EfficientRollout: System-Aware Self-Speculative Decoding for RL Rollouts

Minseo Kim [*1] **Minjae Lee** [*1] **Seunghyuk Oh** [1] **Kevin Galim** [1] **Donghoon Kim** [1] **Coleman Hooper** [2]
Harman Singh [2] **Amir Gholami** [2] **Hyung Il Koo** [1] **Wonjun Kang** [1]

Abstract

Reinforcement learning (RL) has become a representative post-training paradigm for large language models (LLMs), but rollout generation remains a dominant latency bottleneck. Autoregressive (AR) sampling decodes responses sequentially, and a small number of long-tailed generations often determine completion time. Speculative decoding (SD) is a well-established technique for serving fixed LLMs that reduces latency by drafting tokens and verifying them in parallel. However, its practical speedups do not directly carry over to RL rollouts: (i) the continuously evolving target model makes static drafters stale, and (ii) active batch sizes shrink throughout rollout decoding, changing whether verification overhead can be amortized. We present **EfficientRollout**, a system-aware self-SD framework for RL rollouts that induces a quantized drafter directly from the target model (*i.e.* self-speculative decoding), keeping it coupled to the evolving policy without separate drafter pretraining or online adaptation. It further coordinates a system-aware SD toggle with acceptance-aware draft-length adaptation, speculating only in beneficial regimes while matching the drafting budget to evolving drafter quality. EfficientRollout reduces rollout and end-to-end latency by up to 19.6% and 12.7%, respectively, over an accelerated AR rollout baseline while preserving model quality. Code is available at `https://github.com/furiosa-ai/EfficientRollout`.

1. Introduction

Reinforcement learning (RL) has become an essential method for post-training frontier large language models (LLMs), enhancing their reasoning, coding, and agentic capabilities (Guo et al., 2025; Yang et al., 2025a; Team et al., 2025; Zeng et al., 2025a). Among RL algorithms, on-policy RL collects rollouts from the same policy being optimized, reducing policy-staleness degradation and improving training stability and model quality (Kumar et al., 2025; Fu et al., 2026). However, rollout generation is the dominant latency bottleneck in the RL pipeline because post-trained models increasingly generate long responses (Chen et al., 2025c; Guo et al., 2025; Yu et al., 2026; Liu et al., 2025b) that must be decoded token by token, dominating other parallelizable stages such as policy updates (Schulman et al., 2017).

Speculative decoding (SD) (Leviathan et al., 2023; Chen et al., 2023) offers a promising way to reduce inference latency while preserving the policy distribution. SD uses a cheaper drafter to propose multiple candidate tokens, which are then verified in parallel by the target model. Its speedup depends on two conditions: algorithmically, the block efficiency must be sufficiently high (Zhou et al., 2024; Liu et al., 2026); system-wise, decoding should not be pushed into a compute-bound regime, so that parallel verification can exploit underutilized compute (Leviathan et al., 2023; Sadhukhan et al., 2025; Liu et al., 2026).

In this regard, RL introduces challenges absent from serving fixed LLMs, making existing SD methods nontrivial to apply to RL rollouts. First, the target policy continuously evolves during training, making static drafters stale and potentially reducing block efficiency (Zhang et al., 2026). Second, rollout generation often starts with large active batch sizes, creating compute-bound intervals where SD may be slower than autoregressive (AR) decoding, before the batch later shrinks as shorter responses finish. At the same time, RL creates an opportunity: as post-training sharpens the policy distribution (Zhao et al., 2025), the target policy can be approximated with small loss by a lower-capacity subset induced from the target itself.

Prior SD methods for RL rollouts mitigate these challenges, but leave a trade-off between drafter construction and adaptation burden and block efficiency. History-based drafting methods reuse previous-epoch rollouts as draft proposals and are easy to deploy, but their low-capacity drafters often yield limited block efficiency because past rollouts sparsely

[*]Equal contribution [1]FuriosaAI [2]University of California, Berkeley. Correspondence to: Minseo Kim <minseo.kim@berkeley.edu>, Minjae Lee <minjae.lee@furiosa.ai>, Wonjun Kang <kangwj1995@furiosa.ai>.

Accepted at AdaptFM, ICML 2026.

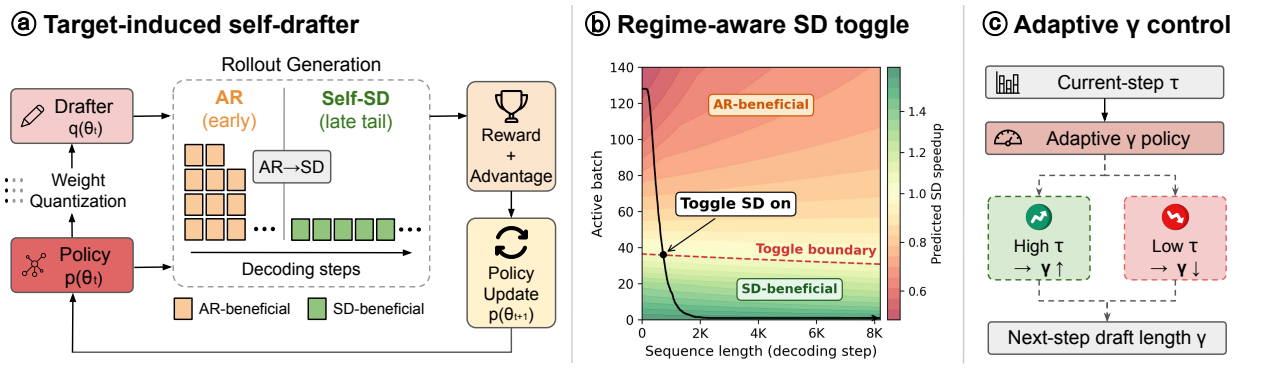


Figure 1. **Overview of EfficientRollout**. (a) per-step self-drafter refresh to track the evolving policy, (b) regime-aware AR-to-SD toggling under tail-heavy rollout dynamics, and (c) adaptive draft length γ from observed block efficiency τ.

cover future trajectories (Liu et al., 2025a; He et al., 2025; Shao et al., 2026). Learned auxiliary drafting methods seek higher block efficiency by training auxiliary drafters, but existing standalone auxiliary drafters do not automatically align with RL rollout distributions; in practice, they often require separate drafter pretraining and continual online adaptation to track the evolving target policy, adding training overhead and system complexity (Zhang et al., 2026; Chen et al., 2026; Hu et al., 2026). This raises a central question: *Can speculative decoding deliver realized rollout speedup while maintaining high block efficiency without an additional pipeline for drafter pretraining and adaptation?*

To this end, we answer this question with **EfficientRollout**, a system-aware SD framework designed for on-policy RL rollouts (Fig. 1). We first characterize RL rollout workloads, showing that self-SD stays coupled to the evolving policy and that dense weight-projection costs dominate rollout-tail decoding. Based on these observations, we induce a weight-quantized drafter from the current target policy at every training step, achieving high block efficiency without separate drafter pretraining or online gradient updates. We further introduce two control policies: a system-aware SD toggle that activates only in favorable regimes, avoiding slowdowns in early compute-bound phases, and an adaptive draft-length policy that matches the drafting budget to evolving acceptance behavior during RL.

Our main contributions are as follows:

- We characterize SD for RL rollouts, showing how evolving policies, shrinking-batch dynamics, and rollout-tail latency motivate a system-aware self-SD design (Sec. 2).
- We present EfficientRollout, an RL-specific SD framework with a target-induced quantized self-drafter, dynamic SD toggling, and adaptive drafting budgets (Sec. 3).
- We show that EfficientRollout reduces rollout and end-to-end latency by up to 19.6% and 12.7% over a standard accelerated AR rollout baseline (Sheng et al., 2025; Kwon et al., 2023) (Sec. 4).

2. Preliminary

2.1. Speedup Factors in Speculative Decoding

For a given batch size B and sequence length S, let $T_q(B,S)$ and $T_p(B,S)$ be the single-token decoding times of the drafter q and target model p, respectively. When the drafter sequentially speculates γ tokens, let $T_V(B,S,\gamma)$ denote the parallel verification time by the target model.[1] Given the block efficiency τ, defined as the expected number of accepted tokens per SD block (Zhou et al., 2024), the SD block time and walltime speedup are

$$T_{\mathrm{SD}}(B,S,\gamma) = \gamma T_q(B,S) + T_V(B,S,\gamma),$$
$$\mathrm{Speedup}_{\mathrm{SD}}(B,S,\gamma) = \tau \frac{T_p(B,S)}{T_{\mathrm{SD}}(B,S,\gamma)}. \quad (1)$$

Thus, SD acceleration depends on both an algorithmic factor, high block efficiency τ, and a system factor, latency ratios induced by (B,S). When decoding becomes compute-bound, often at large B, verification has little underutilized compute to exploit. This increases T_V/T_p and can make SD slower than AR decoding, an effect often hidden in $B=1$ SD benchmarks (Sadhukhan et al., 2025; Xia et al., 2024).

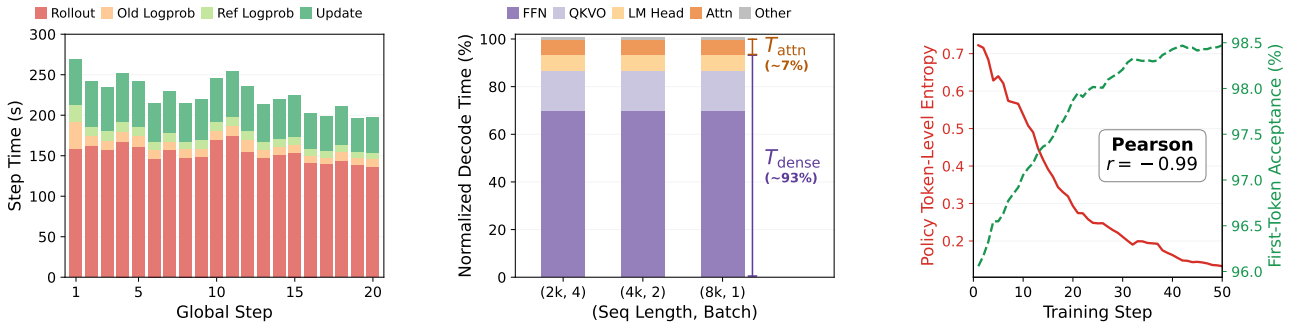
2.2. Roofline Modeling for LLM Decoding Latency

To identify the decoding regime induced by the hardware system, we use roofline modeling, a standard system-aware approach (Williams et al., 2009; Yuan et al., 2024). It distinguishes between arithmetic work $\mathrm{C}_{\mathrm{FLOPs}}$, executed with effective compute capacity $\mathrm{F}_{\mathrm{eff}}$, and data movement $\mathrm{M}_{\mathrm{Bytes}}$, served by effective memory bandwidth $\mathrm{BW}_{\mathrm{eff}}$. Assuming computation and memory access can ideally overlap, the workload latency is approximated as

$$T_{\mathrm{roofline}} \approx \max\left(\frac{\mathrm{M}_{\mathrm{Bytes}}}{\mathrm{BW}_{\mathrm{eff}}}, \frac{\mathrm{C}_{\mathrm{FLOPs}}}{\mathrm{F}_{\mathrm{eff}}}\right). \quad (2)$$

For LLM decoding, $\mathrm{M}_{\mathrm{Bytes}}$ includes weight and KV-cache traffic, while $\mathrm{C}_{\mathrm{FLOPs}}$ includes dense projections such as

[1] For brevity, we omit B, S, and γ when clear.



(a) (b) (c)

Figure 2. **Empirical characteristics of RL rollout decoding.** (a) Step-time decomposition over the first 20 training steps. (b) Single-token decode-time breakdown in representative RL rollout-tail phases. (c) Inverse correlation between target-policy entropy and quantized-drafter first-token acceptance over training steps.

Table 1. Comparison of drafter categories for SD in RL rollouts. We compare methods along three dimensions: whether they use parameterized drafters, avoid continuous online adaptation, and can accelerate from the first epoch without category-specific warm-up, such as rollout-history collection or drafter pretraining. See Sec. A.3 for a detailed discussion.

Drafter category	Param. drafter	Adapt. free	Warm-up free
(1) History-based [33; 45; 18]	No	Yes	No
(2) Learned auxiliary [68; 21; 8; 23]	Yes	No	Cond.
(3) Target-induced **EfficientRollout (ours)**	**Yes**	**Yes**	**Yes**

QKVO, FFN, and LM-head computations, as well as attention operations (Sadhukhan et al., 2025).

2.3. RL Rollout Latency Bottlenecks

Rollout is the primary latency bottleneck in RL post-training for LLMs. As shown in Fig. 2a, sequential rollout decoding accounts for nearly 70% of total latency on average. This bottleneck is amplified in RL because post-training produces long reasoning trajectories decoded token by token, whereas policy updates and log-probability computation remain more parallelizable (Gholami et al., 2024). Within a rollout, after shorter responses finish and the active batch shrinks, a small number of long responses often determine the makespan (Fig. 1) while leaving compute underutilized. This underutilized compute provides the runtime budget that SD can exploit. Section B provides analysis of the long-tail completion behavior of rollout.

3. EfficientRollout: From RL Rollout Characteristics to Runtime Design

EfficientRollout combines three components: a weight-quantized self-drafter, a regime-aware SD toggle, and an adaptive draft-length policy. Algorithm 1 summarizes their runtime interaction.

3.1. Weight-Quantized Drafter for Self-SD

Self-SD. Among the SD designs in Tab. 1, self-SD with a target-induced drafter (Zhang et al., 2024; Yue et al., 2026; Tiwari et al., 2025) is well suited to RL, where the target model evolves throughout training. Because the drafter is derived directly from the current target, it remains synchronized with the evolving policy without RL-specific drafter pretraining or online adaptation. This contrasts with learned auxiliary drafters, which can become stale relative to the evolving target policy unless continually adapted online to track it (Zhang et al., 2026).

