ReSpec: Towards Optimizing Speculative Decoding in Reinforcement Learning Systems

Qiaoling Chen [1 2] **Zijun Liu** [2 3] **Peng Sun** [2] **Shenggui Li** [1] **Guoteng Wang** [2] **Ziming Liu** [2 4]
Yonggang Wen [1] **Siyuan Feng** [2 5] **Tianwei Zhang** [1]

Abstract

Adapting large language models (LLMs) via reinforcement learning (RL) is often bottlenecked by the generation stage, which can consume over 75% of the training time. Speculative decoding (SD) accelerates autoregressive generation in serving systems, but its behavior under RL training remains largely unexplored. We identify **three critical gaps** that hinder the naïve integration of SD into RL systems: diminishing speedups at large batch sizes, drafter staleness under continual actor updates, and drafter-induced policy degradation.

To address these gaps, we present ReSpec, a system that adapts SD to RL through three complementary mechanisms: dynamically tuning SD configurations, evolving the drafter via knowledge distillation, and weighting updates by rollout rewards. On Qwen models (3B–14B), ReSpec achieves up to $4.5\times$ speedup while preserving reward convergence and training stability, providing a practical solution for efficient RL-based LLM adaptation.

1 Introduction

Reinforcement learning (RL) has become a practical route to adapt large language models (LLMs) for complex, high-level objectives, such as improved alignment, reasoning, and domain-specific skills (Yang et al., 2025; Comanici et al., 2025; Team, 2023; 2025). An RL training iteration typically consists of three stages: *generation*, *inference*, and *training*. In the generation stage, the actor model produces rollout trajectories for a batch of prompts via autoregressive decoding; the inference stage evaluates these trajectories using auxiliary models such as reward or critic networks; the training stage consumes the resulting signals to update the actor parameters. Among these stages, **generation is consistently the dominant bottleneck** (Zhong et al., 2025a). As shown in Table 1, for LLMs trained with a maximum response length of 8K tokens, generation accounts for up to 86% and 75% of the wall-clock iteration time in math and code models, respectively.

This bottleneck is further exacerbated by the algorithms widely used in practice (Shao et al., 2024; Yang et al., 2024). Classic policy optimization methods such as PPO (Schulman et al., 2017; 2015) combine trajectory-level rewards with critic-based feedback to stabilize updates, while more recent group-based approaches, e.g., GRPO (Shao et al., 2024) and DAPO (Yu et al., 2025), generate multiple candidate completions per prompt and compute updates from relative comparisons inside each group. Although group sampling improves exploration and gradient quality, it also multiplies the number of decoded tokens per prompt, further inflating the cost of the generation stage.

A natural optimization to address this bottleneck is *speculative decoding* (SD) (Leviathan et al., 2023; Chen et al., 2023; Miao et al., 2025). Standard auto-regressive decoding requires one expensive forward pass of the target model per token. SD reduces this cost by introducing a lightweight *drafter* that proposes short token sequences, which are then validated by the target model in parallel. If the drafter and target align closely, many drafted tokens can be accepted at once, thereby reducing the number of target forwards per generated token. SD has already been widely adopted in LLM serving systems (e.g., SGLang (Zheng et al., 2024), TensorRT-LLM (NVDIA, 2024)) and commercial platforms (e.g., OpenAI (OpenAI, 2024), AWS SageMaker (AWS, 2024)). Among various SD variants, **EAGLE-3** (Li et al., 2025) represents the current state of the art, achieving the highest reported speedup among lossless SD methods (Cai et al., 2024; Fu et al., 2024; Ankner et al., 2024). Accordingly, we adopt EAGLE-3 as the default SD algorithm in our analysis, implementation, and experiments.

Research Gaps. Despite the widespread success of SD in serving systems, its behavior under RL training has never been systematically examined. Through our analysis of

[1]Nanyang Technological University, Singapore [2]Shanghai Qiji Zhifeng Co., Ltd., Shanghai, China [3]Tsinghua University, Beijing, China [4]National University of Singapore, Singapore [5]Shanghai Innovation Institute, Shanghai, China. Correspondence to: Tianwei Zhang <tianwei.zhang@ntu.edu.sg>.

Proceedings of the 9^{th} MLSys Conference, Bellevue, WA, USA, 2026.

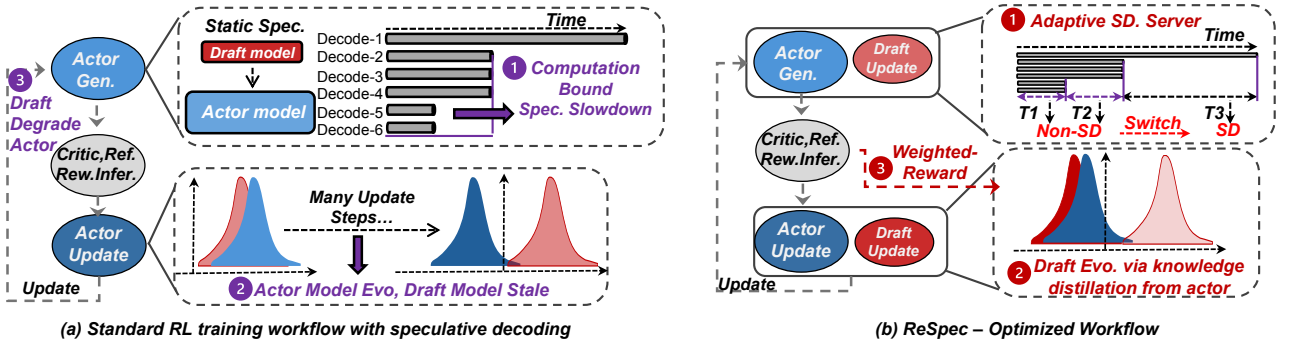

Figure 1. Overview of SD in RL training and our proposed ReSpec system. (a) The RL training workflow integrated with SD, highlighting three fundamental **gaps**. (b) The design of **ReSpec**, which addresses these gaps through three complementary mechanisms: ❶ dynamically tuning SD configurations to match workload conditions, ❷ continuously aligning the drafter with the evolving actor using on-policy signals, and ❸ weighting drafter updates by rollout quality to maintain stable policy learning. Together, these mechanisms enable stable and efficient SD throughout RL training.

RL workflows and the prototype integration of SD into the generation stage, we identify three fundamental gaps that explain why naïve application of SD fails to deliver consistent gains in RL training, as illustrated in Figure 1(a). (G1) *Diminishing speedups at large decoding batch sizes.* When generation already runs at high GPU utilization with large batch sizes, the marginal parallelism provided by SD becomes limited and can even be offset by drafting and synchronization overheads (Yan et al., 2024). Consequently, a single static SD configuration cannot provide reliable speedups across diverse RL workloads, as shown in Figure 3. (G2) *Drafter staleness under continual actor updates.* As the actor (i.e., target) model evolves with each policy update, a fixed drafter rapidly becomes misaligned with the actor distribution, leading to the loss of acceleration benefits, as observed in Figure 4. (G3) *Practical optimization degradation under SD.* Although lossless SD preserves the target distribution in expectation, we observe that in practice, the interaction of non-deterministic verification paths, drafter staleness (G2), and variance amplification under non-stationary policy updates degrades the effective training signal. These compounding factors reduce trajectory quality and diversity, leading to measurable reward degradation during RL training, as demonstrated in Figure 5.

