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

Minseo Kim[1,2*‡] **Minjae Lee**[1*] **Seunghyuk Oh**[1] **Kevin Galim**[1] **Donghoon Kim**[1]
Coleman Hooper[2] **Harman Singh**[2] **Amir Gholami**[2] **Hyung Il Koo**[1,3†] **Wonjun Kang**[1†]
[1]FuriosaAI [2]University of California, Berkeley [3]Ajou University
minseo.kim@berkeley.edu {minjae.lee,kangwj1995}@furiosa.ai

Abstract

Reinforcement learning (RL) has become a representative post-training paradigm for large language models (LLMs), enabling strong reasoning and agentic capabilities. However, rollout generation remains a dominant latency bottleneck because autoregressive (AR) sampling decodes responses sequentially and a small number of long-tailed generations often determine completion time. Speculative decoding (SD) offers a natural way to address this bottleneck, as it is a well-established technique for serving fixed LLMs that reduces latency by rapidly drafting tokens and accepting them through parallel verification while preserving the target-model distribution. However, its practical speedups do not directly carry over to RL rollouts: (i) the evolving target policy makes any fixed drafter increasingly mismatched with the policy's output distribution; and (ii) active batch sizes shrink throughout rollout decoding, shifting decoding from compute-bound to memory-bound regimes where parallel verification can exploit underutilized compute. Therefore, accelerating RL rollouts requires both a drafter that remains effective under long, high-temperature generations from an evolving policy and system-aware use of SD that avoids compute-bound regimes. We present **EfficientRollout**, a system-aware self-SD framework designed to address this gap for RL rollouts. EfficientRollout induces a quantized drafter 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 policy with acceptance-aware draft-length adaptation, enabling speculation only in beneficial regimes while matching the drafting budget to evolving drafter quality. EfficientRollout reduces rollout and end-to-end latency by up to 24.2% and 15.6%, respectively, over an accelerated AR rollout baseline, while preserving final model quality. [1]

1 Introduction

Reinforcement learning (RL) has become an essential method for post-training frontier large language models (LLMs), enhancing their reasoning, math, and agentic capabilities [18, 55, 61, 68]. 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 [15, 27]. However, rollout generation has emerged as the dominant latency bottleneck in the RL pipeline because post-trained models increasingly generate long responses [9, 18, 37, 64] that must be decoded token by token. Consequently, reducing rollout latency is critical for accelerating RL post-training, since it often dominates other parallelizable stages such as policy updates [46].

*Equal contribution. †Corresponding authors.
‡Work done during an internship at FuriosaAI.
[1]Code is available at https://github.com/furiosa-ai/EfficientRollout.

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

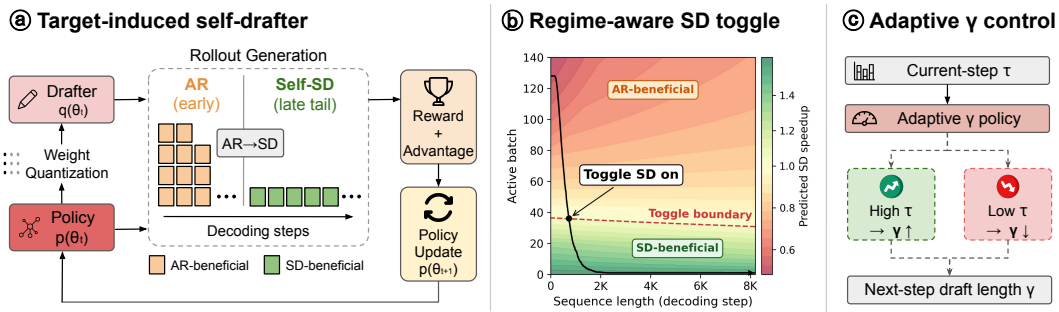


Figure 1: **Overview of EfficientRollout**. EfficientRollout bridges the gap between SD for fixed-model serving and RL rollout decoding through three coordinated components: (a) per-step self-drafter refresh to track the evolving policy, (b) regime-aware toggling from AR to SD under tail-heavy rollout dynamics (see Fig. 7), and (c) adaptive draft length γ control based on observed block efficiency τ.

Speculative decoding (SD) [6, 29] 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, *i.e.*, the policy being sampled. Its speedup depends on two conditions: algorithmically, the block efficiency τ must be sufficiently high, where τ measures how many target-distributed tokens each draft-verify iteration produces on average [36, 75]; system-wise, decoding should not be pushed into a compute-bound regime, so that parallel target-model verification can exploit underutilized compute [29, 36, 45].

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 [72]. 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 [73], the target policy can be approximated by a lower-capacity subset induced from the target itself, which can still cover the concentrated probability mass. Together, these factors raise a practical question: *how can SD realize speedup in RL rollouts under the distinct conditions introduced by RL?*

Prior SD methods for RL rollouts address this problem, but still face either low block efficiency or the burden of drafter construction and adaptation. 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 cover future trajectories [19, 35, 47]. Learned auxiliary drafting methods seek higher block efficiency by training auxiliary drafters, but existing learned 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 [8, 22, 72].

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 rollout workloads, showing that target-induced drafters naturally stay coupled to the evolving policy and that dense weight projections dominate 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 drafter updates. We further introduce two control policies: a system-aware SD toggle that activates SD only in favorable regimes, avoiding slowdowns during early compute-bound large-batch phases, and an adaptive draft-length policy that matches the drafting budget to evolving acceptance behavior during RL. We show that EfficientRollout reduces rollout and end-to-end latency by up to 24.2% and 15.6%, respectively, over prevalent RL training and serving stacks [28, 50].

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. 3).
- We present EfficientRollout, an RL-specific SD framework with a target-induced quantized drafter, dynamic SD toggling, and adaptive drafting budgets (Sec. 4).

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.

Drafter category	Parameterized	Online-adaptation free	Warm-up free	Examples
(1) History-based	No	Yes	No	[19, 35, 47]
(2) Learned auxiliary	Yes	No	Conditional	[8, 22, 24, 72]
(3) Target-induced	**Yes**	**Yes**	**Yes**	**EfficientRollout (ours)**

- We show that EfficientRollout reduces rollout and end-to-end latency by up to 24.2% and 15.6% over a standard accelerated AR rollout baseline [28, 50], and analyze why existing SD approaches do not consistently resolve the emergent challenges of RL rollouts (Sec. 5).

2 Related Work

2.1 Reinforcement Learning for LLMs

Frontier LLMs post-trained with RL have achieved state-of-the-art reasoning and agentic capabilities [18, 55, 61, 68], enabled by advances in policy optimization [5, 46, 48, 64]. 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 [18, 57]. 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 [15, 27]. However, rollout generation is often the main latency bottleneck because RL tends to induce longer outputs [9, 37], amplifying the cost of sequential, memory-bandwidth-bound AR decoding that can leave compute underutilized [16]. By contrast, other stages of the RL loop, including log-probability computation and policy updates, are more parallelizable.

2.2 Speculative Decoding for LLM Inference

SD accelerates target LLM decoding through a sequential drafting and parallel verification scheme [6, 29]. 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 [45]. SD methods differ mainly in how they construct the drafter [59]. Canonical SD uses an independent smaller drafter model to approximate the target model [6, 29]. To make drafting much cheaper, non-parametric drafting methods generate drafts from text using n-gram, prefix-tree, or suffix-tree matching [20, 40, 53]. Learned auxiliary drafting methods attach lightweight modules to the target model for efficient token prediction, such as FFN heads or one-layer drafters [4, 31]. Self-speculative decoding (self-SD) instead derives the drafter directly from the target model via layer skipping [70], sparse attention [66], or quantization [56].

2.3 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 [19, 35, 47]. 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 adapt auxiliary-drafter approaches from fixed-model SD, often using EAGLE-style drafters [31, 32], and aim to match the drafter to long-reasoning rollout distributions [8, 22, 24, 72]. 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 [72]. Tracking the evolving RL policy also requires online adaptation, adding system complexity and management

overhead [22]. **(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 without separate online adaptation while maintaining high block efficiency. We focus on lossless on-policy RL acceleration and therefore exclude rollout-efficiency techniques that change the rollout distribution or termination process, such as lower-precision rollout inference [30], proxy rollout models [10], and early rollout termination [71, 76].

3 Preliminary

3.1 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 $\mathcal{M}_q$ and target model $\mathcal{M}_p$, respectively. When the drafter sequentially speculates γ tokens, let $T_V(B,S,\gamma)$ denote the parallel verification time by the target model.[2] We define block efficiency τ as the average number of accepted draft tokens plus one target-sampled token per drafting block [75]. Under the i.i.d. acceptance assumption, τ can be written in terms of a per-token acceptance rate α as $\tau = (1-\alpha^{\gamma+1})/(1-\alpha)$ [29, 45]. Given τ, the SD block time and wall-clock speedup are

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

Thus, SD acceleration depends on both high τ and the latency ratios T_q/T_p and T_V/T_p 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 common $B=1$ SD benchmarks [45, 59].

3.2 Roofline Modeling for LLM Decoding Latency

To identify the decoding regime on the actual hardware system, we use roofline modeling, a standard system-aware approach [58, 65]. Roofline modeling distinguishes between arithmetic work C_{FLOPs}, executed with effective compute capacity $\mathrm{F}_{\mathrm{eff}}$, and data movement M_{Bytes}, served by effective memory bandwidth $\mathrm{BW}_{\mathrm{eff}}$. Assuming computation and memory access can ideally overlap, the workload latency is approximated as

$$T_{\mathrm{roofline}} \approx \max\left(\frac{M_{\mathrm{Bytes}}}{\mathrm{BW}_{\mathrm{eff}}}, \frac{C_{\mathrm{FLOPs}}}{\mathrm{F}_{\mathrm{eff}}}\right). \tag{2}$$

For LLM decoding, M_{Bytes} includes weight and KV-cache traffic, while C_{FLOPs} includes dense projections such as QKVO, FFN, and LM-head computations, as well as attention operations [45].

