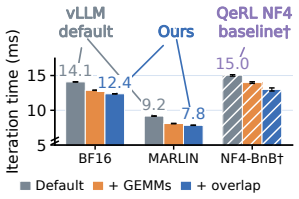

Figure 2: Decoding latency with one live request on Qwen3.5-9B; bars show successive LoRA optimizations.

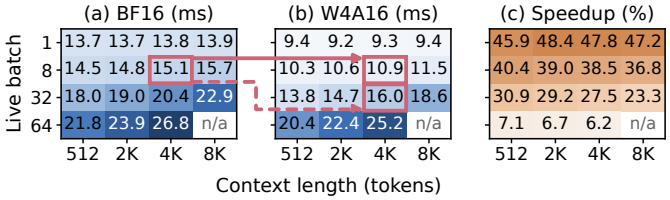

Figure 3: Decoding latency and W4A16 speedup across live batch sizes and context lengths on Qwen3.5-9B. Arrows illustrate a hypothetical workload shift in §3.3; n/a denotes cells beyond the KV budget.

Dequantization overhead varies across quantization methods and can limit the decoding benefit of compressed weights. QeRL (Huang et al., 2026) reports that its NF4 QLoRA baseline slows rollout because weights must be unpacked and mapped to floating-point values before matrix multiplication. We evaluate NF4 with BitsAndBytes as well as integer W4 in GPTQ format with MARLIN kernels (Frantar et al., 2025). With vLLM's default LoRA path and one live request, MARLIN reduces median decoding-iteration time from 14.09 ms to 9.17 ms, a $1.54\times$ speedup over BF16 (Figure 2).

Accelerating the base-weight branch exposes the overhead of the BF16 LoRA adapter (Section 2.3). At small live batches, vLLM's Punica-based Triton kernels (Chen et al., 2024) may expose too little parallel work to occupy the GPU's streaming multiprocessors (SMs). For a single-adapter workload, packed PyTorch GEMMs mitigate this inefficiency and avoid multi-adapter metadata preparation. We also overlap the independent base and adapter branches on separate CUDA streams. At batch one, packed GEMMs reduce W4 iteration time by 1.07 ms; stream overlap saves another 0.26 ms, reaching 7.84 ms. The corresponding BF16 reductions are 1.21 ms and 0.50 ms, reaching 12.37 ms (Appendix A.1).

With this optimized forward path, W4 is most effective at small live batches, including long contexts. On the hybrid-attention Qwen3.5-9B model, batch-one speedup remains 1.46–$1.49\times$ across contexts from 512 to 16K tokens (Figure 3 shows a subset through 8K). At batch 64, the gain falls to roughly 6%. Larger batches perform more arithmetic per weight read, while each additional request adds attention work and unquantized KV-cache traffic, leaving a smaller fraction of decoding time that weight compression can save.

3.2 Quantization changes token sampling during rollout

Quantization can inflate the number of sampled tokens. Lian et al. (2026) report longer reasoning traces under low-bit weight quantization, even when answers remain correct. QeRL (Huang et al., 2026) also finds that quantization noise changes sampling by increasing policy entropy and encouraging exploration during RL. We examine how the resulting response lengths change the rollout workload.

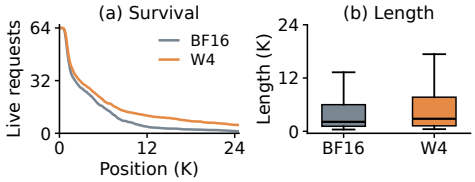

Figure 4: Response survival (a) and length (b) under BF16 and uniform W4 on the same prompts: Qwen3.5-9B at B=64 with a 24K cap. W4 samples 44% more tokens and reaches the cap four times as often (7.8% vs 2.0%).

Uniform W4 generates more tokens and keeps more requests active in our matched-prompt rollouts (Figure 4). Across 960 responses per precision from 15 steps ($B = 64$, 24K cap), W4 produces 44% more response tokens and raises median response length from 2,125.5 to 2,818 tokens, a 32.6% increase. At the same decoding position, more requests remain unfinished under W4 than under BF16, as shown by the higher survival curve in Figure 4a.

3.3 Token inflation erodes efficiency gains throughout the RL step

Token inflation can shift rollout into a larger-batch regime that offsets W4's per-iteration savings. Consider the red box in Figure 3a: BF16 takes 15.1 ms with eight live requests at 4K context. If quantization preserved the live batch, W4 would take 10.9 ms (solid arrow). If longer responses instead left 32 requests active, W4 would take 16.0 ms (dashed arrow), exceeding BF16's 15.1 ms.

Additional sampled tokens also increase downstream log-probability and training time. Both grow approximately linearly with the number of processed tokens (see Appendix A.2), with fitted slopes summing to 0.694 s per thousand tokens.

3.4 Scheduling precision for complete RL-step time

These observations lead to a simple intuition: *we could generate most tokens in BF16 and then apply W4 quantization to the small-batch long tail.* This limits exposure to token inflation, but switching too early risks extra tokens, while switching too late leaves less work to accelerate. We therefore need to decide when to switch. Let π be a precision schedule. For scheduling, we predict its complete RL-step time as

$$\widehat{T}_{\text{step}}(\pi) = \widehat{T}_{\text{roll}}(\pi) + \lambda\, \widehat{N}_\pi + c, \tag{2}$$

where $\widehat{T}_{\text{roll}}(\pi)$ predicts decoding and precision-switching costs, $\widehat{N}_\pi$ predicts the number of generated response tokens, λ is the fitted downstream cost per token, and c collects schedule-independent costs, including prompt processing. The objective is to minimize expected complete-step time, accounting for both the execution cost and the workload induced by π. This model guides scheduling; experimental RL-step times are measured directly.

4 PREDICTIVE TAIL-W4

TAIL-W4 uses predicted response lengths and execution costs to choose when a rollout should switch to W4. Fixed schedules can be costly to evaluate and do not revise their rules as the workload changes (Section 4.1). To address these challenges, TAIL-W4 uses calibration data to dynamically choose a precision-switching policy for each RL step (Sections 4.2–4.3) and updates its internal state as training progresses (Section 4.4). Precomputed decisions keep this search off the decoding path (Section 4.5).

4.1 WHY FIXED SCHEDULES CAN BE COSTLY TO TUNE

A fixed rollout schedule is defined as follows: apply quantization to model base weights after K decoding steps or when at most t requests remain. Such fixed schedules are simple to apply, but comparing candidates can be costly: RL samples responses at nonzero temperature for exploration (Shao et al., 2024), so response lengths can vary across random seeds. vLLM also does not guarantee reproducible inference by default (vLLM Team, 2026). Both generated work and execution time can therefore vary between trials. To reduce the effect of step-to-step variation, we average times over dozens of RL steps (Section 5.2). For example, on Qwen3.5-4B with $B = 32$ and a 24K cap, the measured BF16 step takes 302 s (Table 1). At that rate, one warm-up and ten measured steps would take about 55 minutes per candidate.

Policy updates can also change which fixed threshold is useful. As response lengths evolve, a previously selected constant may become less effective, motivating further trials or an adaptive decision.

4.2 ONLINE PREDICTION FROM CALIBRATION

TAIL-W4 combines response-length predictions, decoding profiles, and downstream token cost to estimate the effect of a switch. Together, they estimate the remaining step cost in Eq. 2. Decisions use state (f, b, q): response frontier f, live batch b, and median prompt length bucket q. The frontier is the longest live response; requests start together with approximately aligned decoding positions.

Remaining response lengths. Calibration estimates how long requests remain active under each precision. In the Qwen3.5-4B setup, we collect 256 BF16 responses before training, plus W4 continuations from BF16 prefixes of various lengths. Let $h_g(x)$ be the probability of finishing in the next Δ tokens, conditional on reaching position x, for calibration group g. A request alive at f survives another k bins with probability

$$S_g(k \mid f) = \prod_{j=0}^{k-1} \big(1 - h_g(f + j\Delta)\big), \qquad \bar{b}_{f+k\Delta} = b\, S_g(k \mid f). \tag{3}$$

Thus survival predicts the remaining live batch and, by summing over future bins, the remaining token count.

W4 survival depends on how much of the response was generated in BF16 and which requests reached the switch. A uniform-W4 calibration misses this conditioning. We therefore collect separate continuation groups at representative BF16 frontiers c. Appendix A.3 details the group selection and survival calculation for candidate switches.

Decoding latency. Decoding profiles account for how W4's benefit varies as requests finish. The profile $T_p(L, b)$ gives time per decoding iteration at precision p, context length L, and live batch b (Figure 3). Combining it with predicted survival estimates remaining rollout time as the batch drains. Appendix A.3 gives the calculation.

Downstream cost. Including downstream work accounts for the cost of extra tokens beyond rollout. The approximately linear relation in Section 3.3 lets the scheduler estimate remaining log-probability and training work as $\lambda\widehat{N}$, where $\widehat{N}$ is the predicted number of future response tokens. Completed work and prompt-processing costs are common to candidate switches and do not affect their ranking.

4.3 SEARCHING FOR THE LARGEST PREDICTED SAVING

Comparing future switch points can reveal savings that an immediate switch would miss. Switching early risks token inflation, while switching late leaves less work to accelerate. TAIL-W4 therefore compares future switches even when switching now looks profitable. For each candidate frontier $s \geq f$, we predict the cost of staying in BF16 until s and using W4 thereafter:

$$J(s \mid f, b, q) = \widehat{T}_{\text{roll}}(s \mid f, b, q) + \lambda\, \widehat{N}(s \mid f, b, q). \tag{4}$$

Minimizing J maximizes predicted savings against the no-switch baseline $s = \infty$.

TAIL-W4 switches now when $s = f$ has the lowest predicted cost and strictly improves on staying in BF16. Otherwise, it continues in BF16 and repeats the lookup at the next frontier with the observed live batch. However, probabilistic predictions (Section 4.2) inevitably incur errors, so scheduling decisions can be suboptimal.

4.4 ADAPTIVE UPDATES DURING TRAINING

Online updates refine survival probabilities estimated from a limited calibration budget. They also help the scheduler follow response-length changes during training. TAIL-W4 refines its initial calibration by updating the survival probabilities defined in Section 4.2 using a *proportional-hazards model* (Cox, 1972). For BF16 or a W4 continuation group g, let $S_g^0(k \mid f)$ be the calibrated probability of remaining unfinished for another k bins after frontier f. After rollout t, a learned positive factor $a_{g,t}$ adjusts it:

$$S_{g,t}(k \mid f) = \left[S_g^0(k \mid f)\right]^{a_{g,t}}. \tag{5}$$

A factor below one predicts longer responses; a factor above one predicts shorter responses. We fit these factors to observed completions and survival intervals, giving more weight to rollout statistics collected under recent policies. Appendix A.4 gives a detailed explanation of the update rule and its assumptions.