Opportunities. Beyond these gaps, we identify two key opportunities in the RL generation stage that can be leveraged to make SD both stable and efficient. First, the *generation stage exhibits strong skewness*: within a batch, most sequences terminate early while a small fraction continue much longer, resulting in a time-varying active batch size. Because the efficiency of SD depends critically on the batch size, this skewness offers an opportunity to adapt SD configurations dynamically rather than relying on static hyperparameters, as shown in Figures 6 and 7. Second, RL rollouts inherently produce rich, *on-policy diagnostic signals* that are routinely logged but never reused, including per-step target, draft logits, and rewards. These signals provide a valuable, untapped supervision source for maintaining drafter alignment with the evolving actor policy, enabling continuous adaptation without introducing additional overhead to the sampling process.

Motivated by the above-mentioned gaps and opportunities, we introduce **ReSpec**, a system that adapts speculative decoding to the unique demands of RL training. ReSpec provides three targeted mechanisms, each directly addressing one of the gaps (Figure 1 (b)). Specifically, (1) for G1, we design *Adaptive Speculative Decoding Server*. It dynamically selects and applies speculative-decoding configurations based on lightweight profiling and runtime workload signals. (2) For G2, we introduce *Drafter Evolution via Knowledge Distillation* with a *Online Learner*. To keep the drafter aligned with the continually updated actor, we evolve the drafter using on-policy distillation from the actor. This distillation pipeline continuously transfers the actor's distributional knowledge into the lightweight drafter so that drafted proposals remain relevant as the policy changes. (3) For G3, we design *Reward-Weighted Adaptation*. To mitigate the practical optimization degradation described above, we weight drafter updates by rollout quality, which prioritizes high-reward trajectories during adaptation. Since the actor is optimized toward high reward, these trajectories are most representative of the actor's evolving distribution, enabling the drafter to track the post-update policy more accurately and reducing the risk of degraded training signals.

Our contributions are summarized as follows:

- ★ We conduct the first systematic study of SD in end-to-end RL training of LLMs. Our analysis reveals three critical gaps, including *diminishing speedups*, *drafter staleness*, and *practical optimization degradation*, that explain why naïve SD integration fails to provide consistent acceleration.

	Generation	Inference	Training
Math	83%–86%	1.7%–2.4%	12.3%–14.6%
Code	70.9%–75.5%	11.4%–14%	13.1%–15.1%

Table 1. Time breakdown during RL iterations for 7B models (max response length 8K tokens).

- ★ We identify two underexplored opportunities in RL generation: (1) *skewed generation workloads* that enable dynamic adaptation of SD configurations, and (2) *on-policy diagnostic signals* that can be repurposed as supervision for continuous drafter alignment.
- ★ We design **ReSpec**, the first system that adapts SD to RL training. ReSpec comprises two tightly coupled components: an *Adaptive SD Server* that performs profiling-guided and runtime-tuned SD scheduling, and an *Online Learner*, which continuously maintains drafter alignment through three complementary mechanisms: evolving the drafter via knowledge distillation from the actor, weighting updates by rollout rewards to preserve policy quality, and performing asynchronous, replay-buffered updates to avoid blocking generation.
- ★ We implement and evaluate ReSpec on Qwen models (3B–14B), showing that it preserves training stability and reward convergence while achieving up to $4.5\times$ speedup over standard RL training, demonstrating its practicality for RL training pipelines.

2 Background and Related Work

2.1 Reinforcement Learning for LLMs

Workflow. RL for LLMs typically proceeds in iterations that consist of three stages: generation, inference, and training. In the *generation* stage, an actor model produces rollout trajectories for a batch of prompts. This stage contains a prefill (prompt encoding) step followed by autoregressive decoding of tokens until termination. In the *inference* stage, auxiliary models (reference, reward, critic) evaluate the generated rollouts by performing forward passes to produce scalar signals or logits used by the update. Finally, in the *training* stage, the actor consumes these signals to compute losses and apply parameter updates.

Algorithms. Classic policy optimization methods such as PPO (Schulman et al., 2017; 2015) combine trajectory-level rewards with critic-based, action-level feedback to reduce the variance of updates. Recent production-oriented recipes, e.g., GRPO (Shao et al., 2024), DAPO (Yu et al., 2025), leverage *group-based* sampling: for each prompt, the actor generates a group (many candidate completions) and updates are computed from relative comparisons inside the group. Group sampling improves the exploration and gradient signal in model optimization but increases the number of decoded tokens per prompt, amplifying the cost of the generation stage and motivating efforts to accelerate decoding without altering the sampling distribution.

Bottleneck. The generation stage consistently dominates the end-to-end RL iteration time. As shown in Table 1, for LLMs trained with a maximum response length of 8K tokens, generation accounts for up to 86% and 75% of the total time for math and code models, respectively. This imbalance highlights that decoding, rather than model updating or reward evaluation, is the primary performance bottleneck in practical RL pipelines.

Existing Frameworks. Numerous frameworks aim to accelerate RL post-training for LLMs. Early systems (Harper et al., 2025; Hu et al., 2024; Lei et al., 2024; Yao et al., 2023) focus on orchestrating multi-stage RL workflows, while later efforts introduce architectural and scheduling optimizations—e.g., hybrid or multi-controller designs (Sheng et al., 2025; Wang et al., 2025), fused or parallelized stages (Zhong et al., 2025b; Mei et al., 2024), and asynchronous post-training for long-tail rollouts (Fu et al., 2025; Gao et al., 2025; He et al., 2025). However, these frameworks largely overlook the generation stage and its optimization opportunities. In contrast, ReSpec explicitly targets this bottleneck by integrating SD into RL training.

Concurrent Work on SD for RL Training. Several concurrent efforts have explored accelerating RL rollout generation via speculative decoding. Beat-the-Long-Tail (Shao et al., 2025) proposes distribution-aware speculative decoding that targets long-tail rollout sequences; SPEC-RL (Liu et al., 2025) accelerates on-policy RL via speculative rollouts; and RhymeRL (He et al., 2025) reuses historical trajectories as draft proposals. These approaches primarily rely on non-parametric or history-based drafting mechanisms. In contrast, ReSpec maintains a *parametric* drafter that is continuously aligned with the non-stationary RL actor through reward-weighted knowledge distillation. FastGRPO (Zhang et al., 2025) focuses on GRPO-specific optimization settings with concurrency-aware speculative decoding and online draft learning, but does not address reward-weighted distillation to prevent policy degradation, nor adaptive SD configuration tuning based on runtime workload dynamics. ReSpec is complementary to these approaches and addresses a broader set of challenges through its integrated Adaptive Server and Online Learner design.

2.2 Speculative Decoding

Standard autoregressive decoding requires one target-model forward per token, which is computationally expensive. Speculative decoding (SD) mitigates this by using a lightweight *drafter* to propose token sequences, which are then verified by the expensive *target* model in batches(Leviathan et al., 2023; Chen et al., 2023). Subsequent works extend this paradigm along multiple dimen-

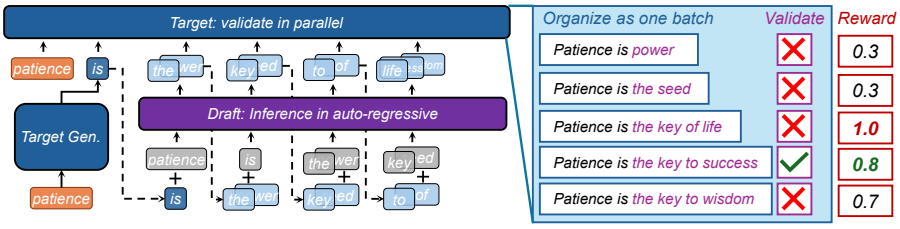

Figure 2. EAGLE-3 workflow. The target model generates one token, while the draft model produces multiple candidates using hidden states. The target verifies all candidates in a single forward pass and accepts four tokens ("the key to success") with only two forward passes.