Rollout-tail latency decomposition. We decompose rollout-tail decoding latency, T_{decode}, to identify which component a self-SD drafter should make cheaper. As shown in Fig. 2b, dense projection time T_{dense}, including QKVO, FFN, and LM-head projections, accounts for around 90% of total latency, exceeding attention time T_{attn} by more than $10\times$. This is consistent with modern LLM architectures that reduce KV-cache sizes (Shazeer, 2019; Ainslie et al., 2023; Liu et al., 2024; Guo et al., 2025) and with analyses showing parameter loading dominates latency in small-batch memory-bound regimes (Sadhukhan et al., 2025).

Weight-quantized drafting. We induce a weight-quantized drafter (Tiwari et al., 2025) from the current target model to reduce the weight-loading cost of the linear layers that dominate T_{dense}, lowering the drafter-to-target latency ratio T_q/T_p in Eq. (1). Specifically, we apply lightweight RTN quantization to the FFN and QKVO projection layers at the beginning of each training step, following prior practice (Lin et al., 2024). We use 4-bit weights (W4) as a practical latency–acceptance trade-off: lower precision makes drafting substantially cheaper while preserving sufficient block efficiency for effective SD. In contrast, sparse-

attention drafting, another representative self-SD approach, mainly targets T_{attn}, which contributes far less than T_{dense} to rollout-tail latency in this regime. Layer skipping is also a self-SD option; however, fixed skipping patterns often provide limited practical speedup (Xia et al., 2024; Song et al., 2026), while adaptive skipping is difficult to combine with static inference optimizations in modern serving engines, such as CUDA graph pre-capture (Xia et al., 2025; Chen et al., 2025b; Kwon et al., 2023). Further details are provided in Secs. C.2 and C.3.

3.2. Regime-Aware SD Toggle Policy

Early rollout phases have large active batches that can saturate compute, leaving little underutilized compute for target verification (*i.e.* high T_V in Eq. (1)). As a result, SD can be slower than AR decoding despite high block efficiency. As shorter responses finish, the active batch shrinks and decoding enters a low-batch tail where SD becomes beneficial. To decide when to enable SD, we use a roofline model to predict speedup (Sec. 2.2), activating SD only when the prediction is favorable.

SD toggle policy. To implement this decision, we first define memory-side and compute-side cost estimates for target, draft, and verification execution:

$$M_T = \frac{W_T + \kappa_{\text{eff}} BS}{\text{BW}_{\text{eff}}}, \quad M_D = \frac{\eta_D W_D + \kappa_{\text{eff}} BS}{\text{BW}_{\text{eff}}},$$
$$C = \frac{B(C_{\text{dense}} + SC_{\text{attn}})}{\text{F}_{\text{eff}}}.$$

Here, M_T and M_D estimate data-movement costs for one target or quantized-drafter forward pass, including weight and effective KV-cache traffic, while C estimates the arithmetic cost. W_T and W_D denote the target and drafter weight sizes, respectively, and C_{dense} and C_{attn} denote architecture-determined dense and attention compute costs. Because the target and self-drafter share the same architecture, they share the same compute term; the quantized drafter's reduced weight traffic is captured by η_D in the memory-side term. Using Eq. (2), we instantiate the roofline latency for target decoding, quantized-drafter decoding, and target verification as

$$\widehat{T_p} = \max\{M_T, C\} + c_T B,$$
$$\widehat{T_q} = \max\{M_D, C\} + c_D B,$$
$$\widehat{T_V} = \max\{M_T, (\gamma+1)C\} + c_V B.$$

The overhead terms capture non-ideal overlap and batch-dependent residual costs. Following Eq. (1), we activate SD when the predicted speedup exceeds a safety margin $\epsilon \geq 0$:

$$\pi_{\text{SD}}(B, S, \tau, \gamma) = \mathbf{1}\left[\frac{\tau \widehat{T_p}}{\gamma \widehat{T_q} + \widehat{T_V}} \geq 1 + \epsilon\right]. \quad (3)$$

Once activated, SD remains enabled until the end of the rollout, as the active batch size monotonically decreases toward the SD-beneficial tail regimes.

Calibrated parameters. The fitted terms κ_{eff}, η_D, and c_T, c_D, c_V adapt the roofline model to different model and system configurations, capturing effective KV-cache traffic, quantized-drafter effects, and residual per-batch overheads for target, draft, and verification execution, respectively. These parameters are calibrated once per model–hardware pair using a pre-hoc profiling sweep and can be reused across RL runs. Section E.3 provides calibration details.

3.3. Adaptive Draft-Length Policy

Acceptance behavior evolves during RL. The weight-quantized drafter can become more effective over training, increasing block efficiency as RL sharpens the target policy. In Fig. 2c, target-output entropy and token-level agreement between p and its weight-quantized copy $Q(p)$ show a strong negative correlation ($r = -0.99$). As training progresses, RL can reduce target-output entropy (Jin et al., 2025; Cui et al., 2025), sharpening the target policy by concentrating probability mass on a smaller set of high-reward tokens (Huang et al., 2025; Yang et al., 2025b). A sharper distribution can make top-token decisions less sensitive to small quantization-induced perturbations, increasing agreement between p and $Q(p)$. These observations motivate adaptive draft budgets that increase as acceptance improves over training. Section D provides full experimental results for policy sharpening and block-efficiency improvement.

Adaptive policy for draft length γ. To exploit the increasing block efficiency observed during RL training, we adapt the draft length γ using the measured block efficiency τ as a proxy for current drafter quality. We maintain an ordered draft-length set $\Gamma = [\gamma_{\text{low}}, \ldots, \gamma_{\text{high}}]$ and update γ only when the block-efficiency condition persists for P consecutive training steps. Specifically, we increase γ when the observed block efficiency is close to the current draft-length ceiling, and decrease it when acceptance is too low to amortize draft computation. The patience window P prevents transient block-efficiency fluctuations across steps. This avoids post-hoc draft-length sweeps, instead adapting γ online from block-efficiency feedback.

4. Experiments

4.1. Setup

We evaluate Qwen2.5-{7B,14B} (Yang et al., 2024) on SimpleRL-8k-hard and Llama3.1-8B (Grattafiori et al., 2024) on SimpleRL-8k-medium, following the math-domain RLVR recipe (Zeng et al., 2025b), accelerating GRPO (Shao et al., 2024) training with train batch size

Table 2. End-to-end training acceleration across models and rollout decoding methods, averaged over 100 training steps ($\bar{\gamma}$ denotes average draft length). Signed percentages next to timing results denote relative changes from the veRL (AR) baseline.

Model	Drafting Method	Prep. (s)	τ / $\bar{\gamma}$	Rollout Gen. (s)	Step Time (s)	Final Acc. (%)
Qwen2.5-7B	veRL (AR) [47; 27]	–	–	82.4	132.6	71.4
	History-based† [33]	1.1	N/A	86.5 (+4.9%)	133.8 (+0.9%)	73.0
	Learned auxiliary	2.2	2.1 / 3.0	81.9 (-0.6%)	132.1 (-0.4%)	72.0
	Quantized self-SD	1.3	7.5 / 7.0	75.6 (-8.2%)	127.6 (-3.7%)	72.6
	EfficientRollout	1.3	8.6 / 8.2	66.3 (-19.6%)	115.7 (-12.7%)	73.2
Qwen2.5-14B	veRL (AR)	–	–	126.6	221.1	76.0
	History-based†	1.1	N/A	129.5 (+2.3%)	218.9 (-1.0%)	75.0
	Learned auxiliary	2.1	2.2 / 3.0	115.4 (-8.9%)	213.8 (-3.3%)	75.2
	Quantized self-SD	2.6	5.6 / 5.0	135.0 (+6.7%)	225.4 (+1.9%)	76.6
	EfficientRollout	2.6	7.0 / 6.6	105.3 (-16.8%)	197.2 (-10.8%)	76.0
Llama3.1-8B	veRL (AR)	–	–	126.4	186.9	53.6
	History-based†	1.1	N/A	131.1 (+3.7%)	187.4 (+0.2%)	50.8
	Learned auxiliary‡	2.6	2.0 / 3.0	172.8 (+36.7%)	234.8 (+25.6%)	52.2
	Quantized self-SD	1.4	5.4 / 5.0	118.9 (-5.9%)	178.2 (-4.7%)	52.8
	EfficientRollout	1.4	5.4 / 5.0	112.9 (-10.7%)	172.2 (-7.9%)	53.4

† History-based drafting uses variable-length prefix matching, so block efficiency is not directly comparable (see Sec. G.4). ‡ Learned auxiliary drafting on Llama3.1-8B shows a large slowdown in our setting (see Sec. G.5).

128, group size 8, rollout temperature 1.0, maximum response length 8k, and 100 steps on a single node of 8 A100-80GB GPUs. All methods share the veRL (Sheng et al., 2025)+vLLM (Kwon et al., 2023) rollout stack; EfficientRollout implements the quantized drafter with the W4A16 Marlin kernel (Frantar et al., 2025), uses a safety margin $\epsilon = 0.05$ for SD toggle policy, and uses $\Gamma = \{5, 7, 9, 11\}$ with $\alpha_{\text{up}} = 0.94$, $\alpha_{\text{down}} = 0.85$, and $P = 2$ for adaptive draft-length policy. We compare EfficientRollout against four baselines: **veRL (AR)**, accelerated AR inference on the same backend without SD; **history-based drafting**, the rollout-history-based Spec-RL (Liu et al., 2025a) using prefix matching in the same stack, with its best-performing lenience factor $e^{0.5}$ (lossy); **learned auxiliary drafting**, an EAGLE3 (Li et al., 2026)-based auxiliary drafter implemented on veRL+vLLM following FastRL (Hu et al., 2026) and NeMo RL (Iso et al., 2026) with fixed $\gamma = 3$; and **quantized self-SD**, always-on SD with the weight-quantized drafter from Sec. 3 at the best fixed $\gamma^* \in \{3, 5, 7\}$. Full experimental details are provided in Sec. F.

4.2. End-to-End RL Training Acceleration

End-to-end timing. Table 2 shows that EfficientRollout achieves the largest latency reduction for every evaluated model. Despite a per-step quantization overhead of 1.3–2.6 s, EfficientRollout reduces rollout-generation time by 15.5% on average and by up to 19.6%, yielding an end-to-end training-step latency reduction of up to 12.7% after accounting for all RL operations. By contrast, the history-based drafting baseline has lightweight preparation overhead (1.1 s) but increases rollout-generation time by 2.3–4.9% and does not consistently improve step time, as reused prefixes are too short to amortize verification overhead.

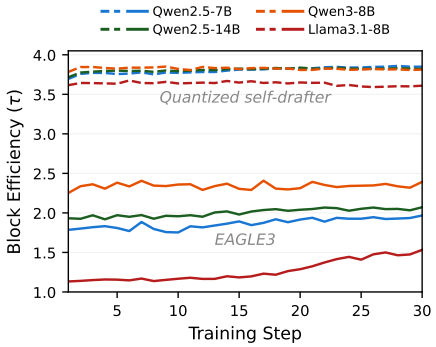
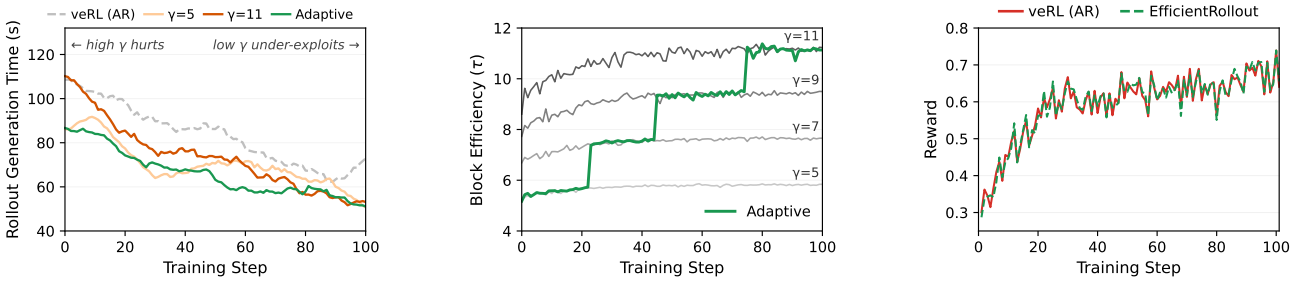


Figure 3. **Additional block efficiency study on DAPO-Math-17K.** Public or broadly pretrained EAGLE3 drafters remain at lower block efficiency than quantized self-drafters.

The learned auxiliary drafting baseline reduces rollout-generation time by 0.6–8.9% and step time by 0.4–3.3% on the Qwen models, but slows down Llama3.1-8B rollout by 36.7%. Quantized self-SD improves rollout and step latency on Qwen2.5-7B and Llama3.1-8B but still falls short of EfficientRollout, and slows down on Qwen2.5-14B where the larger quantized drafter makes harmful high-batch speculation more expensive. Detailed analyses of the history-based drafting and learned auxiliary drafting slowdowns are in Secs. G.4 and G.5.

Block efficiency and realized speedup. EfficientRollout maintains the highest average block efficiency while adapting the draft length, reaching $\tau = 8.6$, 7.0, and 5.4 on Qwen2.5-7B, Qwen2.5-14B, and Llama3.1-8B, respectively. Quantized self-SD also achieves high block efficiency, but its fixed draft length cannot exploit training-time improvements in τ, while learned auxiliary drafting remains limited to moderate block efficiency ($\tau = 2.0$–2.2). Thus,



(a) Rollout generation time *(b)* Block efficiency *(c)* Training reward

Figure 4. **Adaptive γ policy and training dynamics on Qwen2.5-7B.** (a) Adaptive γ reduces rollout-generation time by avoiding overly large drafts early and exploiting longer drafts later. (b) The controller raises γ as block efficiency τ improves during training. (c) EfficientRollout follows the veRL (AR) reward trajectory, indicating preserved training dynamics.

despite a higher per-draft computation cost, the quantized self-drafter's stronger block efficiency, regime-aware activation, and adaptive γ yield the largest end-to-end speedup.