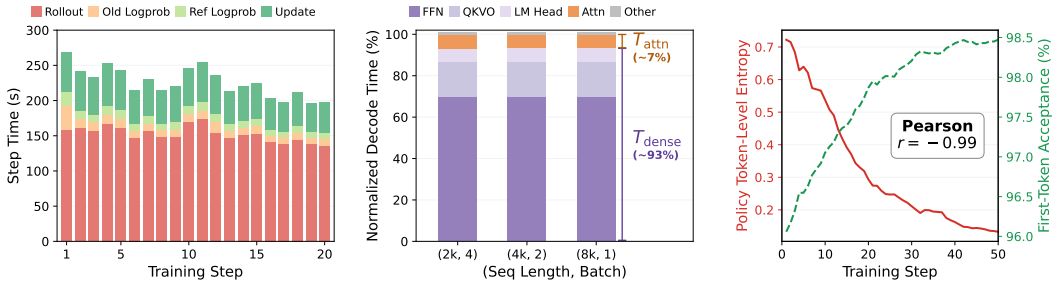
3.3 RL Rollout Latency Bottlenecks from Long-Tail Decoding

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 workloads increasingly produce long reasoning trajectories, and rollout generation must sample these trajectories token by token under AR decoding [16]. By contrast, policy updates and log-probability computation are more parallelizable, allowing them to better utilize compute capacity. 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 A provides additional analysis of the long-tail completion behavior of rollout during RL.

4 EfficientRollout: From RL Rollout Dynamics to System-Aware Self-SD

This section develops **EfficientRollout** from three RL rollout characteristics. First, the evolving target policy and rollout-tail latency profile motivate a weight-quantized drafter for self-SD. Second, shifting

[2] 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 RL rollout-tail phases. (c) Inverse correlation between target-policy entropy and quantized-drafter first-token acceptance over steps.

runtime regimes motivate an SD toggle policy that activates only when beneficial. Third, changing acceptance behavior during RL motivates adaptive draft lengths. We describe each component below; Algorithm 1 summarizes the full EfficientRollout pipeline for Secs. 4.1 to 4.3.

4.1 Weight-Quantized Drafter for Self-Speculative Decoding

Self-speculative decoding. Among the SD designs in Tab. 1, self-SD with a target-induced drafter [56, 66, 70] 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 separate drafter pretraining or online adaptation. This contrasts with learned auxiliary drafters, which can become stale unless continually adapted online to track the evolving policy [72].

Rollout-tail latency decomposition. We decompose rollout-tail single-token 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 observation is consistent with modern LLM architectures that reduce KV-cache sizes [2, 18, 34, 49] and with system analyses showing that parameter loading dominates latency in small-batch memory-bound regimes [45].

Weight-quantized drafting. We induce a weight-quantized drafter [56] from the current target model to reduce the weight-loading cost of T_{dense}, the dominant component of rollout-tail latency (Fig. 2b). This lowers 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 [33]. We use 4-bit weights (W4) as a practical latency–acceptance trade-off: lower precision makes drafting substantially cheaper while preserving sufficient block efficiency τ 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 [52, 59], while adaptive skipping is difficult to combine with static inference optimizations in modern serving engines, such as CUDA graph pre-capture [7, 28, 60]. Sections B.2 and B.3 further analyze self-SD alternatives and W4–W8 latency–acceptance trade-off.

4.2 Regime-Aware SD Toggle Policy via Roofline Modeling

Even with a cheap quantized drafter, SD is not beneficial throughout the entire rollout. Early rollout phases have large active batches that can saturate compute, leaving little underutilized compute for target verification (*i.e.* high T_V/T_p in Eq. (1)). As a result, SD can be slower than AR decoding despite high τ. 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. 3.2) from the current decoding state (B, S) and SD runtime state (τ, γ), activating SD only when the prediction is favorable.

SD toggle policy. To implement this decision, the SD toggle policy uses a calibrated model to predict the SD-over-AR speedup for the current decoding state (B, S). We first define memory-side and

compute-side cost estimates for target, draft, and verification execution:

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

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 of one forward pass. 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

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

The overhead terms capture non-ideal overlap and batch-dependent residual costs beyond ideal roofline approximation. Following the SD speedup model in Eq. (1), we define the SD toggle policy π_{SD} to activate SD when the predicted speedup exceeds a safety margin:

$$\pi_{\text{SD}}(B, S, \tau, \gamma) = \mathbf{1}\left[\frac{\tau \widehat{T_p}}{\gamma \widehat{T_q} + \widehat{T_V}} \geq 1 + \epsilon\right], \qquad \epsilon \geq 0. \tag{3}$$

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. We introduce additional calibration parameters to adapt the roofline model and the corresponding policy to different model and system configurations. The fitted terms κ_{eff}, η_D, and c_T, c_D, c_V capture 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 D.2 provides calibration details.

4.3 Adaptive Draft-Length Policy

Acceptance behavior evolves during RL. The weight-quantized drafter can become more effective over training, increasing τ as RL sharpens the target policy. In Fig. 2c, target-output entropy and token-level agreement between $\mathcal{M}_p$ and its weight-quantized copy $Q(\mathcal{M}_p)$ show a strong negative correlation ($r = -0.99$). As training progresses, RL can reduce target-output entropy [12, 25], sharpening the target policy by concentrating probability mass on a smaller set of high-reward tokens [23, 62]. A sharper distribution can make top-token decisions less sensitive to small quantization-induced perturbations, increasing agreement between $\mathcal{M}_p$ and $Q(\mathcal{M}_p)$. Consistent with this explanation, we observe that the quantized self-drafter's τ improves over training, motivating adaptive draft budgets that increase as acceptance improves. Section C provides full experimental results for policy sharpening and block-efficiency improvement.

Adaptive policy for draft length γ. To exploit the increasing τ observed during RL training, we choose the next draft length γ_{t+1} using the measured block efficiency τ_t as a proxy for current drafter quality. We maintain an ordered draft-length set $\Gamma = [\gamma_{\text{low}}, \ldots, \gamma_{\text{high}}]$ and update from γ_t to γ_{t+1} only when the condition on τ_t persists for P consecutive training steps. Specifically, we increase the draft length when τ_t 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 fluctuations in τ_t across steps from changing the draft length. This removes the need to select a single fixed γ through post-hoc sweeps, instead letting the adaptive policy move within a pre-specified draft-length set using observed τ_t feedback during RL training.

5 Experiments

5.1 Setup

Training details. We evaluate representative open-source models across model families and scales: Qwen2.5-{7B,14B} [63] on SimpleRL-8k-hard, and Llama3.1-8B [17] on SimpleRL-8k-medium,

Algorithm 1 EfficientRollout Pipeline

Parameter: 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 $\gamma_1 \leftarrow \gamma_{\text{low}}$ and $\tau_0 \leftarrow \gamma_{\text{low}}$
for training timestep $t = 1, 2, \ldots$ **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 $\pi_{\text{SD}}(B, S, \tau_{t-1}, \gamma_t) = 0$ **then**
            Decode autoregressively
        else if `sd_on` $=$ **false** and $\pi_{\text{SD}}(B, S, \tau_{t-1}, \gamma_t) = 1$ **then**
            `sd_on` $\leftarrow$ **true**
        else
            Decode with SD using draft length γ_t
    Measure step-level block efficiency τ_t
    $$\gamma_{t+1} \leftarrow \begin{cases} \text{next}_\Gamma(\gamma_t), & \min\limits_{t' \in [t-P+1,t]} \tau_{t'} \geq 1 + \gamma_t \alpha_{\text{up}},\ \gamma_t < \gamma_{\text{high}}, \\ \text{prev}_\Gamma(\gamma_t), & \max\limits_{t' \in [t-P+1,t]} \tau_{t'} \leq 1 + \gamma_t \alpha_{\text{down}},\ \gamma_t > \gamma_{\text{low}}, \\ \gamma_t, & \text{otherwise.} \end{cases}$$
    Run standard post-rollout optimization

following the math-domain RLVR recipe [69]. We evaluate acceleration of RL with GRPO [48], using train batch size 128, group size 8, rollout temperature 1.0, and maximum response length 8k for 100 training steps. Unless otherwise noted, experiments are conducted on a single node with 8 A100-80GB GPUs using data parallelism (DP) [47]. Additional evaluations on H100, Qwen2.5-32B with TP$=2$ on A100, and broader workload settings are provided in Sec. F.8.

System configuration. To ensure both deployability and measured speedup, we implement the weight-quantized drafter on top of veRL [50] and vLLM [28]. Our integration uses the Marlin weight-only quantization kernel [14], which reduces weight-loading cost while preserving compatibility with production-level inference optimizations. We use a fixed safety margin $\epsilon = 0.05$ for SD toggle policy and a shared draft-length set $\Gamma = \{5, 7, 9, 11\}$ for adaptive draft-length policy across all models. Full details are provided in Sec. E.3.

Baselines. We compare EfficientRollout against four baselines. First, **veRL (AR)** denotes accelerated AR inference using the standard rollout backend built on veRL and vLLM [28, 50], without SD. Second, the **history-based drafting** baseline evaluates Spec-RL [35], a rollout-history-based SD method using prefix matching, in the same rollout stack. We use its best-performing lenience factor $e^{0.5}$, which is needed for meaningful speedup but makes the method lossy and deviate from the target distribution. Third, the **learned auxiliary drafting** baseline evaluates an EAGLE3 [32]-based auxiliary drafter implemented on top of veRL and vLLM, following the RL-SD pipelines of FastRL [22] and NeMo RL [24]. We use a fixed draft length of $\gamma = 3$, following the best-performing setting in Iso et al. [24]. Lastly, **quantized self-SD** applies SD with the weight-quantized drafter from Sec. 4, with SD always enabled. It uses the same production-level kernel optimization as EfficientRollout, and we select the best fixed draft length $\gamma^* \in \{5, 7, 9, 11\}$ for each configuration. Additional details on the history-based drafting and learned auxiliary drafting baselines are provided in Secs. E.4 and E.5.