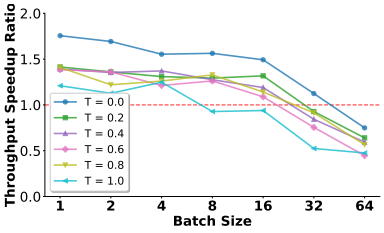

Figure 3. Speedup ratio under different batch sizes and temperature on H100 and Qwen2.5-7B-instruct for the MT-Bench dataset.

sions: Medusa (Cai et al., 2024) increases proposal parallelism, Cascade Speculative Drafting (Chen et al., 2024) introduces multi-stage chaining, and EAGLE-series methods (Li et al., 2024a;b; 2025) progressively enhance feature utilization, draft-tree construction, and parallelization, achieving state-of-the-art inference efficiency.

Speculative decoding procedure. We adopt a representative speculative sampling procedure similar to the recent SOTA method EAGLE-3 (Li et al., 2025) as our reference implementation. Starting from a realized prefix $T_{1:j}$, SD repeats a two-stage cycle. (1) *Drafting:* the drafter autoregressively proposes a block $\hat{T}_{j+1:j+k} = (\hat{t}_{j+1}, \ldots, \hat{t}_{j+k})$, where $\hat{t}$ represents individual tokens; (2) *Validation:* the target evaluates the proposed block in a batched forward and accepts drafted tokens sequentially according to an acceptance test; if a token is rejected, it is replaced by sampling from a residual distribution derived from the target logits.

Acceptance rule (single-token). For a single drafted token $\hat{t}$ with drafter probability $q(\hat{t})$ and target probability $p(\hat{t})$, the acceptance probability is (omitting conditions on prefixes):

$$\Pr\left[\text{accept } \hat{t}\right] \;=\; \min\!\left(1,\; \frac{p(\hat{t})}{q(\hat{t})}\right). \tag{1}$$

If rejected, a replacement token is sampled from the residual distribution:

$$r(x) = \frac{\max(0,\, p(x) - q(x))}{\sum_y \max(0,\, p(y) - q(y))}, \tag{2}$$

where x, y denote single tokens. It guarantees that the marginal sample follows p (Leviathan et al., 2023). We emphasize that lossless speculative decoding with correct verification (i.e., rejection sampling) is *unbiased in expectation*: the rollout distribution is identical to that of the target model under standard autoregressive decoding. Our work does not claim that speculative decoding violates this theoretical guarantee.

Multi-token drafting and accumulated mismatch. When drafting multiple tokens sequentially, each proposed token depends on the previously accepted tokens. We calculate the variance of the acceptance probability $\frac{p}{q}$ for sampled sequences as:

$$\mathrm{Var}_{T\sim q}\left[\prod_{t\in T}\frac{p(t)}{q(t)}\right] = \prod_{t\in T}\left(1 + D_{\chi^2}(p(t)||q(t))\right) - 1, \tag{3}$$

where D_{χ^2} is the Chi-squared divergence. If $D_{\chi^2}(p(t)||q(t))$ is approximated by a constant δ, the variance $(1+\delta)^{|T|} - 1$ increases exponentially with the sequence length. As the target model distribution evolves, this divergence typically widens, amplifying the variance of acceptance probabilities: some tokens become increasingly unlikely to be drafted ($\frac{p(t)}{q(t)} \uparrow$), while others are drafted often but rarely accepted ($\frac{p(t)}{q(t)} \downarrow$).

Acceptance length. A key practical metric in SD is the *acceptance length*, defined as the number of consecutive tokens in a drafted block that are accepted by the target model before the first rejection. Longer acceptance lengths generally lead to higher speedups because more tokens can be validated per forward step, but also increase the likelihood of multi-token distribution mismatch, as described above.

Speculative decoding cost model. Let C_p denote the average per-token cost of the **target model** under naïve decoding (i.e., one forward pass per token), and C_q denote the average per-token cost of the **drafter model**, with $C_q \ll C_p$. Suppose we draft blocks of k tokens, where the average acceptance rate per token is r, i.e., the expected fraction of drafted tokens ultimately accepted. If validating a block of k tokens requires target-model computation roughly proportional to k, but can be parallelized with an efficiency factor α, then the effective target cost for validating the entire block is approximately kC_p/α. Under these assumptions, the expected cost per accepted token in SD is estimated as:

$$\mathrm{Cost}_{\mathrm{SD}} \approx \frac{kC_q + kC_p/\alpha}{rk} \;=\; \frac{C_q + C_p/\alpha}{r}. \tag{4}$$

3 Challenges of Applying SD in RL

Applying SD inside RL training is attractive because the generation stage frequently dominates the iteration time

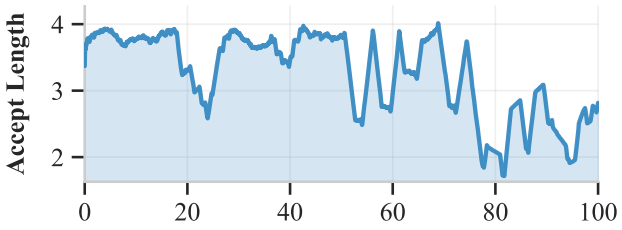

Figure 4. Acceptance length of the draft model during 100 RL steps with Qwen2.5-7B-instruct model and math dataset. As training progresses, the EAGLE-3 drafter quickly becomes stale and its acceptance length drops.

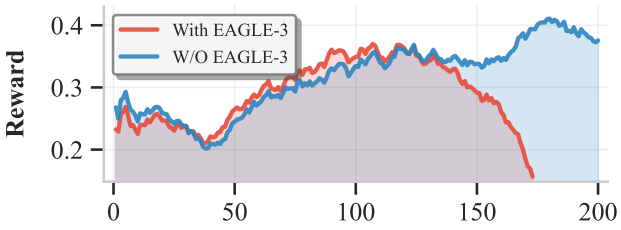

Figure 5. Evolution of rewards over 200 RL steps for Qwen2.5-7B on the math dataset. Naïve application of EAGLE-3 leads to a measurable drop in reward, illustrating practical optimization degradation when SD interacts with on-policy RL training dynamics.

(Table 1). However, prior work on online SD has focused on inference-serving scenarios, where draft models are updated online to track shifting request distributions (Zhou et al., 2023; Liu et al., 2023). In contrast, RL training requires the drafter to adapt to the evolving actor policy itself, posing three practical gaps not addressed by existing methods.

GAP 1: diminishing speedup at large batch sizes. In RL training, we enlarge the batch size to fit GPU utilization in the decoding phase. However, when the batch size in decoding is already large, GPUs are typically operating near high utilization through straightforward batching of independent sequences. Therefore, the extra parallelism that SD provides yields little marginal benefit. Besides, SD introduces additional overheads, as illustrated in Equation 4. This added draft cost and verification synchronization offset, and even exceeded the speedup. As illustrated in Figure 3, speedups from SD shrink as batch size increases.

GAP 2: SD staleness during RL training. In RL training, the actor parameters are continually updated. A drafter distilled from an earlier snapshot may become misaligned with the evolving actor, leading to lower acceptance length and thus diminishing the benefits of SD. Figure 4 illustrates this effect: the acceptance length of the EAGLE-3 drafter decreases as training advances.