Learned auxiliary drafting in RL workloads. We probe the difficulty of obtaining an auxiliary drafter aligned with long, high-temperature RL rollouts by evaluating EAGLE3 drafters in the NeMo RL SD stack (Iso et al., 2026) on DAPO-Math-17K (Yu et al., 2026) with $\gamma = 3$. As shown in Fig. 3, the auxiliary drafters (RedHatAI, 2025a; 2026) achieve 1.2–2.4 block efficiency over the first 30 RL steps, whereas the quantized self-drafter reaches 3.6–3.9 under the same setting. Realizing effective speedups with learned auxiliary drafting may instead require a more specialized training pipeline (Zhou et al., 2024; Iso et al., 2026), such as collecting in-distribution rollouts and training drafters offline on target-generated data. EfficientRollout provides a simpler point in this trade-off: it requires no external drafter checkpoint or offline pretraining, yet achieves high block efficiency from the start of RL training. Detailed analyses with additional auxiliary-drafter checkpoints are in Sec. G.6.

Training dynamics and quality preservation. EfficientRollout accelerates training while preserving training dynamics: Tab. 2 shows final accuracy matches veRL (AR) across models, and Fig. 4c shows that EfficientRollout tracks the veRL (AR) reward trajectory on Qwen2.5-7B, with the same trend across all model series in Sec. G.2.

4.3. Ablation of EfficientRollout Components

Regime-aware SD toggle policy. We validate the roofline-based toggle boundary in Sec. E.4. Figure 5 isolates the effect of regime-aware toggling by comparing always-on quantized self-SD with the same drafter under the SD toggle policy. On Qwen2.5-7B, disabling SD for only the initial 11% of decoding steps improves the rollout-generation reduction from 11.5% to 19.1%. The same trend holds on Qwen2.5-14B and Llama3.1-8B, where toggling improves the reduction from 0.4% to 9.9% and from 3.3% to 9.5%, respectively. Since these gains appear despite similar block efficiency between the two settings, high block efficiency alone does not imply realized speedup; SD must also be activated in system-beneficial regimes.

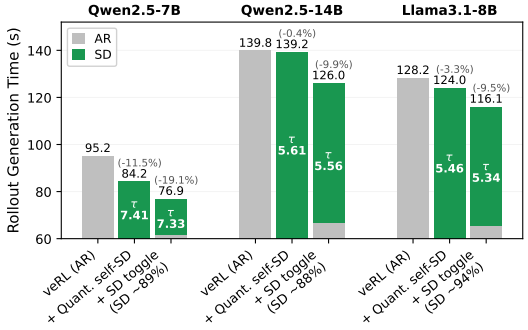


Figure 5. **Effect of regime-aware SD toggling.** Average rollout time over the first 50 training steps, showing that toggling avoids harmful early-regime speculation across all evaluated models.

Adaptive draft-length policy. The adaptive draft-length policy outperforms fixed-γ choices by starting conservatively and increasing γ as block efficiency improves. We isolate this effect by comparing EfficientRollout against variants that use the same quantized self-drafter and toggle policy but fix γ throughout training. As shown in Fig. 4a, fixed $\gamma = 5$ gives lower rollout-generation time than larger fixed draft lengths in early training, whereas fixed $\gamma = 11$ becomes better in later steps. Using the observed block efficiency τ, adaptive draft-length policy raises γ from 5 to 7, 9, and 11 at steps 24, 46, and 76, respectively (Fig. 4b). This yields a 19.6% rollout-generation reduction, compared with 13.5% for fixed $\gamma = 5$ and 11.8% for fixed $\gamma = 11$.

5. Conclusion

We present EfficientRollout, a system-aware self-SD framework that combines a target-induced quantized drafter, dynamic SD toggling, and adaptive draft budgets to reduce rollout and end-to-end latency without separate drafter pretraining, online adaptation, or invasive pipeline changes.

Acknowledgments

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 04-26-03-0081, Energy-Efficient Training–Inference System Optimization for Reinforcement Learning-Based Post-Training). This work was also supported by the "Advanced GPU Utilization Support Program" funded by the Government of the Republic of Korea (Ministry of Science and ICT). We thank all members of the FuriosaAI team for their support, with special thanks to Hanjoon Kim and June Paik for their vision and commitment to research.

References

Aeala. ShareGPT_Vicuna_unfiltered. https://huggingface.co/datasets/Aeala/ShareGPT_Vicuna_unfiltered, 2023. Hugging Face dataset. Accessed: 2026-05-03.

Ainslie, J., Lee-Thorp, J., De Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 4895–4901, 2023.

AngelSlim. Qwen3-8B_eagle3. https://huggingface.co/AngelSlim/Qwen3-8B_eagle3, 2025. Hugging Face model. Accessed: 2026-06-06.

Cai, T., Li, Y., Geng, Z., Peng, H., Lee, J. D., Chen, D., and Dao, T. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=PEpbUobfJv.

Chen, A., Li, A., Gong, B., Jiang, B., Fei, B., Yang, B., Shan, B., Yu, C., Wang, C., Zhu, C., et al. Minimax-m1: Scaling test-time compute efficiently with lightning attention. *arXiv preprint arXiv:2506.13585*, 2025a.

Chen, C., Borgeaud, S., Irving, G., Lespiau, J.-B., Sifre, L., and Jumper, J. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

Chen, L., Shan, R., Wang, H., Wang, L., Liu, Z., Luo, R., Wang, J., Alinejad-Rokny, H., and Yang, M. Clasp: In-context layer skip for self-speculative decoding. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 31608–31618, 2025b.

Chen, Q., Liu, Z., Sun, P., Li, S., Wang, G., Liu, Z., Wen, Y., Feng, S., and Zhang, T. Respec: Towards optimizing speculative decoding in reinforcement learning systems. In *Ninth Conference on Machine Learning and Systems*, 2026. URL https://openreview.net/forum?id=HhDSxs7x2R.

Chen, X., Xu, J., Liang, T., He, Z., Pang, J., Yu, D., Song, L., Liu, Q., Zhou, M., Zhang, Z., Wang, R., Tu, Z., Mi, H., and Yu, D. Do NOT think that much for 2+3=? on the overthinking of long reasoning models. In *Forty-second International Conference on Machine Learning*, 2025c. URL https://openreview.net/forum?id=MSbU3L7V00.

Cordonnier, J.-B., Loukas, A., and Jaggi, M. Multi-head attention: Collaborate instead of concatenate. *arXiv preprint arXiv:2006.16362*, 2020.

Cui, G., Zhang, Y., Chen, J., Yuan, L., Wang, Z., Zuo, Y., Li, H., Fan, Y., Chen, H., Chen, W., et al. The entropy mechanism of reinforcement learning for reasoning language models. *arXiv preprint arXiv:2505.22617*, 2025.

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. QLoRA: Efficient finetuning of quantized LLMs. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=OUIFPHEgJU.

Frantar, E., Castro, R. L., Chen, J., Hoefler, T., and Alistarh, D. Marlin: Mixed-precision auto-regressive parallel inference on large language models. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 239–251, 2025.

Fu, W., Gao, J., Shen, X., Zhu, C., Mei, Z., He, C., Xu, S., Wei, G., Mei, J., JIASHU, W., Yang, T., Yuan, B., and Wu, Y. AREAL: A large-scale asynchronous reinforcement learning system for language reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026. URL https://openreview.net/forum?id=X9diEuva9R.

Gholami, A., Yao, Z., Kim, S., Hooper, C., Mahoney, M. W., and Keutzer, K. Ai and memory wall. *IEEE Micro*, 44 (3):33–39, 2024.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

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

He, J., Li, T., Feng, E., Du, D., Liu, Q., Liu, T., Xia, Y., and Chen, H. History rhymes: Accelerating llm reinforcement learning with rhymerl. *arXiv preprint arXiv:2508.18588*, 2025.

He, Z., Zhong, Z., Cai, T., Lee, J., and He, D. Rest: Retrieval-based speculative decoding. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 1582–1595, 2024.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Hu, Q., Yang, S., Guo, J., Yao, X., Lin, Y., Gu, Y., Cai, H., Gan, C., Klimovic, A., and Han, S. Taming the long-tail: Efficient reasoning rl training with adaptive drafter. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 1933–1948, 2026.

Huang, A., Block, A., Foster, D. J., Rohatgi, D., Zhang, C., Simchowitz, M., Ash, J. T., and Krishnamurthy, A. Self-improvement in language models: The sharpening mechanism. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=WJaUkwci9o.

Iso, H., Mitra, T., Mondal, S., Shafipour, R., Elango, V., Kong, T., Huang, Y., Na, S., Putterman, I., Chislett, B., et al. Accelerating rl post-training rollouts via system-integrated speculative decoding. *arXiv preprint arXiv:2604.26779*, 2026.

Jin, R., Gao, P., Ren, Y., Han, Z., Zhang, T., Huang, W., Liu, W., Luan, J., and Xiong, D. Revisiting entropy in reinforcement learning for large reasoning models. *arXiv preprint arXiv:2511.05993*, 2025.

Kim, M., Hooper, C., Tomar, A., Xu, C., Farajtabar, M., Mahoney, M. W., Keutzer, K., and Gholami, A. Beyond next-token prediction: A performance characterization of diffusion versus autoregressive language models. *arXiv preprint arXiv:2510.04146*, 2025.

Kumar, K., Ashraf, T., Thawakar, O., Anwer, R. M., Cholakkal, H., Shah, M., Yang, M.-H., Torr, P. H., Khan, F. S., and Khan, S. Llm post-training: A deep dive into reasoning large language models. *arXiv preprint arXiv:2502.21321*, 2025.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pp. 19274–19286. PMLR, 2023.

Li, Y., Wei, F., Zhang, C., and Zhang, H. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=1NdN7eXyb4.

Li, Y., Wei, F., Zhang, C., and Zhang, H. EAGLE-3: Scaling up inference acceleration of large language models via training-time test. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026. URL https://openreview.net/forum?id=4exx1hUffq.

Lin, J., Tang, J., Tang, H., Yang, S., Chen, W.-M., Wang, W.-C., Xiao, G., Dang, X., Gan, C., and Han, S. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of machine learning and systems*, 6:87–100, 2024.

Liu, A., Feng, B., Wang, B., Wang, B., Liu, B., Zhao, C., Dengr, C., Ruan, C., Dai, D., Guo, D., et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

Liu, B., Wang, A., Min, Z., Yao, L., Zhang, H., Liu, Y., Zeng, A., and Su, J. Spec-rl: Accelerating on-policy reinforcement learning via speculative rollouts. *arXiv preprint arXiv:2509.23232*, 2025a.

Liu, X., Yu, J., Park, J., Stoica, I., and Cheung, A. Speculative decoding: Performance or illusion? In *Ninth Conference on Machine Learning and Systems*, 2026. URL https://openreview.net/forum?id=fzkqtezFEi.

Liu, Z., Chen, C., Li, W., Qi, P., Pang, T., Du, C., Lee, W. S., and Lin, M. Understanding r1-zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025b.

MIT HAN Lab. Qwen2.5-7B-Eagle-RL. https://huggingface.co/mit-han-lab/Qwen2.5-7B-Eagle-RL, 2025. Hugging Face model. Accessed: 2026-06-08.

open-thoughts. OpenThoughts2-1M. https://huggingface.co/datasets/open-thoughts/OpenThoughts2-1M, 2025. Hugging Face dataset. Accessed: 2026-06-08.

Ou, J., Chen, Y., et al. Lossless acceleration of large language model via adaptive n-gram parallel decoding. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, pp. 10–22, 2024.

RedHatAI. Llama-3.1-8B-Instruct-speculator.eagle3. https://huggingface.co/RedHatAI/Llama-3.1-8B-Instruct-speculator.eagle3, 2025a. Hugging Face model. Accessed: 2026-05-04.

RedHatAI. Qwen3-8B-speculator.eagle3. https://huggingface.co/RedHatAI/Qwen3-8B-speculator.eagle3, 2025b. Hugging Face model. Accessed: 2026-06-06.

RedHatAI. Qwen3-8B-Thinking-speculator.eagle3. https://huggingface.co/RedHatAI/Qwen3-8B-Thinking-speculator.eagle3, 2026. Hugging Face model. Accessed: 2026-06-06.

Sadhukhan, R., Chen, J., Chen, Z., Tiwari, V., Lai, R., Shi, J., Yen, I. E.-H., May, A., Chen, T., and Chen, B. Magicdec: Breaking the latency-throughput tradeoff for long context generation with speculative decoding. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=CS2JWaziYr.

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

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

Shao, Z., Srivatsa, V., Srivastava, S., Wu, Q., Ariyak, A., wu, X., Patel, A., Wang, J., Liang, P., Dao, T., Zhang, C., Zhang, Y., Athiwaratkun, B., Xu, C., and Wang, J. Beat the long tail: Distribution-aware speculative decoding for RL training. In *Ninth Conference on Machine Learning and Systems*, 2026. URL https://openreview.net/forum?id=kMeqqPBjSl.

Shazeer, N. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

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

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Song, M., Xia, H., Zhang, J., Leong, C. T., Xu, Q., Li, W., and Li, S. Knn-ssd: Enabling dynamic self-speculative decoding via nearest neighbor layer set optimization. In *Findings of the Association for Computational Linguistics: EACL 2026*, pp. 641–655, 2026.

Stewart, L., Trager, M., Gonugondla, S. K., and Soatto, S. The n-grammys: Accelerating autoregressive inference with learning-free batched speculation. *arXiv preprint arXiv:2411.03786*, 2024.

Tang, J., Zhao, Y., Zhu, K., Xiao, G., Kasikci, B., and Han, S. QUEST: Query-aware sparsity for efficient long-context LLM inference. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=KzACYw0MTV.