Metrics. We report per-step averaged latency metrics that separate SD-specific overhead, rollout latency, and end-to-end training time. Preparation time (Prep.) measures the extra work required by each SD method, including rollout-history lookup, online drafter forward/backward passes and parameter updates, and per-step drafter quantization. Rollout Gen. measures rollout-generation time, while Step Time measures end-to-end training-step time, including non-rollout components such as log-probability computation and actor updates. τ, $\bar{\gamma}$, and α denote the average block efficiency, average draft length, and per-token acceptance rate used to compare methods with different $\bar{\gamma}$ values, respectively. Additional measurement details are provided in Sec. E.2.

Table 2: End-to-end training acceleration across models and rollout decoding methods. We report SD-specific preparation time, average block efficiency τ ($\bar{\gamma}$ denotes the average draft length), per-token acceptance rate α, rollout-generation time, and training-step time. Signed percentages denote relative changes from the veRL (AR) baseline. In arrow-marked columns, bold and underlined entries indicate the **best** and second-best values within each model group.

Model	Drafting Method	Prep. (s) ↓	τ / $\bar{\gamma}$	α (%) ↑	Rollout Gen. (s) ↓	Step Time (s) ↓
Qwen2.5-7B	veRL (AR)	–	–	–	82.4	132.6
	History-based†	**1.1**	N/A	N/A	86.5 (+4.9%)	133.8 (+0.9%)
	Learned auxiliary	2.2	2.1 / 3.0	57.3	81.9 (-0.6%)	132.1 (-0.4%)
	Quantized self-SD	1.3	7.5 / 7.0	**98.1**	66.1 (-19.7%)	115.7 (-12.8%)
	EfficientRollout	1.3	8.6 / 8.3	**98.1**	**62.4 (-24.2%)**	**111.9 (-15.6%)**
Qwen2.5-14B	veRL (AR)	–	–	–	126.6	221.1
	History-based†	**1.1**	N/A	N/A	129.5 (+2.3%)	218.9 (-1.0%)
	Learned auxiliary	2.1	2.2 / 3.0	60.3	115.4 (-8.9%)	213.8 (-3.3%)
	Quantized self-SD	2.6	5.6 / 5.0	97.3	131.4 (+3.8%)	223.2 (+0.9%)
	EfficientRollout	2.6	7.4 / 7.0	**97.6**	**104.9 (-17.1%)**	**197.7 (-10.6%)**
Llama3.1-8B	veRL (AR)	–	–	–	126.4	186.9
	History-based†	**1.1**	N/A	N/A	131.1 (+3.7%)	187.4 (+0.2%)
	Learned auxiliary‡	2.6	2.0 / 3.0	54.3	172.8 (+36.7%)	234.8 (+25.6%)
	Quantized self-SD	1.4	5.5 / 5.0	**96.4**	116.7 (-7.6%)	178.1 (-4.7%)
	EfficientRollout	1.4	5.3 / 5.0	95.1	**112.2 (-11.2%)**	**173.3 (-7.3%)**

† For history-based drafting, τ and α are not directly comparable; prefix-reuse statistics are reported in Sec. F.5.
‡ We further analyze the Llama3.1-8B slowdown of learned auxiliary drafting in Sec. F.6.

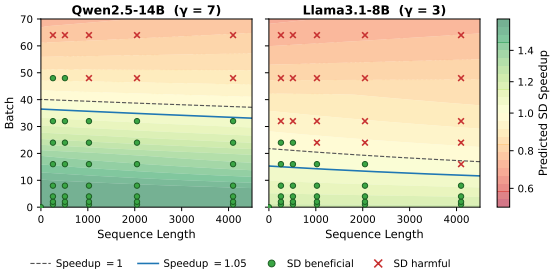


Figure 3: **Validation of the roofline-based toggle boundary.** Colors show the predicted speedup over the batch-size and sequence-length plane, and markers indicate whether measured SD is beneficial or harmful at the corresponding coordinates.

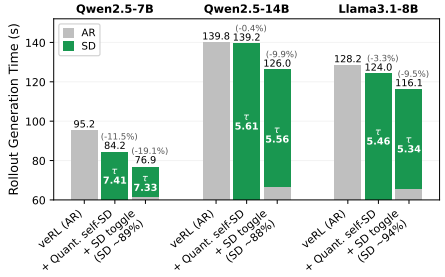
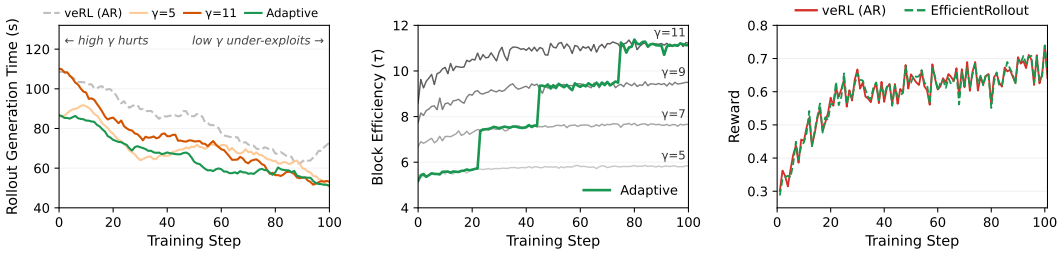


Figure 4: **Effect of regime-aware SD toggling.** Average rollout time over the first 50 training steps, showing that toggling avoids harmful early-regime speculation across all evaluated models.

5.2 End-to-End RL Training Acceleration

End-to-end latency. Table 2 shows that EfficientRollout achieves the largest latency reduction among all methods for every evaluated model. The per-step quantization overhead is modest (1.3–2.6 s) relative to the rollout-time savings: EfficientRollout reduces rollout-generation latency by up to 24.2%, yielding an end-to-end training-step latency reduction of up to 15.6% after accounting for all RL operations. We further quantify the individual contributions of W4 self-drafting, the Roofline toggle, and adaptive γ to rollout-generation and full-step latency reductions in Sec. F.4. The history-based drafting baseline has the lowest preparation overhead from history lookup (1.1 s), but does not reduce rollout-generation latency. Even after rollout history becomes available, it reuses only 4.4–50.2% of response tokens while adding 10.7–19.5 s of verification overhead (Tab. 8). Intuitively, prefix reuse acts like a large draft-and-verify block over the full candidate sequence, and even the highest reuse rate (50.2%) is insufficient to amortize the added verification cost in our setting. The learned auxiliary drafting baseline shows model-dependent, limited gains because its auxiliary drafter does not consistently provide sufficient block efficiency to offset SD overheads (Sec. 5.3). For example, it reduces Qwen2.5-14B step latency by 3.3% but increases Llama3.1-8B step latency by 25.6%. Finally, quantized self-SD reduces step latency on Qwen2.5-7B and Llama3.1-8B, but



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

Figure 5: **Adaptive draft-length 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.

the Qwen2.5-14B slowdown shows that the quantized self-drafter alone is not sufficient for robust speedups. We provide detailed analyses of slowdowns in the history-based drafting and learned auxiliary drafting baselines in Secs. F.5 and F.6.

Acceptance rate and block efficiency. Table 2 shows that EfficientRollout attains consistently high α (95.1–98.1%) together with large τ. Quantized self-SD also maintains high α across models, indicating strong alignment between the target model and the quantized self-drafter. EfficientRollout preserves this high α while using larger $\bar{\gamma}$, indicating that adaptive γ control can exploit training-time improvements in acceptance as it increases the draft budget. By contrast, the learned auxiliary drafting baseline remains limited to much lower α (54.3–60.3%) even with shorter $\bar{\gamma}$, reflecting limited auxiliary-drafter effectiveness in long, high-temperature RL rollouts (Sec. 5.3).

Training dynamics and quality preservation. EfficientRollout accelerates training while preserving training dynamics. Figure 5c shows that EfficientRollout closely tracks the veRL (AR) reward trajectory on Qwen2.5-7B. This aligns with the lossless nature of SD, which preserves the target distribution [6, 29] and is especially important in RL where rollout-distribution shifts can affect training stability [15, 64]. Full reward and validation-accuracy trajectories across all evaluated models are provided in Sec. F.2.

Broader empirical validation. We extend the primary evaluation by varying hardware, model scale and parallelism, rollout temperature, and maximum response length. On H100, it reduces rollout-generation and full-step latency by up to 16.7% and 12.5%, respectively. For Qwen2.5-32B using TP$=2$ on A100, it reduces rollout-generation latency by 13.0%. With a lower temperature (0.6) and a 16k maximum response length on A100, the reductions reach 40.1% and 34.0%, respectively. Full settings and results are provided in Sec. F.8.

5.3 Further Analysis of Rollout Acceleration

Regime-aware SD toggle policy. High τ alone is insufficient for realized speedup, and SD must be activated only in system-beneficial regimes via the toggle policy π_{SD}. Figure 3 validates the calibrated roofline boundary of π_{SD} by showing that the predicted speedup correctly separates SD-beneficial and SD-harmful coordinates in the (B, S) plane. Figure 4 isolates the effect of π_{SD} by comparing rollout latency with the same quantized self-drafter when SD is always enabled versus enabled according to π_{SD}. By disabling SD during only the early 6–11% of decoding steps, the π_{SD} policy yields consistently larger rollout-generation latency reductions relative to AR than always-on SD, despite slightly lower τ. These gains therefore cannot be attributed to better draft quality; they come from avoiding verification in the initial large-batch regime, where T_V/T_p is high. Further A100 and H100 boundary validation and calibration sensitivity for π_{SD} are reported in Sec. D.3.