GAP 3: practical optimization degradation under SD. As established in Section 2.2, lossless speculative decoding with correct rejection-sampling verification preserves the target model's rollout distribution in expectation. However, we observe that in practice, naively applying SD during on-policy RL training leads to measurable reward degradation. Importantly, this degradation does *not* arise from a violation of the theoretical sampling guarantee, but rather from the interaction of several practical factors unique to the RL training loop.

First, *non-deterministic verification paths*: during the verification stage, the forward pass operates as a chunked prefill with masked tree attention. Differences in GPU kernel implementations (e.g., split-K strategies in GeMM, split-KV in attention with varying atomic merging) can introduce

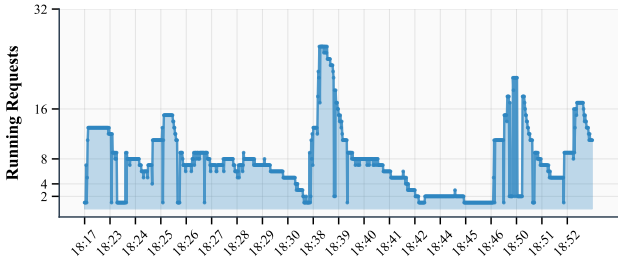

Figure 6. Number of active sequences during decoding in Qwen2.5-7B RL training (Math). It illustrates the skewed generation workload: most sequences finish quickly while a few persist, causing fluctuations in effective batch size.

numerical non-determinism. While individually negligible, these discrepancies accumulate over long rollouts and across many training iterations. Second, *drafter staleness* (compounding Gap 2): as the actor evolves, the drafter's proposals become increasingly misaligned. Although each individual verification step remains correct, the stale drafter constrains the *effective exploration* of the policy by repeatedly proposing similar continuations, reducing the diversity of collected trajectories. Third, *variance amplification under non-stationary updates*: as shown in Eq. (3), the variance of acceptance probabilities grows exponentially with draft length. Under continual policy updates, this variance disproportionately affects trajectory quality, producing a skewed effective training signal for policy optimization even though the expected distribution remains unbiased. Together, these practical effects influence the quality and diversity of the training signal rather than introducing formal sampling bias.

Empirically, we observe that naively applying EAGLE-3 in our RL runs produces a measurable drop in reward after approximately 100 update steps (Figure 5), consistent with these compounding practical factors degrading the optimization dynamics.

4 KEY INSIGHTS

4.1 Generation-stage Skewness and Its Consequences for Speculative Decoding

We observe that the generation stage in RL training exhibits pronounced *skewness*: within a single decoding batch, most sequences finish quickly while a small fraction continue for much longer. Figure 6 shows this pattern concretely for Qwen-2.5–7B: the number of active requests fluctuates heavily, occasionally spiking above 16, spending much of the run near 8, and then decaying to a long tail near 1. Because autoregressive decoding is sequential, these short-lived sequences drop out early, and the active batch size decreases as decoding proceeds, producing a highly non-uniform workload and intermittent under-utilization of compute.

Crucially, we find that this time-varying active batch size substantially changes the cost–benefit trade-offs of SD. SD performance is governed by three key hyperparameters: (1) the number of speculative rounds s, (2) the target branching factor t, i.e., number of candidate branches validated in parallel, and (3) the draft length per round n. Importantly, our measurements reveal that the optimal SD configuration is not fixed, but depends strongly on the *active batch size*. When the batch size is small, aggressive drafting (large s, n, t) can yield clear gains, whereas at larger batch sizes the same configuration may even hurt performance due to overheads outweighing the benefits. For instance, in our experiments (shown in Figure 7), the best SD configuration at batch size 2 achieves $1.46\times$ speedup, while the same setting degrades to $0.76\times$ at batch size 32.

This coupling highlights that static SD hyperparameters cannot deliver consistent benefits across the dynamic regime of RL training. Instead, adaptive strategies that *tune SD parameters online based on the current active batch size* are required to fully realize the potential acceleration.

4.2 On-policy Signals and Knowledge Distillation for Drafter Alignment

Knowledge distillation provides a principled way to align the predictive distribution of a smaller *student* model with that of a larger *teacher*. Prior studies (Zhou et al., 2023; Liu et al., 2023) have shown that this framework is particularly effective for SD in online serving, where incoming request distributions shift over time and the draft model must continually adapt. In these settings, distillation is applied online to fit the varying data distribution.

In contrast, SD within RL training presents a fundamentally different challenge. The request distribution is relatively stable, but the *target model itself* evolves during training. This dynamic teacher–student relationship means that static draft models quickly become misaligned, leading to reduced acceptance rates and diminished decoding efficiency.

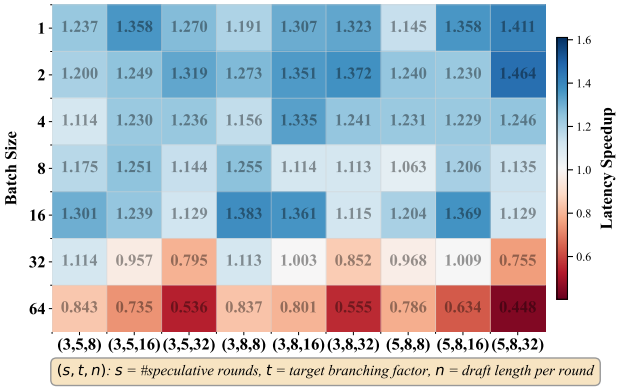

Figure 7. Latency speedup of EAGLE-3 (temperature = 0.6) on Qwen2.5-7B-instruct for the MTBench dataset. Each cell shows speedup for a specific (s, t, n) configuration, showing that optimal SD depends strongly on active batch size.

Fortunately, we observe an underexploited opportunity in the verification step of SD: during validation, the system has access to *dense, on-policy diagnostic signals*, such as the target model's per-step logits, drafter's contemporaneous log-probabilities, and scalar trajectory rewards. These signals are produced naturally during generation and reflect the actor's current behavior under the actual sampling policy, which enables the construction of distillation objectives that directly align the draft model with the current target model, effectively tracking its evolving distribution. By exploiting such information, we can continuously refine the draft model to increase accept length and stabilize RL training.

5 SYSTEM DESIGN

5.1 System Overview

Inspired by the above insights, we present ReSpec, a novel system that efficiently adapts SD to RL. Figure 8 presents its architecture, which consists of two major components:

(1) The **Adaptive Server** accelerates the generation stage with SD. It integrates two modules: t he *Solver*, which searches for and selects the optimal SD hyperparameter combinations, and the *Scheduler*, which monitors the active batch size at runtime and dynamically switches SD configurations while managing the KV cache.

(2) The **Online Learner** maintains the draft model through online training. It incorporates three modules: The *Reward-Weighted Knowledge Distillation algorithm*, which distills knowledge from the evolving target model to keep the draft model aligned and ensure high acceptance length across training steps, and the *Async Update Overlap* mechanism, which overlaps draft model training with RL generation.