Team, K., Bai, Y., Bao, Y., Charles, Y., Chen, C., Chen, G., Chen, H., Chen, H., Chen, J., Chen, N., et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Tiwari, R., Xi, H., Tomar, A., Hooper, C. R. C., Kim, S., Horton, M., Najibi, M., Mahoney, M. W., Keutzer, K., and Gholami, A. Quantspec: Self-speculative decoding with hierarchical quantized KV cache. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=7SHbJENgHX.

Wen, X., Liu, Z., Zheng, S., Ye, S., Wu, Z., Wang, Y., Xu, Z., Liang, X., Li, J., Miao, Z., Bian, J., and Yang, M. Reinforcement learning with verifiable rewards implicitly incentivizes correct reasoning in base LLMs. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=jGbRWwIidy.

Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Xia, H., Yang, Z., Dong, Q., Wang, P., Li, Y., Ge, T., Liu, T., Li, W., and Sui, Z. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding. *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 7655–7671, 2024.

Xia, H., Li, Y., Zhang, J., Du, C., and Li, W. SWIFT: On-the-fly self-speculative decoding for LLM inference acceleration. In *The Thirteenth International Conference*

on Learning Representations, 2025. URL https://openreview.net/forum?id=EKJhH5D5wA.

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

Yang, C., Li, S., and Holtzman, A. Llm probability concentration: How alignment shrinks the generative horizon. *arXiv preprint arXiv:2506.17871*, 2025b.

Yang, Q. A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Li, C., Liu, D., Huang, F., Dong, G., Wei, H., Lin, H., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Lin, J., Dang, K., Lu, K., Bao, K., Yang, K., Yu, L., Li, M., Xue, M., Zhang, P., Zhu, Q., Men, R., Lin, R., Li, T., Xia, T., Ren, X., Ren, X., Fan, Y., Su, Y., Zhang, Y.-C., Wan, Y., Liu, Y., Cui, Z., Zhang, Z., Qiu, Z., Quan, S., and Wang, Z. Qwen2.5 technical report. *ArXiv*, abs/2412.15115, 2024. URL https://api.semanticscholar.org/CorpusID:274859421.

Yu, Q., Zhang, Z., Zhu, R., Yuan, Y., Zuo, X., YuYue, Dai, W., Fan, T., Liu, G., Liu, J., Liu, L., Liu, X., Lin, H., Lin, Z., Ma, B., Sheng, G., Tong, Y., Zhang, C., Zhang, M., Zhang, R., Zhang, W., Zhu, H., Zhu, J., Chen, J., Chen, J., Wang, C., Yu, H., Song, Y., Wei, X., Zhou, H., Liu, J., Ma, W.-Y., Zhang, Y.-Q., Yan, L., Wu, Y., and Wang, M. DAPO: An open-source LLM reinforcement learning system at scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026. URL https://openreview.net/forum?id=2a36EMSSTp.

Yuan, Z., Shang, Y., Zhou, Y., Dong, Z., Zhou, Z., Xue, C., Wu, B., Li, Z., Gu, Q., Lee, Y. J., et al. Llm inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363*, 2024.

Yue, Y., Xue, Y., and Huang, J. Specattn: Co-designing sparse attention with self-speculative decoding. *arXiv preprint arXiv:2602.07223*, 2026.

Yuhui Li. EAGLE3-LLaMA3.1-Instruct-8B. https://huggingface.co/yuhuili/EAGLE3-LLaMA3.1-Instruct-8B, 2024. Hugging Face model. Accessed: 2026-06-06.

Zeng, A., Lv, X., Zheng, Q., Hou, Z., Chen, B., Xie, C., Wang, C., Yin, D., Zeng, H., Zhang, J., et al. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*, 2025a.

Zeng, W., Huang, Y., Liu, Q., Liu, W., He, K., MA, Z., and He, J. SimpleRL-zoo: Investigating and taming zero reinforcement learning for open base models in the wild. In *Second Conference on Language Modeling*, 2025b. URL https://openreview.net/forum?id=vSMCBUgrQj.

Zhang, J., Wang, J., Li, H., Shou, L., Chen, K., Chen, G., and Mehrotra, S. Draft& verify: Lossless large language model acceleration via self-speculative decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11263–11282, 2024.

Zhang, Y., Lv, N., Wang, T., and Dang, J. FastGRPO: Accelerating policy optimization via concurrency-aware speculative decoding and online draft learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=zuGt6TYYtS.

Zhao, R., Meterez, A., Kakade, S. M., Pehlevan, C., Jelassi, S., and Malach, E. Echo chamber: RL post-training amplifies behaviors learned in pretraining. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=dp4KWuSDzj.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. SGLang: Efficient execution of structured language model programs. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=VqkAKQibpq.

Zhou, Y., Lyu, K., Rawat, A. S., Menon, A. K., Rostamizadeh, A., Kumar, S., Kagy, J.-F., and Agarwal, R. Distillspec: Improving speculative decoding via knowledge distillation. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=rsY6J3ZaTF.

Appendix

A. Related Work

A.1. Reinforcement Learning for LLMs

Frontier LLMs post-trained with RL have achieved state-of-the-art reasoning and agentic capabilities (Guo et al., 2025; Yang et al., 2025a; Team et al., 2025; Zeng et al., 2025a), enabled by advances in policy optimization (Schulman et al., 2017; Shao et al., 2024; Yu et al., 2026; Chen et al., 2025a). In particular, RL with verifiable rewards (RLVR) has become a dominant paradigm for reasoning-heavy domains such as math and coding, where supervision can be obtained from rule-based checkers or unit tests rather than learned reward models (Wen et al., 2026; Guo et al., 2025). In on-policy RL, the policy being optimized is the same policy used for rollout, which avoids degradation from policy staleness and improves training stability and model quality (Kumar et al., 2025; Fu et al., 2026). However, rollout is often the main latency bottleneck because RL tends to induce longer outputs, and AR decoding is inherently sequential. By contrast, other stages of the RL loop, including log-probability computation and policy updates, are more parallelizable.

A.2. Speculative Decoding for LLM Inference

SD accelerates target LLM decoding through a sequential drafting and parallel verification scheme (Leviathan et al., 2023; Chen et al., 2023). This draft-and-verify structure preserves the target-model distribution through rejection sampling while exploiting underutilized compute in memory-bound decoding regimes. However, in compute-bound regimes, SD can be slower than standard AR decoding because parallel verification has little underutilized compute to exploit (Sadhukhan et al., 2025). SD methods differ mainly in how they construct the drafter (Xia et al., 2024). Canonical SD uses an independent smaller drafter model to approximate the target model (Leviathan et al., 2023; Chen et al., 2023). To make drafting much cheaper, non-parametric drafting methods generate drafts from text using n-gram, prefix-tree, or suffix-tree matching (He et al., 2024; Ou et al., 2024; Stewart et al., 2024). Learned auxiliary drafting methods attach lightweight modules to the target model for efficient token prediction, such as FFN heads or one-layer drafters (Li et al., 2024; Cai et al., 2024). Self-speculative decoding (self-SD) instead derives the drafter directly from the target model via layer skipping (Zhang et al., 2024), sparse attention (Yue et al., 2026), or quantization (Tiwari et al., 2025).

A.3. Speculative Decoding for Rollout in Reinforcement Learning

Several prior works use SD to accelerate rollout, a major RL latency bottleneck. Table 1 summarizes the main drafting methods for RL-specific SD: **(1) History-based drafting** methods adapt non-parametric drafting by reusing rollouts from previous epochs as draft proposals, e.g., via prefix- or suffix-based matching (Liu et al., 2025a; He et al., 2025; Shao et al., 2026). They are easy to deploy and require no parameterized drafter, but often yield short accepted drafts because past trajectories sparsely cover future rollouts. They also require a warm-up period to collect rollout history and may rely on acceptance-boosting heuristics such as reduced temperatures or lossy lenience settings. **(2) Learned auxiliary drafting** methods, often inspired by Li et al. (2024), can achieve higher block efficiency in RL rollouts when their drafters are sufficiently adapted to long-reasoning rollout distributions (Zhang et al., 2026; Chen et al., 2026; Hu et al., 2026; Iso et al., 2026). However, such drafters are not generally available as public checkpoints, so obtaining them often requires dedicated drafter pretraining or several warm-up epochs before achieving effective speedup (Zhang et al., 2026). Tracking the evolving RL policy also requires online adaptation, adding system complexity and management overhead (Hu et al., 2026). **(3) Target-induced drafting** methods correspond to self-SD methods that derive the drafter solely from the current target model. Our work instantiates this category with **EfficientRollout**, which uses a weight-quantized copy of the target model as the drafter, staying synchronized with the evolving policy while maintaining high block efficiency.

B. Extended Analysis of Shrinking-Batch Dynamics

Figure 6 provides additional evidence that rollout generation remains the dominant bottleneck and exhibits a long-tail completion pattern across models. For Qwen2.5-7B, Fig. 6b shows the step-time decomposition over the first 20 training steps. Although the total step time is shorter than that of Llama3.1-8B-Instruct, rollout is still the largest component. Across the first 20 steps, rollout accounts for 58.0–71.4% of the measured four-phase step time, with a mean of 64.3%. This confirms that rollout generation is the primary systems bottleneck for Qwen2.5-7B as well.

The completion curves show why rollout dominates wall-clock time. For Llama3.1-8B-Instruct, Fig. 6a shows a pronounced long tail at the first step of the second epoch. Roughly 50% of requests finish within 10 s, 90% within 16 s, and 99% within 30 s, but the final request completes only after about 123 s. Thus, a small number of long generations determine the rollout makespan. Qwen2.5-7B exhibits the same mechanism, although with a milder tail. As shown in Fig. 6c, about 50% of requests finish within 13 s, 90% within 22 s, and 99% within 30 s, while the total rollout makespan is about 88 s. Compared with Llama3.1-8B-Instruct, the Qwen tail is shorter, but the final few requests still domi-

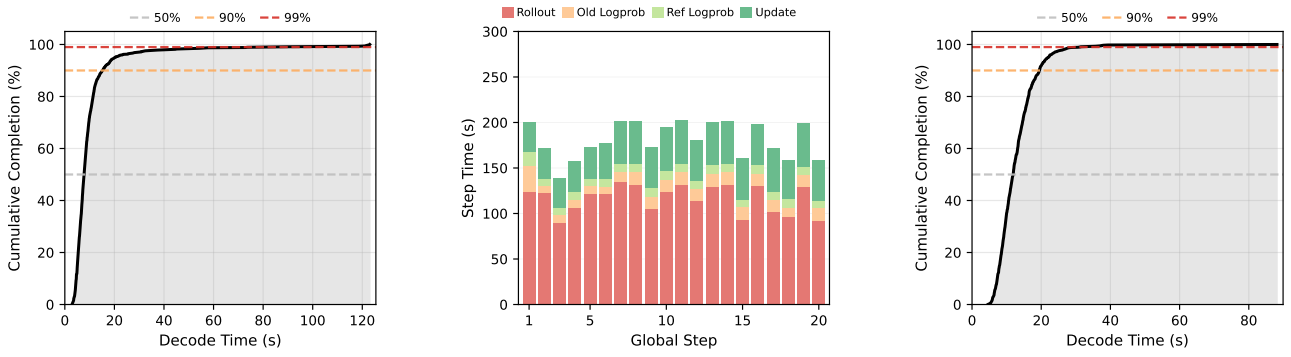


Figure 6. Extended rollout-tail analysis. (a) Cumulative request-completion curve for Llama3.1-8B-Instruct at the first step of the second epoch; the corresponding step-time decomposition is shown in Fig. 2a. (b) Step-time decomposition over the first 20 training steps for Qwen2.5-7B. (c) Cumulative request-completion curve for Qwen2.5-7B at the first step of the second epoch.

nate the end of rollout generation. Once most requests have completed, the active batch size sharply decreases, leaving a shrinking-batch tail where direct acceleration of the remaining decode steps is most valuable.

C. System Rationale for Weight-Quantized Self-Drafting in RL Rollouts

C.1. Detailed Decode-Time Decomposition in Rollout-Tail Regimes

We follow the roofline-based simulation methodology of prior work (Kim et al., 2025) and decompose the single-token decode cost for representative rollout-tail regimes, where each component latency is estimated as the maximum of its compute time and memory-transfer time. Table 3 reports the resulting breakdown for Qwen2.5-7B/14B (Yang et al., 2024) and Llama3.1-8B-Instruct (Grattafiori et al., 2024).

The same qualitative trend holds across the evaluated models. In shrinking-batch rollout regimes, FFN and QKVO projection together dominate the decode cost, while attention contributes only a relatively small fraction. These additional results further support the design choice of quantization-based self-SD, which directly reduces the dominant linear component, rather than sparse-attention drafting, which mainly targets the much smaller attention component. We do not consider KV-cache quantization in this work, although it could become useful in much longer rollout settings, e.g., beyond 64k tokens, where attention cost may again become significant.

C.2. Practical Trade-offs between Quantized and Sparse-Attention Self-Drafting

From a practical systems perspective, sparse-attention drafting interacts poorly with the serving engine. To achieve high acceptance, sparse-attention self-SD typically requires token-wise sparsity patterns that are updated dynamically at each draft step (Tang et al., 2024; Yue et al., 2026). However, such fine-grained sparsity is difficult to realize efficiently in vLLM, whose memory management is page-granular rather than token-granular (Kwon et al., 2023). If we instead use vLLM-friendly sparse patterns, such as page-level sparsity or sliding-window attention, acceptance drops noticeably. We implemented a sparse-attention self-SD method following prior work (Yue et al., 2026). Under sparsity budgets that are cheap enough to yield speedup in practice, e.g., retaining 256 tokens, the resulting drafter remains relatively weak: with $\gamma = 5$, block efficiency τ typically stays around 3–4. In contrast, a 4-bit weight-quantized drafter works well in our regime. Even with the simplest asymmetric RTN (round-to-nearest) quantization, we achieve block efficiency $\tau = 5.2$ at $\gamma = 5$. This design also naturally synergizes with vLLM through optimized W4A16 Marlin kernels (Frantar et al., 2025). Taken together, these observations make quantized self-SD the most effective and practical self-drafting strategy for RL rollout acceleration in our setting.