Adaptive draft-length policy. The adaptive γ policy effectively tracks the latency-reducing γ as the evolving target policy shifts over training. We isolate this effect by replacing only the adaptive γ policy in EfficientRollout with fixed-γ variants throughout training. In this controlled policy ablation, the adaptive γ policy achieves an overall 19.6% reduction in rollout-generation latency, compared with 13.5% and 11.8% for fixed $\gamma = \gamma_{\mathrm{low}} = 5$ and $\gamma = \gamma_{\mathrm{high}} = 11$, respectively, as shown in Fig. 5a.

This improvement comes from leveraging larger latency reductions at different stages of training, as smaller and larger γ values are more effective early and later in training, respectively. Rather than selecting a single fixed γ through post-hoc sweeps, the adaptive γ policy moves within the pre-specified Γ using observed τ feedback.

Learned auxiliary drafting in RL workloads. The lower τ and α of learned auxiliary drafting in Tab. 2 reflect the practical difficulty of obtaining an auxiliary drafter aligned with long, high-temperature RL rollouts. Under the prior RL-SD setup of Iso et al. [24], we further evaluate auxiliary drafters on DAPO-Math-17K [64], the rollout workload used in that setup. As shown in Fig. 6, the evaluated auxiliary drafters [42, 44] achieve only $\tau = 1.2$–2.4 over the first 30 steps, whereas the quantized self-drafter consistently reaches $\tau = 3.6$–3.9 under the same setting. These results suggest that directly reusing public checkpoints or drafters pretrained with fixed-target LLM-serving recipes [32, 72] is insufficient to reliably obtain high τ in RL rollouts. Realizing effective speedups with learned auxiliary drafting may require a more specialized auxiliary-drafter training pipeline [24, 75], such as collecting in-distribution rollouts, training drafters on target-generated synthetic data, and possibly using more aggressive online adaptation. Detailed evaluation settings and analyses with additional auxiliary-drafter checkpoints are presented in Sec. F.7.

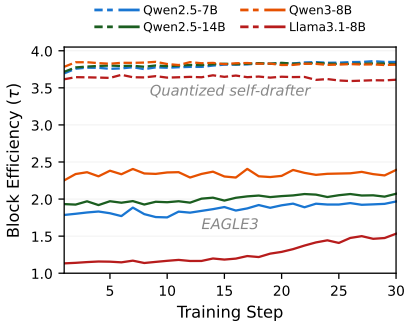


Figure 6: **Block efficiency across pretrained auxiliary drafters.** On DAPO-Math-17K, the evaluated pretrained auxiliary drafters achieve lower block efficiency than quantized self-drafters.

6 Discussion and Future Directions

We present EfficientRollout, a system-aware self-SD framework for RL rollouts, designed around evolving policies, shrinking-batch dynamics, and rollout-tail latency. By combining a target-induced quantized drafter, dynamic SD toggling, and adaptive draft budgets, EfficientRollout reduces rollout and end-to-end latency without separate drafter pretraining, warm-up, online adaptation, or invasive RL pipeline changes.

Future directions. Several directions remain for improving and extending system-aware self-SD for RL rollouts. First, we focus on chain verification because it is practical under dynamic batch sizes [36]; tree verification is orthogonal and could be incorporated into our framework [31, 32]. 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. Adaptive, less-frequent drafter refreshes may further reduce preparation overhead while preserving policy alignment. Finally, when KV-cache loading dominates in long-context or dense-attention regimes [11], sparse-attention drafting may further reduce quantized self-SD cost.

Acknowledgments and Disclosures

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). This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) under the Artificial Intelligence Convergence Innovation Human Resources Development (IITP-2026-RS-2023-00255968) grant funded by the Korea government (MSIT). This work was supported by Artificial intelligence industrial convergence cluster development project funded by the Ministry of Science and ICT (MSIT, Korea) & Gwangju Metropolitan City. 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.

Competing interests. The authors declare no competing interests beyond the affiliations and funding disclosed in this paper.

References

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

[2] Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4895–4901, 2023.

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

[4] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 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.

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

[6] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

[7] Longze Chen, Renke Shan, Huiming Wang, Lu Wang, Ziqiang Liu, Run Luo, Jiawei Wang, Hamid Alinejad-Rokny, and Min Yang. 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)*, pages 31608–31618, 2025.

[8] Qiaoling Chen, Zijun Liu, Peng Sun, Shenggui Li, Guoteng Wang, Ziming Liu, Yonggang Wen, Siyuan Feng, and Tianwei Zhang. 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.

[9] Xingyu Chen, Jiahao Xu, Tian Liang, Zhiwei He, Jianhui Pang, Dian Yu, Linfeng Song, Qiuzhi Liu, Mengfei Zhou, Zhuosheng Zhang, Rui Wang, Zhaopeng Tu, Haitao Mi, and Dong Yu. Do NOT think that much for 2+3=? on the overthinking of long reasoning models. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=MSbU3L7V00.

[10] Zhuoming Chen, Hongyi Liu, Yang Zhou, Haizhong Zheng, and Beidi Chen. Jackpot: Optimal budgeted rejection sampling for extreme actor-policy mismatch reinforcement learning. *arXiv preprint arXiv:2602.06107*, 2026.

[11] Jean-Baptiste Cordonnier, Andreas Loukas, and Martin Jaggi. Multi-head attention: Collaborate instead of concatenate. *arXiv preprint arXiv:2006.16362*, 2020.

[12] Ganqu Cui, Yuchen Zhang, Jiacheng Chen, Lifan Yuan, Zhi Wang, Yuxin Zuo, Haozhan Li, Yuchen Fan, Huayu Chen, Weize Chen, et al. The entropy mechanism of reinforcement learning for reasoning language models. *arXiv preprint arXiv:2505.22617*, 2025.

[13] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=OUIFPHEgJU.

[14] Elias Frantar, Roberto L Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. 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*, pages 239–251, 2025.

[15] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, WANG JIASHU, Tongkai Yang, Binhang Yuan, and Yi Wu. 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`.

[16] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. Ai and memory wall. *IEEE Micro*, 44(3):33–39, 2024.

[17] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[18] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[19] Jingkai He, Tianjian Li, Erhu Feng, Dong Du, Qian Liu, Tao Liu, Yubin Xia, and Haibo Chen. History rhymes: Accelerating llm reinforcement learning with rhymerl. *arXiv preprint arXiv:2508.18588*, 2025.

[20] Zhenyu He, Zexuan Zhong, Tianle Cai, Jason Lee, and Di He. 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)*, pages 1582–1595, 2024.

[21] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

[22] Qinghao Hu, Shang Yang, Junxian Guo, Xiaozhe Yao, Yujun Lin, Yuxian Gu, Han Cai, Chuang Gan, Ana Klimovic, and Song Han. Taming the long-tail: Efficient reasoning rl training with adaptive drafter. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1933–1948, 2026.

[23] Audrey Huang, Adam Block, Dylan J Foster, Dhruv Rohatgi, Cyril Zhang, Max Simchowitz, Jordan T. Ash, and Akshay Krishnamurthy. Self-improvement in language models: The sharpening mechanism. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=WJaUkwci9o`.

[24] Hayate Iso, Tiyasa Mitra, Sudipta Mondal, Rasoul Shafipour, Venmugil Elango, Terry Kong, Yuki Huang, Seonjin Na, Izzy Putterman, Benjamin Chislett, et al. Accelerating rl post-training rollouts via system-integrated speculative decoding. *arXiv preprint arXiv:2604.26779*, 2026.

[25] Renren Jin, Pengzhi Gao, Yuqi Ren, Zhuowen Han, Tongxuan Zhang, Wuwei Huang, Wei Liu, Jian Luan, and Deyi Xiong. Revisiting entropy in reinforcement learning for large reasoning models. *arXiv preprint arXiv:2511.05993*, 2025.

[26] Minseo Kim, Coleman Hooper, Aditya Tomar, Chenfeng Xu, Mehrdad Farajtabar, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. Beyond next-token prediction: A performance characterization of diffusion versus autoregressive language models. *arXiv preprint arXiv:2510.04146*, 2025.

[27] Komal Kumar, Tajamul Ashraf, Omkar Thawakar, Rao Muhammad Anwer, Hisham Cholakkal, Mubarak Shah, Ming-Hsuan Yang, Phillip HS Torr, Fahad Shahbaz Khan, and Salman Khan. Llm post-training: A deep dive into reasoning large language models. *arXiv preprint arXiv:2502.21321*, 2025.

[28] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

[29] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[30] Yuhang Li, Reena Elangovan, Xin Dong, Priyadarshini Panda, and Brucek Khailany. QuRL: Low-precision reinforcement learning for efficient reasoning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=eG0bpCwdKn.

[31] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=1NdN7eXyb4.

[32] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 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.

[33] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of machine learning and systems*, 6:87–100, 2024.

[34] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

[35] Bingshuai Liu, Ante Wang, Zijun Min, Liang Yao, Haibo Zhang, Yang Liu, Anxiang Zeng, and Jinsong Su. Spec-rl: Accelerating on-policy reinforcement learning via speculative rollouts. *arXiv preprint arXiv:2509.23232*, 2025.

[36] Xiaoxuan Liu, Jiaxiang Yu, Jongseok Park, Ion Stoica, and Alvin Cheung. Speculative decoding: Performance or illusion? In *Ninth Conference on Machine Learning and Systems*, 2026. URL https://openreview.net/forum?id=fzkqtezFEi.

[37] Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025.

[38] 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.

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

[40] Jie Ou, Yueming Chen, 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)*, pages 10–22, 2024.

[41] Siru Ouyang, Shuohang Wang, Minhao Jiang, Ming Zhong, Donghan Yu, Jiawei Han, and Yelong Shen. Temperature-centric investigation of speculative decoding with knowledge distillation. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 13125–13137, 2024.