Workflow. ❶ ReSpec starts with a user-specified configuration, such as batch size and temperature. The Solver

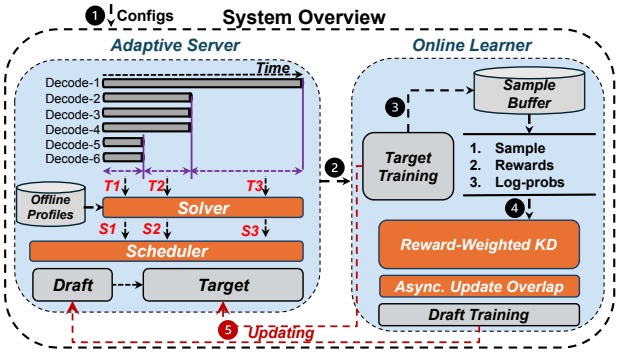

Figure 8. Workflow of ReSpec during RL training. The system iteratively generates responses, stores rollout samples with associated rewards, updates the drafter via reward-weighted knowledge distillation with asynchronous overlap, and applies improved speculative decoding in the next RL step to accelerate and stabilize training.

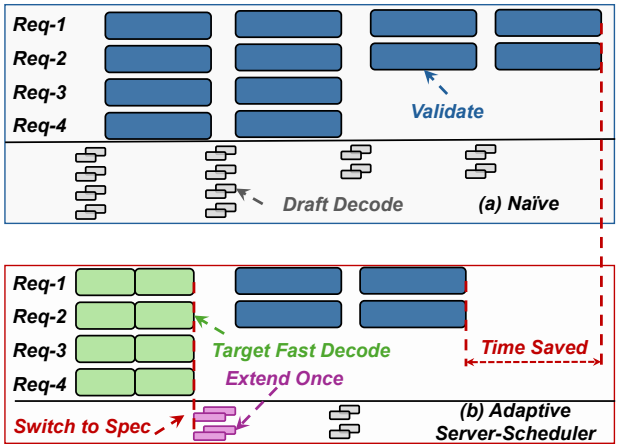

Figure 9. Workflow comparison between standard speculative decoding (a) and the proposed Adaptive Server (b).

provides optimal SD parameters (s, t, n) for the given decoding conditions, and the Scheduler adjusts them online without additional overhead. ❷ The target model generates responses and updates its weights following the standard RL training loop. ❸ Generated samples, along with log-probabilities and rewards, are stored in the Sample Buffer. ❹ The Online Learner selectively draws samples to train the draft model using the reward-weighted KD algorithm, with Async Update Overlap strategies to maximize efficiency. ❺ Updated draft weights are pushed back to the inference engine, enabling the next RL training step to proceed with improved speculative decoding.

5.2 Adaptive Server

SD accelerates generation by drafting multiple candidate tokens in parallel and verifying them against the target model. However, its efficiency depends heavily on runtime conditions such as the active batch size, which fluctuates significantly during RL training. When GPU utilization is already saturated (e.g., large active batch sizes), enabling SD can add redundant overhead; conversely, when batch sizes shrink, SD offers substantial speedup. To balance these dynamics, ReSpec introduces an *Adaptive Server* that dynamically switches between speculative and non-speculative execution modes according to the current system state.

Figure 9(a) shows the standard SD workflow without adaptation: for a batch size of 4, each request independently undergoes draft and validate stages, resulting in redundant compute when the GPU is already well utilized. In contrast, Figure 9(b) depicts the adaptive mode: when the batch size is large, the Scheduler opts for pure target decoding to maintain throughput; as the active batch size drops (e.g., from 4 to 2), the system transitions to the speculative state. During this transition, the Scheduler extends the previous non-speculative decoding state to build an updated KV cache, then resumes normal speculative decoding to maximize parallelism at lower utilization.

The Adaptive Server consists of two key components: (1) a **Solver** that searches for efficient SD configurations via offline profiling, and (2) a **Scheduler** that controls runtime switching between modes with minimal overhead.

Solver: Offline Profiling Guided Search. To determine efficient SD configurations, ReSpec employs a profiling-based solver. As shown in Figure 7, the runtime performance of SD is highly sensitive to the choice of hyperparameters, including the number of speculative rounds s, branching factor t, and number of drafted tokens per round n. The optimal configuration varies across different batch sizes due to the evolving active batch during RL training.

We first conduct an offline profiling stage, where we benchmark the execution time of draft and target models under various SD hyperparameter settings and batch sizes. The results are used to fit a predictive performance model that estimates the throughput speedup of different SD configurations as a function of the active batch size. This profiling is lightweight and performed once before RL training. At runtime, the solver consults the offline-derived performance model and the observed active batch size to dynamically select the SD configuration expected to maximize speedup.

Scheduler. The Scheduler is the runtime controller of the Adaptive Server. It determines when and how SD should be activated during generation, while minimizing system overhead. This is non-trivial in RL training, where active batch sizes shrink rapidly due to data skewness, and the benefits of SD vary across decoding stages. Switching between speculative and non-speculative execution incurs overhead. To achieve zero-overhead runtime switching, we model the

decoding phase as a two-state process: *spec-enabled* and *non-spec*. Each request carries a lightweight flag indicating whether SD is active. The Scheduler tracks the state of the current batch and applies lightweight transitions:

- **Non-spec → Spec.** When the Solver advises switching on SD, the Scheduler promotes the current decode batch into a spec-enabled batch by reusing the *prefill* interface. This allows the draft model to generate candidates without modifying the decoding kernel.
- **Spec → Non-spec.** If SD is no longer beneficial, the Scheduler simply discards speculative metadata (e.g., cached candidates) and continues with regular decoding.

The Scheduler operates in a closed loop with the Solver. At runtime, it monitors active batch size and uses these signals to decide whether SD should remain enabled.

5.3 Online Learner

Algorithm 1 summarizes the online learner workflow used throughout our experiments. During rollouts, we run SD with the current draft (q_θ) and verify each candidate with the target model (p). For every rollout, we store the input and response together with the target logits and the scalar reward. The online learner extracts distillation targets from the rollout buffer, accumulates them in a replay buffer (Q), and performs periodic reward-weighted updates every (I) iterations. These updates (i) keep the draft aligned with the target distribution to preserve high acceptance rates in SD, and (ii) bias the draft toward behaviors that empirically yield higher RL reward. The distillation data is derived directly from on-policy trajectories generated during RL training and requires no additional data collection. Concretely, for Qwen-3B, each drafter update uses 32 samples with approximately 64K tokens; for Qwen-7B and Qwen-14B, each update uses 28 samples, corresponding to about 223K and 112K tokens, respectively.

5.3.1 Reward-Weighted Knowledge Distillation

A central mismatch between RL and SD is that the target (p) evolves, whereas classical KD assumes a fixed teacher. Treating all collected rollouts equally (standard KD) can therefore pull the draft toward low-quality behavior because many rollouts are erroneous or low-reward.

We perform **reward-weighted KD**. For a rollout sample $((x, y))$ with per-step target distributions $(p(\cdot \mid x, y_{<t})$ and scalar reward (r), we minimize the sample-wise loss:

$$\mathcal{L}_{\mathrm{KD}}(x, y) \;=\; w(r) \sum_{t=1}^{T} \mathrm{KL}\big(p(\cdot \mid x, y_{<t}) \mid q_\theta(\cdot \mid x, y_{<t})\big).$$

where $(w(r) > 0)$ is a reward-based weighting function (by default $(w(r) = r)$; in practice we normalize and clip $(w(r))$ for stability). Using stored logits $(\log p)$, we materialize soft targets via $(\mathrm{softmax}(\log p))$ at update time and compute the weighted cross-entropy against the current draft (q_θ). Note that gradients are always computed with respect to the current draft (q_θ), not the recorded $(\log q_\theta)$.