4.5 SCHEDULER EXECUTION AND OVERHEAD

TAIL-W4 hides online scheduling work behind GPU execution. After each rollout, a CPU process updates its state and rebuilds a decision table while the GPU computes log-probabilities and trains. The table stores the planned switch frontier for each state (f, b, q), so decoding only performs table lookups. Section 5.4 visualizes these decisions; Algorithm 1 in Appendix A.3 gives the pseudocode for building and using the table. Every updated table was ready before the next rollout in our measurements (Section 5.6).

We use one vLLM engine to keep both weight copies and CUDA graphs resident. Appendix A.5 evaluates prefix-state reuse at the switch.

5 EVALUATION

We evaluate TAIL-W4 on complete RL-step time (rollout, log-probability computation, and training) against BF16, uniform W4, and fixed precision schedules (Section 5.2). We then examine the components responsible for its gains and visualize how its decisions adapt (Sections 5.3–5.4). We extend the evaluation to distributed rollout (Appendix A.6), measure cumulative training cost and evaluate held-out checkpoint accuracy (Section 5.5), and report system overhead (Section 5.6).

5.1 SETUP

Configuration. We run GRPO on GSM8K with Qwen3.5-4B, Qwen3.5-9B, and Phi-4-mini-reasoning, using verl, Megatron-Core, and vLLM on A100-80GB GPUs unless stated otherwise. Qwen uses AutoRound W4 checkpoints; Phi uses a GPTQ checkpoint from llm-compressor. W4 weights use MARLIN; the output head remains BF16, with a shared rank-16 adapter. Other components use BF16; under TAIL-W4, only the rollout tail uses W4 base weights. Policies use the same prompt order and independently sample at temperature 1.0. TAIL-W4's calibration precedes training; online updates start with the run. Single-worker runs use eight prompts with four responses each ($B = 32$), with response caps given in Table 1.

Baselines. *BF16* and *uniform W4* use one precision throughout rollout. *Fixed* K switches after K decoding iterations; *threshold* t switches when at most t requests remain. We sweep these constants on Qwen3.5-4B (Figure 6). TAIL-W4 updates response-length predictions and revisits the switch decision during decoding (Sections 4.3–4.4).

Measurements. Rollout, downstream, and complete RL-step times are measured directly. Stochastic responses change both generated work and execution time. We report mean times over steps 9–48 to reduce startup effects (Table 1).

Table 1: Measured RL-step results on GSM8K ($B = 32$; 24K response cap for Qwen3.5, 4K for Phi-4-mini). Speedups are ratios of mean times (seconds in parentheses). ΔReward is relative to BF16. Tokens gives the sequence-token change from BF16; W4 exp. is the fraction of response tokens sampled under W4. Bold marks the lowest mean step time.

	Policy	Rollout	Downstream	Step speedup	Tokens	W4 exp.	ΔReward
4B	BF16	1.00× (220 s)	1.00× (82 s)	1.00× (302 s)	+0%	0%	0.859 (abs.)
	Uniform W4	1.12× (197 s)	0.84× (97 s)	1.03× (294 s)	+20%	100%	−0.014
	TAIL-W4	1.38× (160 s)	1.03× (79 s)	**1.26× (239 s)**	−3%	33%	−0.001
9B	BF16	1.00× (261 s)	1.00× (98 s)	1.00× (359 s)	+0%	0%	0.866 (abs.)
	Uniform W4	1.00× (261 s)	0.67× (147 s)	0.88× (408 s)	+53%	100%	−0.030
	TAIL-W4	1.19× (219 s)	0.89× (110 s)	**1.09× (329 s)**	+13%	39%	−0.016
Phi-4 mini	BF16	1.00× (38 s)	1.00× (21 s)	1.00× (58 s)	+0%	0%	0.921 (abs.)
	Uniform W4	1.12× (33 s)	0.89× (23 s)	1.02× (57 s)	+16%	100%	−0.043
	TAIL-W4	1.23× (31 s)	0.97× (21 s)	**1.12× (52 s)**	+4%	57%	−0.024

5.2 END-TO-END RL-STEP SPEEDUP

TAIL-W4 achieves 1.09–1.26× mean RL-step speedup over BF16 across all three models, with smaller mean reward drops than uniform W4 (Table 1). Uniform W4 offers little step-level benefit and even slows Qwen3.5-9B: extra sampled tokens consume its decoding savings and add downstream work. Delaying W4 reduces the extra sampled tokens, while retaining acceleration in the small-batch tail.

Figure 5 illustrates this tradeoff in one rollout. Before the switch, TAIL-W4's live-request trajectory stays close to BF16, avoiding the early token inflation seen under uniform W4. After switching, the remaining requests generate additional tokens and finish at a later decoding position than BF16. Faster W4 decoding nevertheless yields a shorter rollout. Appendix A.7 shows the trajectories across all 40 evaluated rollout steps.

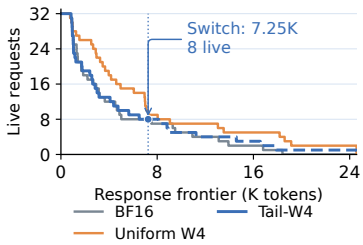

Figure 5: Qwen3.5-4B, step 30: switch at 7,250 tokens with eight live requests. Dashed blue marks W4; baselines run independently.

Fixed schedules can recover much of this benefit while quantizing only part of the rollout, but finding them requires costly trials (Figure 6). Although the broad plateau contains useful choices, each fixed-schedule bar requires about four GPU-hours of profiling, so even a few trials take hours. The 12-candidate sweep cost 46.69 GPU-hours. This cost limits the sweep to fewer candidates on the other models (Appendix A.11). TAIL-W4 avoids this per-threshold search and achieves a mean step time close to that of the best fixed arm in this sweep. It requires initial calibration, but subsequent updates overlap GPU work (Section 5.6), making adaptation inexpensive during training.

We also extend our end-to-end measurements to distributed training. On Qwen3.5-4B with eight data-parallel workers (DP8), TAIL-W4 achieves a measured 1.25× RL-step speedup over BF16. Appendix A.6 details these distributed results and additional rollout configurations.

5.3 WHICH PARTS OF THE SCHEDULER MATTER

TAIL-W4 combines four components to guide its decisions: online length updates (Section 4.4), downstream token cost and decoding latency profiles (Section 4.2), and replanning from the current rollout state (Section 4.3). We disable each in turn to test whether its effect matches the design intuition (Table 2).

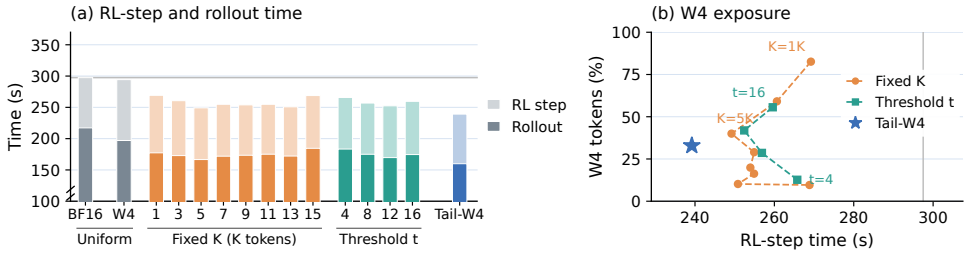

Figure 6: **Completed schedule sweep.** Qwen3.5-4B ($B = 32$, 24K response cap). (a) Mean measured RL-step time (light) with rollout time shaded darker. (b) Percentage of tokens generated with W4 versus step time; dashed lines connect increasing K or t, and the star marks TAIL-W4. Light-gray lines mark BF16 step time in both panels. Every arm uses 40 steps (9–48).

Table 2: Qwen3.5-4B ablations ($B = 32$, 24K cap, steps 9–48). Columns follow Table 1: speedups and ΔReward are relative to the same BF16 run; Uniform W4 is included as a reference.

Variant	Rollout	Downstream	Step speedup	Tokens	W4 exp.	ΔReward
Uniform W4	1.12× (197 s)	0.84× (97 s)	1.03× (294 s)	+20%	100%	−0.014
TAIL-W4	1.38× (160 s)	1.03× (79 s)	**1.26× (239 s)**	−3%	33%	−0.001
No downstream cost	1.30× (170 s)	1.00× (82 s)	1.20× (252 s)	+1%	40%	−0.007
No online update	1.24× (177 s)	0.92× (89 s)	1.14× (266 s)	+9%	62%	−0.010
Constant TPOT	1.31× (169 s)	0.98× (84 s)	1.20× (252 s)	+3%	48%	+0.004
No state replanning	1.30× (170 s)	0.99× (83 s)	1.20× (252 s)	+1%	36%	+0.003

Learning response lengths. Online updates revise the initial length predictions using observed rollouts. Without them (*No online update*), the scheduler retains its early switch and generates more tokens. Step speedup falls from 1.26× to 1.14×, the largest mean loss among the ablations.

Accounting for all work. Downstream token cost helps the scheduler weigh rollout savings against later RL work. Ignoring it (*No downstream cost*) makes early switching look more attractive because extra tokens are charged only for decoding. Step speedup falls to 1.20× when their cost is overlooked.

Knowing when W4 is fast. Batch- and context-dependent latency helps identify where W4 is useful. *Constant TPOT* obscures this variation, leading to earlier switching and 2.9% more tokens than BF16.

Reacting to the current rollout. Replanning uses the requests that remain active to revise the switch. *No state replanning* keeps the frontier chosen at rollout start. The precision-switching point K, measured in decoding steps, has the same median as TAIL-W4. Without state replanning, step speedup reaches only 1.20×: TAIL-W4 better accounts for changes in live batch size throughout each rollout.