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

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

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

[45] Ranajoy Sadhukhan, Jian Chen, Zhuoming Chen, Vashisth Tiwari, Ruihang Lai, Jinyuan Shi, Ian En-Hsu Yen, Avner May, Tianqi Chen, and Beidi Chen. 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.

[46] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[47] Zelei Shao, Vikranth Srivatsa, Sanjana Srivastava, Qingyang Wu, Alpay Ariyak, Xiaoxia wu, Ameen Patel, Jue Wang, Percy Liang, Tri Dao, Ce Zhang, Yiying Zhang, Ben Athiwaratkun, Chenfeng Xu, and Junxiong Wang. 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.

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

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

[50] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297, 2025.

[51] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[52] Mingbo Song, Heming Xia, Jun Zhang, Chak Tou Leong, Qiancheng Xu, Wenjie Li, and Sujian Li. Knn-ssd: Enabling dynamic self-speculative decoding via nearest neighbor layer set optimization. In *Findings of the Association for Computational Linguistics: EACL 2026*, pages 641–655, 2026.

[53] Lawrence Stewart, Matthew Trager, Sujan Kumar Gonugondla, and Stefano Soatto. The n-grammys: Accelerating autoregressive inference with learning-free batched speculation. *arXiv preprint arXiv:2411.03786*, 2024.

[54] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. 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.

[55] Kimi Team, Yifan Bai, Yiping Bao, Y Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

[56] Rishabh Tiwari, Haocheng Xi, Aditya Tomar, Coleman Richard Charles Hooper, Sehoon Kim, Maxwell Horton, Mahyar Najibi, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. 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.

[57] Xumeng Wen, Zihan Liu, Shun Zheng, Shengyu Ye, Zhirong Wu, Yang Wang, Zhijian Xu, Xiao Liang, Junjie Li, Ziming Miao, Jiang Bian, and Mao Yang. 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.

[58] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[59] Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding. *Findings of the Association for Computational Linguistics: ACL 2024*, pages 7655–7671, 2024.

[60] Heming Xia, Yongqi Li, Jun Zhang, Cunxiao Du, and Wenjie Li. 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.

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

[62] Chenghao Yang, Sida Li, and Ari Holtzman. Llm probability concentration: How alignment shrinks the generative horizon. *arXiv preprint arXiv:2506.17871*, 2025.

[63] Qwen An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxin Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yi-Chao Zhang, Yunyang Wan, Yuqi Liu, Zeyu Cui, Zhenru Zhang, Zihan Qiu, Shanghaoran Quan, and Zekun Wang. Qwen2.5 technical report. *ArXiv*, abs/2412.15115, 2024. URL https://api.semanticscholar.org/CorpusID:274859421.

[64] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, YuYue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. 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.

[65] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, et al. Llm inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363*, 2024.

[66] Yikang Yue, Yuqi Xue, and Jian Huang. Specattn: Co-designing sparse attention with self-speculative decoding. *arXiv preprint arXiv:2602.07223*, 2026.

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

[68] Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

[69] Weihao Zeng, Yuzhen Huang, Qian Liu, Wei Liu, Keqing He, Zejun MA, and Junxian He. SimpleRL-zoo: Investigating and taming zero reinforcement learning for open base models in the wild. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=vSMCBUgrQj.

[70] Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. 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)*, pages 11263–11282, 2024.

[71] Yiqi Zhang, Huiqiang Jiang, Xufang Luo, Zhihe Yang, Chengruidong Zhang, Yifei Shen, Dongsheng Li, Yuqing Yang, Lili Qiu, and Yang You. Sortedrl: Accelerating rl training for llms through online length-aware scheduling. *arXiv preprint arXiv:2603.23414*, 2026.

[72] Yizhou Zhang, Ning Lv, Teng Wang, and Jisheng Dang. 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.

[73] Rosie Zhao, Alexandru Meterez, Sham M. Kakade, Cengiz Pehlevan, Samy Jelassi, and Eran Malach. Echo chamber: RL post-training amplifies behaviors learned in pretraining. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=dp4KWuSDzj.

[74] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=VqkAKQibpq.

[75] Yongchao Zhou, Kaifeng Lyu, Ankit Singh Rawat, Aditya Krishna Menon, Afshin Rostamizadeh, Sanjiv Kumar, Jean-François Kagy, and Rishabh Agarwal. Distillspec: Improving speculative decoding via knowledge distillation. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=rsY6J3ZaTF.

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

Appendix

A Extended Analysis of Shrinking-Batch Dynamics **18**

B System Rationale for Weight-Quantized Self-Drafting in RL Rollouts **18**

B.1 Detailed Decode-Time Decomposition in Rollout-Tail Regimes 18

B.2 Practical Trade-offs between Quantized and Sparse-Attention Self-Drafting 18

B.3 W4 versus W8: Latency–Acceptance Trade-off 19

B.4 Choosing RTN for Step-Wise Drafter Refresh 20

C Policy Sharpening Improves Quantized Drafter Alignment **20**

D Implementation and Calibration Details of EfficientRollout **21**

D.1 KV-Cache Sharing between the Quantized Drafter and Target 21

D.2 Roofline-Based SD Toggle Model and Calibration 22

D.3 Validation of the Roofline-Based SD Toggle Policy 23

E Detailed Experimental Setup **24**

E.1 Infrastructure, Datasets, and Reporting Window 24

E.2 Metric Measurement Details 24

E.3 Training and Method Configuration 25

E.4 Rollout-History-Based Drafting Baseline 25

E.5 Learned Auxiliary Drafting Baseline 26

F Additional Evaluation Results **26**

F.1 Per-Step Rollout Generation Time 26

F.2 Training Dynamics and Quality Preservation 27

F.3 Adaptive Draft-Length Schedules Across Models 27

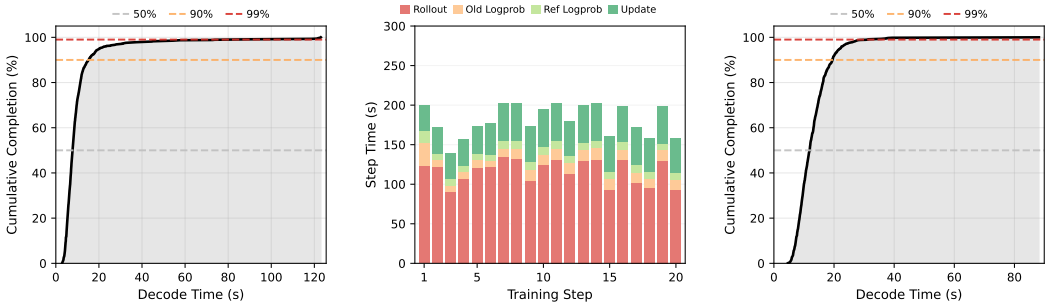
F.4 Component-Level Decomposition 28

F.5 Why History-Based Drafting Does Not Accelerate 28

F.6 Why Learned Auxiliary Drafting Remains Challenging 28

F.7 Auxiliary Drafters Aligned with RL Rollout Distributions Are Difficult to Obtain . 30

F.8 Extended Evaluation Across Hardware, TP, and Workload Settings 32



(a) (b) (c)

Figure 7: 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.

A Extended Analysis of Shrinking-Batch Dynamics

Figure 7 provides additional evidence that rollout generation remains the dominant bottleneck and exhibits a long-tail completion pattern across models. For Qwen2.5-7B, Fig. 7b 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. 7a 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. 7c, 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 dominate 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.

B System Rationale for Weight-Quantized Self-Drafting in RL Rollouts

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

We follow the roofline-based simulation methodology of prior work [26] 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 [63] and Llama3.1-8B-Instruct [17].

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.

B.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

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

patterns that are updated dynamically at each draft step [54, 66]. However, such fine-grained sparsity is difficult to realize efficiently in vLLM, whose memory management is page-granular rather than token-granular [28]. 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 [66]. 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 [14]. Taken together, these observations make quantized self-SD the most effective and practical self-drafting strategy for RL rollout acceleration in our setting.

B.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 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.

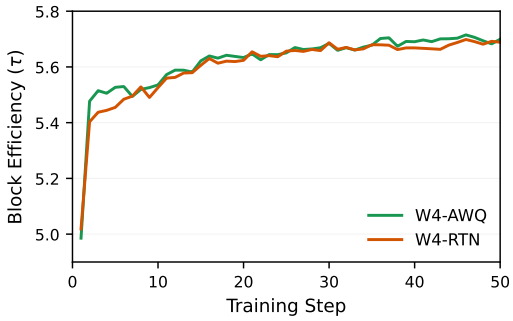


Figure 8: Block efficiency τ over the first 50 training steps on Qwen2.5-7B with W4-RTN and W4-AWQ drafters.

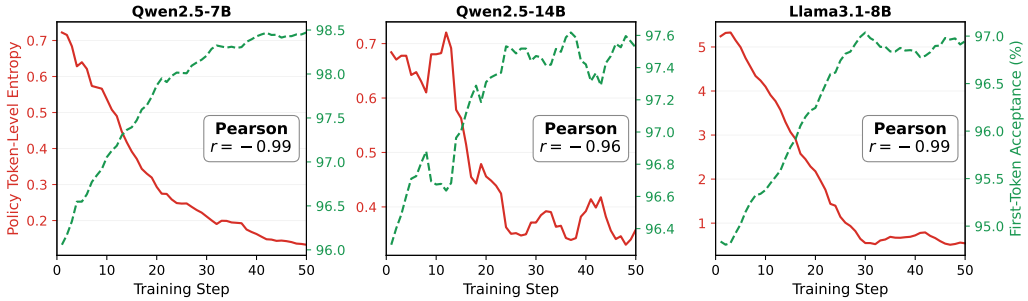


Figure 9: Policy sharpening improves quantized drafter alignment. As target-policy entropy decreases, first-token acceptance of the RTN W4 drafter increases across models.

B.4 Choosing RTN for Step-Wise Drafter Refresh

We also experimented with an activation-aware quantization (AWQ) pipeline [33] 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 8 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 [13]. 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.

