RACER: Retrieval-Augmented Contextual Rapid Speculative Decoding

Zihong Zhang[1], Zuchao Li[1]*, Lefei Zhang[2], Ping Wang[3], Hai Zhao[4],
[1]School of Artificial Intelligence, Wuhan University, Wuhan, China
[2]School of Computer Science, Wuhan University, Wuhan, China
[3]School of Information Management, Wuhan University, Wuhan, China
[4]School of Computer Science, Shanghai Jiao Tong University, China
{zhangzihong, zcli-charlie, zhanglefei, wangping}@whu.edu.cn
zhaohai@cs.sjtu.edu.cn

Abstract

Autoregressive decoding in Large Language Models (LLMs) generates one token per step, causing high inference latency. Speculative decoding (SD) mitigates this through a guess-and-verify strategy, but existing training-free variants face trade-offs: retrieval-based drafts break when no exact match exists, while logits-based drafts lack structural guidance. We propose **RACER** (**R**etrieval-**A**ugmented **C**ont**e**xtual **R**apid Speculative Decoding), a lightweight and training-free method that integrates retrieved exact patterns with logit-driven future cues. This unification supplies both reliable anchors and flexible extrapolation, yielding richer speculative drafts. Experiments on Spec-Bench, HumanEval, and MGSM-ZH demonstrate that RACER consistently accelerates inference, achieving more than $2\times$ speedup over autoregressive decoding, and outperforms prior training-free methods, offering a scalable, plug-and-play solution for efficient LLM decoding. Our source code is available at https://github.com/hkr04/RACER.

1 Introduction

Large Language Models (LLMs) such as GPT (OpenAI, 2025), LLaMA (Team, 2024), and Qwen (Bai et al., 2023) have achieved remarkable success across diverse natural language processing tasks. However, their autoregressive decoding paradigm, which generates one token per step, fundamentally limits inference efficiency. The sequential dependency causes inference latency to scale linearly with sequence length and model size, creating a key bottleneck for real-world deployment.

Speculative Decoding (SD) has emerged as a promising approach to address this challenge. By adopting a *guess-and-verify* strategy, SD enables multiple tokens to be proposed and verified in parallel, achieving acceleration without sacrificing output quality. Existing methods fall into two categories. Model-based approaches rely on lightweight auxiliary models – either separately trained (Cai et al., 2024; Zhang et al., 2025; Li et al., 2024b,a, 2025; Shi et al., 2026) or inherited from smaller variants of the same model family (Leviathan et al., 2023; Liu et al., 2025a) – to generate draft tokens, at the cost of additional memory, training, and integration overhead. Model-free approaches, in contrast, construct draft tokens directly from signals available during inference. Among model-free approaches, most are retrieval-based, leveraging exact token sequence matches from static corpora or dynamically generated contexts (Saxena, 2023; He et al., 2023). Recent work further exploits the predictive power of last logit (Liu et al., 2025b), or recycles candidate logits (Luo et al., 2025), showing that LLMs inherently encode richer cues for near-future tokens than previously assumed.

Despite these advances in model-free methods, two key limitations remain. First, retrieval-based methods depend on exact token matching, which breaks down when no continuation can be directly aligned. Second, logits-based methods are restricted to last-step or self-drafted candidates and lack external structural guidance, making it difficult to extrapolate toward more suitable tokens. As a result, their predictions tend to be narrow in scope and suboptimal in quality.

To address these limitations, we propose **RACER** (**R**etrieval-**A**ugmented **C**ont**e**xtual **R**apid Speculative Decoding), a **plug-and-play, training-free** method that unifies the strengths of both paradigms. Retrieval provides **seen information** through exact pattern matches, offering structural guidance, while logits supply **unseen information**, enabling extrapolation beyond strict matches. By augmenting logit predictions with retrieval signals, RACER generates richer and

*Corresponding author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 19962–19988

more accurate speculative drafts. In this way, retrieval functions not as an independent generator but as structural guidance that empowers logits to hypothesize plausible continuations beyond their immediate horizon.

We conduct comprehensive experiments across general benchmark Spec-Bench (Xia et al., 2024), code generation benchmark HumanEval (Chen et al., 2021) and Chinese math reasoning benchmark MGSM-ZH (Shi et al., 2022). Overall, we demonstrate that RACER provides a lightweight, training-free SD solution that effectively leverages complementary seen and unseen information via a unified cache-like module, delivering consistent inference acceleration with stable memory usage.