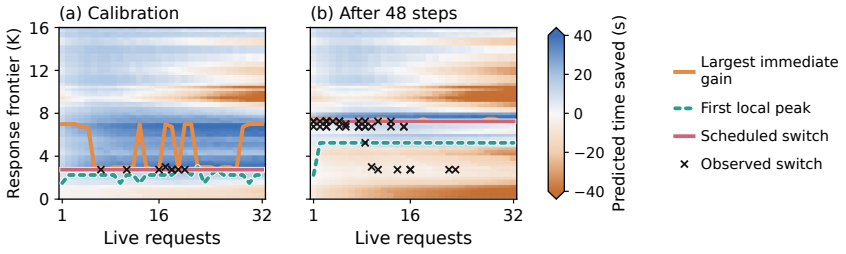

Figure 7: Qwen3.5-4B's switch decisions, (a) at calibration and (b) after 48 training steps. Each cell predicts the step time saved by switching now, relative to staying in BF16: blue is faster and orange slower (colors saturate at ±40 s). Red curves show schedule snapshots; crosses mark actual switches under evolving schedules in steps 1–8 and 9–48, respectively.

5.4 HOW DECISIONS ADAPT DURING TRAINING

Online observations update TAIL-W4's switching decisions. In the 4B run, revised response-length predictions move the scheduled frontier from 2.75K to 7.25K tokens, retaining BF16 for more of the rollout (Figure 7). Across multiple runs with the same calibration, TAIL-W4 makes similar adjustments in response to online observations (Appendix A.8).

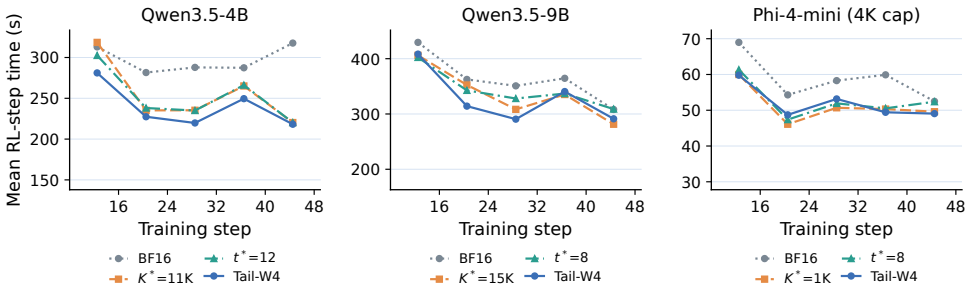

Figure 8: RL-step time over training steps; points average eight steps.

We compare TAIL-W4 with fixed-K and threshold-t schedules selected greedily from the tested candidates. For each model, we select K^* and t^* separately to minimize mean RL-step time over steps 1–8 of each candidate's run, then fix them for steps 9–48. In this window, TAIL-W4 achieves the best or competitive RL-step time (Figure 8). Some fixed choices are faster: TAIL-W4 aims to maintain competitive performance, without guaranteeing the best switch at every step.

5.5 LEARNING BEHAVIOR AND END-TO-END TRAINING EFFICIENCY

TAIL-W4 keeps the rollout prefix in BF16 and confines quantized sampling to the tail, reducing exposure to quantization-induced rollout–training mismatch. We train Phi-4-mini for 32 steps on 211 selected Big-Math questions (Open R1, 2025), using FSDP2 with DP4 and a 12K response cap. TAIL-W4 uses token-level truncated importance sampling (TIS, threshold 2). Both adapters are evaluated with the same BF16 base on 256 held-out questions with two responses each (Appendix A.11).

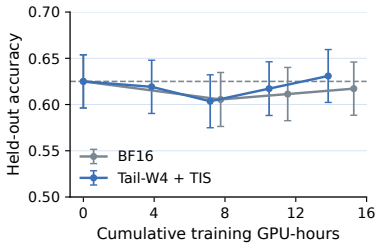

Figure 9: Phi-4-mini held-out accuracy versus cumulative training GPU-hours. Dashed: initialization; bars: ± 1 standard error.

After initial adaptation, TAIL-W4 achieves $1.15\times$ rollout speedup and $1.13\times$ RL-step speedup over steps 9–32. Across all 32 steps, cumulative training cost falls from 15.3 to 13.8 GPU-hours (9.5%; Figure 9). Final held-out accuracy is 61.7% for BF16 and 63.1% for TAIL-W4 + TIS.

These results show lower training cost with comparable held-out accuracy. Existing training-side methods address optimization and rollout–training mismatch (Huang et al., 2026; Li et al., 2026; Xi et al., 2026; Gu et al., 2026), while TAIL-W4 schedules precision to reduce the combined cost of decoding and downstream processing.

5.6 OVERHEAD

Optimized single-GPU preparation is estimated at 3.02 BF16 steps of additional work if BF16 calibration reuses training rollouts (Appendix A.9). Online updates add no exposed delay in the measured runs: the CPU rebuilds decisions during GPU log-probability computation and training, and every table is ready before the next rollout. Decoding uses lookups without additional calibration rollouts (Appendix A.9).

6 CONCLUSION

We presented TAIL-W4, an adaptive precision scheduler for long-tail RL rollouts. By combining response-length predictions, state-dependent decoding latency, and downstream token cost, TAIL-W4 chooses when to switch from BF16 to W4 and adapts as training progresses. Across three evaluated models, it achieves 1.09–$1.26\times$ mean RL-step speedup over BF16, with smaller mean reward drops than uniform W4. These execution savings also reduce cumulative training cost in our 32-step Phi-4-mini experiment, with held-out accuracy comparable to BF16 training. Together, these results show the value of scheduling precision around its effect on both execution speed and generated work.

AI USE STATEMENT

In this work, we used generative AI tools to assist with interpreting experimental results, drafting and revising the manuscript, editing plotting and analysis code, and preparing figures and LaTeX formatting. Research ideas, method implementation, experimental design and execution, and theoretical analysis were developed or performed by the authors without assistance from these tools. We did not use these tools to generate experimental data. AI-assisted numerical summaries and figures were checked against source data and run logs, and cross-references were checked against the manuscript source. Changes to plotting and document-generation code were checked by running the affected workflows and inspecting their outputs. The authors take responsibility for the final content of this work, including text, claims, code, and figures produced with AI assistance.

REFERENCES

Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant LoRA serving. In *Conference on Machine Learning and Systems*, 2024.

Wenhua Cheng, Weiwei Zhang, Haihao Shen, Yiyang Cai, Xin He, Kaokao Lv, and Yi Liu. Optimize weight rounding via signed gradient descent for the quantization of LLMs. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 11332–11350, 2024. doi: 10.18653/v1/2024.findings-emnlp.662. URL https://aclanthology.org/2024.findings-emnlp.662/.

Wenhua Cheng, Weiwei Zhang, Heng Guo, Haihao Shen, and Zaner Ma. SignRoundV2: Toward closing the performance gap in extremely low-bit post-training quantization for LLMs. *arXiv preprint arXiv:2512.04746*, 2025. URL https://arxiv.org/abs/2512.04746v2. Version 2, revised May 18, 2026.

D. R. Cox. Regression models and life-tables. *Journal of the Royal Statistical Society: Series B (Methodological)*, 34(2):187–202, 1972. doi: 10.1111/j.2517-6161.1972.tb00899.x. URL https://doi.org/10.1111/j.2517-6161.1972.tb00899.x.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. In *Advances in Neural Information Processing Systems*, 2023.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.17323.

Elias Frantar, Roberto L. Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. MARLIN: Mixed-precision auto-regressive parallel inference on large language models. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 239–251, 2025. doi: 10.1145/3710848.3710871. URL https://doi.org/10.1145/3710848.3710871.

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. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-1218. URL https://papers.neurips.cc/paper_files/paper/2025/hash/33c00862bfa29ac72ecf630a41e19352-Abstract-Conference.html.

Hao Gu, Hao Wang, Jiacheng Liu, Lujun Li, Qiyuan Zhu, Bei Liu, Binxing Xu, Lei Wang, Xintong Yang, Sida Lin, Sirui Han, and Yike Guo. QaRL: Rollout-aligned quantization-aware RL for fast and stable training under training–inference mismatch. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17051–17064, 2026. doi: 10.18653/v1/2026.findings-acl.841. URL https://aclanthology.org/2026.findings-acl.841/.

Daya Guo et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645:633–638, 2025. doi: 10.1038/s41586-025-09422-z. URL https://doi.org/10.1038/s41586-025-09422-z.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

Qinghao Hu, Shang Yang, Junxian Guo, Xiaozhe Yao, Yujun Lin, Yuxian Gu, Han Cai, Chuang Gan, Ana Klimovic, and Song Han. Taming the long-tail: Efficient reasoning RL training with adaptive drafter. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1933–1948, 2026. doi: 10.1145/3779212.3790231. URL https://doi.org/10.1145/3779212.3790231.

Wei Huang, Yi Ge, Shuai Yang, Yicheng Xiao, Huizi Mao, Yujun Lin, Hanrong Ye, Sifei Liu, Ka Chun Cheung, Hongxu Yin, Yao Lu, Xiaojuan Qi, Song Han, and Yukang Chen. QeRL: Quantization-enhanced low-rank reinforcement learning for LLMs. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/a15a2ece7f0663d1ba7db91103ac61c9-Paper-Conference.pdf.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan O. Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-R1: Training LLMs to reason and leverage search engines with reinforcement learning. In *Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=Rwhi91ideu.

Minseo Kim, Minjae Lee, Seunghyuk Oh, Kevin Galim, Donghoon Kim, Coleman Hooper, Harman Singh, Amir Gholami, Hyung Il Koo, and Wonjun Kang. EfficientRollout: System-aware self-speculative decoding for RL rollouts. *arXiv preprint arXiv:2606.18967*, 2026. URL https://arxiv.org/abs/2606.18967.

Qingyuan Li, Bo Zhang, Liang Ye, Yifan Zhang, Wei Wu, Yerui Sun, Lin Ma, and Yuchen Xie. Flash communication: Reducing tensor parallelization bottleneck for fast large language model inference. *arXiv preprint arXiv:2412.04964*, 2024. URL https://arxiv.org/abs/2412.04964.

Yuhang Li, Reena Elangovan, Xin Dong, Priyadarshini Panda, and Brucek Khailany. QuRL: Efficient reinforcement learning with quantized rollout. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=eG0bpCwdKn.

Xinyu Lian, Walid Krichene, Beichen Huang, Masahiro Tanaka, Olatunji Ruwase, Li Zhang, and Minjia Zhang. Quantization inflates reasoning: Token inflation as a hidden cost of low-bit reasoning models. *arXiv preprint arXiv:2606.25519*, 2026. URL https://arxiv.org/abs/2606.25519.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration. In *Proceedings of Machine Learning and Systems*, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/hash/42a452cbafa9dd64e9ba4aa95cc1ef21-Abstract-Conference.html.

Open R1. Big-Math-RL-Verified-Processed. https://huggingface.co/datasets/open-r1/Big-Math-RL-Verified-Processed, 2025. Processed Big-Math dataset; revision c79efbb6d3b75e3a2bcc27a5c569119918132345.

