History Doesn't Repeat Itself but Rollouts Rhyme: Accelerating Reinforcement Learning with RhymeRL

Jingkai He
Shanghai Jiao Tong University
Shanghai, China

Tianjian Li
ByteDance
Shanghai, China

Erhu Feng
Shanghai Jiao Tong University
Shanghai, China

Dong Du
Shanghai Jiao Tong University
Shanghai, China

Qian Liu
ByteDance
Shanghai, China

Tao Liu
ByteDance
Shanghai, China

Yubin Xia
Shanghai Jiao Tong University
Shanghai, China

Haibo Chen
Shanghai Jiao Tong University
Shanghai, China

Abstract

With the rapid advancement of large language models (LLMs), reinforcement learning (RL) has emerged as a pivotal methodology for enhancing the reasoning capabilities of LLMs. Unlike traditional pre-training approaches, RL encompasses multiple stages: *rollout*, *reward*, and *training*, which necessitates collaboration among various worker types. However, current RL systems continue to grapple with substantial GPU underutilization, due to two primary factors: (1) The rollout stage dominates the overall RL process due to test-time scaling; (2) Imbalances in rollout lengths (within the same batch) result in GPU bubbles. While prior solutions like asynchronous execution and truncation offer partial relief, they may compromise training accuracy for efficiency.

Our key insight stems from a previously overlooked observation: *rollout responses exhibit remarkable similarity across adjacent training epochs*. Based on the insight, we introduce RhymeRL, an LLM RL system designed to accelerate RL training with two key innovations. First, to enhance rollout generation, we present HistoSpec, a speculative decoding inference engine that utilizes the similarity of historical rollout token sequences to obtain accurate drafts. Second, to tackle rollout bubbles, we introduce HistoPipe, a two-tier scheduling strategy that leverages the similarity of historical rollout distributions to balance workload among rollout workers. Experimental results demonstrate that RhymeRL achieves up to a 2.6x performance improvement over existing methods, without compromising accuracy or modifying the RL paradigm.

CCS Concepts: • **Computing methodologies** → **Distributed computing methodologies**; **Machine learning**.

Keywords: Large language models, Reinforcement Learning

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790172

ACM Reference Format:
Jingkai He, Tianjian Li, Erhu Feng, Dong Du, Qian Liu, Tao Liu, Yubin Xia, Haibo Chen. 2026. History Doesn't Repeat Itself but Rollouts Rhyme: Accelerating Reinforcement Learning with RhymeRL. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3779212.3790172

1 Introduction

Post-training with reinforcement learning (RL) has emerged as a new paradigm for scaling and enhancing the capabilities of LLMs [1–6]. Representative post-training models, such as DeepSeek-R1 [1], have demonstrated significant improvements in areas including coding [5], mathematics [7] and many others [8–11]. A standard RL pipeline comprises three main stages: *rollout*, *reward*, and *training*. In the rollout stage, the LLM generates a large number of tokens, with extended reasoning to improve the quality of final responses (test-time scaling [12]). In the reward stage, a reward score is assigned to each response (i.e., sample). In the subsequent training stage, the model's weights are updated by computing new loss values based on the assigned rewards. As a result, RL systems are inherently complex and distributed, typically involving coordination among multiple heterogeneous workers to efficiently execute the various pipeline stages.

Current LLM RL systems face significant GPU underutilization, primarily arising from two sources. First, *overly long rollout phases*. During each RL step, the rollout phase, which entails generating a substantial number of thinking tokens, typically consumes 84% to 91% of the total time. Additionally, the autoregressive nature of LLMs prevents the rollout phase from fully utilizing GPU computational resources, resulting in substantial underutilization. Second, *bubbles caused by imbalanced rollouts*. Within a single batch, the response lengths of rollouts generated by different prompts vary dramatically, leading to a pronounced long-tail effect. As a result, completed rollout tasks must wait until the longest rollout in the batch finishes before advancing to reward and training stages. Due to these factors, we observe that SOTA RL systems, such as veRL [13], experience over 46% GPU resource

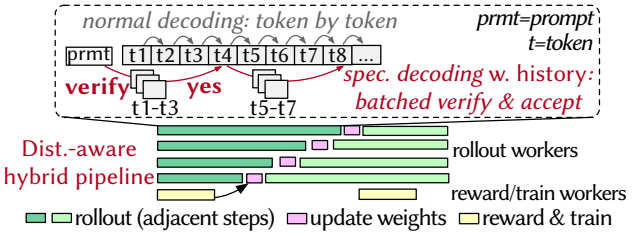


Figure 1. An overview of RhymeRL's designs.

idleness, significantly compromising the efficiency of RL.

To address these problems, recent research has mainly focused on scheduling optimizations. Some systems mitigate the performance degradation caused by long-tail rollouts by truncation (e.g., Kimi-K2 [14]) or reducing their batch size (e.g., StreamRL [15]). However, truncation methods inherently introduce a trade-off between accuracy and efficiency, while adjusting batch sizes provides only limited alleviation of the long-tail problem (~10% [15]). Some systems (e.g., AReaL [16]) depart from conventional RL systems by enabling full asynchronization between the rollout and training phases to reduce GPU idle time. However, this results in rollouts utilizing stale model weights, which changes the paradigm of RL. Additionally, model weight updates during rollout trigger the recomputation of all tokens in the ongoing rollouts, leading to considerable overhead. Therefore, enhancing the efficiency of rollout generation and minimizing GPU resource idleness remain critical challenges in the development of LLM RL systems.

Through a detailed analysis of the RL training practice, we observe that the rollout process in RL is distinct from conventional LLM inference. Specifically, a complete RL training consists of performing multiple inference passes on the same prompt in different RL steps (several *epochs* [7, 17, 18]). While the model weights vary between these steps due to continuous updates during training, current RL algorithms (such as GRPO [7], etc. [17, 19]) apply mechanisms (e.g., *PPO clipping* [20] and *gradient clipping* [21]) to restrict the magnitude of the model's updates at each step, thus maintaining stable model evolution. This stability leads to a *high degree of similarity* between rollout responses across different epochs: (1) *Token sequence similarity.* The responses generated for each prompt exhibit substantial similarity to their corresponding previous rollout responses, with 75%—95% of tokens being reusable. (2) *Length distribution similarity.* Although a prompt generates responses of varying lengths in different epochs, their position within the epoch's response length ranking remains stable, with only 2%-4% of responses experiencing significant rank changes. Motivated by this observation, our central insight is to *exploit the* ***similarity from historical rollouts*** *to accelerate the rollout process and achieve effective load balancing across rollout workers.*

We propose a novel RL system, RhymeRL, which significantly improves LLM RL efficiency without modifying existing RL training paradigms or sacrificing accuracy, as Fig. 1 shows. First, leveraging the historical token sequence similarity, we propose a lightweight yet effective *speculative decoding* [22] mechanism, *HistoSpec*, to accelerate the rollout process and improve the computational density. HistoSpec adopts a reward-aware, tree-based management strategy to organize draft tokens, and incorporates a novel speculative strategy inspired by TCP congestion control mechanisms (AIMD [23]) to improve prediction accuracy. Secondly, leveraging the historical length distribution similarity, we introduce the *HistoPipe* scheduling. By complementing long and short rollouts between adjacent steps, HistoPipe effectively balances workload across rollout workers and eliminates GPU bubbles caused by the imbalanced distribution of rollout lengths. HistoPipe further tackles outliers with migration-based tail rebalancing, and proposes a two-tier scheduling mechanism for better complementarity.

We have implemented our system on veRL and successfully deployed it in industrial RL environments, achieving a performance improvement of up to 2.6x. RhymeRL consistently attains SOTA performance across GPU clusters of varying scales while maintaining training accuracy without compromise. HistoSpec is merged into veRL's codebase [24].

In summary, we make the following contributions:

- We reveal the historical similarities during RL training. We also provide the key implications from our RL practices.
- We leverage speculative decoding for RL acceleration. We also design the reward-aware, tree-based distributed cache management and the AIMD-inspired token speculation.
- We propose the distribution-aware rollout pipeline. We also propose the migration-based tail rebalancing, and the two-tier scheduling mechanism.

2 Background

2.1 LLM Reinforcement Learning

RL-based post-training refines LLMs through interaction with environments. As the marginal benefits of pretraining diminish, RL is widely acknowledged as a pivotal approach for enhancing LLMs' reasoning capabilities [1, 2, 25, 26].

Workflow. In brief, the LLM RL training workflow comprises three stages, as Fig. 2 shows: (1) *Rollout*: the model generates responses to input prompts (processed in batches). (2) *Reward*: Responses are scored using rule-based functions or reward models. (3) *Train*: Loss is computed based on rewards, followed by backpropagation to optimize the model.

Algorithm. Early algorithms like PPO [27, 28] employ not only rule-based functions/reward models to generate sample-level rewards, but also simultaneously train a critic model to produce action-level rewards. This strategy stabilized training by reducing the volatility of rewards from sample-level evaluations. Contemporary mainstream methods, exemplified by GRPO [7] and DAPO [17], have superseded critic models with *Group Relative Advantage (GRA).* This paradigm

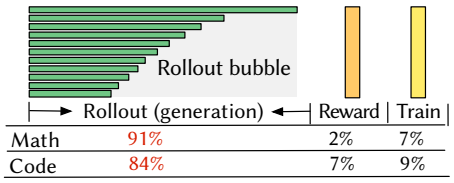


Figure 2. Phases and time distribution in LLM RL. *We train 32B models with math/code datasets using veRL [13] and GRPO [7].*

leverages the inherent stochasticity of LLMs during rollout: for each prompt, the LLM generates *a group of responses* (usually 16 in our production). Policy updates are then guided by relative scoring within the groups. The process of generating multiple responses per prompt can be interpreted as the model exploring diverse solution pathways to a problem.

2.2 Speculative Decoding

During LLM decoding, each attention operation requires accessing the entire KV cache, making the decoding process memory bandwidth-bound and preventing full utilization of computational resources. Unlike conventional decoding, which generates one token per iteration (i.e., LLM forward pass), speculative decoding [22, 29–33] first predicts multiple draft tokens using a lightweight method (e.g., using a small draft model [34–37] or retrieving from corpora [38–40]). The LLM then verifies these tokens in a single forward pass by computing their logits and accepting valid tokens. Since verifying multiple tokens incurs nearly identical memory access overhead (traversing the KV cache once) as generating one token conventionally, but with higher computational intensity, speculative decoding amortizes the memory cost across multiple tokens. This approach is theoretically proven to preserve output distribution integrity. When acceptance rates are favorable, it significantly accelerates LLM decoding [41].

3 Characterising RL Training in the Wild

Despite the widespread adoption of RL for LLM training, current RL systems still face significant performance challenges.

3.1 Rollout as the Major Bottleneck in RL Training

Implication-1: Rollout dominates the RL training timeline. During RL training, we observe that the rollout phase dominates the RL time. Specifically, as Fig. 2 shows, for LLMs trained with a maximum response length of 16K tokens, rollout accounted for 91% of the entire RL processing time for the math model and 84% for the code model. When the maximum response length was increased to 32K tokens or longer, the rollout time overhead exceeded 95%.

This significant overhead stems from three key factors: (1) *Complex reasoning demands.* During RL, LLMs are required to generate complex reasoning chains in responses. This results in long responses, and crucially, the response length tends to increase progressively as training advances. As Fig. 3a shows, the mean response length grows from 1K to 10K in 320 steps.