2 Background

Speculative Decoding Speculative Decoding (SD) typically proceeds in two phases: a *drafting phase* and a *verification phase*.

Given a prefix $\mathbf{x}$, a lightweight **draft model** M_q generates γ candidate tokens $\tilde{x}_1, \ldots, \tilde{x}_\gamma$. These tokens, together with the prefix, are then passed to the **target model** M_p, which produces logits $p_1, \ldots, p_\gamma$. Each draft token $\tilde{x}_i$ is verified by comparing its probability under M_p with that under M_q (Leviathan et al., 2023; Chen et al., 2023):

$$\alpha_i = \begin{cases} 1 & \text{if } p_i[\tilde{x}_i] \geq q_i[\tilde{x}_i], \\ \dfrac{p_i[\tilde{x}_i]}{q_i[\tilde{x}_i]} & \text{otherwise.} \end{cases} \tag{1}$$

If $\tilde{x}_i$ is accepted (with probability α_i), it is appended to the sequence; otherwise, $\tilde{x}_i, \ldots, \tilde{x}_\gamma$ are discarded, and the SD step terminates early. The next iteration then resumes from the last accepted prefix, using the last accepted logits p_{i-1} of the target model to resample x_i as the new continuation token. This guarantees that every iteration produces at least one token, while leveraging accepted drafts whenever possible to accelerate generation.

Regardless of whether greedy or nucleus sampling is employed, validation always leverages the logits from the target model. In expectation, a single SD step can advance by up to $\gamma + 1$ tokens, significantly reducing the number of target model invocations compared to standard autoregressive decoding.

Retrieval-based Methods Retrieval-based SD methods bypass the draft model M_q and instead rely on pattern matching within the token sequence. PLD (Saxena, 2023), for example, stores past n-grams together with their succeeding m-grams as predictions. This method is simple and effective in pattern-repeating scenarios such as code generation, but can only propose a single continuation at a time. Moreover, because pattern matches are sparse and fail to capture the full diversity of target model outputs, PLD is constrained to specific domains and cannot generalize broadly.

Tree Attention The standard *guess-and-verify* scheme assumes a linear draft sequence. *Tree attention* (Miao et al., 2024; Cai et al., 2024) generalizes this by allowing the draft model to propose a branching tree of candidates. During verification, the target model processes all nodes in parallel, with position encodings set by depth and attention masks restricting each node to its ancestors:

$$\begin{aligned} &\text{pos}[i] = \text{pos}[\text{parent}(i)] + 1, \\ &\text{mask}[i, j] = \mathbb{1}[\, j = i \text{ or } j \in \text{ancestor}(i)\,]. \end{aligned} \tag{2}$$

This transforms SD into a branching search process, enabling higher parallelism and more effective utilization of the target model when multiple plausible continuations exist.

For model-based methods, Medusa (Cai et al., 2024) attaches multiple additional language model (LM) heads to the top layer, each predicting draft tokens for different tree depths. EAGLE-3 (Li et al., 2025) further integrates low-, mid-, and high-level features of the target model, with its core structured as a Transformer decoder layer. For model-free methods, REST (He et al., 2023) constructs a suffix array to identify the longest suffix match and expands the matched continuation into a trie, while SAM Decoding (Hu et al., 2025) employs both dynamic and static suffix automata to capture contextual as well as pre-built suffix patterns, providing more flexible retrieval-guided expansions.

3 Methodology

In this section, we introduce our approach in three parts: *Logits Tree*, *Retrieval Tree with LRU (Least Recently Used) eviction*, and their integration strategy.