C.3. W4 versus W8: Latency–Acceptance Trade-off

Table 4 compares W4 and W8 RTN quantized drafters. As expected, W8 provides higher drafting quality and achieves larger block efficiency across draft lengths. However, W4 remains reasonably accurate while making the draft path substantially cheaper: in the idealized memory-bound model, W4 reduces the draft-to-target latency ratio to 0.360, compared with 0.573 for W8. This difference is important because SD speedup depends not only on accepted length, but also on whether the drafter is cheap enough to amortize target verification overhead.

W8 also introduces a larger memory footprint in the rollout engine. For example, its drafter weights require roughly twice the memory of W4, reaching about 16.3 GiB for the

Table 3. Simulation-based single-token decode-time breakdown for Qwen2.5-7B, Qwen2.5-14B, Llama3.1-8B-Instruct at different sequence lengths and batch sizes representing RL rollout tail regimes. Results assume a single A100-80GiB SXM GPU, FP16 inference, no tensor parallelism, and full attention. Bold entries indicate weight-loading costs targeted by quantization; sparse attention targets the attention term.

Model	(Seq, Batch)	FFN	QKVO Proj	Attn ($QK \cdot V$)	LM Head	Nonlinear
Qwen2.5-7B	(2k, 8)	**75.2%**	**10.9%**	6.2%	7.2%	0.5%
	(2k, 4)	**77.8%**	**11.2%**	3.2%	7.4%	0.3%
	(4k, 2)	**77.9%**	**11.2%**	3.2%	7.4%	0.2%
	(8k, 1)	**78.0%**	**11.2%**	3.2%	7.5%	0.1%
Qwen2.5-14B	(2k, 8)	**65.0%**	**19.3%**	10.3%	5.0%	0.4%
	(2k, 4)	**68.7%**	**20.4%**	5.4%	5.2%	0.2%
	(4k, 2)	**68.8%**	**20.4%**	5.4%	5.3%	0.1%
	(8k, 1)	**68.8%**	**20.4%**	5.4%	5.3%	0.1%
Llama3.1-8B	(2k, 8)	**65.4%**	**15.6%**	12.4%	6.1%	0.5%
	(2k, 4)	**69.9%**	**16.7%**	6.7%	6.5%	0.3%
	(4k, 2)	**70.0%**	**16.7%**	6.7%	6.5%	0.2%
	(8k, 1)	**70.0%**	**16.7%**	6.7%	6.5%	0.1%

Table 4. W4–W8 latency–acceptance trade-off. We report $\widehat{T_q}/\widehat{T_p}$ predicted by our roofline model under memory-bound, zero-overhead assumptions, and block efficiency τ for $\gamma \in \{3, 5, 7\}$ measured on Qwen2.5-7B-Instruct at batch size 1 and sequence length 2k.

Drafter	Idealized $\widehat{T_q}/\widehat{T_p}$	τ ($\gamma = 3$)	τ ($\gamma = 5$)	τ ($\gamma = 7$)
RTN W8	0.573	3.94	5.87	7.79
RTN W4	0.360	3.59	5.18	6.70

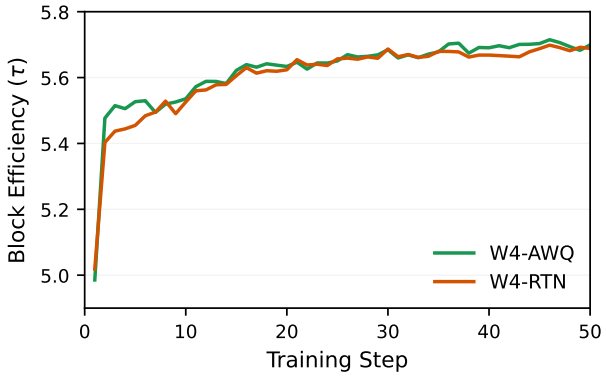


Figure 7. Block efficiency τ over the first 50 training steps on Qwen2.5-7B with W4-RTN and W4-AWQ drafters.

Qwen2.5-14B drafter. This additional memory pressure is undesirable during RL rollouts, where persistent training weights, rollout weights, and vLLM KV-cache memory already compete for GPU capacity. Moreover, although W8 improves block efficiency, its draft path is not sufficiently cheaper than target AR decoding to reliably translate the higher acceptance into end-to-end speedup when considering realistic overheads.

C.4. Choosing RTN for Step-Wise Drafter Refresh

We also experimented with an activation-aware quantization (AWQ) pipeline (Lin et al., 2024) for constructing the W4 drafter. We collect hidden-state statistics from the previous training step and use them to perform AWQ quantization, with the goal of improving drafter quality over naive round-to-nearest (RTN) quantization. Figure 7 compares the block efficiency τ achieved by W4-RTN and W4-AWQ on Qwen2.5-7B over the first 50 training steps. W4-AWQ provides a small advantage during the first few steps, but the gap quickly disappears as the RTN drafter quality improves and saturates. This suggests that, for Qwen2.5-7B, simple RTN quantization is already sufficient for maintaining a high-quality self-drafter during RL training.

The main drawback of AWQ-style quantization is its additional overhead. Unlike RTN, activation-aware methods require collecting calibration data and running a more expensive quantization procedure before each drafter refresh. Since EfficientRollout refreshes the drafter every training step, this overhead can become a significant burden unless the quantization pipeline is heavily optimized. Nevertheless, less lossy quantization may be useful for models whose RTN-quantized drafter has low initial acceptance. We leave more advanced step-wise quantization schemes as future work.

We also do not use bitsandbytes (BNB) 4-bit quantization (Dettmers et al., 2023). Although BNB-style 4-bit quantization can provide more calibrated low-bit weights, its execution path is not as well optimized for production-grade vLLM rollout inference as the AWQ-compatible W4A16 Marlin kernels used in our implementation.

D. Policy Sharpening Improves Quantized Drafter Alignment

We analyze how RL training affects quantized self-drafting under a fixed SD configuration, without the regime-aware toggle or adaptive draft-length control. At each training step, we construct the drafter by applying vanilla round-to-nearest (RTN) 4-bit quantization to the current target model, and use this quantized copy to propose drafts with a fixed draft length. We measure two quantities: first-token acceptance rate, defined as the fraction of SD iterations in which the first drafted token is accepted by the full-precision target model, and block efficiency τ, defined as the average number of target-distributed tokens produced per SD iteration. The former directly reflects local drafter–target alignment at the current prefix, while the latter captures the resulting effectiveness of each SD block.

Figure 8 shows the target-policy entropy and first-token acceptance rate over training for Qwen2.5-7B, Qwen2.5-14B, and Llama3.1-8B-Instruct. Across all three models, RL training reduces policy entropy while first-token acceptance steadily increases, with strong negative Pearson correlations ranging from -0.96 to -0.99. Figure 9 shows that block efficiency follows the same trend, with Pearson correlations ranging from -0.92 to -0.98. These results suggest that RL post-training sharpens the target distribution, making small quantization-induced perturbations less likely to change accepted draft tokens and thereby improving block efficiency over training.

E. Implementation and Calibration Details of EfficientRollout

E.1. Full Runtime Policy

Algorithm 1 summarizes how EfficientRollout refreshes the quantized drafter, toggles SD during rollout, and adapts the draft length across training steps.

Algorithm 1 EfficientRollout Runtime Policy

Parameter: initial draft length γ_0, draft-length set $\Gamma = [\gamma_{\text{low}}, \ldots, \gamma_{\text{high}}]$, toggle margin ϵ, block-efficiency thresholds $\alpha_{\text{up}}, \alpha_{\text{down}}$, patience P
Input: target model, rollout states with runtime batch size B, and sequence length S.

Initialize $\gamma \leftarrow \gamma_0$ and $\tau_{\text{prev}} \leftarrow \gamma_0$
for training timestep t **do**
  Refresh the drafter by quantizing the current target model
  Initialize `sd_on` $\leftarrow$ **false**
  while rollout not finished **do**
    if `sd_on` $=$ **false** and $\pi_{\text{SD}}(B, S, \tau_{\text{prev}}, \gamma) = 0$ **then**
      Decode autoregressively
    else if `sd_on` $=$ **false** and $\pi_{\text{SD}}(B, S, \tau_{\text{prev}}, \gamma) = 1$ **then**
      `sd_on` $\leftarrow$ **true**
    else
      Decode with SD using draft length γ
    end if
  end while
  Measure step-level block efficiency τ
  if $\min_{t' \in [t-P+1,t]} \tau_{t'} \geq 1 + \gamma\alpha_{\text{up}}$ and $\gamma < \gamma_{\text{high}}$ **then**
    $\gamma \leftarrow \text{next}_{\Gamma}(\gamma)$
  else if $\max_{t' \in [t-P+1,t]} \tau_{t'} \leq 1 + \gamma\alpha_{\text{down}}$ and $\gamma > \gamma_{\text{low}}$ **then**
    $\gamma \leftarrow \text{prev}_{\Gamma}(\gamma)$
  end if
  $\tau_{\text{prev}} \leftarrow \tau$
  Run standard post-rollout optimization
end for

E.2. KV-Cache Sharing between the Quantized Drafter and Target

Figure 10 illustrates the shared-KV execution pipeline used by EfficientRollout. The quantized drafter shares the target model's KV-cache storage, so drafting does not require an additional KV cache for the draft model. At the start of each SD iteration, the prefix KV cache consists only of clean states produced by the full-precision target model. The W4 drafter then proposes γ tokens using quantized weights and writes provisional KV states for the drafted positions into the shared KV cache.

The target model verifies the drafted tokens in parallel using full-precision weights. During verification, the target overwrites the provisional KV entries at the verified positions with clean target-generated KV states and performs rejection sampling to preserve the target-model sampling distribution. If all drafted tokens are accepted, the target additionally samples the bonus token and the next SD iteration continues from the updated clean KV cache. If a drafted token is rejected, the target resamples at the first rejected position from the adjusted target distribution, and all KV states after that position are discarded. Thus, every new SD iteration begins from a prefix whose KV states are generated by the full-precision target model.

This design has two practical advantages for RL rollouts. First, sharing the KV cache avoids the memory overhead of maintaining a separate drafter KV cache, which is important when rollout generation uses long maximum response lengths. Second, because the drafter is reconstructed by quantizing the current target model at each training step, it remains synchronized with the evolving RL policy without auxiliary drafter training or online adaptation.

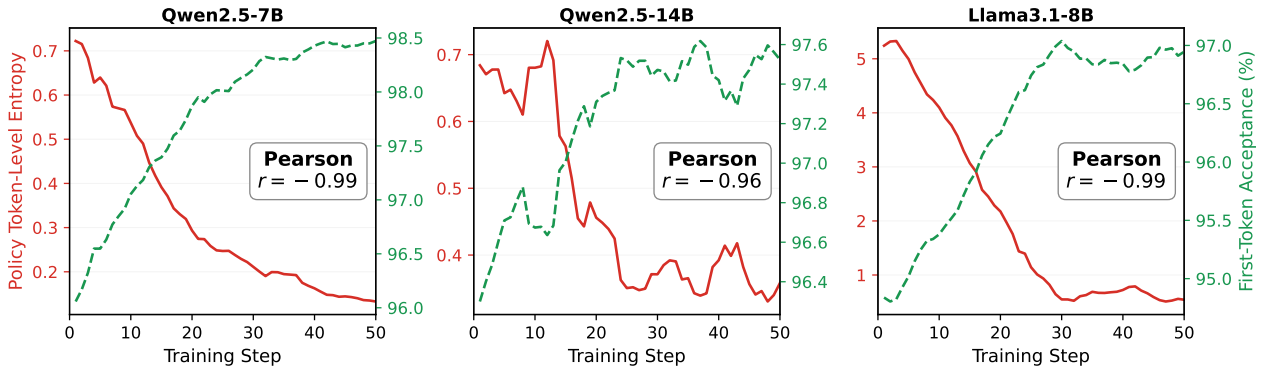


Figure 8. Policy sharpening improves quantized drafter alignment. As target-policy entropy decreases, first-token acceptance of the RTN W4 drafter increases across models.

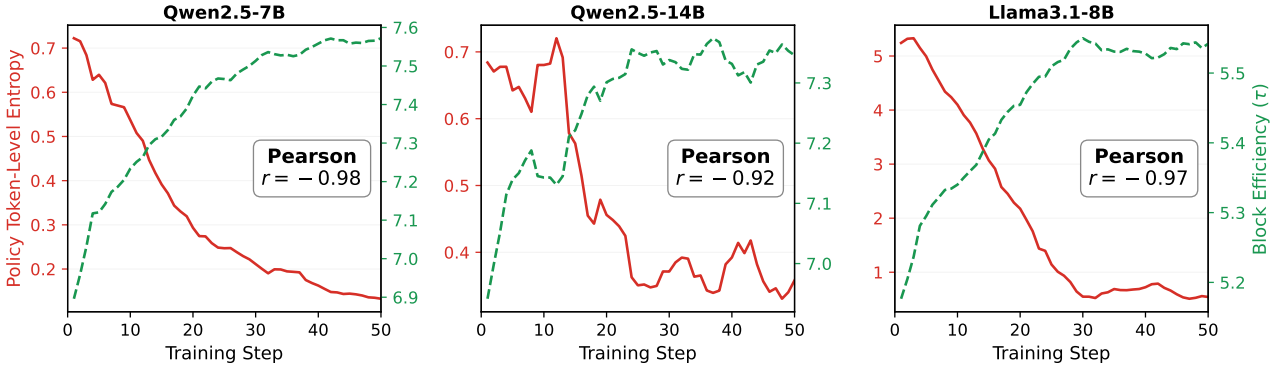


Figure 9. Policy sharpening improves quantized drafter alignment. As target-policy entropy decreases, block efficiency τ of the RTN W4 drafter increases across models.