Systems	Reduce rollout time	Tackle rollout bubbles	Rollout -train pipeline	Core techniques
RLHFuse [45]	✗	Partially	✗	PPO-specific optimizations
HybridFlow [13]	✗	✗	✗	Hybrid programming model
Kimi K2 [14]	✗	Partially	✗	Partial rollout
DeepScaleR [18]	✗	✗	✓	Pipelined rollout/train
StreamRL [15]	✗	Partially	✓	Skewness-aware scheduling
AReaL [16, 42]	✗	✓	✓	Fully async rollout
AsyncFlow [43]	✗	Partially	✓	Streaming pipeline
DistFlow [44]	✗	✗	✓	Distributed multi-controller
RhymeRL	✓	✓	✓	HistoSpec + HistoPipe

Table 1. Existing LLM RL optimizations. *RLHFuse [45] proposes RLHF-specific optimizations (i.e., PPO-specific), and cannot optimize current RL algorithms (GRPO/DAPO). It fuses the inference of actor/-critic/reference/reward models and the training of actor/critic models. But current algorithms usually only rollout/train with the actor model.*

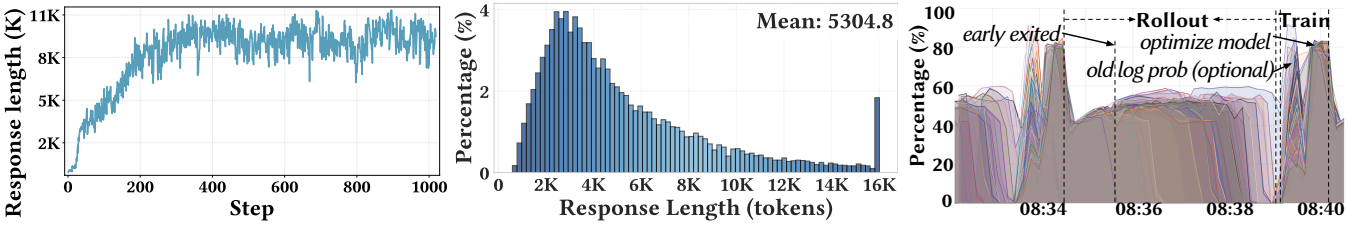
(2) *Memory-bound decoding.* During rollout, generating each token requires an LLM forward pass, which is constrained by memory bandwidth. (3) *Sequential dependency.* Due to the dependency between rollout and the subsequent training step, the training phase cannot commence until the longest sequence within the batch completes its rollout.

Implication-2: Rollout imbalance leads to GPU underutilization. Preserving the rollout-train dependency is critical for accuracy, but it introduces significant compute bubbles and leads to GPU idleness during rollout. We observe a significant long-tail effect within a batch, as shown in Fig. 3b — rollout response lengths vary widely across sequences. Due to the imbalanced distribution of rollout lengths, some rollout workers (i.e., GPUs cooperating with tensor parallelism) finish early and become idle, yet must remain inactive until all workers complete their tasks. As Fig. 3c shows, in a step, GPU monitoring reveals that the earliest-finishing GPU remains idle for ~76% of the total rollout duration.

3.2 State-of-the-Art Efforts and the Limitations

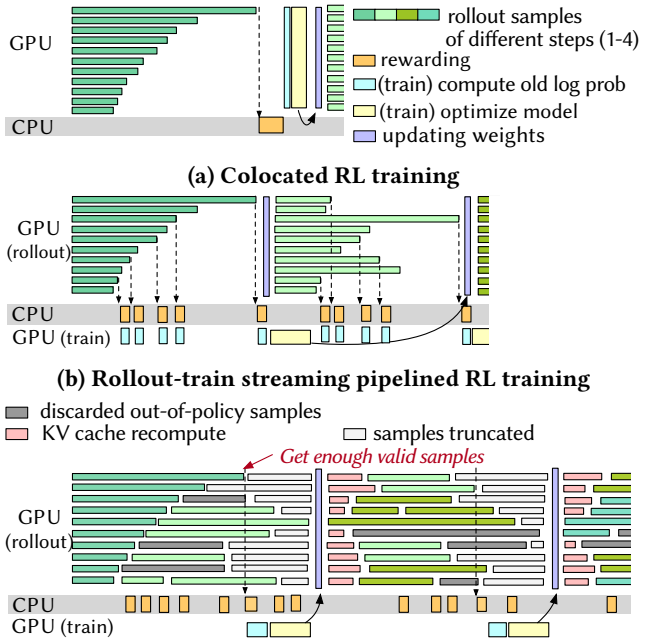
We summarize existing efforts on optimizing LLM RL systems [13–16, 18, 42–48] in Table. 1.

Pipelining rollout and training. Early RL systems like veRL [13] and OpenRLHF [47] adopt a *colocated architecture*, where GPUs repeatedly switch between rollout and training workloads, as Fig. 4a shows. This design incurs significant overhead from context switching between workers and substantial bubbles due to strict step-wise rollout-train dependency. Consequently, DeepScalaR [18] introduces a *decoupled architecture*, relaxing the dependency by using stale weights that are one step behind for rollout (i.e., the off-policyness is 1), which is widely proven to maintain training accuracy [15, 18, 49]. This enables coarse-grained rollout-train pipeline, and is adopted by systems including veRL [49]. AsyncFlow [43] further refines the rollout-train pipeline by granularizing dependencies from batch-level to mini-batch level, enhancing pipeline efficiency (as Fig. 4b shows, the reward and train processes proceed after a mini-batch's roll-



(a) Mean response lengths across steps (b) Response length distribution in a step (c) Real-world GPU SM utilization

Figure 3. Diving into real-world RL training of a 32B LLM. *We train the model using math datasets (>200K samples) with 64 Hopper GPUs. The max response length is set to 16K tokens. In Fig.-c, each line represents a GPU's SM utilization. The training stage can be further divided into old log prob computing (optional), which computes the generated tokens' logits via a model forward pass , and model optimizing.*



(a) Colocated RL training

(b) Rollout-train streaming pipelined RL training

(c) Fully asynchronous rollout RL (max off-policyness = 1)

Figure 4. Existing LLM RL pipelines. *Dashed lines represent the dependency between rollout samples and reward/train, while solid lines represent the dependency between training and weight updating.*

out finishes). Nevertheless, imbalance persists among rollout workers, leaving significant bubbles within rollout stages.

Tackling long-tail rollout. StreamRL [15] proposes *skewness-aware scheduling*, which allocates additional data-parallel GPUs to the prompts likely to generate long responses. By reducing batch sizes for these prompts, it alleviates the long-tail effect in rollouts. However, due to the autoregressive nature of LLM rollouts, this approach only marginally reduces the long-tail and the rollout bubbles (~10% [15]). Kimi-K2 [14] introduces *partial rollout*, which truncates excessively long responses and retains generated segments for continuation in subsequent steps. This method faces an efficiency-accuracy dilemma: Excessive truncation causes significant portions of responses to be generated by stale model weights, ultimately producing outdated reward signals; conservative truncation diminishes its effectiveness in reducing rollout bubbles.

Fully asynchronous rollout. AReaL [16] adopts a radical approach (Fig. 4c): (1) Fully async rollout. Rollout workers continuously generate responses while train workers select usable samples based on the maximum off-policyness threshold. (2) Aggressive dependency relaxation. Training can utilize rewards derived from rollouts generated by (potentially) stale model weights many steps prior. When new model weights are produced, ongoing rollouts are truncated to propagate updated weights. The truncated rollouts are continued after recomputing the KV cache with new weights. This method has critical limitations: (1) Fully async rollout changes the paradigm of RL. Excessive off-policyness floods training with obsolete signals. (2) Rollout imbalance-induced GPU underutilization remains. Rollouts exceeding the off-policyness threshold are discarded, and frequent KV cache recomputation wastes GPU resources.

Summary. In existing systems, rollout imbalance remains a persistent bottleneck, resulting in significant GPU resource underutilization. Critically, no existing approach reduces the time required for rollout execution and improves the GPU computational underutilization of the rollout stages.

4 RhymeRL Overview

To resolve the persistent challenges in LLM RL: time-consuming rollouts and rollout imbalance, we design and implement RhymeRL, which leverages historical rollout information to accelerate rollout execution and minimize imbalance, thereby achieving significant speedups in RL training.

4.1 Observations and Insights

Observation. A complete LLM RL training typically spans several *epochs* [7, 17, 18], each comprising multiple *steps* that iterate through the full dataset[1]. In the long-term practice of LLM RL training, we observe strong ***historical similarity*** in rollout responses across epochs, characterized by:

- High *token similarity* in responses of the same prompt

[1]For example, Qwen3 [4] trains 170 steps with only 3995 query-verifier pairs, DeepSeekMath-RL [7] (>8000 steps) trains >55 epochs, JustRL-DeepSeek [50] (4380 steps) trains >6 epochs, M2PO [51] (4000 steps) trains >25 epochs, DeepScaleR [18] (1750 steps) trains >5 epochs, SimpleTIR [52] (1200 steps, multi-turn) trains 4 epochs.

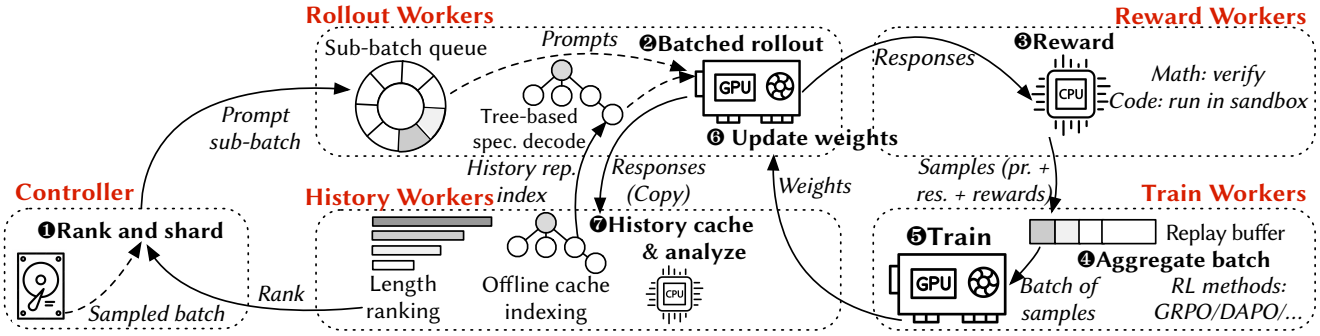


Figure 5. RhymeRL overview. *Solid lines indicate data flow across workers. Dashed lines indicate data dependencies within a worker.*

across adjacent epochs, i.e., responses in each epoch contain a large proportion of consecutive token sequences that are identical to the sequences in the previous epoch (§5.1);

• High *response length distribution similarity* across adjacent epochs. If we rank the prompts using their response lengths, the ranking across adjacent epochs remains similar (§6.1).

Root cause. To maintain stable model evolution and prevent catastrophic forgetting, current RL algorithms [7, 17, 19, 27, 51] employ mechanisms to constrain the magnitude of policy updates, such as PPO clipping [20] and gradient clipping [21].

$$\nabla_\theta L \propto \underbrace{\hat{A}_t}_{\text{Advantage}} \cdot \underbrace{r_t(\theta)}_{\text{Importance weight}} \cdot \underbrace{\nabla_\theta \log \pi_\theta(a_t|s_t)}_{\text{Score Function}} \quad (1)$$

As Equation 1 shows, the magnitude of the update depends on the reward signal (i.e., advantage), the difference between old and new policies (i.e., importance weight), and the gradient of log-policy (i.e., score function). The advantage is typically normalized, while the importance weight is constrained by PPO clipping (e.g., usually 0.2–0.28 [17]). During stable iterating RL, the score function usually remains moderate. Furthermore, the algorithms apply gradient clipping to bound the final gradient. In practice, after the initial phase, each step's gradient norm (the gradient's Euclidean norm) generally stabilizes at the order of 10^{-1} [53].