Qwen Team. Qwen2.5-Coder-7B-Instruct-AWQ: 4-bit model on Hugging Face. https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct-AWQ, 2024. Accessed September 23, 2026.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

vLLM Team. vLLM documentation: Reproducibility. https://docs.vllm.ai/en/stable/usage/reproducibility/, 2026. Accessed September 23, 2026.

Haocheng Xi, Charlie Ruan, Peiyuan Liao, Yujun Lin, Han Cai, Yilong Zhao, Shuo Yang, Kurt Keutzer, Song Han, and Ligeng Zhu. Jet-RL: Enabling on-policy FP8 reinforcement learning with unified training and rollout precision flow. *arXiv preprint arXiv:2601.14243*, 2026. URL https://arxiv.org/abs/2601.14243.

A Additional Methods and Results

A.1 Shared LoRA adapter: execution and GPU profiles

Small live batches leave little useful work in the default kernel tiles. Without tuned configurations, vLLM's Triton shrink kernel uses 32×16 output tiles and split-K 64 for $b < 128$. For one adapter and projection, the grid has $64\lceil b/32 \rceil \lceil r/16 \rceil$ blocks, or 64 at $b = 1, r = 16$. Each block then has only one useful output row, and reduction partitions beyond the input dimension add no useful products. The expand kernel's 64-row tiles have a similar mismatch. We replace this default path with a compiled pair of PyTorch GEMMs.

Packing projections reduces the number of adapter operations. Projections such as QKV or gate/up share an input, so we stack their A_i matrices along the rank dimension and form a block-diagonal matrix from their U_i matrices. One GEMM pair then produces the concatenated output. Packed weights are refreshed after adapter updates. Our intermediate is BF16, whereas Punica uses FP32 for its shrink output, so numerical agreement is within tolerance rather than bitwise equality.

Separate streams overlap the independent base and adapter branches. In the captured dual-stream path, offloaded base GEMMs run on auxiliary streams while LoRA remains on the main stream. CUDA events synchronize the branches before their outputs are added. The profiles below show this assignment directly, rather than inferring it from the variant name.

Table 3: Median decoding-iteration time (ms): Qwen3.5-9B, A100-80GB, rank-16 zero-delta adapter, GSM8K, temperature 1.0. $B = 32$ reports the median over a full decode run starting with 32 requests. The rows compare the three distinct implementation stages.

	BF16		Integer W4	
Adapter / stream configuration	$b = 1$	$B = 32$	$b = 1$	$B = 32$
Default Punica, sequential	14.09	16.49	9.17	12.41
Packed PyTorch GEMMs, sequential	12.87	16.15	8.10	12.07
Packed GEMMs + overlap	12.37	15.39	7.84	11.02

The three stages isolate packing and overlap, not launch-order effects. The configuration named `base_first` follows the same stream assignment as the captured default dual-stream path, so it is excluded as a duplicate. These measurements do not establish a benefit from reversing the assignment or changing stream priority.

GPU profiles of the LoRA fast path. Kernel profiles show where the adapter optimization saves GPU time. We capture Qwen3.5-9B with BF16 base weights, a rank-16 zero-delta adapter, and one live request on an A100-80GB. Each variant generates 256 identical tokens under greedy decoding; summaries use the final 128 decode steps. These controlled profiles complement the rollout latencies in Table 3.

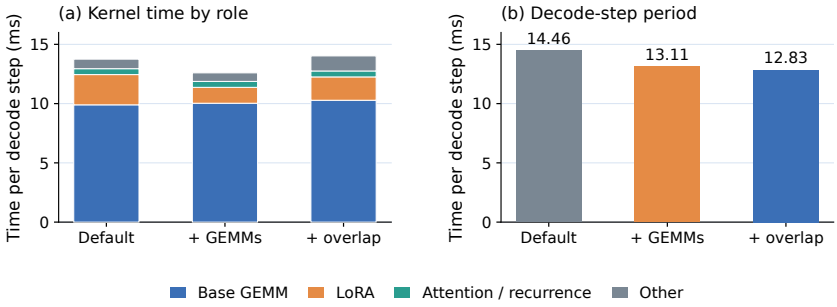

Figure 10: LoRA fast-path profiles on Qwen3.5-9B, BF16, rank 16, batch one. (a) Median kernel time by role, summed across streams; stacked segments are separately computed medians. (b) Median decode-step period, measured between successive step-start kernels, including idle gaps.

Packed GEMMs reduce adapter time even though they launch more kernels. The LoRA kernel-time sum falls from 2.55 to 1.34 ms per step, while the kernel count rises from 304 to 408. The median

decode-step period falls from 14.46 to 13.11 ms (Figure 10), indicating that kernel count alone does not explain the gain.

Stream overlap helps full-attention layers more than linear-attention layers in this hybrid model. After packed GEMMs, overlap reduces their median periods by 6.4% and 0.9%, respectively (Figure 11). The full-attention path occupies about 27 μs, compared with 12 μs for convolution and recurrence, providing more work behind which auxiliary-stream GEMMs can overlap. Because most layers use linear attention, the decode-step period reduction is only 2.1%, from 13.11 to 12.83 ms. These profiles do not establish the overlap benefit for other architectures.

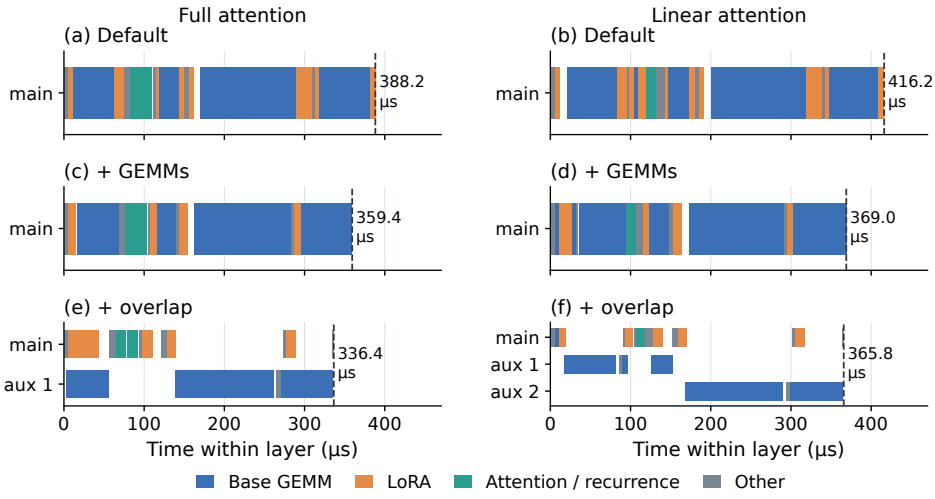

Figure 11: Representative layers of Qwen3.5-9B under the three LoRA execution paths. Columns separate full and linear attention; all panels share the time axis. Each example is selected closest to its type's median period, marked by a dashed line. Offloaded base GEMMs overlap main-stream work. Medians use 1,016 full-attention and 2,921 linear-attention windows per variant.

Layer windows separate the two attention types using their actual kernels. We delimit windows by successive input RMSNorm kernels and classify them by FlashAttention or convolution/recurrence kernels, rather than layer index. Each of 127 complete steady-state steps contributes eight full-attention and 23 linear-attention windows; the final window is excluded because it also contains the output head and sampling. Weighted layer medians recover the measured 31-layer span within 0.1% for the default and overlap variants. Under overlap, per-role duration sums count concurrent work separately and can increase even when the layer finishes sooner; elapsed periods determine the speedup.

Step boundaries follow a repeated kernel sequence rather than an idle-gap threshold. The once-per-step `index_elementwise_kernel` with grid $(1, 1, 1)$ follows sampling and precedes slot mapping in all three variants. Successive occurrences delimit a full step, including the sampling and scheduling gaps that GPU busy time excludes.

The profiles resolve kernels inside CUDA graphs. Captures use node-level CUDA-graph tracing, and decode windows are anchored on a GPU kernel that occurs once per step. Kernel families identify base GEMMs, LoRA, attention or recurrence, and other work. Signatures shared with the Punica baseline are conservatively assigned to the base branch; this attribution may undercount LoRA time.

A.2 Downstream time as a function of token count

Figure 12 shows how actor log-probability computation and training time vary with sequence-token count in 20 randomly sampled Qwen3.5-9B RL steps at $B = 32, 64$. Both phases are approximately linear over the sampled range: the fits have $R^2 = 0.996$ and $R^2 = 0.997$, respectively, with slopes summing to 0.694 s per thousand tokens. These fits supply the scheduler's downstream cost model. Experimental RL-step times are recorded directly from the evaluated runs.

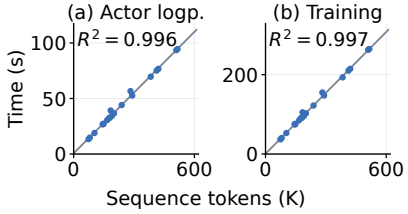

Figure 12: Actor log-probability and training time for Qwen3.5-9B (20 randomly sampled RL steps; $B = 32, 64$). Lines show linear fits; reference replay is excluded.

A.3 Calculating the remaining cost of a switch

A candidate trajectory combines BF16 survival before the switch with W4 survival after it. At current state (f, b, q), let s be a candidate switch frontier, $g(s)$ its closest preceding calibration group, and $r = (s - f)/\Delta$ its distance in bins. For a request alive at f, the probability of reaching future bin k is

$$a_s(k) = \begin{cases} S_{\mathrm{BF16}}(k \mid f), & k < r, \\ S_{\mathrm{BF16}}(r \mid f)\, S_{\mathrm{W4},g(s)}(k - r \mid s), & k \geq r. \end{cases} \tag{6}$$

The factor $S_{\mathrm{BF16}}(r \mid f)$ accounts for requests that finish before reaching the switch. The no-switch candidate, $s = \infty$, uses BF16 survival throughout.

Survival probabilities estimate both the remaining live batch and the chance that decoding continues. Approximating the b requests as independent with survival probability $a_s(k)$ gives a nonempty-batch probability $\nu_s(k) = 1 - (1 - a_s(k))^b$. Conditional on a nonempty batch, the expected live batch is $b\, a_s(k)/\nu_s(k)$. We round this value to the nearest integer and clip it to $[1, B]$ to obtain the latency-lookup batch $\widetilde{b}_s(k)$. Bins with $\nu_s(k) = 0$ contribute no decoding time.