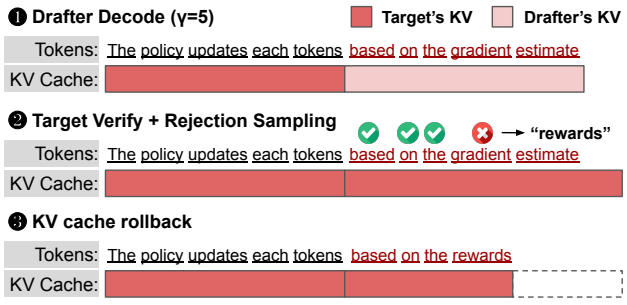


Figure 10. Quantized self-SD with shared KV cache, target verification via rejection sampling, and rollback of rejected KV cache.

E.3. Roofline-Based SD Toggle Model and Calibration

We provide additional details on the parameters used in the SD toggle model and on the calibration procedure.

Parameter details. W_T denotes the weight size of the target model in bytes, and W_D denotes the weight size of the W4 drafter in bytes. Because only the FFN and QKVO projection layers are quantized to 4 bits, while the embedding layer, LM head, and normalization layers remain in higher precision, the ratio W_D/W_T is not exactly 25%; in practice, it is about 33–36% of the original model size. C_{dense} is the per-token dense compute cost, including FFN, QKVO projection, and LM-head computation, and C_{attn} is the per-token attention compute cost. These quantities can be determined statically by the model architecture.

The effective per-token KV-cache traffic is modeled as

$$\kappa_{\text{eff}} = \rho_\kappa \, \kappa_{\text{theoretical}},$$

where $\kappa_{\text{theoretical}}$ is the theoretical KV traffic determined by the model configuration,

$$\kappa_{\text{theoretical}} = (\#\text{layers}) \times 2 \times (\#\text{KV heads}) \times (\text{head dim}) \times 2 \text{ bytes},$$

with the factor of 2 for key/value tensors and the final 2 bytes corresponding to FP16 precision. This quantity can be computed statically from the model architecture. The scaling factor ρ_κ is typically smaller than 1, reflecting effects such as cache reuse and overlap with other operations.

The factor η_D captures the practical overhead of W4A16 drafting. Although weight quantization reduces the raw

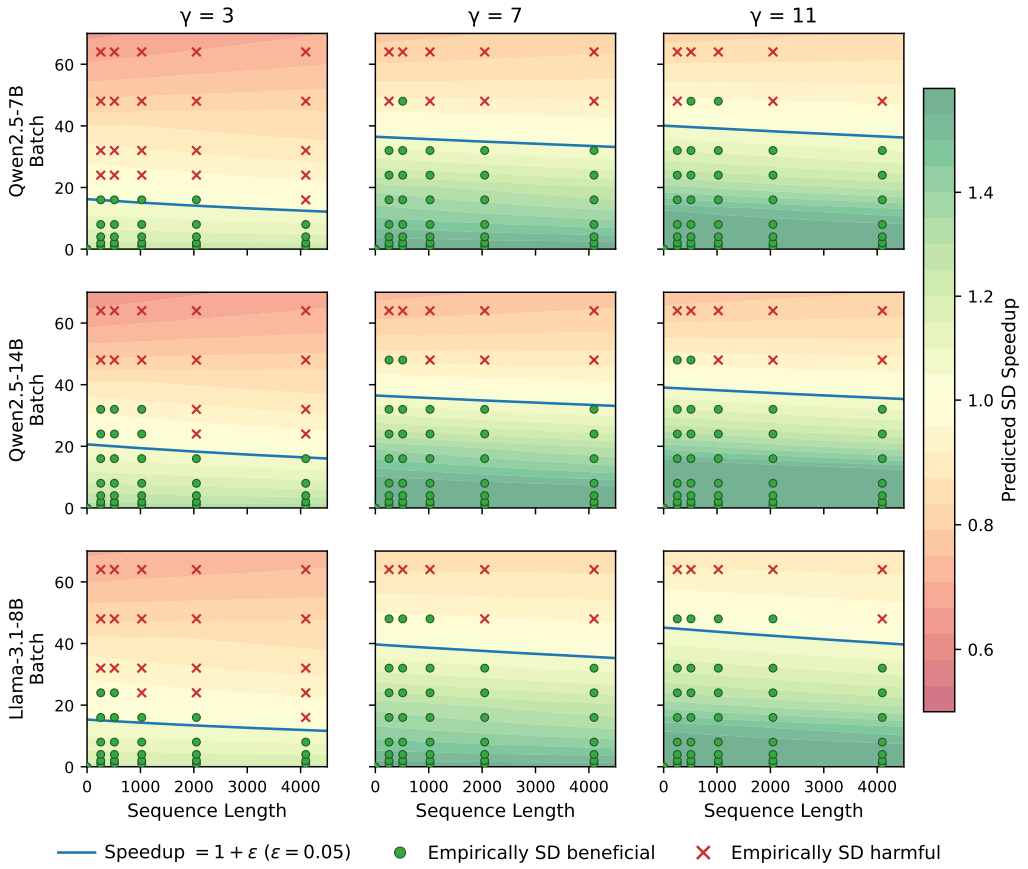


Figure 11. Validation of the roofline-based SD toggle policy. Background colors show predicted speedup, the blue line marks $\mathrm{Speedup}_{\mathrm{SD}} = 1 + \epsilon$, and markers indicate empirical SD-beneficial/harmful points.

model weight size substantially, this does not translate directly into proportional memory-time reduction, because quantized decoding still incurs additional costs such as on-the-fly dequantization and kernel-level overhead. Thus, η_D models the aggregate overhead beyond the idealized weight-byte reduction.

Finally, we model the residual overhead terms as $c_T B$, $c_D B$, and $c_V B$ for target decoding, drafter decoding, and verification, respectively. Empirically, we found that these residual overheads are dominated by batch-dependent effects, making a linear-in-batch approximation sufficiently accurate.

Our current formulation focuses on the TP$= 1$ setting. In principle, it can be generalized to TP> 1 by augmenting each latency term with an explicit communication-overhead component, e.g., an all-reduce cost c_{comm}. We leave this extension to future work.

Calibration procedure. We calibrate the model in three stages.

First, we measure the effective compute throughput $\mathrm{F}_{\mathrm{eff}}$ using a hardware-specific microbenchmark based on repeated dense matrix multiplications. Second, for each model of interest, we sweep full-precision target decoding, verification, and W4 quantized drafter decoding over a range of batch sizes and sequence lengths, with draft lengths $\gamma \in \{3, 7, 11, 15\}$. Third, we fit the remaining parameters in two steps. We first fit $\mathrm{BW}_{\mathrm{eff}}$, ρ_κ, and c_T jointly using the target-model sweep data, and then fix them. We next fit η_D, c_D, and c_V using the drafter and verification sweep data. Because self-SD uses the same architecture for the target and drafter, the compute term $C(B, S)$ is shared between target decoding and drafter decoding. All fitted parameters are independent of γ; the dependence on draft length enters only through the verification compute term and the speedup model in Eq. (1). This allows a model calibrated with a small set of draft lengths to generalize across other draft lengths without re-calibration.

In principle, $\mathrm{BW}_{\mathrm{eff}}$ should depend only on the hardware platform. However, unlike $\mathrm{F}_{\mathrm{eff}}$, it is difficult to benchmark memory bandwidth in a directly comparable standalone form for our decoding workloads. We therefore fit $\mathrm{BW}_{\mathrm{eff}}$

separately for each model. In practice, the fitted values are consistent across models, typically falling in the range of 1560–1620 GB/s.

E.4. Validation of the Roofline-Based SD Toggle Boundary

We validate the calibrated roofline model used by the SD toggle policy against measured W4 self-SD component timings. For each model, we sweep active batch size B, sequence length S, and draft length γ, and measure the target decode time $T_p(B, S)$, quantized-drafter decode time $T_q(B, S)$, and target verification time $T_V(B, S, \gamma)$ on a single A100 GPU with TP$= 1$. Using these measured components, we compute the empirical SD speedup using the same notation as Eq. (1):

$$\text{Speedup}_{\text{emp}}(B, S, \gamma) = \frac{\tau T_p(B, S)}{\gamma T_q(B, S) + T_V(B, S, \gamma)}. \quad (4)$$

For this validation, we use the optimistic setting $\tau = \gamma$, corresponding to full draft acceptance, to isolate the cost-model accuracy from block-efficiency variation.

Figure 11 compares the policy-predicted speedup surface with empirical measurements. The background color shows the speedup predicted by the calibrated roofline model, and the blue contour indicates the toggle boundary at $\text{Speedup}_{\text{SD}} = 1 + \epsilon$ with $\epsilon = 0.05$. Green circles denote measured points where SD is empirically beneficial, while red crosses denote points where SD is harmful. Across models and draft lengths, the predicted boundary closely tracks the empirical beneficial/harmful frontier. The policy is slightly conservative because of the safety margin ϵ: several empirically beneficial points lie just above the blue boundary, but the policy intentionally keeps SD disabled unless it predicts a sufficient margin.

This boundary also motivates our monotone toggle policy. During RL rollout, the active batch size monotonically decreases as requests finish, while the surviving sequences become longer. Thus, the decoding state follows a structured trajectory in the (S, B) plane, moving from a dense-batch regime toward a shrinking-batch tail. Since the batch-size effect dominates the boundary movement, this trajectory naturally crosses the SD-beneficial region once in the regimes we target. We therefore activate SD only after the predicted speedup exceeds $1 + \epsilon$, and keep SD enabled until the end.

F. Detailed Experimental Setup

F.1. Infrastructure, Datasets, and Reporting Window

We use veRL (Sheng et al., 2025) v0.7.0 with vLLM (Kwon et al., 2023) v0.11.2. The training pipeline uses Ray for distributed execution, FSDP for actor training, and vLLM as the rollout backend. All experiments are conducted on a single node with 8 NVIDIA A100-SXM4-80GB GPUs. We use the SimpleRL math datasets (Zeng et al., 2025b). SimpleRL-8k-hard and SimpleRL-8k-medium correspond to MATH (Hendrycks et al., 2021) Level 3–5 and Level 1–4 problems, respectively. Following the SimpleRL recipe, we use SimpleRL-8k-hard for the Qwen models and easier one for Llama3.1-8B-Instruct, reflecting the stronger reasoning capability of the Qwen base models in this setting (Zeng et al., 2025b). For timing summaries, we exclude the first training step because it includes one-time distributed setup and CUDA-graph initialization overheads that do not repeat in steady-state training. All reported averages are computed over the following 100 training steps unless otherwise specified.

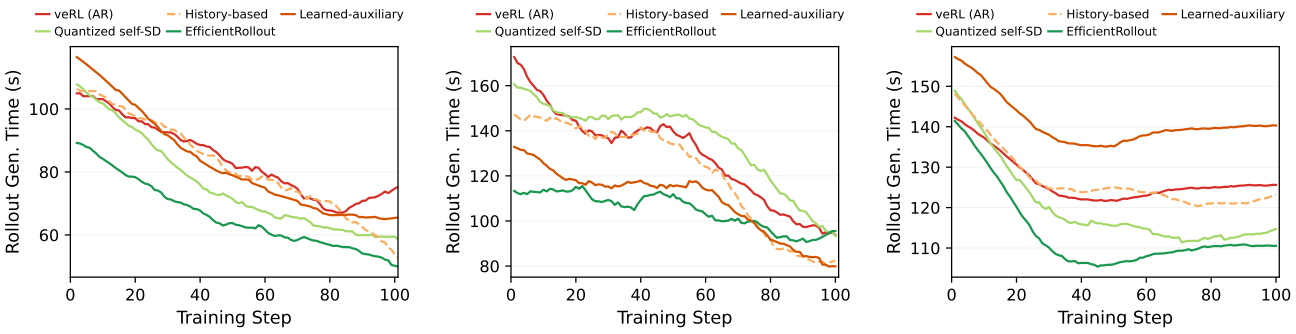
F.2. Metric Measurement Details

For all methods, preparation time, rollout-generation time, and step time are measured throughout training and averaged over training steps. For EfficientRollout, the draft length can change only at step boundaries; we therefore report τ and $\bar{\gamma}$ averaged over training steps, so $\bar{\gamma}$ need not be an integer.

Preparation time includes method-specific overhead required to enable SD. For history-based drafting, it includes rollout-history caching and loading overhead; prefix-matching costs incurred during generation are included in Rollout Gen. For Learned auxiliary drafting, it includes hidden-state extraction, which incurs negligible overhead in our implementation, and online drafter-training overhead, including drafter forward/backward passes and optimizer updates. For Quantized self-SD and EfficientRollout, it includes per-step drafter quantization overhead.

F.3. Training and Method Configuration

We use a train batch size of 128 and generate 8 rollouts per prompt, with a maximum rollout length of 8,192 tokens. Training is performed with a learning rate of 5×10^{-7} and a mini-batch size of 128. Because each mini-batch contains samples from the current policy only, training is purely on-policy and does not require policy-ratio clipping. The default sampling temperature is set to 1.0 with data parallelism (TP=1). For the Qwen base models, we use a KL loss coefficient of 10^{-4} following SimpleRL (Zeng et al., 2025b), and for the Llama instruct model, we use 10^{-2} following FastGRPO (Zhang et al., 2026). For training stability, we further apply token-level rejection sampling [0.5, 2.0] and frequency penalty 0.05 for Qwen2.5-14B. For adaptive draft-length policy, we omit $\gamma = 3$ because it is consistently dominated by, or comparable to, $\gamma = 5$ in the rollout regimes encountered during training.