Insights. The key insight of RhymeRL is motivated by the above novel observations. By systematically organizing and utilizing historical information, which has been overlooked in all existing RL systems, we unlock new opportunities to further enhance the overall performance of RL system. Specifically, to accelerate the rollout process, we propose a dedicated speculative decoding mechanism that leverages historical rollouts as accurate drafts. Furthermore, to balance the workload among rollout workers, we introduce a two-tier, distribution-aware pipeline scheduling strategy that exploits the distributional similarity present in historical rollouts.

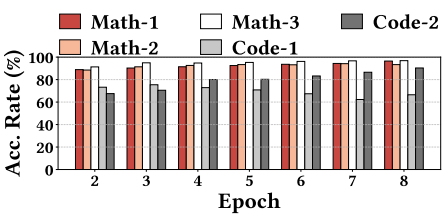
4.2 System Overview

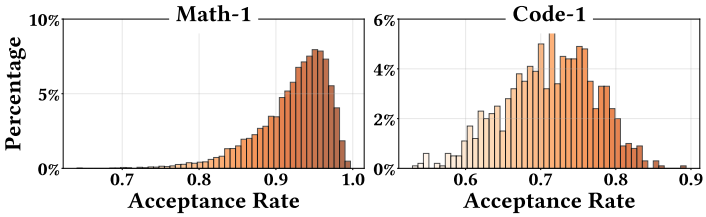
As Fig.5 shows, RhymeRL inherits the hybrid controller architecture proposed by HybridFlow [13] and the disaggregated rollout-train structure (Fig. 4b), utilizing dedicated *rollout workers* for response generation, *reward workers* for reward computation, and *train workers* for policy optimization to form a pipelined RL workflow.

To improve the RL system's overall efficiency, we adopt a *streaming pipeline* architecture. During each step, each rollout worker retrieves a sub-batch of prompts asynchronously dispatched by the controller from its *prompt sub-batch queue*, and generates responses using the inference engine (❷). Completed rollout responses (i.e., samples) proceed to reward workers for scoring (❸) before being transferred to train workers' replay buffer. Once the replay buffer accumulates sufficient samples matching the batch size (❹), the train workers execute user-defined RL algorithms to optimize the model (❺). Following full-batch policy optimization, updated model weights are propagated from train workers to the *weight buffers* (on host memory) of rollout workers. Before processing each step's sub-batch, rollout workers synchronize the weights to their GPUs (❻). This *worker-controlled asynchronous weight update* strategy removes global synchronization overhead and minimizes idle waiting times.

To accelerate rollout generation via speculative decoding (HistoSpec), RhymeRL employs a *reward-aware suffix-tree-based approach* (§5.3) that efficiently generates speculation drafts from historical data with minimal overhead. We further propose an *AIMD-inspired token speculation strategy* (§5.4) to dynamically adjust the length of speculative tokens, thereby achieving higher acceptance rates while improving computational density. Moreover, to balance the workload among rollout workers, RhymeRL implements a *distribution-aware scheduling strategy* (HistoPipe), which leverages inter-step complementarity to achieve overall workload balance. It addresses anomalous outliers with *migration-based tail rebalancing* (§6.3), and tackles highly skewed rollout time distributions with a *two-tier scheduling* mechanism (§6.4).

We introduce the cluster-scale *history workers* to manage, index and analyzethe historical responses, which transfer the indexed cache (asynchronously) to rollout workers for speculation, and provide the length ranking information to the controller for rollout scheduling. They operate on idle CPU resources within the RL cluster. With the integration of history workers, the RL workflow incorporates two additional stages. First, the controller leverages the historical





Dataset	Size
Math-1	230K
Math-2	40K
Math-3	10K
Code-1	40K
Code-2	12K

(a) Acceptance rate across epochs **(b) Responses' acceptance rate distribution (Math-1 and Code-1)** **(c) Datasets used**

Figure 6. Characterizing historical token similarity. *We use five datasets to train 14B LLMs respectively using the GRPO algorithm.*

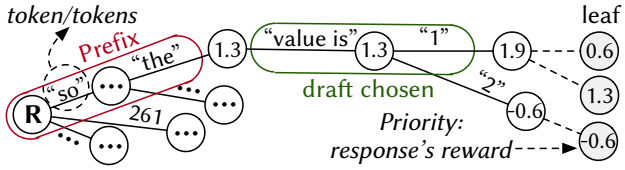


Figure 7. Suffix-tree based draft generation. *R = Root. For ease of understanding, we convert token ids into words.*

length ranking to enable more effective task scheduling (❶), while it also distributes relevant historical responses to the assigned rollout workers. Second, after a rollout worker generates responses, it asynchronously sends them to history workers (❼), which update corresponding data structures.

5 HistoSpec: Speculative Rollout Generation

Since the rollout phase constitutes the majority of execution time in RL workflows, accelerating rollout is critical for improving overall efficiency. Although existing RL systems utilize SOTA inference engines [54, 55] during rollout, their performance is fundamentally constrained by memory bandwidth, due to the extremely long rollout sequences. We observe that rollout is a specialized LLM inference that demonstrates high historical similarity. Motivated by this, we introduce a novel speculative strategy dedicated to rollout.

5.1 Observation: Token Similarity in Rollout

RL training comprises multiple iterative epochs, during which the same prompt will be repeatedly sampled across different epochs. Mainstream RL algorithms [7, 17, 19, 27] restrict the magnitude of the model's updates. Furthermore, they use *Group Relative Advantage (GRA) Optimization*, generating multiple responses per prompt (8—64 responses), which enables comprehensive exploration of diverse reasoning trajectories throughout each epoch. Consequently, the outputs generated for the same prompt during rollouts at adjacent epochs exhibit a high degree of similarity.

To quantitatively evaluate the similarity of tokens generated during rollouts, we analyze the response traces generated during RL training across five math/code datasets, as listed in Fig. 6c. For each response, we simulate its rollout "generation" from the beginning, and use the last three "generated" tokens as the prefix to search for exact matches in the historical responses from the previous epoch. When a match is found, we record the length of identical token sequences that follow the prefix in both historical and current responses, labeling them as "accept" tokens. Otherwise, we will forward the response to "generate" one token. The routine is repeated until the end of the response. As Fig. 6a shows, across 8 epochs, for math tasks, 93% of tokens could be successfully "accept" using this routine (on average, 75% for code). Furthermore, as training progresses (i.e., with increasing epochs), the similarity increases. Fig. 6b shows the distribution of the responses' acceptance rates.

5.2 Speculative Rollout with History

Leveraging historical token similarity, we design HistoSpec, which utilizes speculative decoding to accelerate rollouts and uses historical responses as draft sources. In each decoding iteration, HistoSpec uses *the last few generated tokens* as the prefix to search for matches within the prompt's historical responses. Upon matching, it extracts a certain number of subsequent tokens following the prefix as drafts. The drafts are then verified via a single LLM forward pass, with some (possibly all) tokens accepted (§2.2). This history-based speculative rollout improves computational resource utilization during rollouts, and reduces the time of the rollout stage.

Technical challenges. However, this approach encounters two challenges: (1) Over-long rollout sequences induce significant prefix matching overhead, and the branching in historical responses complicates draft token selection; (2) Unpredictable draft acceptance length creates a trade-off between underutilizing compute capacity (when predicting too few tokens) and wasting resources on verifying rejected tokens (when over-predicting).

5.3 Tree-based Historical Rollout Management

Draft generation overhead is a pivotal factor determining speculative decoding's effectiveness. However, existing corpus-based speculation methods face a dilemma between retrieval costs and index building costs. E.g., n-gram [39, 56] requires time-consuming linear scans for prefix matching, and SuffixDecoding [38] limits the lengths of draft source sequences to reduce index building costs. During LLM RL, each epoch generates multiple long responses (totaling hundreds of thousands of tokens per prompt). Traditional draft generation methods struggle to operate efficiently under the constraints.

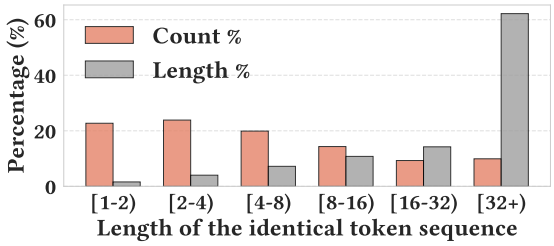


Figure 8. Distribution of speculative hit lengths.

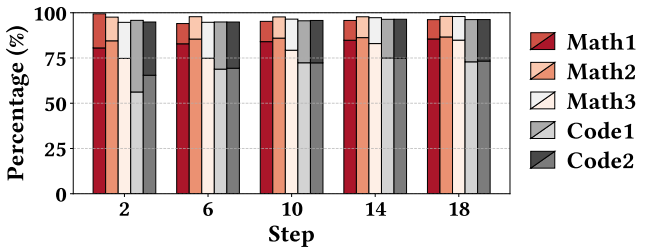


Figure 9. The ranking group changes of responses across 20 epochs. *Since each prompt generate multiple responses, we analyze the previous epoch by assigning all responses of each prompt to the same ranking group based on their* ***median*** *length, yielding the predicted group. For the current epoch, we sort all responses by their actual lengths to obtain the real group, and compare it with the predicted group to collect the data. In the figure, the lower parts of the bars represent the proportion of responses that remain in the same or a lower group; the upper parts represent the proportion of responses that, despite changing groups, only move up one group and have lengths in the shorter 50% of the new group (i.e., shift near the group boundary).*

Async cache building. The RL workloads allow us to relax the constraints on draft generation. In RL sampling strategies, the same request is generally not re-sampled until multiple steps later. Therefore, we are able to shift the computational overhead associated with retrieval to the indexing phase. RhymeRL employs history workers to index historical rollout sequences asynchronously. Upon generating new responses, the controller dispatches them to history workers for index construction. When scheduling prompts to rollout workers (asynchronously), the controller notifies the corresponding history workers to transfer the relevant indexed cache.

Reward-aware tree-based management. RhymeRL employs suffix trees [57] to index cached responses, which enables $O(m)$-time matching for length-m prefixes. RhymeRL maintains a dedicated tree for each prompt, indexing all its historical responses generated in its last rollout. Since modern RL algorithms generate multiple responses per prompt to explore multiple solution paths, a prefix may branch to divergent suffixes, complicating draft token selection. We observe that, during training, RL algorithms optimize the model towards generating high-reward solutions with higher probability. Therefore, we add a *priority* value to each tree node, weighted by the sum of the rewards of its branch. For every suffix branch present, HistoSpec selects the one with the highest priority. Through this RL-algorithm-guided design, HistoSpec maximizes the speculation acceptance rate.

As Fig. 7 shows, in the suffix tree, each node represents a subsequence formed by the path from root to that node, where each leaf node corresponds to a complete suffix of a response, and each edge represents one or more tokens extending the sequence of parent node to the sequence of the child. For each leaf node, its priority (marked on the nodes in Fig. 7) is the sum of the rewards for the responses that end with this suffix (that the leaf node represents). A parent node's priority is the sum of its children's priorities.