Algorithm 1 RL-Integrated Speculative Draft Update

Input: Target model $(p(\cdot \mid x))$, draft model $(q_\theta(\cdot \mid x))$, rollout buffer $(\mathcal{B})$ storing tuples ((x, y, $\log p$, r)), update interval (I), replay buffer (Q).
Initialization: Pre-train (q_θ) with offline KD on warmup data; set $Q \leftarrow [], i \leftarrow 0$;
while *RL training not converged* **do**
  Sample a mini-batch $((x, y, \log p, r))$ from rollout buffer $(\mathcal{B})$
  Run speculative decoding with draft (q_θ) and verify outputs using target (p)
  Collect token-level alignment info $((\log q_\theta, \log p))$ for each position
  Store $((x, y, \log q_\theta, \log p, r))$ in replay buffer (Q)
  $i \leftarrow i + 1$
  if $i \bmod I = 0$ **then**
    $\mathcal{L}_{\mathrm{tot}} \leftarrow 0$ **for** *each sample* $(x, y, \log q_\theta, \log p, r) \in Q$ **do**
      Construct soft targets $p(\cdot \mid x, y_{<t}) \leftarrow \mathrm{softmax}(\log p)$
      Compute per-sample reward-weighted KD loss:
      $\mathcal{L}_{\mathrm{KD}}(x, y) = w(r) \sum_{t=1}^{T} \mathrm{KL}\big(\tilde{p}(\cdot \mid x, y_{<t}) \,\big\|\, q_\theta(\cdot \mid x, y_{<t})\big)$
      $\mathcal{L}_{\mathrm{tot}} \leftarrow \mathcal{L}_{\mathrm{tot}} + \mathcal{L}_{\mathrm{KD}}(x, y)$
    end
    Update draft parameters with accumulated loss:
    $\theta \;\leftarrow\; \theta - \eta \nabla_\theta \mathcal{L}_{\mathrm{tot}}$
    Reset replay buffer: $Q \leftarrow []$
  end
end

We compare three variants: (i) **No-reward KD**: standard KL on all stored samples with no weighting; (ii) **Eagle-only**: SD applied, but the draft is *never* updated; (iii) **Reward-weighted KD**. On Qwen2.5-7B both no-reward KD and eagle-only begin to degrade around step (≈ 125). The no-reward KD run collapses fastest (near-zero reward by (≈ 150)), while eagle-only degrades more slowly (near-zero by (≈ 175)). Reward-weighted KD maintains steadily increasing rewards over the same horizon (see Figure 10).

Mechanism. Naive KD treats all rollouts equally, so noisy or low-reward trajectories pull the draft away from the target's high-quality modes. Because SD uses the draft to accelerate rollouts, a biased draft increases the probability of generating further low-quality rollouts, creating a *positive feedback loop* that degrades policy quality. Reward-weighted KD mitigates this by attenuating the contribution of low-reward samples and amplifying high-reward ones, steering the draft toward behavior that matches the target and empirically produces high reward. We note that the goal of reward weighting is *not* to make the drafter itself perform the task well, but rather to prioritize trajectories that are most representative of the actor's post-update distribution. In on-policy RL, the actor is optimized to maximize reward; consequently, high-reward trajectories better reflect

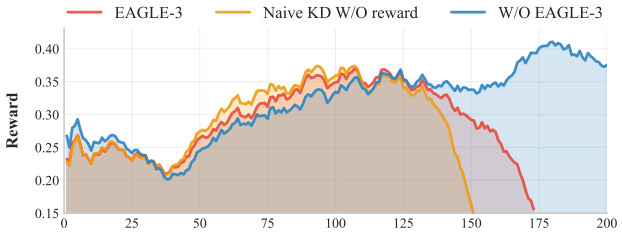

Figure 10. Qwen2.5-7B reward trajectories under three strategies: No-reward KD (standard KL), Eagle-only (no draft updates), and Reward-weighted KD (ours). Reward-weighted KD prevents the rapid collapse seen in baselines.

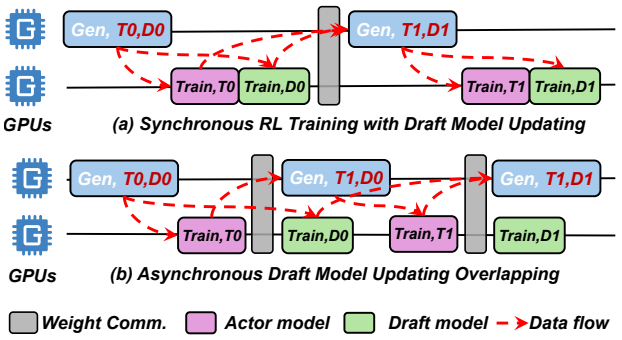

Figure 11. Asynchronous draft model updating overlapping: draft updates are scheduled to run during pipeline idle slots to avoid extra wall-clock overhead.

the direction in which the actor's distribution is moving. By focusing distillation on these trajectories, the drafter more accurately tracks the evolving target policy, leading to higher acceptance rates and more stable SD throughout training.

5.3.2 *Async Update Overlap*

Updating a draft model during RL training introduces unique system inefficiencies. In a naive design, draft updates are executed synchronously after each generation step, which creates pipeline bubbles: generation must wait for optimization to finish, leading to under-utilized hardware, as illustrated in Figure 11 (a). Moreover, as the draft is smaller than the target, training it at the same frequency and scale as the target model is both computationally heavy and algorithmically unnecessary, given the scaling-law mismatch.

To address these challenges, we design an *asynchronous update overlap* mechanism. The core idea is twofold: (1) We maintain a replay buffer that accumulates distillation targets over multiple rollouts. The draft model is then updated every I iterations using only $1/I$ of the buffer data, which amortizes optimization cost and reduces the burden of frequent updates while keeping the draft approximately aligned with the target. (2) We overlap draft model updates with the generation stage by exploiting idle slots in the RL pipeline. Specifically, draft training runs in parallel with target rollouts during these bubbles, ensuring that updates incur negligible additional wall-clock latency, as shown in Figure 11 (b).

This design balances system efficiency and algorithmic freshness: by avoiding blocking synchronization while still providing frequent enough updates, the draft model can continuously track the target's evolving distribution without slowing down overall training.

6 Evaluation

To systematically evaluate ReSpec, we structure our experiments around the following research questions:

Model	Gen GPU	Train GPU	Temp.	Seq Len
Qwen2.5-3B	1	1	0.6	3072
Qwen2.5-7B	8	8	0.6	8192
Qwen2.5-14B	8	8	0.4	8192

Table 2. Configurations for various models.

- RQ1: Can ReSpec maintain stable training and reward convergence while applying SD in RL? (§ 6.1)
- RQ2: How much end-to-end speedup does ReSpec provide, and how does it scale with model size? (§ 6.1)
- RQ3: What are the contributions of each component of ReSpec to training acceleration and stability? (§ 6.2)
- RQ4: How does asynchronous update frequency affect drafter alignment and policy performance? (§ 6.3)

Implementation. We build ReSpec on top of VeRL (Sheng et al., 2025) and SGLang (Zheng et al., 2024). The implementation consists of roughly 2K lines of code (LOC), with about 500 LOC devoted to the Adaptive Server and 1500 LOC for the Online Learner. To maximize the efficiency of draft model training, we adopt tailored parallelization strategies between the actor and draft models. In addition, we implement an optimized EALGE-3 training backend to support training draft models for diverse target architectures.