(a) Qwen2.5-7B *(b)* Qwen2.5-14B *(c)* Llama3.1-8B

Figure 12. Rollout-generation time over training steps (smoothed). EfficientRollout consistently reduces rollout time, whereas other SD baselines provide limited gains or can be slower than veRL (AR).

F.4. Rollout-History-Based Drafting Baseline

We use Spec-RL (Liu et al., 2025a) as a representative rollout-history-based drafting baseline. To our knowledge, it is the only publicly available baseline in this category implemented in a veRL+vLLM training stack. Spec-RL collects generations from previous epochs and, starting from the second epoch, reuses matching prefixes from this rollout history as draft tokens for the current policy.

This design is not strictly target-distribution preserving. First, after a token is rejected, Spec-RL does not perform rejection sampling from the corrected target distribution. Exact SD requires resampling from an adjusted distribution at the first rejected position (Leviathan et al., 2023), which in turn requires access to the full probability distribution at that position. In contrast, rollout-history-based drafting makes exact rejection sampling difficult because it stores only past tokens and their corresponding token probabilities, rather than full logits over the vocabulary for every generated position. Storing full logits for all positions across an epoch would introduce prohibitive memory and storage overhead.

Second, Spec-RL uses a lenience factor of $e^{0.5}$ in its best-performing configuration to increase the length of reused prefixes. This heuristic makes token acceptance more permissive, but further departs from exact target-distribution-preserving speculative decoding. We therefore treat Spec-RL as a lossy rollout-history-based baseline.

F.5. Learned Auxiliary Drafting Baseline

Existing RL rollout acceleration systems with learned auxiliary drafting use different execution stacks. FastGRPO (Zhang et al., 2026) uses a HuggingFace Transformers-based rollout pipeline rather than a serving backend such as vLLM or SGLang. FastRL (Hu et al., 2026) uses SGLang (Zheng et al., 2024) for rollout generation, while NeMo RL (Iso et al., 2026) uses vLLM with NVIDIA's own post-training framework. A direct system-level comparison would therefore mix drafter design with backend and framework differences, including batching, KV-cache management, SD execution paths, and training-loop implementation.

To isolate the drafter design, we implement an EAGLE3-style learned auxiliary drafting baseline in the same veRL+vLLM stack used by the other methods in our paper. A practical requirement of this baseline is that a compatible drafter must be available for each target model family before RL post-training begins. For Qwen2.5-7B and Qwen2.5-14B, we did not find publicly available compatible EAGLE3 drafters, so we pretrain the drafters on ShareGPT (Aeala, 2023) using the official vLLM SD training pipeline following EAGLE3 (Li et al., 2026). Each drafter is trained for 10 epochs, and we select the checkpoint with the best validation performance. For Llama3.1-8B-Instruct, we use the publicly available RedHatAI pretrained EAGLE3 drafter (RedHatAI, 2025a).

For online adaptation during RL, we carefully follow the rollout-SD design choices of FastRL (Hu et al., 2026) and NeMo RL (Iso et al., 2026). Specifically, we capture hidden states during the actor forward pass in the log-probability computation phase and train the drafter inline immediately after each micro-batch of the actor forward. This avoids cross-step buffering and CPU offload while allowing the drafter to consume the freshly captured target hidden states. We follow FastRL's packed layout, next-position shift convention, and combined loss with a hidden-regression term and a soft cross-entropy probability-matching term at the 1:0.1 weight ratio; the drafter is trained with a small fixed AdamW learning rate (1e-6). As a loss-design check, removing the hidden-regression term while keeping only the soft cross-entropy term (matching the NeMo RL formulation) produced similar online block-efficiency trajectories, so we use the FastRL-style combined loss for the primary baseline. We use a separate AdamW optimizer for the drafter and

report the direct drafter-update wall time as method-specific preparation time.

We do not implement FastRL's asynchronous background drafter training, CPU offload, or overlap optimizations. These techniques introduce additional system-level design choices beyond the drafter itself and would make the comparison depend on overlap efficiency rather than only rollout-time SD behavior. For decoding, we use chain drafting with a fixed draft length $\gamma = 3$, following the best-performing setting reported by Iso et al. (2026).

G. Additional Evaluation Results

G.1. Per-Step Rollout Generation Time

Figure 12 shows rollout-generation time over training steps across models and SD baselines for RL rollouts. EfficientRollout achieves the lowest generation time throughout training on Qwen2.5-7B and Llama3.1-8B, and for most Qwen2.5-14B steps. The small late-stage slowdown on Qwen2.5-14B coincides with longer sampled responses under EfficientRollout, whose maximum response length reaches about 4,108 tokens, compared with 3,383 for learned auxiliary drafting and 2,916 for history-based drafting. We view this fluctuation as response-length stochasticity rather than a systematic toggle-policy failure.

Quantized self-SD can improve over veRL (AR) on Qwen2.5-7B and Llama3.1-8B, but can become slower when SD is enabled throughout rollout, especially on Qwen2.5-14B where the larger quantized drafter increases drafting cost. Learned auxiliary drafting achieves reasonable rollout speedup on Qwen2.5-14B because its auxiliary drafter combines low proposal cost with moderate block efficiency, as shown in Tab. 6. These per-step trends are consistent with the aggregate timing results in Tab. 2.

G.2. Training Dynamics and Quality Preservation

Figures 13 and 15 report training reward and validation accuracy trajectories over RL training. Across Qwen2.5-7B, Qwen2.5-14B, and Llama3.1-8B-Instruct, EfficientRollout closely follows the veRL (AR) baseline in both metrics. These results support that the rollout speedups do not materially alter training dynamics or degrade downstream model quality in our evaluated settings. This is consistent with the lossless nature of SD (Leviathan et al., 2023; Chen et al., 2023), which preserves the target-model distribution through rejection sampling and is especially important in RL, where rollout-distribution shifts can affect training stability (Yu et al., 2026; Fu et al., 2026).

Table 5. Active-window analysis of history-based drafting after rollout history becomes available (from the second epoch onward). Response and reused-prefix lengths are in tokens; verification overhead is per step; ΔGen. is the generation-time change relative to No-SD.

Model	Resp. len.	Prefix	Reuse %	Verify (s)	ΔGen. (s)
Qwen2.5-7B	661.4	28.9	4.4	10.7	+12.8
Qwen2.5-14B	622.9	42.3	6.8	19.5	+5.8
Llama3.1-8B	646.2	324.6	50.2	15.9	+11.1

G.3. Adaptive Draft-Length Schedules Across Models

Figure 14 shows the γ schedule selected by the adaptive controller across models. Qwen2.5-7B increases from $5 \rightarrow 7 \rightarrow 9 \rightarrow 11$ as its block efficiency repeatedly approaches the effective ceiling of the current draft length. Before each elevation, its block efficiency reaches 5.73, 7.60, and 9.48, and after increasing γ the block efficiency further rises to 7.37, 9.35, and 11.25, respectively. Qwen2.5-14B follows the same pattern but later in training, increasing from $5 \rightarrow 7 \rightarrow 9$ when its block efficiency reaches 5.73 and 7.61. In contrast, Llama3.1-8B-Instruct stays at $\gamma = 5$: although its block efficiency improves from 5.07 to around 5.5, it never crosses the elevation threshold. No downward adjustment is triggered in these runs.

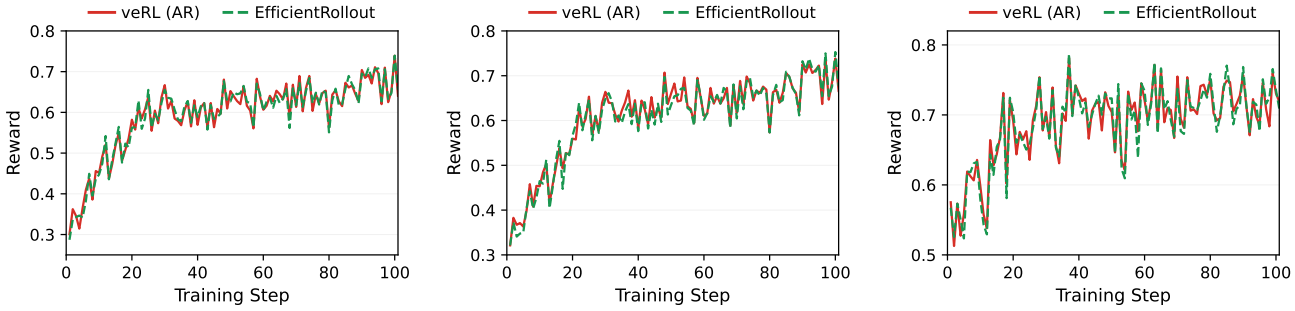
G.4. Why History-Based Drafting Does Not Accelerate

Table 5 analyzes the rollout-history-based baseline after rollout history becomes available, i.e., from the second epoch onward. This isolates the active window where history-based drafting can actually be applied. Even in this favorable window, the method does not reduce rollout generation time.

The main reason is that history reuse does not provide enough accepted tokens to amortize verification cost. For Qwen2.5-7B, the Spec-RL implementation reuses only 30 tokens on average out of a 660-token response, corresponding to a 4.5% reuse rate. For Llama3.1-8B-Instruct, the reused prefix is longer, but verification is also more expensive, adding 15.9 s of overhead per active step. As a result, even after excluding the first epoch and using the original lossy lenience factor to increase acceptance, rollout-history-based drafting increases rollout generation time by 19.4% on Qwen2.5-7B and 8.8% on Llama3.1-8B-Instruct. These results show that the history-based approach is limited not only by its cold start, but also by low effective reuse relative to verification cost.

G.5. Why Learned Auxiliary Drafting Remains Challenging

Block efficiency of learned auxiliary drafting. EAGLE3-style auxiliary drafters typically require pretraining and on-



(a) Qwen2.5-7B *(b)* Qwen2.5-14B *(c)* Llama3.1-8B

Figure 13. Average training reward over RL steps for three evaluated models. Across all evaluated models, EfficientRollout closely follows the No-SD reward trajectory, suggesting that rollout acceleration preserves the main training dynamics.

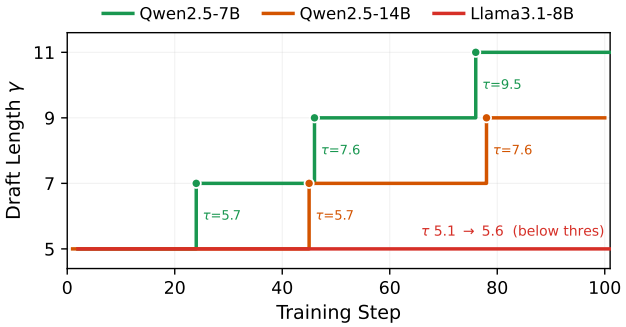


Figure 14. Adaptive γ schedules across models. The controller increases γ only when block efficiency τ is high enough to support a longer draft; otherwise, it keeps γ unchanged.

Table 6. Timing decomposition for learned auxiliary drafting. T_p is the single-token target forward time, T_q is the one-token drafter forward time, and T_V is the target verification time for $\gamma + 1$ tokens.

Model	T_p (ms)	T_q (ms)	T_V (ms)	$\mathbf{T_q/T_p}$	$\mathbf{T_V/T_p}$
Qwen2.5-7B	10.45	0.82	14.27	0.078	1.37
Llama3.1-8B	13.66	0.85	19.75	0.062	1.45
Qwen2.5-14B	20.31	0.91	28.50	0.045	1.40

line adaptation to match the target model and rollout data distribution (Zhang et al., 2026; Hu et al., 2026; Iso et al., 2026). In our setting, a randomly initialized drafter yields block efficiency close to 1.0, meaning that it rarely predicts even the first token correctly. As shown in Fig. 16, starting from a generally pretrained drafter improves early-stage behavior, but the initial block efficiency remains limited: with a ShareGPT-pretrained drafter evaluated on SimpleRL-math rollouts, block efficiency starts around 1.4–2.0. A drafter pretrained directly on the target RL post-training distribution would likely provide higher initial block efficiency (Iso et al., 2026); however, such in-distribution drafter pretraining is not always available in practice, so we use a broadly pretrained drafter as a more general setting.

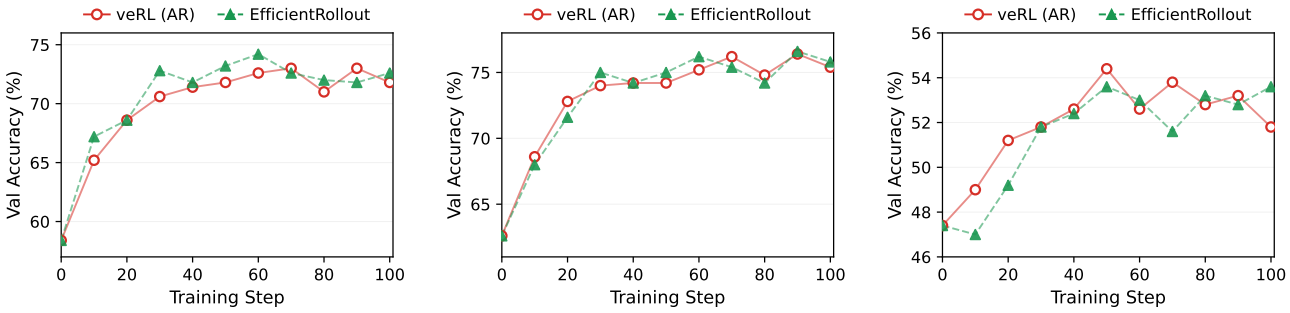
Table 7. Depth-dependent block efficiency for EAGLE3-based learned auxiliary drafting before online adaptation. Llama3.1-8B-Instruct already drops close to one after 512 generated tokens, while the Qwen models degrade more gradually.