Resources. Both the tree's construction time overhead and memory overhead are $O(n)$ for n tokens [58]. GPU clusters have sufficient CPU resources (64–128+ cores and multi-TB host memory), which are mostly idle during previous RL flows. RhymeRL strictly limits the resources used by history workers using OS methods [59], preventing them from interfering with other workers. For RL training with a 230K dataset and 16K max response length, the host memory overhead of suffix trees is $<$ 80GB per node with 8 nodes.

HistoSpec also supports compression and swapping to SSD for memory saving, and checkpoint for fault tolerance.

5.4 AIMD-like Token Speculation

To determine the length of tokens predicted by HistoSpec in each iteration, we conduct a detailed analysis of the length distribution of identical token sequences in rollout responses. As shown in Fig. 8, short segments (1-2 tokens) dominate in quantity, while long segments account for the majority in the total length. This poses a challenge for HistoSpec: If we predict many tokens per iteration, most tokens would be rejected, wasting significant computation on verifying them. If we predict a small number of tokens, computational resources cannot be fully utilized, undermining speculative decoding's advantage in improving computational density.

We find that network congestion control encounters similar challenges. Classical TCP congestion control employs the AIMD (*Additive Increase, Multiplicative Decrease* [23]) principle: gradually increasing the window size when network is uncongested, but aggressively reducing it upon congestion. Inspired by AIMD, HistoSpec designs a dynamic speculation window for each response, initialized to two tokens. When all speculated tokens are accepted, HistoSpec additively increases the window size by 2, until it meets the upper threshold (32 by default). But when any token is rejected, it resets the size directly to 2. This approach guarantees high computational density during generating long matching sequences, and minimal wasted computation for short sequences.

As for the prefix length, HistoSpec sets it to 7 initially. If no matching suffix is found, HistoSpec progressively reduces it until it reaches 3. These hyperparameters are adjustable.

HistoSpec also considers *batch size*. At large batch sizes, increased decoding parallelism yields higher GPU computational utilization, where excessively low speculative acceptance rates degrades the overall throughput. To mitigate this,

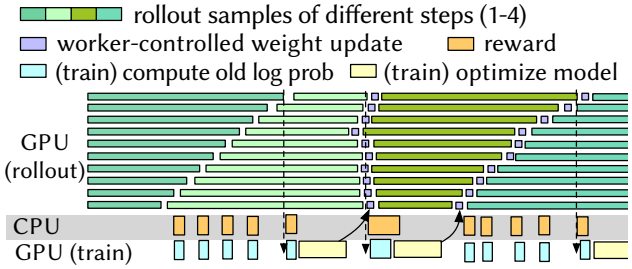


Figure 10. HistoPipe design. *Leveraging the **worker-controlled async weight updating** (§4.2), updated model weights are propagated asynchronously to the rollout workers' weight buffers and updated to GPUs by rollout workers upon completing a rollout step. In rare cases where the corresponding weights have not been propagated to the weight buffer, the rollout worker will await the weights.*

HistoSpec monitors the acceptance rates and adaptively disables speculation. Through pre-profiling, HistoSpec gets the maximum viable batch size for throughput gains at different acceptance rates. When a rollout worker's current batch size exceeds the threshold, speculation is automatically disabled to preserve system efficiency.

We provide a dedicated proof for the sampling mechanism of HistoSpec, to rigorously demonstrate that it does not alter the model's output token distribution (§A.2).

6 HistoPipe: Hybrid Rollout Pipeline

Although HistoSpec improves rollout efficiency, it does not address the issue of imbalanced rollout distributions, resulting in non-trivial GPU resource underutilization. We observe that leveraging the information from historical rollouts can effectively maintain load balance throughout the rollout process, and propose the HistoPipe scheduling design.

6.1 Observation: Distribution Similarity in Rollout

Although different prompts generate varying numbers of tokens within a single rollout, we observe that the response length (i.e., token count) distribution for the same prompt across adjacent epochs remains similar. In other words, if a prompt generates a relatively large number of tokens in one rollout iteration, it is likely to produce a similarly large token count in subsequent rollouts. Therefore, *we can effectively predict the ranking of response lengths using historical lengths.*

To validate this observation, we analyze the response length rankings across multiple rollout epochs for five datasets (Fig. 6c). Specifically, we group the responses into 8 ranking groups based on their lengths, ranked from low to high. As Fig. 9 shows, across 20 epochs (300—1000 steps), for math tasks, an average of only 16% of responses change their group to higher ones (28% for code tasks). Among them, 13% of the (all) responses only shift near the group boundary without significantly altering the distribution (24% for code). These results demonstrate that not only do generated tokens exhibit similarity across rollouts, but the *distribution of rollout lengths also remains relatively consistent over epochs.*

Dataset name	Math-1		Math-2		Math-3		Code-1		Code-2	
Rank	8	16	8	16	8	16	8	16	8	16
Accurate	85.7	79.7	83.2	76.9	79.0	71.6	71.8	62.6	73.2	62.5
Not last 10%	12.2	16.8	12.7	17.5	16.9	22.8	23.9	30.4	22.9	30.1
Within 1.1x	0.61	1.03	0.55	1.02	0.86	1.47	1.44	2.34	1.46	2.32
Migrated	**1.49**	**2.47**	**3.55**	**4.58**	**3.24**	**4.13**	**2.86**	**4.66**	**2.44**	**5.08**

Table 2. Accuracy of predicting the samples' ranks with historical lengths and migration rates (14B models). *We list the average value of the metrics of the 20+ epochs.*

6.2 Distribution-aware Hybrid Pipeline

Leveraging the similarity in rollout length distributions can enable more balanced scheduling of rollout tasks, reducing load imbalance during rollout. We propose a distribution-aware rollout pipeline strategy, named HistoPipe.

Hybrid Pipeline. Preserving the rollout-reward-train dependency is essential for algorithmic integrity. However, the imbalanced distribution of rollout lengths and the autoregressive nature of LLMs induce rollout bubbles within individual steps. Inspired by Dual-Pipe's philosophy [60], instead of seeking per-step balance, HistoPipe shifts focus to *inter-step workload complementarity*, which constructs synergistic balancing across consecutive training steps.

Historical distribution enables HistoPipe to rank rollout prompts based on their historical response lengths, obtaining several **ranking groups** by equally dividing the ranked prompts. As Fig. 10 shows, during odd-numbered rollout steps, the scheduler assigns ranking groups to workers (0 through *N-1*) in *ascending order* of rollout length.

Conversely, during even-numbered steps, ranking groups are assigned in *descending order* of rollout length. By complementing rollout lengths in this alternating manner, we fill idle time that occurs during rollout execution, thereby improving the overall efficiency of the rollout workers. [2]

HistoPipe is not enabled during the first epoch as no historical information exists, which is acceptable because RL training typically spans several epochs, and the first epoch exhibits the shortest duration. For RL algorithms using Group Relative Advantage, HistoPipe uses the median lengths within each response group as the *historical response lengths* to rank the prompts. Although we do not preclude more complicated algorithms or model-based methods [15], the current method is sufficiently robust and effective (Fig. 9).

Technical challenges. HistoPipe also faces several challenges under real workloads. (1) Although the rollout length distributions exhibit high similarity, the occurrence of an anomalously long rollout can damage the complementarity and disrupt the pipelines. (2) In practice, the long-tail distribution of rollout lengths prevents consecutive steps from achieving perfect workload complementarity.

[2] After gathering the entire batch from rollout/reward workers, the controller randomly shuffles the batch, and then divides it into mini-batches for training. This ensures that the training process does not inherit any bias from the ranking-based grouping during rollout.

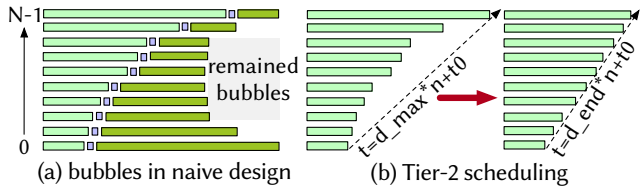


Figure 11. Bubbles caused by skewed rollout time distribution, and how the two-tier scheduling reduces bubbles.

6.3 Migration-based Tail Rebalancing

To mitigate the impact of anomalous rollout lengths on the hybrid pipeline, we employ two strategies. (1) *Intra-step migration.* Migrating excessively long rollouts to other groups that are still in the same step's rollout process. (2) *Inter-step migration.* Migrating the outliers to the next step.

More specifically, we set a threshold for each ranking group. When a rollout length exceeds the threshold, it will be migrated. For the first strategy, long rollouts are dynamically reassigned to other groups during execution. The generated tokens are preserved and the rollouts continue after RhymeRL recomputes the KV cache (i.e., prefill) of the prompt and the tokens. For the second strategy, the migrated rollout is added to the next step. Similarly, the generated tokens are preserved and the rollouts continue after KV cache recomputation. Since the RL algorithms and systems inherently use oversampling, inter-step migrating a small number of rollouts and deferring their completion to the next step do not adversely affect training. In our scheduling design, anomalous rollouts in groups with short rollout lengths are preferentially intra-step migrated, whereas inter-step migration is used for anomalous rollouts in groups with generally long rollout lengths. In practice, intra-step migration efficiently handles most of the outliers.

Threshold. RhymeRL migrates rollouts exhibiting: (1) within the last remaining α% of the rollouts in the group; (2) generated response length exceeding β-times the maximum historical response length of the prompts in the same group. Currently, α=10 and β is determined by the 75th percentile of the growth rates in response lengths of the previous epoch (β=1.1 if the percentile < 1.1). Analysis across 5 datasets over 20 epochs reveals only 1.49%—3.55% (8 groups, on average) of responses undergo migration, demonstrating acceptable overhead and negligible impact on training accuracy.

6.4 Two-tier Scheduling

Achieving near-bubble-free inter-step complementarity requires approximately linear execution time distributions across rollout groups. In practice, however, long-tailed rollouts (Fig. 3b) cause the time distributions to approximate exponential patterns, which leads to bubbles on some rollout workers, as Fig. 11a shows. HistoPipe proposes a two-tier scheduling mechanism to tackle this issue:

- **Tier-1: assigning prompts to ranking groups.** First, HistoPipe equally divides the ranked prompts into groups.
- **Tier-2: mapping ranking groups to GPUs.** If we allocate equal GPUs per ranking group, their execution times will exhibit exponential distributions due to long-tail rollouts. Therefore, HistoPipe designs a distribution reshaping strategy: instead of evenly distributing GPUs among the groups, it allocates fewer GPUs to the short-length and medium-length groups while provisioning extra GPUs to the few groups with long responses. In this way, HistoPipe reshapes the groups' rollout time distribution from *exponential to linear*, thereby further reducing rollout bubbles.

In RhymeRL, each rollout worker manages several GPUs cooperating via model parallelism with user-specified configurations. During Tier-2 scheduling, RhymeRL allocates GPUs at rollout-worker granularity. Rollout workers operate via data parallelism, thus, increasing rollout workers reduces per-worker batch size and shortens the execution time.