Summing over future bins converts these predictions into rollout time and response tokens. With $\tau_k = \min(\Delta, C - f - k\Delta)$, the terms in Eq. 4 are

$$\widehat{T}_{\mathrm{roll}}(s \mid f, b, q) = \sum_k \tau_k\, \nu_s(k)\, T_{p_s(k)}\big(q + f + k\Delta, \widetilde{b}_s(k)\big), \tag{7}$$

$$\widehat{N}(s \mid f, b, q) = b \sum_k \tau_k\, a_s(k). \tag{8}$$

Here $p_s(k)$ is BF16 before s and W4 thereafter, T is measured in seconds per decoding iteration, and the sums stop at the response cap. The approximation evaluates latency at a conditional mean batch size instead of integrating over every possible batch history. The current cost calculation has no separate switch-overhead term because weights and prefix state remain resident.

Algorithm 1 summarizes the table search and online execution.

Algorithm 1 TAIL-W4: update and search on CPU, lookup during decoding

function BUILDTABLE(survival curves, T, λ)
  for each state (f, b, q) **do**
    $s^* \leftarrow \infty$; $J^* \leftarrow J(\infty \mid f, b, q)$
    for $s = f, f + \Delta, \ldots < C$ **do** ▷ earliest profitable switch wins ties
      $J_s \leftarrow J(s \mid f, b, q)$ using the W4 group selected for s
      if $J_s < J^*$ **then** $s^* \leftarrow s$; $J^* \leftarrow J_s$
    $D[f, b, q] \leftarrow s^*$
  return D
Initialize: calibrate survival, T, λ; build D.
While in BF16: at each frontier, switch if $D[f, b, q] \leq f$.
After rollout, on CPU alongside GPU training:
  update survival (§4.4); rebuild and atomically publish D.

A.4 UPDATING RESPONSE-LENGTH PREDICTIONS ONLINE

Observed survival and natural completion provide the two inputs to the update in Eq. 5. Let $H_g^0(x) = -\log S_g^0(x)$ be the calibrated cumulative hazard. For request i, observed in group g from position s_i to y_i, define exposure $e_i = H_g^0(y_i) - H_g^0(s_i)$ and completion indicator δ_i. A natural completion has $\delta_i = 1$; a precision switch or response-cap hit has $\delta_i = 0$. Thus a request that remains unfinished still contributes evidence through its exposure. BF16 observation ends at the switch, and W4 observation begins there; W4 cohorts use the group selected by their median entry frontier. Cumulative hazard is interpolated between bin boundaries.

The multiplier balances weighted completions against weighted exposure. Under the model, request i contributes likelihood $\mathcal{L}_i(a_g) \propto a_g^{\delta_i} \exp(-a_g e_i)$, giving

$$a_{g,t} = \frac{\kappa_g + \sum_i w_i \delta_i}{\kappa_g + \sum_i w_i e_i}, \qquad w_i = \rho^{t - t_i}. \tag{9}$$

Both sums cover group g through rollout t, and t_i is the rollout that produced observation i. Longer survival without completion increases the denominator and lowers the multiplier; more completions increase the numerator. The term κ_g keeps the multiplier near one when observations are scarce. Equivalently, this update maximizes the discounted log-likelihood with regularizer $\kappa_g(\log a_g - a_g)$.

The decay factor $0 < \rho \leq 1$ controls the weight of earlier rollouts. The adapted bin hazard, $h_{g,t}(x) = 1 - (1 - h_g^0(x))^{a_{g,t}}$, feeds directly into the same cost calculation and table search.

For the main-table runs, we use $\rho = 0.9$, $\kappa_{\mathrm{BF16}} = 64$, and $\kappa_g = 64 n_g / 256$ for each W4 continuation group, where n_g is its calibration response count. Before publishing an updated table, we limit each switch frontier's movement relative to the previous table to 2,000 tokens for Qwen and 1,000 tokens for the main-table Phi 4K configuration. This limit applies when both entries specify a switch; transitions to or from the no-switch decision are unrestricted.

Sharing a multiplier also defines what the update can learn. It assumes that changes across a group's response lengths can be represented by scaling the calibrated cumulative hazard. BF16 prefix observations can therefore revise predicted tail lengths, but cannot identify a change confined to an unobserved tail.

A.5 REUSING PREFIX STATE AT THE SWITCH

Teacher forcing isolates the effect of reusing BF16 prefix state. We use 16 BF16 traces with a switch after 4,096 output tokens and give every branch the same recorded tokens. At offsets $o \in \{0, 512, 1024, 2048, 4096\}$ after the switch, we score a 64-token window under BF16, W4 with reused BF16 state, and W4 after re-prefill. The metric is mean next-token KL divergence from BF16, $D(q) = \frac{1}{64} \sum_j D_{\mathrm{KL}}(p_j^{\mathrm{BF16}} \,\|\, q_j)$.

State reuse stays closer to BF16 at every measured offset (Figure 13). This diagnostic uses symmetric group-128 simulated INT4 and supports the choice to retain prefix state. Free-generation results with production MARLIN kernels are evaluated separately in Section 5.2.

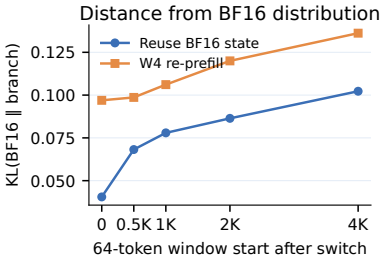

Figure 13: State reuse gives lower divergence from BF16 than W4 re-prefill. Mean KL divergence over 16 teacher-forced traces with simulated INT4.

A.6 Extended rollout configurations and data-parallel scaling

Evaluation goal and scope. We test whether TAIL-W4's efficiency benefits extend across response caps, training backends, and data-parallel group sizes. Table 4 covers caps from 4K to 24K, Megatron-Core and FSDP training, and DP sizes from one to eight. Each comparison uses BF16 with the same configuration as its reference; uniform W4 is included where available. The reported windows differ across configurations, so these results assess the scheduler within each setting rather than rank the backends or response caps.

Table 4: Results across extended rollout configurations. Speedups and ΔReward are relative to BF16 in the same configuration. Downstream entries give BF16 → TAIL-W4 seconds. MG denotes Megatron-Core; the number after the backend is the DP size. Dashes indicate unavailable uniform-W4 comparisons. The Phi-4-mini 12K row uses the TIS configuration. The 16K MG runs and the 9B 24K MG/2 run contain 12 steps.

			TAIL-W4			Uniform W4	ΔReward
Model	Cap	Backend/DP	Rollout	Downstream (s)	Step	Step	TAIL-W4/ W4
Phi-4 mini	4K	MG/1	1.24×	21.6 → 22.3	**1.115**×	1.018×	−0.021/ − 0.040
	4K	FSDP/1	1.20×	12.8 → 13.3	**1.105**×	—	−0.017/—
	8K	FSDP/8	1.16×	12.9 → 13.3	**1.123**×	1.068×	−0.005/ − 0.039
	12K	FSDP/4	1.14×	72.7 → 68.6	**1.122**×	—	−0.022/—
	16K	MG/1	1.16×	26.3 → 25.1	**1.118**×	0.768×	−0.013/ − 0.055
Qwen3.5 4B	16K	MG/1	1.22×	96.3 → 98.5	**1.123**×	1.033×	+0.021/ + 0.003
	24K	MG/1	1.36×	93.4 → 92.2	**1.226**×	0.994×	0.000/ − 0.012
	24K	FSDP/1	1.28×	51.7 → 53.2	**1.199**×	—	−0.013/—
	24K	FSDP/2	1.37×	52.7 → 54.7	**1.274**×	—	−0.002/—
	24K	FSDP/4	1.34×	55.2 → 57.7	**1.253**×	—	−0.001/—
	24K	FSDP/8	1.34×	54.4 → 57.1	**1.251**×	1.159×	−0.006/ − 0.023
Qwen3.5 9B	16K	MG/1	1.25×	131 → 138	**1.122**×	1.013×	−0.013/ − 0.065
	24K	MG/1	1.21×	116 → 132	**1.084**×	0.870×	−0.018/ − 0.033
	24K	MG/2	1.18×	220 → 220	**1.100**×	0.832×	+0.008/ − 0.026

TAIL-W4 improves mean RL-step time over BF16 in all 14 listed configurations, with speedups of 1.084–1.274×. In the nine settings with a uniform-W4 comparison, TAIL-W4 provides 1.05–1.46× step speedup over uniform W4 and higher reported mean reward. Uniform W4 can accelerate rollout yet lose the benefit at the step level as extra sampled tokens increase downstream work; the Qwen3.5-9B and 16K Phi-4-mini results illustrate this effect.

We do not repeat fixed-rule sweeps for these extended configurations because selecting a useful constant requires costly trials across candidate rules (Section 4.1; Appendix A.11). A well-tuned fixed schedule can match or exceed TAIL-W4's performance, but finding that rule requires evaluating candidates that may offer little or no benefit. Our goal is a best-effort scheduler that improves on uniform W4 without this per-configuration trial-and-error search, rather than a guarantee of beating every tuned fixed rule. TAIL-W4 starts from calibration and runtime profiles, then adapts its decisions using online observations.

Data-parallel scaling. We compare TAIL-W4 with BF16 at the same data-parallel (DP) degree, separating rollout speedup from complete RL-step speedup. Figure 14 combines measured Qwen3.5-4B results at DP2, DP4, and DP8 with projections from single-worker rollouts for all three models. DP1 reuses the measured summaries in Table 1. Models fit on one GPU per rollout worker; increasing DP replicates workers rather than sharding model weights.

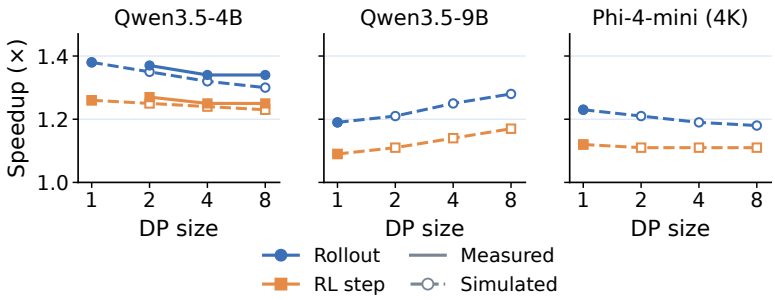

Figure 14: TAIL-W4 speedup over BF16 at equal data-parallel (DP) size. Solid: Qwen3.5-4B measurements at DP2–8; dashed: simulations. DP1 reuses Table 1. The 9B and Phi-4-mini DP curves are projections; their distributed measurements in Table 4 use other configurations.