3.1 Logits Tree

First Step beyond Next-Token We examine two *logit-reuse* strategies when extending the tree

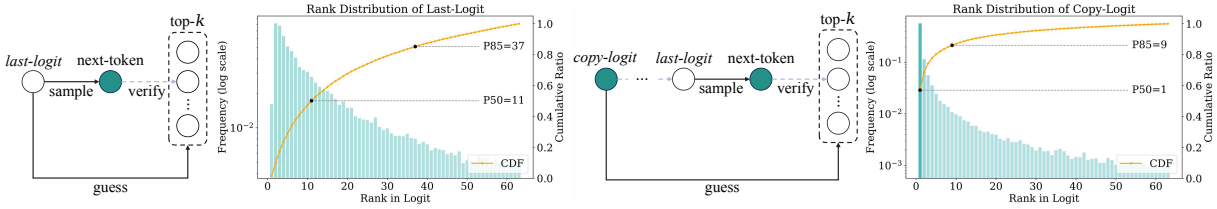

Figure 1: The *last-logit* node (white) produces both the next-token sample and the draft tokens immediately after it. The *copy-logit* node (green) marks the same token ID as the next-token, whose logit is reused to approximate the next-token's logits when generating subsequent draft tokens.

beyond the next token. Let the verified prefix be $x_{<t}$, and let the target model produce logits $\mathbf{z}_t = f(x_{<t})$. The next token is sampled as $x_t \sim \text{Softmax}(\mathbf{z}_t)$ without computing $\mathbf{z}_{t+1}$. To approximate the unknown future logits, it's natural to construct a surrogate distribution $\tilde{\mathbf{z}}_{t+1}$ and expand its top-k tokens as candidates for $x_{t+1} \sim \text{Softmax}(\mathbf{z}_{t+1})$:

$$\tilde{x}_{t+1}^{(k)} = \text{TopK}_k\big(\text{Softmax}(\tilde{\mathbf{z}}_{t+1})\big).$$

We define two strategies: *last-logit* with $\tilde{\mathbf{z}}_{t+1} = \mathbf{z}_t$, and *copy-logit* with $\tilde{\mathbf{z}}_{t+1} = \mathbf{z}_{i+1}$ where $i < t$ is the nearest index such that $x_i = x_t$.

The *last-logit* strategy reuses the logit distribution from which the next-token was sampled to expand all of its children candidates, assuming local smoothness in the token space. The *copy-logit* strategy instead reuses the logit from the most recent occurrence of the same token, assuming that identical vocabulary tokens tend to preserve similar semantic tendencies when appearing in comparable contexts.

To evaluate these strategies, we conducted experiments on Spec-Bench using Vicuna-7B, OpenPangu-7B and Qwen3-1.7B under greedy decoding. For each speculative step, we selected the top-63 tokens from the corresponding logits to form the first layer, and measured their effectiveness by the mean accepted tokens (MAT) and the distribution of accepted ranks (from 1 to 63). Since all models exhibited similar trends, we report only the results of Vicuna-7B here. Additional results and discussions can be found in Appendix F.

The MAT values of *last-logit* and *copy-logit* are 1.57 and 1.87, respectively, indicating that *copy-logit* achieves higher acceptance rate. As shown in Figure 1, the *copy-logit* strategy also exhibits a pronounced heavy-tail property: its accepted tokens concentrate strongly at the top ranks, with rank-1 alone accounting for more than 50% of accepted cases. For *copy-logit*, the 50th and 85th percentile accepted ranks are 1 and 9, compared with 11 and 37 for *last-logit*. These results demonstrate that *copy-logit* provides a sharper and more reliable distribution for speculative expansion, and we therefore adopt it as the basic expansion strategy of Logits Tree.

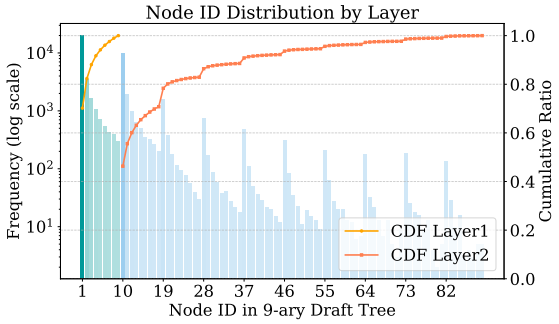

Figure 2: Analogy experiments with a fixed 9-ary draft tree of height 3.

k-ary Analogy and Pruning Motivated by the heavy-tail property observed in Figure 1, we next extend the expansion recursively to deeper layers. Two principles guide this design: (i) the 85th percentile rank is around 9, indicating that useful candidates concentrate in the head of the distribution; and (ii) because SD proceeds prefix-wise, the breadth of any child node should not exceed that of the root expansion.