Algorithm. HistoPipe determines the GPU allocation plan using the algorithm detailed in Fig. 12. It uses the mean value of the prompts' historical response lengths within each ranking group (*representative length*) to estimate the group's execution time. Through pre-profiling of the inference engine (i.e., HistoSpec, also considering the impact of speculative decoding), we can obtain the relationship between a ranking group's execution time *(t)*, its representative rollout length *(l)*, and DP worker number *(dp)*: $t{=}\tau(l, dp)$. Within the RL cluster, there are *wks* rollout workers to serve N ranking

```
►MIN_WKS/MAX_WKS: a ranking group' min/max worker number
►Tool func: calculate workers needed for a given d
func calWks(d, lens, wks, t0)
  wks_needed = 0
  for i in range(0, N):
    target_t = t0+i*d
    ►find min workers to meet the group's target time
    group_wks = -1
    for k in range(MIN_WKS,
            MAX_WKS+1):
      exec_t = τ(lens[i], k)
      if exec_t <= target_t:
        group_wks = k
        break
    if group_wks == -1:
      return (Inf, [])
    wks_needed += group_wks
    plan[i] = group_wks
  return (wks_needed, plan)
```

```
►Find best worker allocation
►The target time of group n: t(n)=d*n+t0, 0<=n<N
func planAllocation(lens,
    wks, t_train)
  t0 = max(τ(lens[0], MAX_WKS),
      t_train)
  d_min = 0.0
  d_max = (τ(lens[N-1],
        MIN_WKS)-t0)/(N-1)
  p = 1 # precision, adjustable
  ►binary search
  while (d_max-d_min) > p:
    d_mid = (d_max+d_min)/2
    result = calWks(d_mid,
              lens, wks, t0)
    if result.wks_needed > wks:
      d_min = d_mid
    else: # feasible solution
      best_plan = result.plan
      d_max = d_mid
  return best_plan
```

Figure 12. Tier-2 scheduling algorithm. *It solves the problem by minimizing the execution time gradient (d) via binary search, as Fig. 11b shows. We use* t_train *(time of the last finished step's training stage) to ensure the execution time of the shortest ranking group is longer than the training stage's time, avoiding the next step's rollout waiting for train workers to generate weights.* $\tau(l, dp)$ *is determined by looking up the table obtained from pre-profiling. The* ***cooperation of HistoPipe with HistoSpec*** *is also considered.* $\tau(l, dp)$ *also considers HistoSpec's current speculation information (e.g., acceptance rate), which is omitted in the algorithm for ease of understanding.*

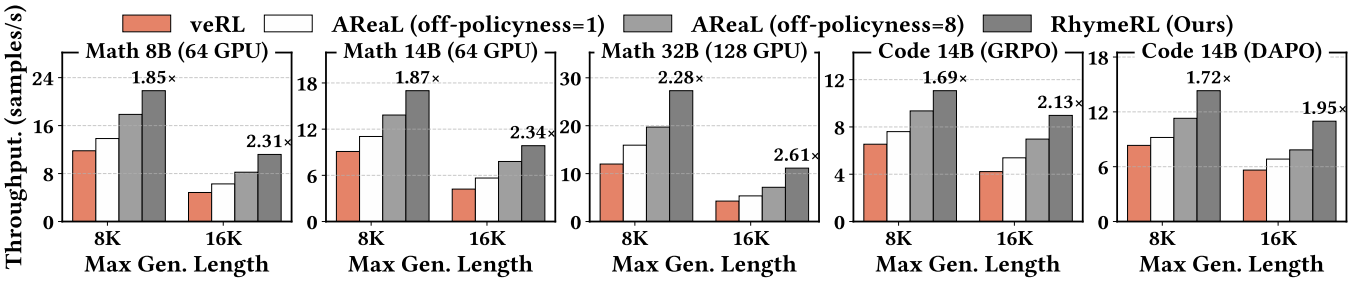


Figure 13. The training throughput of RhymeRL, veRL, and AReaL. *For each setting, we use the same configurations (e.g., number of rollout/train workers, clip ratio, etc.) for the three systems. We use 4-GPU tensor parallelism (TP) for 32B LLMs, and 2-GPU TP for 8B/14B LLMs.*

Model	Layers	Attn heads	K/V heads	Hidden size
Qwen3-8B-Base	36	32	8	4096
Qwen3-14B-Base	40	40	8	5120
Qwen2.5-32B	64	40	8	5120
Llama3.1-8B	32	32	8	4096
DeepSeek-R1-Distill-Qwen-7B	28	28	4	3584

Table 3. The specifications of the evaluated LLMs.

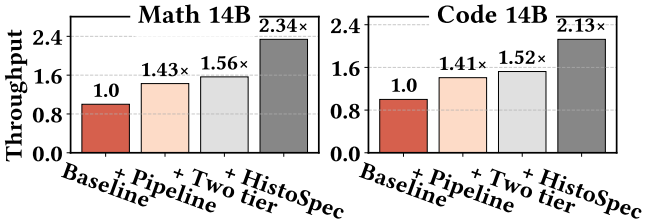


Figure 14. Breakdown of RhymeRL's improvements. *The max response length is set to 16K tokens. The throughputs are normalized.*

groups, whose representative lengths are *lens*. We have to assign varying numbers of rollout workers to each group, and make their completion times close to linear distribution. This constitutes an *integer nonlinear programming* problem and we converge to the solution via binary search.

7 Evaluation

We evaluate RhymeRL with LLMs of sizes ranging from 8B to 32B, on real-world math and code datasets.

Experiments. First, we compare RhymeRL's end-to-end training throughput with SOTA LLM RL systems on training LLMs of different sizes using different datasets (§7.1). Then, we break down the improvements of RhymeRL's designs (§7.2.1). To gain a deeper understanding of HistoSpec (§7.2.2), we study HistoSpec's improvements on rollout throughput across steps, speculation rates, and acceptance rates. To further understand HistoPipe (§7.2.3), we study its improvements on training throughput across steps, and its migration rates. Finally, we study the impact of RhymeRL on training accuracy and algorithm behavior (§7.3). We also study HistoSpec's generality (§7.4), and compare HistoSpec with other speculative decoding methods (§8.1).

Hardware settings. We deploy RhymeRL on 16 nodes and 128 Hopper GPUs (80 GB). Each node is equipped with 8 GPUs, 2 Intel Xeon Sapphire Rapids CPUs (96 cores), 1.9 TB host DRAM, and 9 * 400 Gb/s ConnectX7 InfiniBand NIC (8 GPU-dedicated NICs and one CPU-dedicated NIC).

Algorithm settings. In all experiments, we set rollout's temperature to 1.0, top_p = 1.0, top_k = -1, min_p = 0.0, and remove the KL penalty. This is the SOTA practice used by DAPO [17], AReaL [16], and is recommended for all models/settings by veRL [61]. We also evaluate RhymeRL under different algorithm settings (different temperature and enabling/disabling KL penalty) in §7.4.1.

Model and algorithm. Our evaluation primarily utilizes Qwen3-8B-Base, Qwen3-14B-Base, and Qwen2.5-32B models [4, 62], which constitute our production model suite for business applications, with detailed architectures listed in Table.3. We also train DeepSeek-R1-Distill-Qwen-7B [1] and LLama-3.1-8B [63] models to validate RhymeRL's generality (§7.4.2). We train these models using GRPO algorithm [7], which is currently the mainstream LLM RL algorithm, powering advanced LLMs such as DeepSeek-R1 [1]. We also verify RhymeRL's efficiency on recent next-generation algorithms like DAPO [17] and provide the results. Based on our practice, the response group size is set to 16, which yields peak algorithmic efficiency.

Metrics. As per convention [15, 45, 64], we report training throughput, measured by the average number of samples generated and processed per second. We sample 8K math-/code prompts from internal datasets (Fig. 6c). Under each setting, we train 80 steps for math models and 100 steps for code models, and use the first 20 steps as warm-up.

7.1 End-to-end Training Performance

We compare the end-to-end training performance of RhymeRL with the following SOTA LLM RL training systems.

- **veRL [13]** (v0.4.1). Featuring the hierarchical hybrid programming model, veRL is used by many projects and companies, and is RhymeRL's codebase. We use the rollout/train disaggregated architecture natively supported by veRL. It adapts the key designs of StreamRL [15], e.g., rollout/train disaggregation and asynchronous training, which is the SOTA RL architecture and outperforms its hybrid mode [49].
- **AReaL [16]** (v0.3.0). Leveraging fully async rollout, AReaL achieves continuous rollout GPU utilization without idle time (§3.2). We evaluate its performance when its max off-

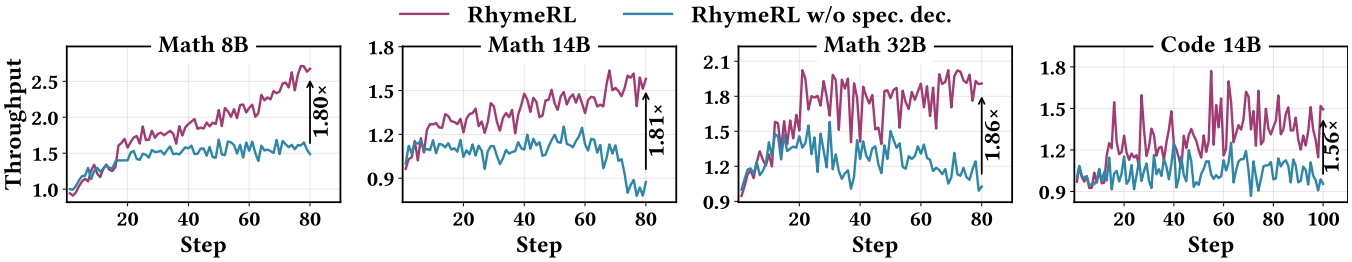


Figure 15. HistoSpec's rollout throughput improvements. *Max response length = 8K. The results are normalized to the baseline's first step.*

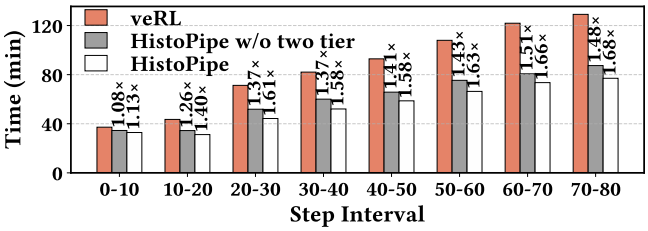


Figure 16. HistoPipe's improvements across steps. *Model: Math-14B. Max response length = 16K.*

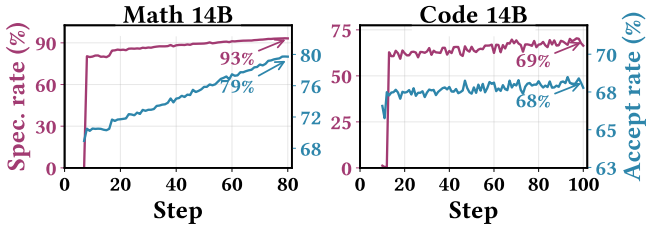


Figure 17. Speculation rate and acceptance rate.

policyness threshold is 1 (equal off-policyness with RhymeRL and veRL) and 8 (the max off-policyness recommended).