Distributed measurements. With FSDP2 training, Qwen3.5-4B retains 1.25–1.27× RL-step speedup across DP2–8. The measurements are:

DP size	Rollout speedup	RL-step speedup
2	1.37×	1.27×
4	1.34×	1.25×
8	1.34×	1.25×

Each BF16–TAIL-W4 comparison takes roughly eight to nine hours in total. Reported speedups compare mean measured times; they are not ratios of whole-run durations, which also include work outside the measurement window.

Rollout projection. The projection models the wait for the slowest worker. For each policy, we sample groups of D single-worker observations with replacement from steps 9–48, preserving each observation's rollout time and response-token count. Each worker has $B = 32$, and each DP degree uses 4,000 sampled groups. The BF16 input averages two reference runs at each step; TAIL-W4 uses its observed run. Groups are sampled independently for the two policies. A synchronous barrier makes group rollout time the maximum worker time, giving

$$\text{speedup}_{\text{roll}}(D) = \frac{\mathbb{E}[\max_{d \le D} T_{\text{BF16},d}]}{\mathbb{E}[\max_{d \le D} T_{\text{Tail-W4},d}]}.$$

The expectation is estimated over sampled groups. Thus the projection compares mean barrier times, rather than averaging individual-worker speedups.

Downstream projection. The model and calibration factors below apply only to the dashed simulation curves. Measured points use directly recorded experimental times. For policy p, let $N_{p,d}$ be the sampled response-token count on worker d. Assuming token balancing across workers, we estimate complete-step time as

$$\widehat{T}_{\text{step},p}(D) = \max_{d \le D} T_{p,d} + \gamma_D \left(\lambda \frac{1}{D} \sum_{d=1}^{D} N_{p,d} + c \right).$$

The fitted downstream slopes are 0.494, 0.694, and 0.428 seconds per thousand tokens for Qwen3.5-4B, Qwen3.5-9B, and Phi-4-mini, respectively, with intercepts c of 4.6, 5.3, and 4.5 seconds. They account for actor log-probability and training time, excluding reference evaluation. Historical FSDP2 calibration uses $\gamma_2 = 1.09$ and $\gamma_4 = 1.05$; DP8 reuses $\gamma_8 = 1.05$ without adjustment to the new measurements. Simulated step speedup is the ratio of mean projected BF16 and TAIL-W4 step times.

Why model trends differ. Increasing DP emphasizes the upper tail of rollout time: a larger group is more likely to contain a slow worker, and all workers wait for it. The gain therefore depends on how

the two policies' slowest rollouts compare, not only on their mean rollout times. Each worker still uses $B = 32$, so these trends do not come from increasing its local batch size.

For Qwen3.5-9B, the projected DP8 barrier is $1.46\times$ the mean single-worker rollout time under BF16, compared with $1.37\times$ under TAIL-W4. BF16 incurs the larger relative penalty from slow workers, so projected rollout speedup rises from $1.19\times$ at DP1 to $1.28\times$ at DP8. For Qwen3.5-4B, the corresponding factors are $1.36\times$ for BF16 and $1.44\times$ for TAIL-W4: its average rollout advantage narrows when the slowest worker determines completion. Projected rollout speedup consequently falls from $1.38\times$ to $1.30\times$. Phi-4-mini shows the same direction, with factors of $1.31\times$ and $1.36\times$ and rollout speedup falling from $1.23\times$ to $1.18\times$.

Complete-step speedup also includes downstream work. In the projection, rollout time is a maximum across workers, whereas token-balanced downstream cost depends on their average token count. These different scaling behaviors produce a rising step-speedup curve for 9B, a modest decline for 4B, and a nearly flat curve for Phi-4-mini. The measured 4B results likewise retain 1.25–$1.27\times$ step speedup across DP2–8; the other two models' distributed trends remain projections.

Agreement and scope. For Qwen3.5-4B, projected rollout speedups differ from measurements by at most 3%, and projected step speedups by at most 1.6%, relative to the measured speedups. At DP8, the projections give $1.23\times$, $1.17\times$, and $1.11\times$ step speedup for 4B, 9B, and Phi-4-mini, respectively. These estimates use observed single-worker variation and an approximate downstream model; they do not simulate a shared distributed training trajectory or correlations between workers. The 4B comparison checks this approximation only in the tested configurations. The 9B and Phi-4-mini measurements in Table 4 use different caps or training backends and do not validate the corresponding projected curves.

Training-rollout accuracy at DP8. The Qwen3.5-4B DP8 experiment runs for 48 steps with FSDP2, sampling 64 prompts and four responses per prompt at each step. This covers 1.5 epochs of the configured 2,048-question GSM8K subset; the adapter and response cap match the single-worker setup. Over all 48 steps, TAIL-W4 achieves $1.25\times$ speedup in accumulated RL-step time, excluding setup, with mean training-rollout accuracy 0.008 below BF16. Uniform W4 achieves $1.16\times$ speedup with accuracy 0.023 below BF16. It generates 25% more tokens than BF16, compared with TAIL-W4's 5%, increasing downstream work. These are accuracies of the training rollouts; the common-precision held-out checkpoint evaluation is reported separately in Section 5.5.

A.7 How switch decisions vary across rollouts

Switch frontiers and live batches vary across the evaluated rollouts. Figure 15 reports both for every Qwen3.5-4B rollout in steps 9–48. Each point is an executed decision, rather than a fixed threshold imposed on the run.

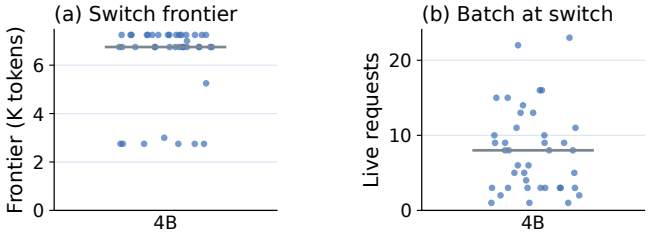

Figure 15: Qwen3.5-4B switch frontiers and live requests over steps 9–48. Each dot is one rollout; horizontal bars mark medians.

The full set of batch trajectories shows the context behind these switches. Figure 16 plots all 40 evaluated rollouts, with each panel marking the response frontier and live batch at the switch. BF16 and uniform-W4 curves come from independent runs at the same training step, so they describe observed trajectories rather than counterfactual continuations of the TAIL-W4 batch.

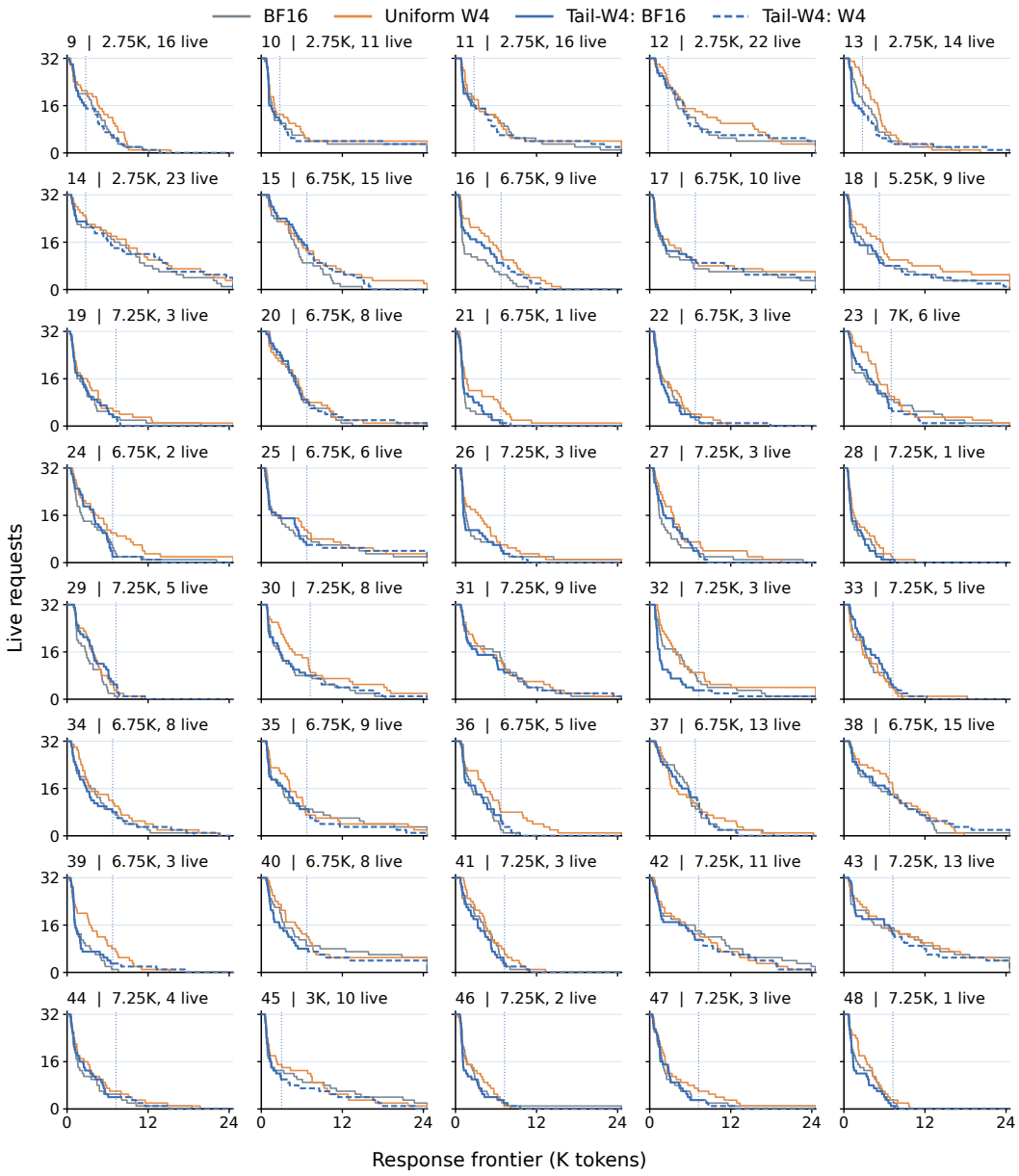

Figure 16: Live requests over the response frontier on Qwen3.5-4B, steps 9–48. Panel labels give the training step, switch frontier in thousands of tokens, and live requests at the switch. Solid and dashed blue segments mark TAIL-W4's BF16 and W4 phases; dotted vertical lines mark switches. BF16 and uniform W4 are independent runs.