C 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:



Figure 10: Policy sharpening improves quantized drafter alignment. As target-policy entropy decreases, block efficiency τ of the RTN W4 drafter increases across models.

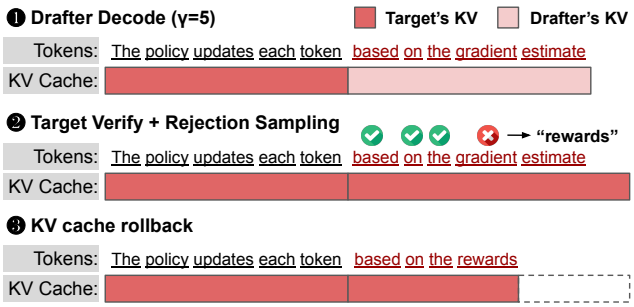


Figure 11: Quantized self-SD with shared KV cache, target verification via rejection sampling, and rollback of rejected KV cache.

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 9 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 10 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.

D Implementation and Calibration Details of EfficientRollout

D.1 KV-Cache Sharing between the Quantized Drafter and Target

Figure 11 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.

D.2 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 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.

For TP> 1, the target and W4 drafter are co-sharded under the same TP layout, and each rank requantizes only its local target shard, so drafter refresh requires no additional model-wide collective. We can account for TP communication by adding the measured collective cost c_{comm} to each calibrated latency estimate:

$$\widehat{T}_{x,\text{TP}}(B, S, \gamma) = \widehat{T}_x(B, S, \gamma) + c_{\text{comm}}, \qquad x \in \{p, q, V\} \tag{4}$$

The adjusted target, drafter, and verification costs enter the same toggle rule in Eq. (3); end-to-end TP$= 2$ results are reported in Sec. F.8.

Calibration procedure. We calibrate the model in three stages. First, we measure the effective compute throughput F_{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 BW_{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

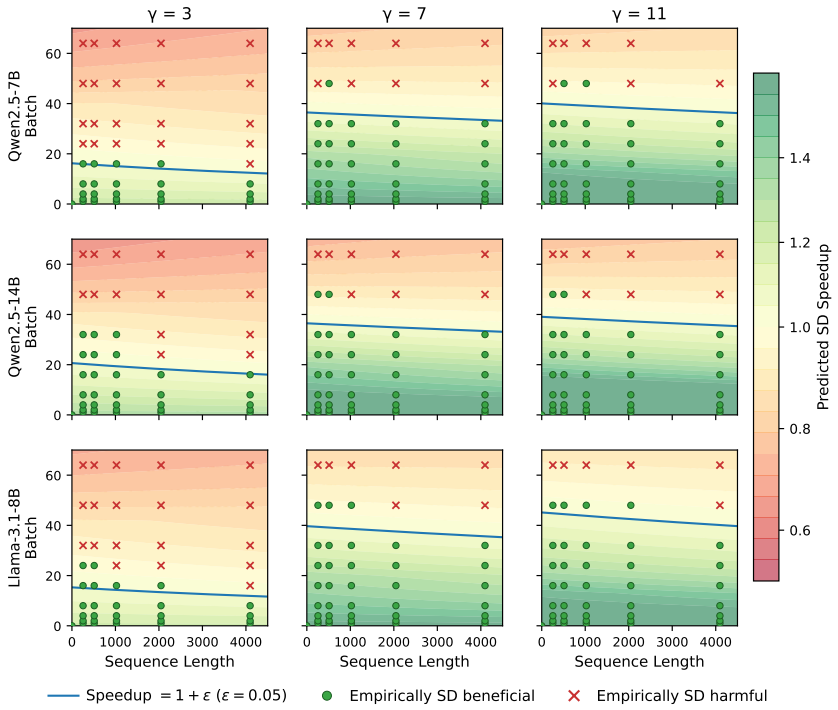


Figure 12: Validation of the roofline-based SD toggle policy. Background colors show predicted speedup, the blue line marks $\mathrm{Speedup}_{\mathrm{SD}} = 1 + \epsilon$, and markers indicate empirical SD-beneficial/harmful points.

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.

D.3 Validation of the Roofline-Based SD Toggle Policy

Toggle-boundary validation. 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 $T_p(B, S)$, $T_q(B, S)$, and $T_V(B, S, \gamma)$ 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):

$$\mathrm{Speedup}_{\mathrm{emp}}(B, S, \gamma) = \frac{\tau T_p(B, S)}{\gamma T_q(B, S) + T_V(B, S, \gamma)}. \tag{5}$$

For this validation, we use the optimistic setting $\tau = \gamma$ to isolate the cost-model accuracy from block-efficiency variation. Figure 12 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 $\mathrm{Speedup}_{\mathrm{SD}} = 1 + \epsilon$ 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.

Calibration sensitivity. We independently perturb each of the seven calibrated terms by $\pm 10\%$ and compare the resulting policy decisions with those from the unperturbed calibration. The two

Table 5: Sensitivity of the Roofline toggle policy to independent $\pm 10\%$ perturbations of each calibrated term. Δ is the worst absolute change in sign accuracy relative to the unperturbed calibration; flips are the fraction of grid cells whose AR/SD decision changes.

Perturbed Term	A100 Δ (pp)	A100 Flips (%)	H100 Δ (pp)	H100 Flips (%)
$\mathrm{BW}_{\mathrm{eff}}$	2.7	4.9	2.2	5.1
$\mathrm{F}_{\mathrm{eff}}$	2.7	2.2	1.7	2.8
η_D	4.9	6.6	7.3	16.9
κ_{eff}	0.0	0.0	0.0	0.0
c_T	2.7	3.3	2.2	2.2
c_D	2.2	4.4	3.4	4.5
c_V	0.5	2.2	1.7	0.6

Table 6: H100 toggle-policy validation after one-time model–hardware recalibration.

Model	Sign Accuracy (%)	Harmful Activation (%)
Qwen2.5-7B	88.6	4.0
Qwen2.5-14B	94.5	0.0
Llama3.1-8B	91.6	0.6

metrics capture complementary effects: Δ measures the largest absolute change in aggregate sign accuracy, whereas flips count grid cells whose AR/SD decision changes. Across the A100 and H100 validation grids, six of the seven terms change sign accuracy by at most 3.4 pp and flip at most 5.1% of decisions, as shown in Tab. 5. The W4 drafter-cost multiplier η_D is the most sensitive term, flipping 6.6% and 16.9% of decisions on A100 and H100, respectively. However, an independent calibration on a second A100 node changes η_D by only -0.5% to $+0.9\%$ across the three models. This cross-node variation is far smaller than the $\pm 10\%$ perturbation range, supporting the practical stability of calibration.

Cross-hardware H100 validation. We repeat the boundary validation on H100 after recalibrating once for each model–hardware pair. Sign accuracy is the fraction of grid cells for which the predicted SD-beneficial or SD-harmful label matches measurement; harmful activation is the fraction of cells activated with $\epsilon = 0.05$ for which measured SD is slower than AR. As shown in Tab. 6, sign accuracy ranges from 88.6% to 94.5%, while harmful activation remains between 0.0% and 4.0%. These results show that the same calibration-and-control procedure recovers a reliable hardware-specific boundary on H100, rather than assuming a universal boundary across hardware.

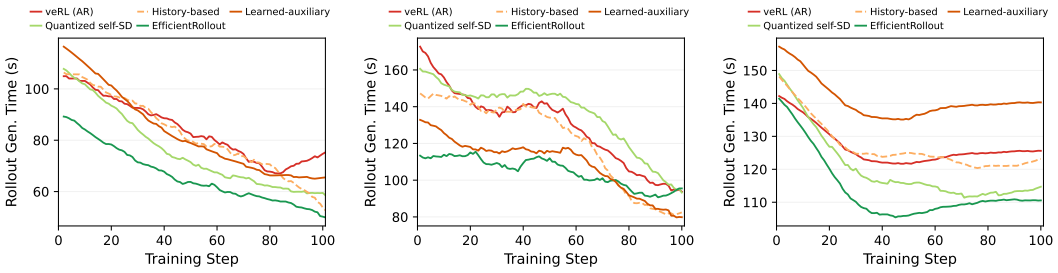
E Detailed Experimental Setup

E.1 Infrastructure, Datasets, and Reporting Window

We use veRL [50] v0.7.0 with vLLM [28] 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 [69]. SimpleRL-8k-hard and SimpleRL-8k-medium correspond to MATH [21] 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 [69]. 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 steps.

E.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



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

Figure 13: Rollout-generation time over training steps. EfficientRollout consistently reduces rollout time, whereas other SD baselines provide limited gains or can be slower than veRL (AR). Curves are smoothed for visualization; Tab. 2 reports averages of the unsmoothed measurements.

integer. We compute α from the measured τ using the equation $\tau = (1 - \alpha^{\gamma+1})/(1 - \alpha)$. When γ varies over training, we apply the same inversion within fixed-γ portions of the trajectory and average the resulting rates weighted by the number of training steps in each portion. 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.

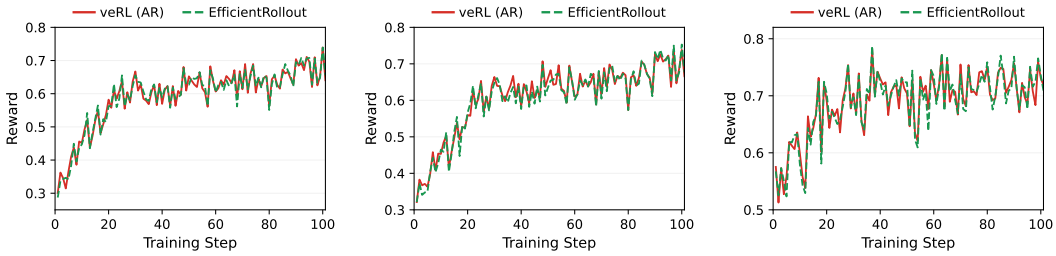
E.3 Training and Method Configuration

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 [69], and for the Llama instruct model, we use 10^{-2} following FastGRPO [72]. 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 use the same configuration for all models: $\Gamma = \{5, 7, 9, 11\}$, $\alpha_{\text{up}} = 0.94$, $\alpha_{\text{down}} = 0.85$, and $P = 2$. We omit $\gamma = 3$ because it is consistently dominated by, or comparable to, $\gamma = 5$ in the rollout regimes encountered during training.