As Fig. 13 shows, RhymeRL outperforms veRL and AReaL on different model sizes (8B—32B), response lengths (8K/16K), tasks (Math/Code) and algorithms (DAPO/GRPO). Compared with veRL, RhymeRL improves the training throughput by up to 2.6x (1.9x on average for 8K max response length, and 2.3x on average for 16K max response length). This is primarily attributed to RhymeRL significantly reducing rollout time and effectively minimizing rollout bubbles. Compared with AReaL, when its off-policyness is 1, RhymeRL improves the training throughput by up to 2.1x (1.6x on average for 8K max response length, and 1.8x on average for 16K max response length). Although AReaL achieves continuous GPU utilization, it introduces rollout truncation and KV cache recomputation overhead. Leveraging historical similarity, RhymeRL achieves a more effective rollout pipeline. When AReaL's off-policyness is 8, RhymeRL improves the training throughput by up to 1.6x (1.3x on average for 8K max response length, and 1.4x on average for 16K max response length). RhymeRL outperforms AReaL (off-policyness = 8) as it significantly reduces rollout time through HistoSpec. Besides, RhymeRL does not change current RL paradigm.

7.2 Extended Studies

7.2.1 Ablation Study. We break down the improvements brought by the designs. As Fig. 14 shows, for the Math-14B LLM, HistoPipe (§ 6.2) achieves 1.43x (1.41x for Code-14B) throughput boosts, with Two-tier Scheduling (§ 6.4) further improving the throughput of the naive hybrid pipeline by 1.10x (1.08x for Code-14B). HistoSpec (§ 5) further improves the training throughput by 1.50x (1.40x for Code-14B).

7.2.2 Detailed Analysis of HistoSpec.

Improvement across steps. We present HistoSpec's rollout throughput gains in each step in Fig. 15. HistoSpec delivers up to 1.86x per-step rollout throughput gains, with progressive enhancement observed throughout training. This acceleration stems from: (1) increasing proportion of tokens generated by speculation and acceptance rate, and (2) growing decoding dominance due to extending response lengths, which exacerbates memory-bound limitations.

Speculation rate and acceptance rate. As shown in Fig. 17, the proportion of tokens generated from speculative decoding (speculation rate) increases as training continues. The acceptance rate, i.e., the proportion of accepted tokens among the draft, remains 65%–79% and also increases as training continues. Utilizing the reward-aware tree-based history management (§ 5.3) and AIMD-like token speculation (§ 5.4), HistoSpec achieves high speculation rate and acceptance rate without wasting much computational resources.

7.2.3 Detailed Analysis of HistoPipe.

Improvement across steps. We evaluate the improvements of HistoPipe with/without Two-tier Scheduling across steps. As Fig. 16 shows, HistoPipe shortens the training time per 10 steps by up to 1.68x, and the Two-tier Scheduling (§ 6.4) accelerates the naive hybrid pipe by up to 1.14x. As the training progresses, the response length distribution becomes more stable, and the improvement becomes more significant.

Migration Rate. We quantify the proportion of rollout samples migrated (intra-step or inter-step, § 6.3) as outliers across steps. We use 16 ranking groups for math models and 8 ranking groups for code models. As Fig. 19 shows, for math tasks, 2.2%—5.5% of samples undergo migration (1.6%—4.6% for code tasks), with the outlier proportion progressively decreasing during training. Such minor migration maintains the efficiency of HistoPipe, while preserving algorithmic integrity (§ 7.3) and introducing negligible overhead.

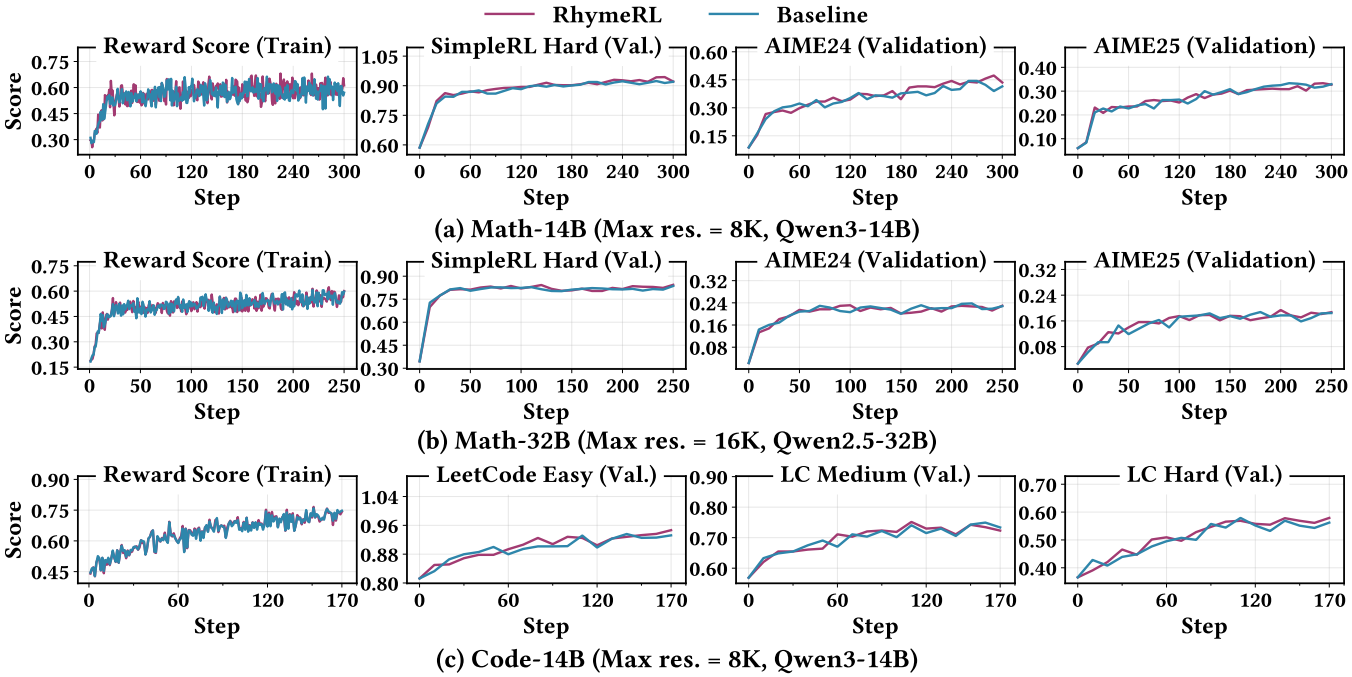


Figure 18. The end-to-end training curves and model evaluation (i.e. validation) scores. *The training set and validation set are disjoint. We report the average scores per step (training) or per dataset (validation).*

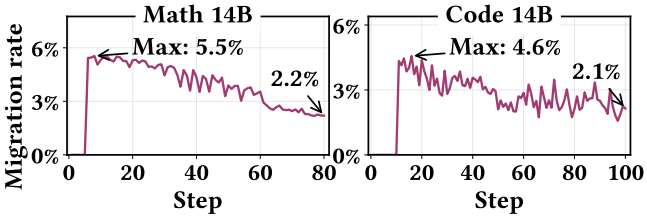


Figure 19. The proportion of samples migrated.

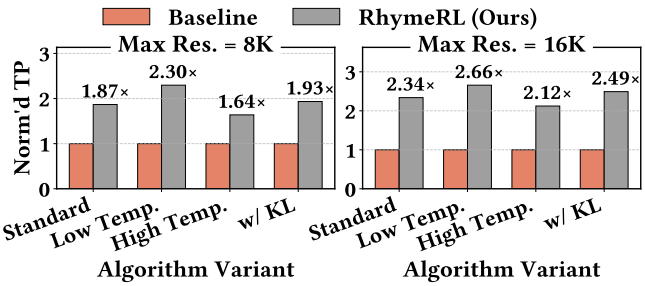


Figure 20. Improvements across algorithm settings (Math-14B). *The results (throughput) are normalized to the baseline.*

7.3 Accuracy and Algorithmic Integrity

We conduct full training of the Math-14B, Code-14B and Math-32B models using RhymeRL and veRL. During math training, we use DAPO-Math-17k [17, 65] as the training set, and evaluate the model every 10 steps, using three datasets (AIME24 [66], AIME25 [67], and SimpleRL Hard (MATH lv.3-5) [68, 69]) as the validation set. During code training, we use the training set of CodeR1-12K [70] as the training set, and evaluate the model using CodeR1-12K's testing set.

Fig.18 presents the per-step reward scores during the training processes, as well as the evaluation scores of the models on the validation datasets (per 10 steps). The results demonstrate that the training trajectory with RhymeRL aligns with that of veRL, and the model trained with RhymeRL delivers consistent end-to-end performance with that of veRL. Collectively, these findings confirm that RhymeRL preserves training integrity and final model accuracy.

7.4 Generality Studies

7.4.1 Generality for Algorithmic Variants. Algorithm configurations, especially temperature and KL penalty, can influence the prominence of historical similarity. The temperature controls the randomness of model outputs, and a higher temperature increases stochasticity. The KL penalty constrains policy updates and limits the divergence between old and new policies. We evaluate their effect on RhymeRL's performance[3], with the following settings:

- **Standard.** Temperature = 1 and disabling KL penalty.
- **Low temperature.** Temperature = 0.8, disabling KL penalty.
- **High temperature.** Temperature = 1.2, disabling KL penalty.
- **KL penalty.** Temperature = 1, enabling KL penalty.

As Fig. 20 shows, with lower temperature and KL penalty, the historical similarity becomes more pronounced, enhancing RhymeRL's effectiveness. Under higher temperature, increased output randomness diminishes the performance of

[3]Please note that, in current practice, it is a standard and largely unvaried configuration to set temperature = 1 (i.e., the original distribution) and disable the KL penalty [16, 17, 61], which is used in other experiments.

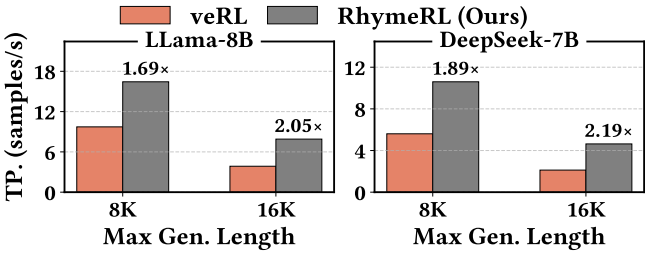


Figure 21. Improvements on different base models.

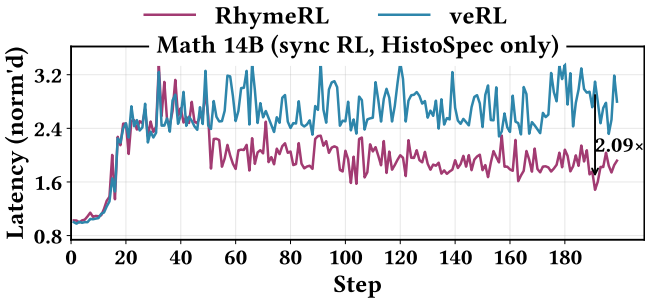


Figure 22. Efficiency of HistoSpec for sync RL (rollout latency per step). *The results are normalized to the baseline's first step. We use a dataset of 16K prompts.*

RhymeRL, yet it still significantly outperforms the baseline.

7.4.2 Generality for Other Model Structures. To evaluate the generality of RhymeRL across different models, we conduct experiments using Llama3.1-8B and DeepSeek-R1-Distill-Qwen-7B (DeepSeek-7B for short, detailed in Table.3). As Fig.21 shows, RhymeRL yields consistent improvements for different base models. The insights and designs of RhymeRL are grounded in general RL algorithms and workflows, ensuring its generality for different base models.