Testbed. All experiments are conducted on two compute nodes, each equipped with 8× NVIDIA H100 GPUs (80GB) interconnected with 900 GB/s NVLink, and connected across nodes via 8× 400 Gbps RoCE. The software stack includes PyTorch 2.7.1, Python 3.12, and CUDA 12.6.

Models and Algorithms. We conduct a thorough evaluation on Qwen-2.5 models ranging from 3B to 14B parameters, trained on real-world math datasets. Detailed configurations are summarized in Table 2. Training is based on the GRPO algorithm (Shao et al., 2024), a widely adopted RL approach for LLM post-training that underpins systems such as DeepSeek-R1 (Guo et al., 2025).

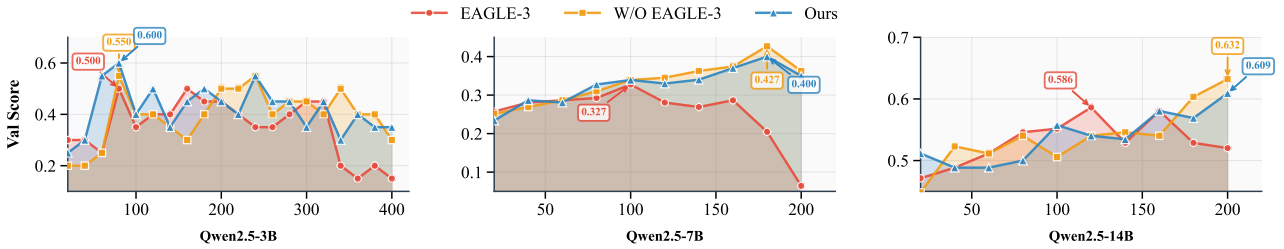

Figure 12. Validation score comparison across Qwen-2.5 models (3B–14B). ReSpec closely tracks the baseline without acceleration, while direct EAGLE-3 integration destabilizes training and leads to significant accuracy degradation.

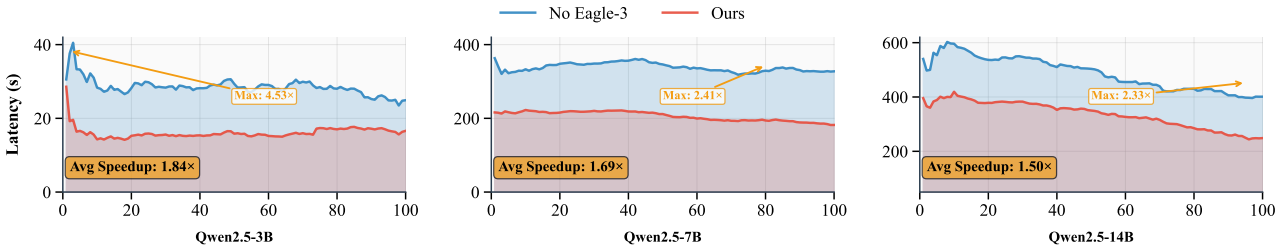

Figure 13. End-to-end training latency speedup. The curves are smoothed using a moving average with a window of 20 to reduce noise. ReSpec achieves consistent $1.5\times$–$4.5\times$ acceleration across models while preserving convergence stability, unlike EAGLE-3 which fails to sustain training.

Metrics. We measure training efficiency via end-to-end wall-clock latency. To assess training performance, we report validation scores every 20 steps and track the mean reward throughout the RL updates.

6.1 End-to-end Evaluation

We evaluate our method in an end-to-end RL setting, focusing on two key aspects: (i) training stability, measured by validation score during the RL updates, and (ii) efficiency, measured by wall-clock speedup relative to the baseline training without acceleration.

In terms of accuracy, we observe that ReSpec's Online Learner maintains stable validation scores comparable to the baseline without speculative acceleration, as shown in Figure 12. Across models of different scales, our method avoids the collapse when directly applying EAGLE-3 in RL training. For instance, in Qwen-3B, our validation score trajectory remains aligned with the no-acceleration baseline even after 400 steps, while EAGLE-3 often causes degradation, e.g., dropping to 0.15 after step 400. A similar pattern is found in Qwen-7B, where our method sustains higher scores (0.4 at steps 160-180) compared to EAGLE-3's early collapse (0.06–0.2). For larger models like Qwen-14B, ReSpec consistently tracks the baseline trends, whereas EAGLE-3 exhibits divergence after longer horizons, with unstable or declining validation scores. These results confirm that our method preserves convergence guarantees while still benefiting from SD.

In terms of efficiency, ReSpec achieves substantial wall-clock acceleration as illustrated in Figure 13. We report both *maximum* and *average* end-to-end speedups to distinguish peak acceleration (typically observed during low-batch-size phases) from sustained gains over the full training run. For Qwen-3B, our approach yields up to $4.53\times$ maximum end-to-end training speedup, with an average improvement of around $1.84\times$, particularly in the early generation stages where speculative execution is most effective. For larger models, the gains remain substantial yet more stable: $1.69\times$ on average for Qwen-7B (peaking at $2.41\times$), and $1.50\times$ average for Qwen-14B (up to $2.60\times$ maximum). We also observe that speculative acceleration is most pronounced during the early generation stages, where token-level parallelism is maximally exploitable.

Taken together, these findings highlight that ReSpec delivers the desired balance: it matches baseline RL training in accuracy and reward stability while offering consistent end-to-end speedups across different model sizes. This makes our method suitable for practical RL training pipelines, where both stability and throughput are critical.

6.2 Speedup Breakdown

We further decompose the end-to-end training acceleration of ReSpec on the Qwen-14B model to quantify the contribution of each component. As shown in Figure 14, the baseline system without SD is normalized to $1.0\times$. Introducing the *Reward-Weighted KD* algorithm in the Online Learner achieves a $1.48\times$ improvement by stabilizing SD and enhancing draft acceptance. Adding the *Adaptive Server*, including the solver and scheduler, brings an addi-

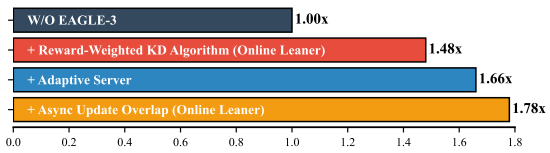

Figure 14. End-to-end training speedup breakdown of ReSpec on Qwen-14B. Starting from the baseline without SD.

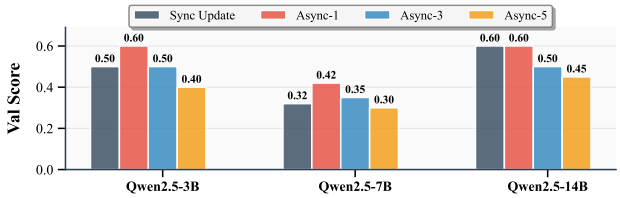

Figure 15. Validation score comparison under synchronous and asynchronous updates. Async-1 yields the best performance, while longer intervals (Async-3, Async-5) cause reward degradation, especially in smaller models.

tional 12% gain (1.66× total) by dynamically selecting the most efficient SD configuration under varying workloads. Finally, enabling the *Async Update Overlap* mechanism further hides draft update latency by overlapping online learning with rollout execution, reaching an overall 1.78× acceleration. These results demonstrate that the algorithmic and system-level optimizations in ReSpec are complementary and jointly contribute to end-to-end efficiency.

6.3 Asynchronization Analysis

We next evaluate the effectiveness of our asynchronous update overlap design across different model scales. Figure 15 reports the validation score under synchronous update and asynchronous updates with varying intervals ($I = 1, 3, 5$).