Model	0–512	512–1k	1k–2k	2k–4k	4k–8k
Qwen2.5-7B	1.939	1.895	1.723	1.526	1.195
Llama3.1-8B	**2.252**	**1.418**	**1.100**	**1.037**	**1.056**
Qwen2.5-14B	2.045	2.037	2.029	2.237	1.866

Table 8. Depth-dependent block efficiency for EAGLE3-based learned auxiliary drafting over the first five RL steps. Llama3.1-8B-Instruct exhibits a sharp drop after 512 generated tokens, whereas the Qwen models maintain or recover block efficiency at deeper positions.

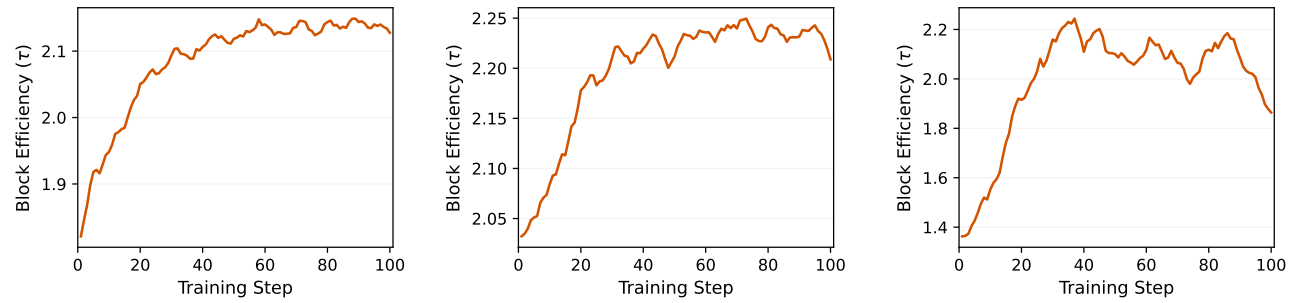
Model	0–512	512–1k	1k–2k	2k–4k	4k–8k
Qwen2.5-7B	1.970	1.921	1.790	1.716	1.604
Llama3.1-8B	**2.288**	**1.557**	**1.252**	**1.190**	**1.294**
Qwen2.5-14B	2.070	2.061	2.103	2.390	2.573

Limited acceleration in Qwen-7B/14B. For the Qwen2.5-series models, learned auxiliary drafting provides limited generation-time acceleration mainly because its block efficiency remains insufficient for our RL rollout workload. Under high-temperature sampling ($T = 1.0$) and long reasoning generation, the pretrained drafters do not provide sufficient proposal quality for our math RL rollout distribution. In addition, system-aware activation remains important: as indicated by the gap between quantized self-SD and EfficientRollout, enabling SD during early large-batch phases can reduce or eliminate its benefit. Higher block efficiency may be achievable with in-distribution drafter initialization or more aggressive online adaptation, such as training the auxiliary drafter for multiple steps per RL iteration or running adaptation asynchronously. However, these choices introduce additional drafter-training, scheduling, and configuration



(a) Qwen2.5-7B *(b)* Qwen2.5-14B *(c)* Llama3.1-8B

Figure 15. Validation accuracy over training steps for the three evaluated models. EfficientRollout achieves validation trajectories comparable to the veRL (AR) baseline, supporting that the acceleration does not degrade downstream model quality in our evaluated settings.



(a) Qwen2.5-7B *(b)* Qwen2.5-14B *(c)* Llama3.1-8B

Figure 16. Block efficiency of learned auxiliary drafting over RL training steps. Across models, the pretrained drafter starts with low block efficiency but improves as online adaptation proceeds.

search burden; we further analyze the workload-matching requirement in Sec. G.6.

Unexpected slowdown on Llama3.1-8B. Li et al. (2026) report strong inference speedups for EAGLE3 in standard serving settings across Qwen and Llama models. However, these results are not directly comparable to our RL rollout setting, which combines vLLM execution, high-temperature sampling ($T = 1.0$), hard reasoning prompts, and long response budgets. To diagnose the Llama3.1-8B-Instruct slowdown, we run short cold-start profiling experiments for the first five RL steps under the same rollout setting as in Sec. 4.1, using vLLM EAGLE3 timing (`VLLM_SD_TIMING=1`), position-binned block efficiency logging, and a custom proposal timer.

Table 6 shows that the slowdown is not primarily due to drafter cost: the one-token drafter time T_q is similar across models, around 0.8–0.9 ms. Llama3.1-8B-Instruct has the largest verification-to-target ratio ($T_V/T_p = 1.45$), but this is only modestly higher than the Qwen models. Thus, timing overhead alone does not explain the much larger slowdown.

The larger difference appears in depth-dependent block efficiency. As shown in Tab. 7, before online adaptation, Llama3.1-8B-Instruct already shows block efficiency close to one after 512 generated tokens, indicating weak long-depth proposal quality from the pretrained drafter. After the first five online-adaptation steps, this weakness persists: Llama3.1-8B-Instruct remains near 1.2–1.3 in the 1k–8k bins, whereas the Qwen models quickly recover block efficiency relative to their pre-adaptation values, especially at deeper positions (Tabs. 7 and 8). This is precisely the regime that matters for rollout latency. Long-tail responses dominate the rollout makespan, and the Llama drafter pays draft and verification overhead while most proposals are rejected, making the long tail even more expensive. This suggests a model- and workload-dependent failure mode of learned auxiliary drafting rather than a stable speedup across RL rollouts.

G.6. Auxiliary Drafters Aligned with RL Rollout Distributions Are Difficult to Obtain

NeMo-RL-native validation setup. To check whether the moderate block efficiency of learned auxiliary drafting stems from our veRL (Sheng et al., 2025)-based implementation or from the drafter itself, we additionally evaluate EAGLE3 drafters in NVIDIA's native NeMo RL SD

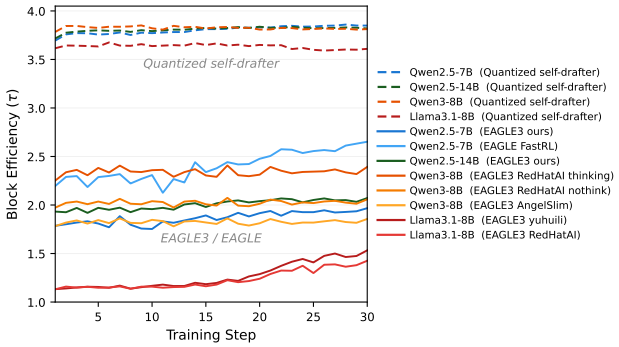


Figure 17. Block efficiency on DAPO-Math-17K. Across the first 30 RL steps, evaluated learned auxiliary drafters remain largely below target-induced quantized drafters.

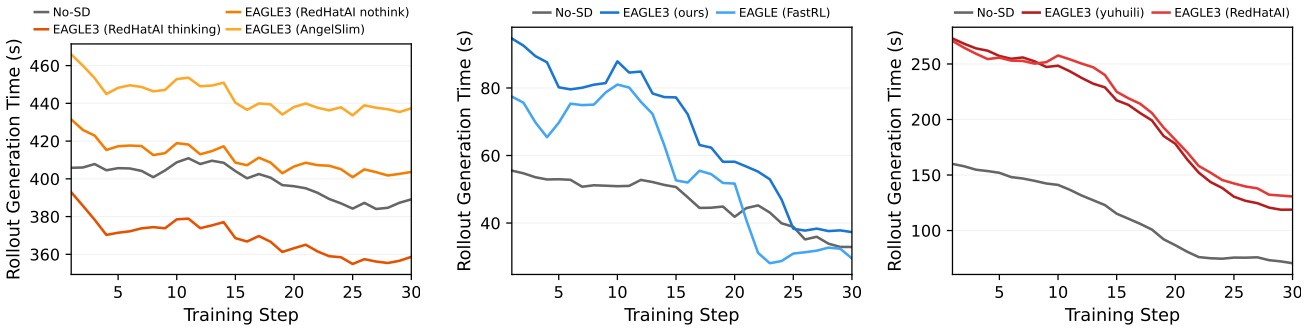
stack (Iso et al., 2026). This stack uses Megatron (Shoeybi et al., 2019)-based online drafter training with vLLM rollout serving, and we leave the core RL, SD, and reward paths unchanged. We train on the DAPO-Math-17K dataset (Yu et al., 2026) using one node with 8 H100-SXM GPUs, $\gamma = 3$ with chain drafting and verification, 30 RL steps, a maximum generation length of 8k, temperature $T = 1.0$, and a global batch size of 1k rollouts. For comparison, we also measure the block efficiency of quantized self-drafters under the same DAPO-Math experimental setting in our self-SD stack.

Evaluated learned auxiliary drafters. Alongside our three model families (Qwen2.5-7B/14B and Llama3.1-8B), we additionally include Qwen3-8B following Iso et al. (2026). For Qwen3-8B, we evaluate three public EAGLE3 drafters: RedHatAI's thinking drafter (RedHatAI, 2026), RedHatAI's non-thinking drafter (RedHatAI, 2025b), and AngelSlim's drafter (AngelSlim, 2025). For Qwen2.5-7B and Qwen2.5-14B, we did not find compatible public EAGLE3 checkpoints, so we use our ShareGPT-pretrained drafters introduced in Sec. F.5. For Llama3.1-8B-Instruct, we evaluate both the RedHatAI public drafter (RedHatAI, 2025a) and the official checkpoint from the EAGLE3 authors released by yuhuili (Yuhui Li, 2024). We also evaluate the public Qwen2.5-7B EAGLE-style drafter (MIT HAN Lab, 2025) released with FastRL (Hu et al., 2026) as an RL-targeted auxiliary-drafter checkpoint. The checkpoint is reported to be trained on the OpenThoughts2-1M dataset (open-thoughts, 2025), a synthetic reasoning dataset spanning math, science, code, and puzzles. Because this checkpoint is not directly compatible with NeMo RL's EAGLE3-specific online update path, we use it as a fixed drafter over the 30 RL steps without online adaptation.

Block efficiency gap. Figure 17 shows that evaluated learned auxiliary drafters achieve substantially lower block efficiency than target-induced quantized self-drafters. Across the first 30 RL steps, quantized self-drafters stay near $\tau = 3.6$–3.9 under $\gamma = 3$, close to the ceiling of 4. By contrast, EAGLE3 drafters remain mostly in the $\tau = 1.2$–2.4 range, with the best public configuration, Qwen3-8B with the RedHatAI thinking drafter, reaching roughly $\tau = 2.2$–2.4. The FastRL Qwen2.5-7B EAGLE-style drafter reaches higher block efficiency ($\tau = 2.2$–2.6) than our ShareGPT-pretrained EAGLE3 drafter, but this is still insufficient for consistent rollout-generation speedup. Within our 30-step window, online adaptation yields only modest block efficiency improvement from the evaluated initializations. These results suggest that the limitation is not the EAGLE/EAGLE3 architecture itself, but the availability of an auxiliary drafter sufficiently aligned with long, high-temperature RL rollout distributions and effective from the beginning of RL training.

Generation-time consequence. Figure 18 shows the practical consequence of low block efficiency within the NeMo RL stack. All timing curves compare learned auxiliary drafting against the corresponding NeMo RL No-SD baseline. Among the evaluated EAGLE3 drafters, Qwen3-8B with the RedHatAI thinking drafter is the only configuration that consistently reduces rollout-generation time, improving it by about 7.7%. The Qwen2.5-7B EAGLE-style FastRL drafter becomes consistently beneficial only late in training, reducing rollout-generation time by 17.6% over steps 22–30. The other configurations are slower than No-SD for much of training because their block efficiency is too low to amortize draft, verification, and online-update overhead. This confirms that learned auxiliary drafting can reduce rollout-generation time when the drafter reaches sufficiently high block efficiency.

Output length and training distribution. One possible explanation is an output-sequence-length mismatch between drafter training and rollout inference. However, when we run ablation probes with shorter output caps (e.g., 1k, 2k, and 4k), most public or broadly pretrained drafters remain weak even under shorter caps, and several configurations are roughly length-flat. This suggests that output length alone does not explain their low block efficiency. A more central factor appears to be the drafter-training distribution: public drafters and our Qwen2.5 drafters are trained on general chat/text corpora such as ShareGPT or UltraChat, rather than target-generated rollouts from the RL workload. Hu et al. (2026) train their drafter on OpenThoughts2-1M (open-thoughts, 2025), a synthetic reasoning dataset closer to long reasoning workloads than general chat corpora. More generally, reaching the high-block efficiency regime reported by Iso et al. (2026) may require a more specialized auxiliary-drafter training pipeline, such as collecting target-generated long rollouts from the RL workload and then training the auxiliary drafter on the resulting data.



(a) Qwen3-8B *(b)* Qwen2.5-7B *(c)* Llama3.1-8B

Figure 18. Rollout-generation time of EAGLE3 in the NeMo RL stack. Among the evaluated EAGLE3 drafters, only Qwen3-8B with the RedHatAI thinking drafter reduces rollout-generation time, while the other configurations are slower than No-SD due to insufficient block efficiency.

Takeaway. Our results show that reaching this regime depends on a well-aligned auxiliary drafter, which is not generally available from public checkpoints or generic drafter-training recipes. Obtaining such a drafter can require a separate pipeline for target-generated long-rollout collection, offline auxiliary-drafter training, and possibly more aggressive online adaptation. This raises the barrier to using learned auxiliary drafting as a drop-in rollout accelerator. Target-induced self-drafting provides a simpler point in this trade-off: it requires no external drafter checkpoint or offline auxiliary-drafter pretraining, yet achieves high block efficiency from the beginning of RL training and translates it into practical rollout speedup.

H. Future Directions

Several directions remain for extending EfficientRollout. First, we focus on chain verification because it is practical under dynamic batch sizes (Liu et al., 2026); tree verification is orthogonal and could be incorporated into our framework (Li et al., 2024; 2026). Second, naive RTN may not yield strong early-stage drafters for all model families, motivating less lossy quantization methods that improve drafter quality without introducing excessive per-step overhead. Finally, when KV-cache loading dominates in long-context or dense-attention regimes (Cordonnier et al., 2020), sparse-attention drafting may further reduce quantized self-SD cost.