7.4.3 Effectiveness of HistoSpec for Synchronous RL. While asynchronous RL is widely adopted for large-scale training due to its high resource efficiency and scalability, synchronous RL remains essential for scenarios such as algorithm validation. RhymeRL also supports using HistoSpec in synchronous mode (i.e., HistoSpec only). As Fig.22 shows, it speeds up the rollout by up to 2.09x for Math-14B at 8K max response length, demonstrating its generality.

8 Discussion

8.1 HistoSpec vs. Model-based Speculation

Although model-based speculation methods can also accelerate LLM inference, they face significant challenges in LLM RL rollout, making HistoSpec outperform them.

Draft model adaptability and model evolution. Small LLMs cannot have peer reasoning capabilities as large LLMs [71]; therefore, SOTA methods utilize the target LLM's hidden states for draft generation [34–37]. However, they still face fundamental challenges in RL due to continuous model evolution. During RL, the LLMs are updated iteratively for thousands of steps. However, SOTA methods typically acquire draft models through distillation from the static model (Eagle1-3 [35–37]) or incremental fine-tuning of frozen base models (Medusa [34]), limiting their adaptability to the dynamically evolving LLMs.[4]

Performance and batch size. HistoSpec outperforms SOTA model-based methods, stemming from three key advantages: (1) ultra-low draft generation overhead (hundreds of CPU cycles vs. millisecond-level costs in model-based speculation), (2) zero GPU computation or HBM footprint for drafting, and (3) consistently high acceptance rates. SOTA methods like Eagle3 [37] suffer throughput degradation beyond a batch size of 64 despite using a single-layer transformer (also reported by Eagle3 [37] and vLLM [72]). In contrast, HistoSpec sustains significant acceleration even at extreme batch sizes. At step 80, automatic oversampling leads to batch sizes of 4,928 rollouts/TP for Math-32B and 2,176 rollouts/TP for Math-8B, where HistoSpec still accelerates rollout by 1.80x—1.86x.

[4] As DAPO [17] mentions, "during training the long-CoT reasoning model, the model distribution can diverge significantly from the initial model".

8.2 Accelerating the First Training Epoch

Some may be concerned that RhymeRL cannot accelerate the first epoch due to the absence of history. This can be addressed through the following approaches: (1) leveraging the multi-response nature of RL algorithms like GRPO/DAPO, and sampling several responses per prompt to serve as the history for the subsequent generations [73]; (2) pre-warming the dataset with fast, large-batch inference before formal training begins; and (3) reusing traces from previous training that runs on the same dataset.

9 Conclusion

We present RhymeRL, an LLM RL system leveraging historical similarity for efficiency optimization. As the first RL system leveraging speculative decoding to shorten rollout time, RhymeRL utilizes historical rollout sequences as draft sources. Leveraging historical distribution, it proposes distribution-aware scheduling to reduce rollout bubbles. It improves RL efficiency without compromising accuracy.

Acknowledgments

We thank the ASPLOS reviewers, Xingda Wei, and Rongxin Cheng for their support, feedback, and suggestions. This work was supported in part by the National Key Research & Development Program of China (No. 2023YFB3308501), the National Natural Science Foundation of China (No. 62432010, No. 62302300, No. 623B2074, No. 62472279), and the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM113).
Corresponding authors: Erhu Feng (fengerhu1@sjtu.edu.cn), and Dong Du (dd_nirvana@sjtu.edu.cn).

A Speculative Decoding: Algorithm and Proof of Distribution Preservation

A.1 Algorithm and Workflow

Let M_p denote the target model with distribution $p(x)$ and M_q denote the draft method with distribution $q(x)$. The speculative decoding process consists of three phases: *target forward*, *speculative sampling*, and *bonus token sampling*.

- **Input.** The draft tokens and their probabilities.
- **Phase 1: target model forward.** The target model processes all draft tokens in a single forward pass to obtain their logits and compute probabilities. This process is identical to a standard model forward pass.
- **Phase 2: speculative sampling.** The sampling process iterates through each draft token. For a token x, the process has three branches:
  - **Branch 1 (direct acceptance).** If x is the draft token, and $p(x) \geq q(x)$, x is always accepted (i.e., with probability 1).
  - **Branch 2 (probabilistic acceptance).** If x is the draft token, and $p(x) < q(x)$, x is accepted with probability $\frac{p(x)}{q(x)}$.
  - **Branch 3 (resampling).** If the draft token is rejected (whether x is the draft token or not), a new token is sampled from the residual distribution $p'(x)$. Let V be the vocabulary of all possible tokens. The variable $z \in V$ represents a specific token in the vocabulary.

$$\begin{aligned} p'(x) &= \text{norm}\left(\max(0, p(x) - q(x))\right) \\ &= \frac{\max(0, p(x) - q(x))}{\sum_{z \in V} \max(0, p(z) - q(z))} \end{aligned} \tag{1}$$

- **Phase 3: bonus token sampling.** If all draft tokens are accepted, an additional token is generated beyond the draft tokens, called the bonus token. In Phase 1, the target model not only computes the logits and probabilities for all draft tokens, but also generates the logits and probabilities for this bonus token in the same forward pass.

A.2 HistoSpec: Setting and Distribution Preservation

To rigorously demonstrate that HistoSpec does not alter the target model's output token distribution, we provide a dedicated mathematical proof for the sampling mechanism of HistoSpec. Please note that this proof is a special case of the general speculative decoding proof [22], which proves that for **any** distributions $p(x)$ and $q(x)$, the tokens sampled via speculative sampling from $p(x)$ and $q(x)$ are distributed identically to those sampled from $p(x)$ alone.

Specific settings. HistoSpec is much simpler than general speculative decoding, because the draft distribution $q(x)$ is one-hot. In other words, HistoSpec sets the probability of the token that it predicts to 1, and sets the probability of other tokens to 0. If the draft token is k, we have:

$$q(x) = \mathbb{I}(x = k) = \begin{cases} 1 & \text{if } x = k \\ 0 & \text{if } x \neq k \end{cases} \tag{2}$$

Consequently, the speculative sampling phase of HistoSpec (Phase 2) is simplified as follows:

Phase 2: speculative sampling. For a token x:

- **Branch 1 (direct acceptance).** Never happens, because $q(k) = 1$, and $p(k) < 1$.[5]
- **Branch 2 (probabilistic acceptance).** The draft token k is accepted with probability $p(k)/q(k) = p(k)/1 = p(k)$.
- **Branch 3 (resampling).** If the draft token is rejected, a new token is sampled. The new token is resampled from the residual distribution $p'(x)$.

From Equation 1, the unnormalized mass (numerator) for $p'(x)$ is $\max(0, p(x) - q(x))$.

- For $x = k$, since $p(k) \leq 1$, we have:

$$\max(0, p(k) - q(k)) = \max(0, p(k) - 1) = 0 \tag{3}$$

- For $x \neq k$:

$$\max(0, p(k) - q(k)) = \max(0, p(x) - 0) = p(x) \tag{4}$$

The normalization constant (denominator) is:

$$\sum_z \max(0, p(z) - q(z)) = 0 + \sum_{z \neq k} p(z) = 1 - p(k) \tag{5}$$

Thus, the resampled distribution is:

$$p'(x) = \begin{cases} 0 & \text{if } x = k \\ \frac{p(x)}{1 - p(k)} & \text{if } x \neq k \end{cases} \tag{6}$$

Notably, $p'(k) = 0$, which means that the resampling will never choose k.

Mathematical proof. For a token position with draft token k, we analyze $P(\text{output} = x)$ for two cases:

- **Case 1: The output token is the draft token ($x = k$).** This happens when speculative sampling accepts the draft token (with probability $p(k)$, Branch 2).[6]

$$P(\text{output} = k) = p(k) \tag{7}$$

- **Case 2: The output token is NOT the draft token ($x \neq k$).** This happens when speculative sampling rejects the draft token (with probability $1 - p(k)$, Branch 2), and resamples x from the residual distribution $p'(x)$ (Branch 3).

$$\begin{aligned} P(\text{output} = x) &= (1 - p(k)) \cdot p'(x) \\ &= (1 - p(k)) \cdot \frac{p(x)}{1 - p(k)} \\ &= p(x) \end{aligned} \tag{8}$$

We have shown that for any token x, whether it is the draft proposal k or not, $P(\text{output} = x) = p(x)$.

Therefore, HistoSpec strictly preserves the target probability distribution. **QED.**

This is also the case for n-gram [39, 56], SuffixDecoding [38], and other model-free drafting methods.

[5] Although it is theoretically possible that $p(k) = 1$, it is extremely rare in practice and can be covered by Branch 2.

[6] From Equation 6, Branch 3 (resampling) will never choose k if it is rejected.

References

[1] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] https://arxiv.org/abs/2501.12948

[2] Google. 2025. Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities. arXiv:2507.06261 [cs.CL] https://arxiv.org/abs/2507.06261

[3] 2025. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. https://ai.meta.com/blog/llama-4-multimodal-intelligence/.

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

[5] 2025. Introducing Claude 4. https://www.anthropic.com/news/claude-4.

[6] Rongxin Cheng, Kai Zhou, Xingda Wei, Siyuan Liu, Mingcong Han, Mingjing Ai, Yeju Zhou, Baoquan Zhong, Wencong Xiao, Rong Chen, and Haibo Chen. 2025. Fast LLM Post-training via Decoupled and Fastest-of-N Speculation. arXiv:2511.16193 [cs.DC] https://arxiv.org/abs/2511.16193

[7] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. arXiv:2402.03300 [cs.CL] https://arxiv.org/abs/2402.03300

[8] Yuxiang Zheng, Dayuan Fu, Xiangkun Hu, Xiaojie Cai, Lyumanshan Ye, Pengrui Lu, and Pengfei Liu. 2025. DeepResearcher: Scaling Deep Research via Reinforcement Learning in Real-world Environments. arXiv:2504.03160 [cs.AI] https://arxiv.org/abs/2504.03160

[9] Junde Wu, Jiayuan Zhu, Yuyuan Liu, Min Xu, and Yueming Jin. 2025. Agentic Reasoning: A Streamlined Framework for Enhancing LLM Reasoning with Agentic Tools. arXiv:2502.04644 [cs.AI] https://arxiv.org/abs/2502.04644

[10] Vignesh Prabhakar, Md Amirul Islam, Adam Atanas, Yao-Ting Wang, Joah Han, Aastha Jhunjhunwala, Rucha Apte, Robert Clark, Kang Xu, Zihan Wang, and Kai Liu. 2025. OmniScience: A Domain-Specialized LLM for Scientific Reasoning and Discovery. arXiv:2503.17604 [cs.AI] https://arxiv.org/abs/2503.17604

[11] Yifei Zhou, Song Jiang, Yuandong Tian, Jason Weston, Sergey Levine, Sainbayar Sukhbaatar, and Xian Li. 2025. SWEET-RL: Training Multi-Turn LLM Agents on Collaborative Reasoning Tasks. arXiv:2503.15478 [cs.LG] https://arxiv.org/abs/2503.15478

[12] Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. s1: Simple test-time scaling. arXiv:2501.19393 [cs.CL] https://arxiv.org/abs/2501.19393

[13] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. HybridFlow: A Flexible and Efficient RLHF Framework. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 1279–1297. https://doi.org/10.1145/3689031.3696075

[14] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Hao Hu, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Chao Hong, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Lijun Lu, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Xinjie Sun, Flood Sung, Heyi Tang, Jiawen Tao, Qifeng Teng, Chensi Wang, Dinglu Wang, Feng Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Xiaofei Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yangkun Zhang, Yizhi Zhang, Yongting

Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Huabin Zheng, Shaojie Zheng, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. 2025. Kimi K2: Open Agentic Intelligence. arXiv:2507.20534 [cs.LG] https://arxiv.org/abs/2507.20534

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

[16] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. 2025. AReaL: A Large-Scale Asynchronous Reinforcement Learning System for Language Reasoning. arXiv:2505.24298 [cs.LG] https://arxiv.org/abs/2505.24298

[17] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan 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, Mu Qiao, Yonghui Wu, and Mingxuan Wang. 2025. DAPO: An Open-Source LLM Reinforcement Learning System at Scale. arXiv:2503.14476 [cs.LG] https://arxiv.org/abs/2503.14476

[18] Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Y. Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Li Erran Li, Raluca Ada Popa, and Ion Stoica. 2025. DeepScaleR: Surpassing O1-Preview with a 1.5B Model by Scaling RL. https://pretty-radio-b75.notion.site/DeepScaleR-Surpassing-O1-Preview-with-a-1-5B-Model-by-Scaling-RL-19681902c1468005bed8ca303013a4e2. Notion Blog.