For Qwen2.5-3B and 7B, we observe a clear advantage of asynchronous updates at small intervals. Specifically, Async-1 achieves the highest reward (0.60 and 0.42, respectively), outperforming synchronous updates by a notable margin. Increasing the interval to 3 steps eliminates this gain, and at 5 steps performance further degrades below synchronous levels. This trend indicates that smaller models rely more critically on timely draft model adaptation to follow the evolving target distribution.

For Qwen2.5-14B, synchronous updates already achieve a relatively high reward (0.60), and Async-1 maintains this level without improvement. However, performance still degrades under longer intervals (Async-3: 0.50, Async-5: 0.45). This suggests that larger models are inherently more robust to stale draft updates, but still benefit from avoiding excessive delays.

Overall, these results validate our design principle: overlapping asynchronous updates with rollouts is most effective when updates are frequent (e.g., Async-1). This balance ensures the draft model remains aligned with target distribution while avoiding system stalls from synchronous training.

7 Conclusion

We present ReSpec, the first system that adapts speculative decoding to reinforcement learning for large language models. By addressing key bottlenecks, diminishing speedups, drafter staleness, and policy degradation,ReSpec integrates adaptive decoding, drafter evolution, and reward-weighted adaptation. Experiments on Qwen models show up to 4.5× faster training with stable reward convergence.

Acknowledgements

This work was supported by Shanghai Qiji Zhifeng Co., Ltd. under Grant No. 2025-GZL-RGZN-01001. We thank the anonymous reviewers for their constructive feedback, which helped improve the quality of this paper.

References

Ankner, Z., Parthasarathy, R., Nrusimha, A., Rinard, C., Ragan-Kelley, J., and Brandon, W. Hydra: Sequentially-dependent draft heads for medusa decoding. *arXiv preprint arXiv:2402.05109*, 2024.

AWS. Aws speculative decoding, 2024. URL https://aws.amazon.com/blogs/machine-learning/.

Cai, T., Li, Y., Geng, Z., Peng, H., Lee, J. D., Chen, D., and Dao, T. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.

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

Chen, Z., Yang, X., Lin, J., Sun, C., Chang, K. C., and Huang, J. Cascade speculative drafting for even faster llm inference. *Advances in Neural Information Processing Systems*, 37:86226–86242, 2024.

Comanici, G., Bieber, E., Schaekermann, M., Pasupat, I., Sachdeva, N., Dhillon, I., Blistein, M., Ram, O., Zhang, D., Rosen, E., et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Fu, W., Gao, J., Shen, X., Zhu, C., Mei, Z., He, C., Xu, S., Wei, G., Mei, J., Wang, J., et al. Areal: A large-scale

asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025.

Fu, Y., Bailis, P., Stoica, I., and Zhang, H. Break the sequential dependency of llm inference using lookahead decoding. *arXiv preprint arXiv:2402.02057*, 2024.

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

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

Harper, E., Majumdar, S., Kuchaiev, O., Jason, L., Zhang, Y., Bakhturina, E., Noroozi, V., Subramanian, S., Nithin, K., Jocelyn, H., Jia, F., Balam, J., Yang, X., Livne, M., Dong, Y., Naren, S., and Ginsburg, B. NeMo: a toolkit for Conversational AI and Large Language Models, 2025. URL https://github.com/NVIDIA/NeMo.

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

Hu, J., Wu, X., Shen, W., Liu, J. K., Zhu, Z., Wang, W., Jiang, S., Wang, H., Chen, H., Chen, B., et al. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. *arXiv preprint arXiv:2405.11143*, 2024.

Lei, K., Jin, Y., Zhai, M., Huang, K., Ye, H., and Zhai, J. {PUZZLE}: Efficiently aligning large language models through {Light-Weight} context switch. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pp. 127–140, 2024.

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

Li, Y., Wei, F., Zhang, C., and Zhang, H. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077*, 2024a.

Li, Y., Wei, F., Zhang, C., and Zhang, H. Eagle-2: Faster inference of language models with dynamic draft trees. *arXiv preprint arXiv:2406.16858*, 2024b.

Li, Y., Wei, F., Zhang, C., and Zhang, H. Eagle-3: Scaling up inference acceleration of large language models via training-time test. *arXiv preprint arXiv:2503.01840*, 2025.

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

Liu, X., Hu, L., Bailis, P., Cheung, A., Deng, Z., Stoica, I., and Zhang, H. Online speculative decoding. *arXiv preprint arXiv:2310.07177*, 2023.

Mei, Z., Fu, W., Li, K., Wang, G., Zhang, H., and Wu, Y. Realhf: Optimized rlhf training for large language models through parameter reallocation. *arXiv e-prints*, pp. arXiv–2406, 2024.

Miao, X., Oliaro, G., Zhang, Z., Cheng, X., Jin, H., Chen, T., and Jia, Z. Towards efficient generative large language model serving: A survey from algorithms to systems. *ACM Computing Surveys*, 58(1):1–37, 2025.

NVDIA. Tensorrt speculative decoding, 2024. URL https://developer.nvidia.com/blog.

OpenAI. Openai speculative decoding, 2024. URL https://platform.openai.com/docs/guides/predicted-outputs.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. Trust region policy optimization. In *International conference on machine learning*, pp. 1889–1897. PMLR, 2015.

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

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

Shao, Z., Srivatsa, V., Srivastava, S., Wu, Q., Ariyak, A., Wu, X., Patel, A., Wang, J., Liang, P., Dao, T., et al. Beat the long tail: Distribution-aware speculative decoding for rl training. *arXiv preprint arXiv:2511.13841*, 2025.

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

Team, A. Claude. https://www.anthropic.com/news/claude4., 2023.

Team, L. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation—ai. meta. com, 2025.

Wang, Z., Zhou, T., Liu, L., Li, A., Hu, J., Yang, D., Hou, J., Feng, S., Cheng, Y., and Qi, Y. Distflow: A fully distributed rl framework for scalable and efficient llm post-training. *arXiv preprint arXiv:2507.13833*, 2025.

Yan, M., Agarwal, S., and Venkataraman, S. Decoding speculative decoding. *arXiv preprint arXiv:2402.01528*, 2024.

Yang, A., Zhang, B., Hui, B., Gao, B., Yu, B., Li, C., Liu, D., Tu, J., Zhou, J., Lin, J., et al. Qwen2. 5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*, 2024.

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

Yao, Z., Aminabadi, R. Y., Ruwase, O., Rajbhandari, S., Wu, X., Awan, A. A., Rasley, J., Zhang, M., Li, C., Holmes, C., et al. Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales. *arXiv preprint arXiv:2308.01320*, 2023.

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

Zhang, Y., Lv, N., Wang, T., and Dang, J. Fastgrpo: Accelerating policy optimization via concurrency-aware speculative decoding and online draft learning. *arXiv preprint arXiv:2509.21792*, 2025.

Zheng, L., Yin, L., Xie, Z., Sun, C. L., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

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

Zhong, Y., Zhang, Z., Wu, B., Liu, S., Chen, Y., Wan, C., Hu, H., Xia, L., Ming, R., Zhu, Y., et al. Optimizing {RLHF} training for large language models with stage fusion. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pp. 489–503, 2025b.

Zhou, Y., Lyu, K., Rawat, A. S., Menon, A. K., Rostamizadeh, A., Kumar, S., Kagy, J.-F., and Agarwal, R. Distillspec: Improving speculative decoding via knowledge distillation. *arXiv preprint arXiv:2310.08461*, 2023.