A.8 Online adaptation from shared calibration data

To examine whether online updates adapt a shared calibration to different training trajectories, we run Qwen3.5-4B with three different random seeds and the same calibration data. We refer to these runs as A, B, and C; Run A is the run shown in Figure 7. Figure 17 compares the three runs in panels (a)–(c), showing decision maps before training and after 48 steps, together with the switches executed during training. The initial predicted gains and scheduled frontiers are identical across runs.

All three runs move beyond the initial 2.75K-token switch frontier as online observations revise the response-length predictions. Runs B and C first switch beyond 5K tokens at steps 4 and 3, respectively, compared with step 15 in Run A. Thus, the two additional runs begin adapting within the first few rollouts. A frontier near 7.25K recurs in all three runs over the later training steps.

The final decision maps also reveal variation across training trajectories. Run A schedules switching at 7.25K across live batch sizes. Run C makes similar choices at most batch sizes, with later switches at some large live batches. Run B returns to a 2.75K frontier in the final five rollouts. These observations show that online updates can revise the same initial calibration in response to the workload encountered during training; they do not establish convergence to a common or optimal switch frontier.

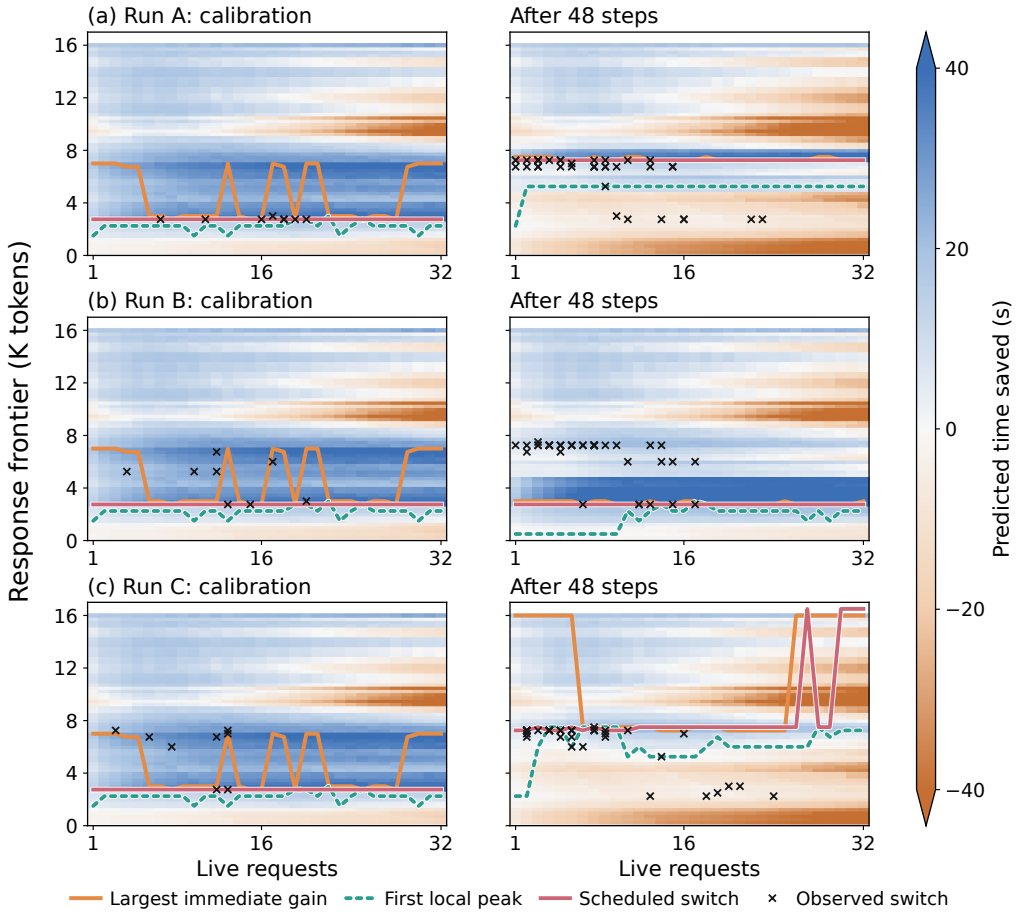

Figure 17: Qwen3.5-4B decision maps for (a) Run A, (b) Run B, and (c) Run C, all initialized from the same calibration. Left: calibration; right: after 48 steps. Run A also appears in Figure 7. Colors show predicted step time saved by switching now relative to BF16, saturating at ± 40 s. Red curves show schedule snapshots; crosses mark actual switches in steps 1–8 (left) and 9–48 (right). The range includes Run C's 16.5K decisions above the gain surface.

A.9 Overhead measurement details

Preparation combines runtime setup, latency profiling, and response calibration.

Grouped W4 rollouts. Grouped W4 rollouts are continuation samples collected from BF16 prefixes of different lengths. Each group captures how the prefix length affects the remaining response length under W4. We select population quantiles of BF16 response lengths, round them down to the $\Delta = 250$ token grid, and remove duplicate frontiers. For Qwen3.5-4B, 256 BF16 responses give continuation frontiers at 2,750, 5,000, 6,250, 7,250, and 8,250 tokens. These groups contain 64, 64, 64, 56, and 45 W4 continuations, respectively; another 256 uniform-W4 responses provide the group with no BF16 prefix.

Each continuation preserves the recorded prefix tokens while measuring how generation proceeds under W4. At frontier c, calibration reissues the original prompt and BF16 prefix to the W4 engine with a remaining budget of $C - c$ tokens. This recomputes prefix state under W4, whereas the runtime switch retains BF16 state. Calibration therefore supplies an initial prediction that online observations refine, rather than an exact replay of the runtime switch.

The uniform-W4 group covers candidates before the first nonzero calibration frontier. Without it, the implementation falls back to the earliest continuation group, so those candidates lack direct observations.

Preparation cost. The component estimate below uses earlier Qwen3.5-9B profiling on an A100-80GB together with the current grouped-W4 calibration measurement. Startup was measured relative to a BF16 worker under matched hot-start conditions, and the decode profile covered 128 precision/context/live-batch cells.

Component	Time	Measurement
Dual-precision startup increment	12.44 s	Earlier 9B profile
W4 kernel warm-up	0.93 s	Earlier 9B profile
Decode profile	47.89 s	Earlier 9B profile
Downstream cost calibration	237.15 s	Earlier 9B profile
Grouped W4 rollouts	13.1 min	Current; 1.75M tokens

Reusing BF16 training rollouts could reduce incremental preparation cost. Eight BF16 batches at $B = 32$ provide the required 256 responses; the estimate charges W4 continuation sampling and runtime preparation as incremental work. With this optimization, the listed additional components sum to $12.44 + 0.93 + 47.89 + 237.15 + 786 = 1{,}084.41$ s. Relative to the single-GPU 9B BF16 step in Table 1 (359 s), this is approximately 3.02 steps.

Online update cost. Scheduler updates finished before the next rollout in all observed transitions. We matched the final request completion, table publication, and next-rollout start across 47 adjacent-rollout transitions in each of the three 48-step runs, for 141 transitions in total. Every publication lag was shorter than that step's GPU actor-training time alone.

Model	Maximum publication lag	Minimum lead before next rollout
Qwen3.5-4B	2.82 s	45.86 s
Qwen3.5-9B	2.80 s	60.86 s
Phi-4-mini	2.75 s	18.26 s

Publication lag includes polling and publication work. These measurements do not isolate lookup, readiness-check, or precision-switch latency.

A.10 Decode-time breakdown across architectures

The fraction of decoding spent in quantizable linear layers differs substantially between the two profiled models. Figure 18 profiles BF16 without LoRA at 16K context and batch sizes one and four on the A100-80GB. With compilation and CUDA graphs enabled, packed linear layers account for

71–75% of Qwen3.5-9B's GPU kernel time, compared with 16–24% for Phi-4-mini. The output head is shown separately because it remains BF16 in our W4 path.

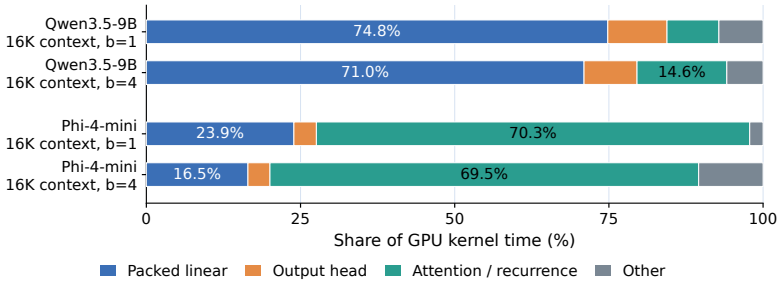

Figure 18: GPU kernel-time shares at 16K context, BF16, without LoRA, using the compiled stack on A100-80GB. Shares are classified by kernel family. Phi-4-mini's attention cost includes the effect of the current FlashAttention split configuration, not just model architecture.

Attention limits the fraction of Phi's runtime that weight compression can accelerate in this configuration. It accounts for about 70% of kernel time, but the batch-one trace launches only 24 attention blocks without a split-combine stage. This limited parallelism contributes to the measured cost. The result is specific to the profiled stack and 16K context; it does not determine the speedup in the main evaluation's 4K Phi workload.

An eager reference checks the kernel attribution under compilation. The eager pass uses module-level NVTX ranges, while the compiled pass classifies kernel families and identifies the output-head GEMM by its vocabulary-sized grid. At Qwen3.5-9B batch one, packed-linear time is 8.90 ms in eager execution and 8.48 ms when compiled; other-kernel time drops from 3.40 to 0.81 ms. We use the compiled measurements in the figure because they retain the production execution path.

A.11 RL-STEP ACCOUNTING AND EVALUATION CONFIGURATION

Table 1 reports directly measured rollout, downstream, and complete RL-step times from the evaluated runs. Component times, total times, and speedups are rounded independently for display.

The Qwen3.5-4B runs keep the training configuration fixed across precision schedules. Runs use learning rate 10^{-6}, rank-16 LoRA, Megatron-Core TP1, and vLLM at 50% GPU memory utilization. Each step samples four responses for each of eight prompts, with temperature and top-p both 1.0. The prompt and response caps are 2,048 and 24,576 tokens, respectively.

Timing excludes startup steps while allowing adaptation from the start. The 4B runs last 48 steps, with steps 9–48 used for reported averages. TAIL-W4 may switch and revise predictions during the first eight steps; excluding their timings does not force those rollouts to stay in BF16.