[19] Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. 2025. Group Sequence Policy Optimization. arXiv:2507.18071 [cs.LG] https://arxiv.org/abs/2507.18071

[20] Nai-Chieh Huang, Ping-Chun Hsieh, Kuo-Hao Ho, and I-Chen Wu. 2024. PPO-Clip Attains Global Optimality: Towards Deeper Understandings of Clipping. arXiv:2312.12065 [cs.LG] https://arxiv.org/abs/2312.12065

[21] Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. 2020. Why gradient clipping accelerates training: A theoretical justification for adaptivity. arXiv:1905.11881 [math.OC] https://arxiv.org/abs/1905.11881

[22] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 795, 13 pages.

[23] Y.R. Yang and S.S. Lam. 2000. General AIMD congestion control. In *Proceedings 2000 International Conference on Network Protocols*. 187–198. https://doi.org/10.1109/ICNP.2000.896303

[24] 2025. (veRL Pull Request) feat: accelerate rollout via model-free speculative decoding. https://github.com/volcengine/verl/pull/4535.

[25] OpenAI. 2024. OpenAI o1 System Card. arXiv:2412.16720 [cs.AI] https://arxiv.org/abs/2412.16720

[26] 2025. Introducing OpenAI o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/.

[27] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG] https://arxiv.org/abs/1707.06347

[28] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust Region Policy Optimization. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 37)*, Francis Bach and David Blei (Eds.). PMLR, Lille, France, 1889–1897. https://proceedings.mlr.press/v37/schulman15.html

[29] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 932–949. https://doi.org/10.1145/3620666.3651335

[30] Zhihao Zhang, Alan Zhu, Lijie Yang, Yihua Xu, Lanting Li, Phitchaya Mangpo Phothilimthana, and Zhihao Jia. 2024. Accelerating Iterative Retrieval-augmented Language Model Serving with Speculation. In *Forty-first International Conference on Machine Learning*. https://openreview.net/forum?id=CDnv4vg02f

[31] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating Large Language Model Decoding with Speculative Sampling. arXiv:2302.01318 [cs.CL] https://arxiv.org/abs/2302.01318

[32] Baohao Liao, Yuhui Xu, Hanze Dong, Junnan Li, Christof Monz, Silvio Savarese, Doyen Sahoo, and Caiming Xiong. 2025. Reward-Guided Speculative Decoding for Efficient LLM Reasoning. arXiv:2501.19324 [cs.CL] https://arxiv.org/abs/2501.19324

[33] Yingpeng Du, Tianjun Wei, Zhu Sun, and Jie Zhang. 2025. Reinforcement Speculative Decoding for Fast Ranking. arXiv:2505.20316 [cs.AI] https://arxiv.org/abs/2505.20316

[34] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads. arXiv:2401.10774 [cs.LG] https://arxiv.org/abs/2401.10774

[35] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2025. EAGLE: Speculative Sampling Requires Rethinking Feature Uncertainty. arXiv:2401.15077 [cs.LG] https://arxiv.org/abs/2401.15077

[36] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. EAGLE-2: Faster Inference of Language Models with Dynamic Draft Trees. arXiv:2406.16858 [cs.CL] https://arxiv.org/abs/2406.16858

[37] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2025. EAGLE-3: Scaling up Inference Acceleration of Large Language Models via Training-Time Test. arXiv:2503.01840 [cs.CL] https://arxiv.org/abs/2503.01840

[38] Gabriele Oliaro, Zhihao Jia, Daniel Campos, and Aurick Qiao. 2025. SuffixDecoding: Extreme Speculative Decoding for Emerging AI Applications. arXiv:2411.04975 [cs.CL] https://arxiv.org/abs/2411.04975

[39] Apoorv Saxena. 2023. Prompt Lookup Decoding. https://github.com/apoorvumang/prompt-lookup-decoding/

[40] Nan Yang, Tao Ge, Liang Wang, Binxing Jiao, Daxin Jiang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023. Inference with Reference: Lossless Acceleration of Large Language Models. arXiv:2304.04487 [cs.CL] https://arxiv.org/abs/2304.04487

[41] 2025. (vLLM official blog) How Speculative Decoding Boosts vLLM Performance by up to 2.8x. https://blog.vllm.ai/2024/10/17/spec-decode.html.

[42] Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. 2025. ReaL: Efficient RLHF Training of Large Language Models with Parameter Reallocation. In *Eighth Conference on Machine Learning and Systems*. https://openreview.net/forum?id=yLU1zRf95d

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

[44] Zhixin Wang, Tianyi Zhou, Liming Liu, Ao Li, Jiarui Hu, Dian Yang,

Jinlong Hou, Siyuan Feng, Yuan Cheng, and Yuan Qi. 2025. DistFlow: A Fully Distributed RL Framework for Scalable and Efficient LLM Post-Training. arXiv:2507.13833 [cs.DC] https://arxiv.org/abs/2507.13833

[45] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, and Xin Jin. 2025. Optimizing RLHF Training for Large Language Models with Stage Fusion. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 489–503. https://www.usenix.org/conference/nsdi25/presentation/zhong

[46] 2025. NeMo: A scalable generative AI framework built for researchers and developers working on Large Language Models, Multimodal, and Speech AI. https://github.com/NVIDIA/NeMo.

[47] Jian Hu, Jason Klein Liu, Haotian Xu, and Wei Shen. 2025. REINFORCE++: An Efficient RLHF Algorithm with Robustness to Both Prompt and Reward Models. arXiv:2501.03262 [cs.CL] https://arxiv.org/abs/2501.03262

[48] Weixun Wang, Shaopan Xiong, Gengru Chen, Wei Gao, Sheng Guo, Yancheng He, Ju Huang, Jiaheng Liu, Zhendong Li, Xiaoyang Li, Zichen Liu, Haizhou Zhao, Dakai An, Lunxi Cao, Qiyang Cao, Wanxi Deng, Feilei Du, Yiliang Gu, Jiahe Li, Xiang Li, Mingjie Liu, Yijia Luo, Zihe Liu, Yadao Wang, Pei Wang, Tianyuan Wu, Yanan Wu, Yuheng Zhao, Shuaibing Zhao, Jin Yang, Siran Yang, Yingshui Tan, Huimin Yi, Yuchi Xu, Yujin Yuan, Xingyao Zhang, Lin Qu, Wenbo Su, Wei Wang, Jiamang Wang, and Bo Zheng. 2025. Reinforcement Learning Optimization for Large-Scale Learning: An Efficient and User-Friendly Scaling Library. arXiv:2506.06122 [cs.LG] https://arxiv.org/abs/2506.06122

[49] 2025. (veRL) One Step Off Policy Async Trainer. https://verl.readthedocs.io/en/latest/advance/one_step_off.html.

[50] Bingxiang He, Zekai Qu, Zeyuan Liu, Yinghao Chen, Yuxin Zuo, Cheng Qian, Kaiyan Zhang, Weize Chen, Chaojun Xiao, Ganqu Cui, Ning Ding, and Zhiyuan Liu. 2025. JustRL: Scaling a 1.5B LLM with a Simple RL Recipe. arXiv:2512.16649 [cs.CL] https://arxiv.org/abs/2512.16649

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

[52] Zhenghai Xue, Longtao Zheng, Qian Liu, Yingru Li, Xiaosen Zheng, Zejun Ma, and Bo An. 2025. SimpleTIR: End-to-End Reinforcement Learning for Multi-Turn Tool-Integrated Reasoning. arXiv:2509.02479 [cs.LG] https://arxiv.org/abs/2509.02479

[53] 2025. Trace of DAPO Training. https://wandb.ai/verl-org/DAPO%20Reproduction%20on%20verl. (2025).

[54] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[55] 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. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 62557–62583. https://proceedings.neurips.cc/paper_files/paper/2024/file/724be4472168f31ba1c9ac630f15dec8-Paper-Conference.pdf

[56] 2025. (vLLM documentation) vLLm Speculative Decoding. https://docs.vllm.ai/en/latest/features/spec_decode.html.

[57] Peter Weiner. 1973. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973) (SWAT '73)*. IEEE Computer Society, USA, 1–11. https://doi.org/10.1109/SWAT.1973.13

[58] E. Ukkonen. 1995. On-line construction of suffix trees. *Algorithmica* 14, 3 (Sept. 1995), 249–260. https://doi.org/10.1007/BF01206331

[59] 2024. cgroups - Linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html.

[60] DeepSeek-AI. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[61] 2025. veRL: Volcano Engine Reinforcement Learning for LLMs. https://github.com/volcengine/verl.

[62] Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] https://arxiv.org/abs/2412.15115

[63] AI @ Meta Llama Team. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[64] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 444–456.

[65] 2025. DAPO Math 17k dataset. https://huggingface.co/datasets/BytedTsinghua-SIA/DAPO-Math-17k.

[66] 2024. Dataset card for American Invitational Mathematics Examination (AIME) 2024. https://huggingface.co/datasets/math-ai/aime24.

[67] 2025. Dataset card for American Invitational Mathematics Examination (AIME) 2025. https://huggingface.co/datasets/math-ai/aime25.

[68] Weihao Zeng, Yuzhen Huang, Qian Liu, Wei Liu, Keqing He, Zejun Ma, and Junxian He. 2025. SimpleRL-Zoo: Investigating and Taming Zero Reinforcement Learning for Open Base Models in the Wild. arXiv:2503.18892 [cs.LG] https://arxiv.org/abs/2503.18892

[69] 2025. SimpleRL-Zoo Math dataset. https://huggingface.co/datasets/hkust-nlp/SimpleRL-Zoo-Data.

[70] Jiawei Liu and Lingming Zhang. 2025. Code-R1: Reproducing R1 for Code with Reliable Rewards. https://github.com/ganler/code-r1. (2025).

[71] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. arXiv:2001.08361 [cs.LG] https://arxiv.org/abs/2001.08361

[72] 2025. (vLLM issue) vLLM Eagle performance is worse than expected. https://github.com/vllm-project/vllm/issues/9565.

[73] Ruoyu Qin, Weiran He, Weixiao Huang, Yangkun Zhang, Yikai Zhao, Bo Pang, Xinran Xu, Yingdi Shan, Yongwei Wu, and Mingxing Zhang. 2025. Seer: Online Context Learning for Fast Synchronous LLM Reinforcement Learning. arXiv:2511.14617 [cs.DC] https://arxiv.org/abs/2511.14617