E.4 Rollout-History-Based Drafting Baseline

We use Spec-RL [35] 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 [29], 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.



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

Figure 14: 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.

E.5 Learned Auxiliary Drafting Baseline

Existing RL rollout acceleration systems with learned auxiliary drafting use different execution stacks. FastGRPO [72] uses a HuggingFace Transformers-based rollout pipeline rather than a serving backend such as vLLM or SGLang. FastRL [22] uses SGLang [74] for rollout generation, while NeMo RL [24] 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 [1] using the official vLLM SD training pipeline following EAGLE3 [32]. 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 [42].

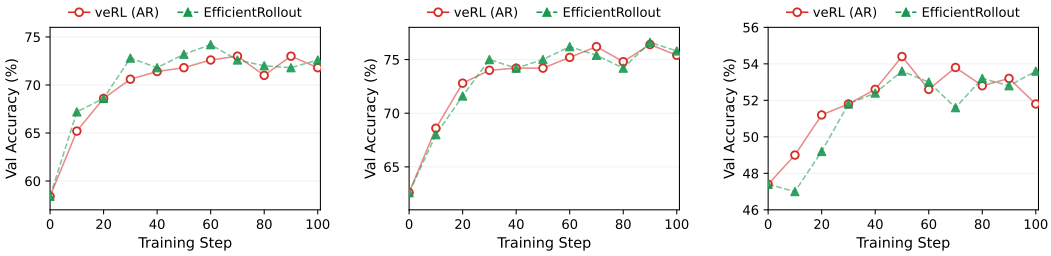
For online adaptation during RL, we carefully follow the rollout-SD design choices of FastRL [22] and NeMo RL [24]. Specifically, we capture hidden states during the actor forward pass in the log-probability computation phase and train the drafter inline immediately after each micro-batch of the actor 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. [24].

F Additional Evaluation Results

F.1 Per-Step Rollout Generation Time

Figure 13 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



(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.

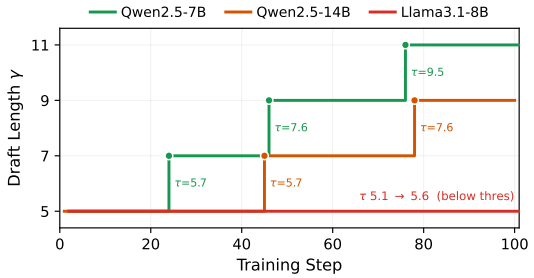


Figure 16: Adaptive γ schedules across models. The controller increases γ only when block efficiency τ is high enough to support a longer draft; otherwise, it keeps γ unchanged.

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. 9. These per-step trends are consistent with the aggregate timing results in Tab. 2.

F.2 Training Dynamics and Quality Preservation

Figures 14 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.

Repeated-run consistency. Across three paired data-order seeds for Qwen2.5-7B, EfficientRollout reduces rollout-generation and full-step latency by $22.2\% \pm 1.8$ pp and $14.0\% \pm 1.5$ pp (mean$\pm$SD), respectively. Final validation accuracy remains comparable between AR and EfficientRollout ($73.2\% \pm 1.6$ pp vs. $73.3\% \pm 0.3$ pp), with no meaningful difference in late-training reward.

F.3 Adaptive Draft-Length Schedules Across Models

Figure 16 shows the γ schedule selected by the adaptive controller across models. Qwen2.5-7B increases from $5 \to 7 \to 9 \to 11$ 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 $5 \to 7 \to 9$ when its block efficiency reaches 5.73 and 7.61. In contrast, Llama3.1-8B-Instruct stays at $\gamma = 5$: although its block

Table 7: Component-level decomposition of EfficientRollout for Qwen2.5-7B over 100 steps. Each row adds one component to the preceding configuration using a common execution path.

Sequential Configuration	**Rollout Gen. (s)**	**Step Time (s)**	**Rollout / Step vs. AR (%)**
veRL (AR)	82.4	132.6	–
+ Always-on W4 self-SD, fixed $\gamma = 7$	75.6	127.6	$-8.2/-3.7$
+ Roofline toggle, fixed $\gamma = 7$	68.2	118.0	$-17.2/-11.0$
+ Adaptive γ (full EfficientRollout)	66.3	115.7	$-19.6/-12.7$

Table 8: Active-window analysis of history-based drafting after rollout history becomes available. Statistics are reported from the second epoch onward.

Model	**Resp. Len. (tokens)**	**Reused Prefix (tokens)**	**Reuse Rate (%)**	**Verify Overhead (s/step)**	**Gen. Time Δ (s vs. No-SD)**
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

efficiency improves from 5.07 to around 5.5, it never crosses the elevation threshold. No downward adjustment is triggered in these runs.

F.4 Component-Level Decomposition

Table 7 attributes 8.2/3.7, 9.0/7.3, and 2.4/1.7 pp of rollout/full-step reduction to W4 self-drafting, the Roofline toggle, and adaptive γ, respectively. The absolute timings were measured before removing CPU-side bubbles from W4 drafting, whereas Tab. 2 includes this optimization; because every row uses the same execution path, the factor-wise attribution remains controlled.

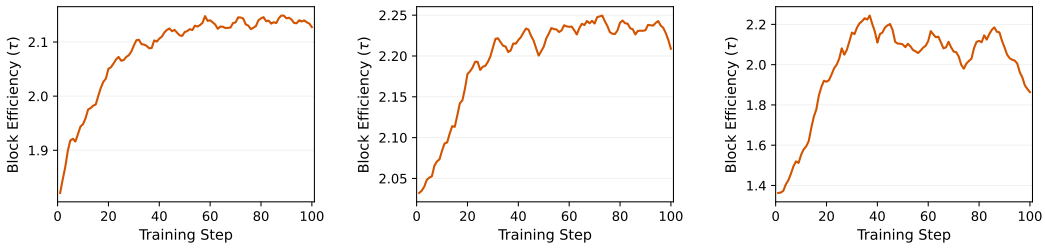
F.5 Why History-Based Drafting Does Not Accelerate

Table 8 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. Unlike fixed-γ chain drafting, the history-based drafting method reuses variable-length prefixes from previous rollouts, so block efficiency τ and effective acceptance rate α are not directly comparable. We therefore analyze prefix reuse, verification overhead, and rollout-generation time in this window. 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.

F.6 Why Learned Auxiliary Drafting Remains Challenging

Block efficiency of learned auxiliary drafting. EAGLE3-style auxiliary drafters typically require pretraining and online adaptation to match the target model and rollout data distribution [22, 24, 72]. 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. 17, 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



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

Figure 17: 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.

Table 9: 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

initial block efficiency [24]; however, such in-distribution drafter pretraining is not always available in practice, so we use a drafter pretrained on general chat/text data [32].

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 search burden; we further analyze the workload-matching requirement in Sec. F.7.

Unexpected slowdown on Llama3.1-8B. Li et al. [32] 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. 5.1, using vLLM EAGLE3 timing (`VLLM_SD_TIMING=1`), position-binned block efficiency logging, and a custom proposal timer.

Table 9 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. 10, 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. 10 and 11). 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

Table 10: 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 11: 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

expensive. This suggests a model- and workload-dependent failure mode of learned auxiliary drafting rather than a stable speedup across RL rollouts.

F.7 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 [50]-based implementation or from the drafter itself, we additionally evaluate EAGLE3 drafters in NVIDIA's native NeMo RL SD stack [24]. This stack uses Megatron [51]-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 [64] 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. For Qwen3-8B, we evaluate three public EAGLE3 drafters: RedHatAI's thinking drafter [44], RedHatAI's non-thinking drafter [43], and AngelSlim's drafter [3]. 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. E.5. For Llama3.1-8B-Instruct, we evaluate both the RedHatAI public drafter [42] and the official checkpoint from the EAGLE3 authors released by yuhuili [67]. We also evaluate the public Qwen2.5-7B EAGLE-style drafter [38] released with FastRL [22] as an RL-targeted auxiliary-drafter checkpoint. The checkpoint is reported to be trained on the OpenThoughts2-1M dataset [39], 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 18 shows that evaluated learned auxiliary drafters achieve substantially lower block efficiency than target-induced quantized 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.

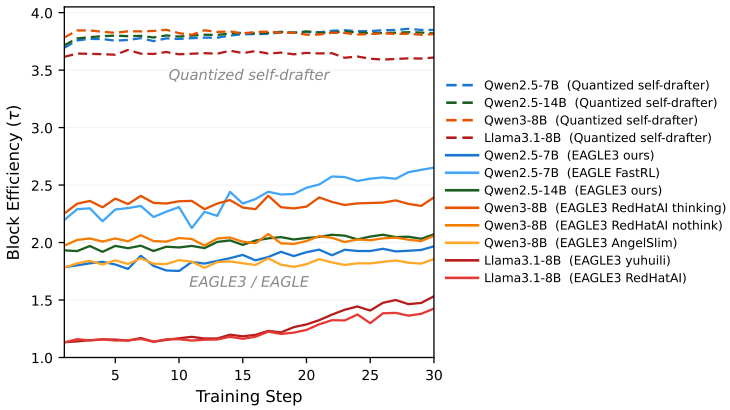
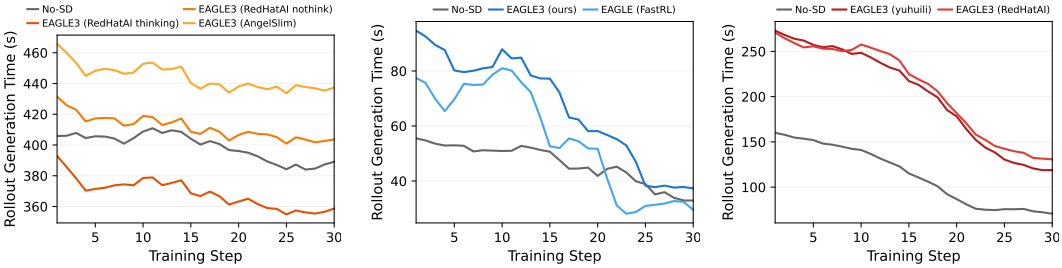


Figure 18: Block efficiency on DAPO-Math-17K. Across the first 30 RL steps, evaluated learned auxiliary drafters remain largely below target-induced quantized drafters.