The completed sweep compares a wider range of fixed schedules under one timing convention. Figure 6 reports mean times over 40 steps (9–48) for every arm. Panel (b) shows the fraction of generated tokens decoded with W4.

The sweep uses the recorded `timing_s/step` and `timing_s/gen` metrics for complete-step and rollout time.

The constant-TPOT ablation uses 13.52 ms for BF16 and 11.77 ms for W4 at every batch size and context length. The no-downstream-cost variant sets $\lambda = 0$; the other variants are defined in Section 5.3.

Held-out checkpoint evaluation and training cost. Section 5.5 evaluates Phi-4-mini on a 211-question subset of the processed Big-Math dataset (Open R1, 2025). The questions have between one and seven correct responses out of eight initial samples, retaining prompts with mixed outcomes. Both runs use learning rate 3×10^{-4}, rank-16 LoRA, FSDP2 with four data-parallel workers, temperature 1.0, and a 12,288-token response cap. Each training step uses 16 prompts with eight responses per prompt. The TAIL-W4 rollout correction uses token-level TIS with threshold 2. We report the completed 32-step budget.

For each checkpoint, evaluation loads the same BF16 base weights and the adapter from the corresponding training run. We evaluate the same 256 held-out questions with two responses per question at temperature 1.0 and the training response cap. Accuracy averages correctness over these responses. Figure 9 shows the shared initialization and checkpoints at steps 16, 24, and 32. Their accuracies are 62.5%, 60.5%, 61.1%, and 61.7% for BF16, and 62.5%, 60.4%, 61.7%, and 63.1% for TAIL-W4 + TIS. The TAIL-W4 curve also includes its step-8 checkpoint (61.9%). The horizontal coordinate sums the recorded complete-step times through each checkpoint and multiplies by four GPUs. Error bars show one standard error across the 256 per-question accuracies, each averaging two responses. The measured costs, 15.3 and 13.8 GPU-hours, sum the 32 complete-step timers. Using unrounded times (15.276 and 13.831 GPU-hours) gives a 9.5% reduction for this fixed training budget, rather than a measurement of time to reach a target accuracy.

After the initial eight steps of adaptation, mean rollout time over steps 9–32 is 325.0 s for BF16 and 283.1 s for TAIL-W4, giving $1.15\times$ speedup. Mean complete-step time is 422.9 s and 374.0 s, respectively, giving $1.13\times$ speedup. Over the successive windows 9–16, 17–24, and 25–32, rollout speedups are $1.16\times$, $1.15\times$, and $1.14\times$; complete-step speedups are $1.14\times$, $1.14\times$, and $1.12\times$. These ratios use the recorded mean times within each window; Figure 9 retains all 32 steps in its cumulative timing.

A separate 48-step configuration. An earlier experiment uses a different 1,024-question Big-Math training subset, the same model, learning rate, DP4 setting and 12K response cap, and 48 training steps. Under common BF16 checkpoint evaluation on 256 held-out questions with four responses each, accuracy is 61.3% at initialization, 60.9% for the final BF16 checkpoint, and 56.8% for the final TAIL-W4 + TIS checkpoint. This run shows that the correction does not always prevent quality degradation. It is not a continuation of the 211-question experiment: both the training subset and budget differ, so this comparison does not isolate the cause of the degradation.

Fixed-rule selection over an early window. Figure 8 uses measured RL-step times for both fixed-rule selection and evaluation. The 4B candidates are $K \in \{1, 3, 5, 7, 9, 11, 13, 15\}$K and $t \in \{4, 8, 12, 16\}$; the 9B candidates are $K \in \{5, 7, 11, 15\}$K and $t \in \{4, 8\}$. Phi-4-mini uses a 4K response cap with $K \in \{1, 2\}$K and $t \in \{4, 8\}$; its selected $K = 1$K is also the in-sample best. Selection uses the beginning of each candidate's own run, not an independent pilot or a shared checkpoint from which all policies are restarted. It also requires evaluating the candidate rules; this comparison does not remove that cost.

The compared runs use matching prompt multisets, verified from saved rollout inputs. As discussed in Section 4.1, stochastic sampling and nondeterministic inference change generated work and execution time. A single training step therefore cannot reliably rank candidate schedules; we run each candidate for 48 steps and evaluate its mean time over steps 9–48.

Table 5 reports the training cost of every candidate, summing the recorded complete-step timers over all 48 steps, including the first eight steps used for selection. Each run uses one A100 GPU, so elapsed step-hours equal GPU-hours. These totals exclude process startup and setup outside the step timers. The 4B candidates cost 3.79–4.10 GPU-hours each and 46.69 GPU-hours in total. The BF16, uniform-W4, and TAIL-W4 runs add 4.45, 4.41, and 3.62 GPU-hours, respectively, bringing the 15 runs behind Figure 6 to 59.18 GPU-hours.

The full sweep is not a required tuning budget. Even if a heuristic based on interval halving needed only three or four trials to identify a promising coarse threshold, each trial would still take hours: under the same 48-step evaluation budget, this smaller search would cost approximately 11.4–16.4 GPU-hours.

The six 9B candidates cost another 29.52 GPU-hours, and the four Phi-4-mini candidates cost 3.09 GPU-hours: 79.31 GPU-hours across all 22 fixed-schedule runs. This tuning cost motivates the smaller candidate sets for those models rather than repeating the full 4B sweep. Even selection alone requires running the first eight steps of every candidate, costing 9.30 GPU-hours for 4B, 5.56 for 9B, and 0.51 for Phi-4-mini. Savings from the selected rule must repay this search cost before it improves overall training efficiency. For the 40-step evaluation window here, the search cost outweighs those savings; a longer deployment would be needed to amortize it.

Table 5: Training cost per fixed-schedule run, in GPU-hours over steps 1–48. Dashes indicate candidates not evaluated. Totals are computed before rounding.

Schedule	Qwen3.5-4B	Qwen3.5-9B	Phi-4-mini
$K = 1$K	4.10	–	0.76
$K = 2$K	–	–	0.79
$K = 3$K	3.93	–	–
$K = 5$K	3.79	4.94	–
$K = 7$K	3.83	4.78	–
$K = 9$K	3.83	–	–
$K = 11$K	3.80	4.83	–
$K = 13$K	3.79	–	–
$K = 15$K	4.01	4.94	–
$t = 4$	4.00	4.91	0.77
$t = 8$	3.87	5.12	0.77
$t = 12$	3.80	–	–
$t = 16$	3.94	–	–
Total	46.69	29.52	3.09

A.12 Extending Tail-W4 to NVFP4

We extend Tail-W4 to NVFP4 weights with BF16 activations. Table 6 reports measured results on Qwen3.5-4B with single-A100 Megatron training. Tail-W4 achieves $1.23\times$ RL-step speedup with INT4 and $1.22\times$ with NVFP4, while uniform W4 slightly slows the complete step in both formats. The scheduling benefit therefore extends beyond integer quantization.

Table 6: Measured Qwen3.5-4B results with single-A100 Megatron training, a 24K response cap, and eight prompts with four responses each. Speedups are relative to BF16; W4 tokens gives the fraction of response tokens sampled with quantized weights.

Policy	Weights	W4 tokens	Rollout	RL step
BF16	BF16	0%	$1.00\times$	$1.00\times$
Uniform W4	INT4	100%	$1.10\times$	$0.99\times$
Uniform W4	NVFP4	100%	$1.06\times$	$0.97\times$
Tail-W4	INT4	27%	$1.36\times$	$\mathbf{1.23}\times$
Tail-W4	NVFP4	27%	$1.34\times$	$\mathbf{1.22}\times$

The generated-work effect also persists: on the same 256 GSM8K prompts, NVFP4 generates 46.4% more tokens than BF16 (Figure 19), motivating precision scheduling across quantization formats.

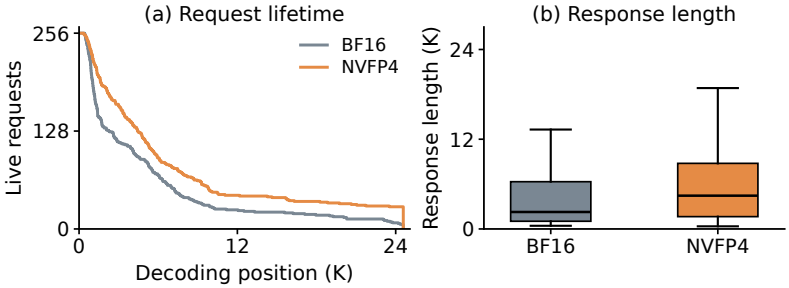

Figure 19: Qwen3.5-4B with BF16 or NVFP4 weights and BF16 activations: 256 matched GSM8K prompts, one sample each, temperature 1.0, and a 24,576-token cap. (a) Response survival by decoding position. (b) Response-length distribution.

B Limitations

Token inflation and mitigation. The benefit of delaying W4 depends on the extent of token inflation. We study it primarily in small models suited to single-GPU PEFT; its extent varies with the model, quantizer, and task. Prior work reports it in reasoning models spanning 4B to 30B parameters (Lian et al., 2026). Reasoning-aligned calibration data and quantization-aware training

can reduce both token inflation and accuracy loss (Lian et al., 2026). Our results concern post-training-quantized weights. If a quantizer or training method mitigates inflation, earlier W4 use may become preferable and the benefit of delaying the switch may shrink.

Models sharded across GPUs. Gains for models sharded across GPUs remain to be evaluated. Our experiments use models that fit on one GPU per rollout worker, with data parallelism replicating these workers. In sharded settings, communication can limit inference speed independently of weight traffic (Li et al., 2024). TAIL-W4 could be extended to these settings by incorporating communication and memory constraints into its cost model; the resulting schedules would need separate evaluation.

C Related Work

Prior work on quantized RL studies how lower precision affects efficiency and learning. QLoRA (Dettmers et al., 2023) enables adaptation on a quantized base; low-precision RL work studies rollout efficiency and precision mismatch (Huang et al., 2026; Li et al., 2026; Xi et al., 2026; Gu et al., 2026). Quantization-induced token inflation has also been documented for reasoning inference (Lian et al., 2026). We account for both the resulting decoding work and its subsequent training cost when choosing a switch within the rollout.

Speculative decoding offers another approach to accelerating the rollout tail, including TLT's adaptive drafter (Hu et al., 2026) and EfficientRollout's quantized self-drafter (Kim et al., 2026). These preserve the target sampling distribution through verification. TAIL-W4 directly samples from W4, so its cost model accounts for the changed response distribution; it does not use a drafter or rollback.