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

Figure 19: 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.

Generation-time consequence. Figure 19 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 checkpoints or drafters pretrained on general chat/text data 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. [22] train their drafter on OpenThoughts2-1M [39], 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. [24] 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.

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

Table 12: Training results on H100 over 100 steps. We report EfficientRollout acceptance rate and AR→EfficientRollout latency; signed percentages are relative to AR.

Model	α **(%)** ↑	**Rollout Gen. (s)** ↓	**Step Time (s)** ↓
Qwen2.5-7B	98.2	60.1 → 50.1 (-16.7%)	86.5 → 76.3 (-11.8%)
Qwen2.5-14B	97.6	90.1 → 80.9 (-10.3%)	139.0 → 129.2 (-7.1%)
Llama3.1-8B	96.2	87.0 → 72.6 (-16.6%)	120.9 → 105.9 (-12.5%)

Table 13: End-to-end evaluation for Qwen2.5-32B with TP$=2$ on A100 over 100 steps. Metrics follow Tab. 2; signed percentages are relative to AR.

Drafting Method	**Prep. (s)** ↓	τ **/** $\bar{\gamma}$	α **(%)** ↑	**Rollout Gen. (s)** ↓	**Step Time (s)** ↓
AR	–	–	–	162.4	308.7
EfficientRollout	3.3	9.0 / 8.6	98.5	141.2 (-13.0%)	289.0 (-6.4%)

Table 14: Additional matched 50-step Qwen2.5-7B evaluations on A100. Response lengths are in tokens; Avg. Max Response is the mean length of the longest sampled response across steps.

Temperature	**Max Response**	**Avg. Max Response**	α **(%)**	**Rollout / Step Reduction (%)**
0.6	8,192	8,157	99.2	27.2 / 19.0
1.0	16,384	7,363	97.6	24.3 / 18.6
0.6	16,384	13,561	99.0	**40.1 / 34.0**

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.

F.8 Extended Evaluation Across Hardware, TP, and Workload Settings

Cross-hardware H100 evaluation. We evaluate on H100 using the main evaluation configuration, recalibrating once per model–hardware pair. Table 12 shows that EfficientRollout maintains a 96.2–98.2% acceptance rate while reducing rollout-generation and full-step latency by 10.3–16.7% and 7.1–12.5%, respectively.

Capacity-driven TP$=2$ execution. EfficientRollout naturally supports end-to-end execution with tensor parallelism (TP) by incorporating the calibrated collective term in Eq. (4) into its toggle policy. Table 13 shows that, for Qwen2.5-32B with capacity-driven TP$=2$ on 8×A100 GPUs, EfficientRollout reduces rollout-generation and full-step latency by 13.0% and 6.4%, respectively. In TP settings, irreducible collective costs raise the effective T_q/T_p ratio and narrow the SD-beneficial region. When the target and drafter fit at TP$=1$, data-parallel rollout workers therefore provide a more favorable latency regime.

Temperature and response-length settings. We additionally evaluate lower-temperature and 16k settings that reflect prior reasoning-RL practice: DAS [47] uses a temperature of 0.6 with a 16k maximum sequence length, while DeepSeek-R1 [18] lowers its temperature to 0.7 in the second RL stage. According to Tab. 14, increasing the maximum response length makes the realized response distribution more long-tailed: at temperature 0.6, the average per-step maximum response nearly saturates the 8k response-length cap (8,157 tokens) and rises to 13,561 tokens under the 16k cap, while rollout-generation/full-step reductions increase from 27.2/19.0% to 40.1/34.0%. At a fixed 16k cap, lowering temperature from 1.0 to 0.6 raises α from 97.6% to 99.0% and increases the reductions from 24.3/18.6% to 40.1/34.0%, consistent with a prior study [41] showing that lower temperatures improve drafter acceptance rate and speedup.

NeurIPS Paper Checklist

1. **Claims**

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

   Answer: [Yes]

   Justification: The abstract and introduction state the paper's main contributions: a target-induced quantized self-drafter, a regime-aware SD toggle policy, and adaptive draft-length control for accelerating RL rollouts. These claims are supported by the method description in Sec. 4 and by the end-to-end results, ablations, and training-dynamics analysis in Sec. 5 and Secs. F.2 and F.4.

   Guidelines:

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

2. **Limitations**

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

   Answer: [Yes]

   Justification: Section 6 discusses the limitations of the proposed method and directions for future work, including the focus on chain verification, the limitations of naive RTN quantization for some model families, and regimes where weight quantization alone may be insufficient, such as extremely long-context or dense-attention settings.

   Guidelines:

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

3. **Theory assumptions and proofs**

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

   Answer: [N/A]

   Justification: The paper does not present theorem-style theoretical results requiring formal proofs. Instead, it uses analytical latency and roofline models to guide system design; the relevant assumptions, equations, calibration procedure, and empirical validation are provided in Secs. 3.1, 3.2 and 4.2 to D.3.

   Guidelines:

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

4. **Experimental result reproducibility**

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

   Answer: [Yes]

   Justification: The paper discloses the main information needed to reproduce the experimental results, including the implementation stack, hardware, models, datasets, RL algorithm, rollout configuration, training hyperparameters, tensor-parallel setting, toggle-policy parameters, adaptive draft-length parameters, and reporting window. These details are provided in Sec. 5.1 and Secs. D and E.

   Guidelines:

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

(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

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

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

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

   Answer: [Yes]

   Justification: The public repository linked in the abstract provides environment setup, data preparation, calibration, and training scripts for the primary A100 evaluations of EfficientRollout, No-SD, and the quantized self-SD variants. The release focuses on the core EfficientRollout implementation; the Spec-RL integration, learned-auxiliary drafter training, and NeMo RL comparison scripts are not included, with their designs and experimental settings described in Secs. E.4, E.5 and F.7.

   Guidelines:

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

6. **Experimental setting/details**

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

   Answer: [Yes]

   Justification: Section 5.1 and Sec. E specify the experimental settings, including model choices, SimpleRL dataset variants, GRPO configuration, train batch size, group size, rollout temperature, maximum response length, learning rate, KL coefficient, data-parallel and tensor-parallel settings, EfficientRollout hyperparameters, and baseline configurations.

   Guidelines:

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

7. **Experiment statistical significance**

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

   Answer: [Yes]

   Justification: For the primary Qwen2.5-7B comparison, Sec. F.2 reports latency reductions and final validation accuracy as mean±SD across three paired data-order seeds. The remaining evaluations report averages and per-step trajectories without repeated-run error bars.

   Guidelines:

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

8. **Experiments compute resources**

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

   Answer: [Yes]

   Justification: The paper reports the compute setup used for the experiments, including a single node with $8\times$ NVIDIA A100-SXM4-80GB GPUs for the primary evaluation, the H100 and tensor-parallel configurations for additional evaluations, and the software stack based on veRL, vLLM, Ray, and FSDP. The latency measurements and Sec. E and Secs. F.7 and F.8 provide additional details needed to estimate reproduction cost.

   Guidelines:

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

9. **Code of ethics**

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

   Answer: [Yes]

Justification: The research follows standard experimental practice for LLM post-training systems, uses public or cited models, datasets, and software frameworks, and does not involve human-subject experiments, private data collection, or unsafe data release.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

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

Answer: [No]

Justification: The paper focuses on the systems and training-efficiency aspects of RL rollout acceleration and does not include a separate broader-impact discussion. The societal implications are indirect and depend on the underlying models, datasets, and deployment contexts.

Guidelines:

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

11. **Safeguards**

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

Answer: [N/A]

Justification: The paper does not release a new pretrained language model, image generator, scraped dataset, or other high-risk asset requiring special safeguards. The released artifact is a training-system implementation for rollout acceleration.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.

- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

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

    Answer: [Yes]

    Justification: The paper cites the assets used, and the code release preserves upstream license notices: the evaluated Qwen models, Qwen EAGLE3 and yuhuili checkpoints, and SimpleRL/DAPO/ShareGPT releases declare Apache-2.0, subject to applicable base-model and third-party terms; Llama3.1 and the RedHatAI Llama drafter use the Llama 3.1 Community License; MATH is MIT-licensed; and the software stacks use Apache-2.0 and BSD-style licenses. The FastRL project code is Apache-2.0.

    Guidelines:

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

13. **New assets**

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

    Answer: [Yes]

    Justification: The paper introduces and releases the EfficientRollout implementation as a new software artifact. The public code repository includes documentation, reproduction scripts, configurations for No-SD and quantized self-SD variants, and an Apache-2.0 license for the released code.

    Guidelines:

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

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

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

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing, user studies, or research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

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

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

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or human-subject research, so IRB approval or equivalent review is not applicable.

    Guidelines:

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

16. **Declaration of LLM usage**

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

    Answer: [N/A]

    Justification: LLM assistance in preparing this work was limited to writing, editing, or formatting and did not affect the scientific methodology, rigor, or originality of the work; therefore, declaration is not required under the checklist policy.

    Guidelines:

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