To further examine the implications of the above principles and to explore how to prune the draft tree effectively, we conducted analogy experiments with a fixed 9-ary draft tree of height 3. This tree contains $1 + 9 + 9^2 = 91$ nodes, indexed from 0 to 90. Node 0 is the root corresponding to the next-token position; nodes 1-9 form the first layer; and nodes 10-90 form the second layer. Under this level-order indexing scheme, each node $i > 0$ has its parent given by $\text{parent}(i) = \lfloor \frac{i-1}{k} \rfloor$.

As illustrated in Figure 2, the second-layer chil-

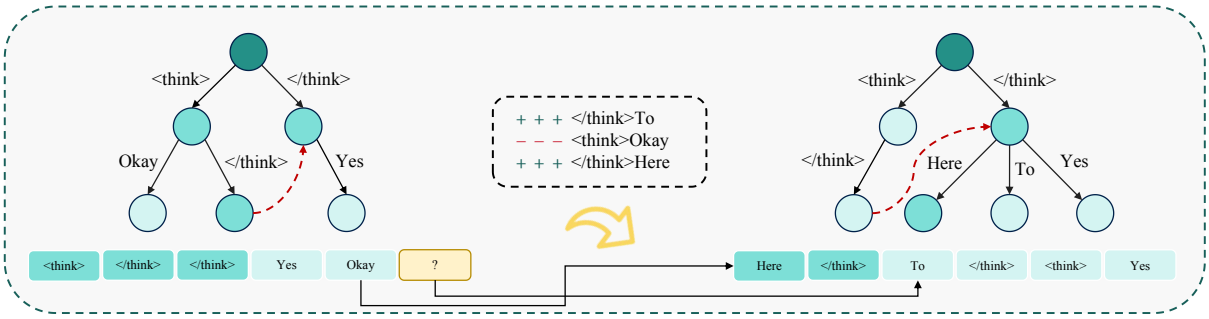

Figure 3: Illustration of the LRU-based eviction strategy in RACER's retrieval automaton. Solid black edges denote standard trie transitions, and dashed red edges denote failure links. Yellow nodes represent unallocated states. Green nodes indicate allocated states, where darker color corresponds to more recent usage. The example demonstrates how inserting the 2-gram `[</think>, To]` creates a new node `To`, and how LRU leaf node `Okay` is evicted and replaced with `Here` when the capacity is reached with the new 2-gram `[</think>, Here]`.

dren of node 1 exhibit a cumulative trend resembling that of the first layer, albeit slightly slower. For children of parents with larger IDs, the growth further slows and the total volume is nearly halved. The distribution remains front-loaded. Compared with *copy-logit*, the MAT increases from 1.87 to 2.34.

To capture this behavior, we define the breadth allocation in the Logits Tree for child nodes as

$$b_{\text{child}(i,j)} = \max\left(1, \left\lfloor \frac{b_i}{2^{j+[i\neq 0]}} \right\rfloor\right), \quad j = 0, 1, \ldots, b_i - 1. \tag{3}$$

where b_i is the breadth of the parent node and j is the child index. Specifically, nodes at the first layer start with the maximum breadth, while deeper layers inherit half of their parent's breadth. This design ensures that the upper part of the Logits Tree expands more aggressively, while deeper layers are progressively pruned. Given a specific draft capacity, the Logits Tree then expands in a breadth-first manner according to this allocation rule. Figure 9 in the Appendix illustrates this process using a 4-ary example, showing: (i) the original k-ary indexing, and (ii) the pruned Logits Tree structure.

3.2 Retrieval Tree with LRU Eviction

Approaches such as SAM Decoding (Hu et al., 2025) and LogitSpec (Liu et al., 2025b) indicate that explicit retrieval drafts can complement the logits and improve acceptance rates. Motivated by these findings, we aim to design an efficient retrieval structure that exploits repeated patterns in the context. Classical indexing structures such as suffix arrays (Manber and Myers, 1993) or suffix automata (Blumer et al., 1984) provide efficient substring matching, but they grow proportionally with the context length and lack a natural mechanism to discard obsolete states. This is undesirable in language modeling, where the distribution of substrings follows a Zipf-like long-tail law, implying that many substrings have little utility and can be safely evicted. To balance efficiency and adaptivity, we propose to use an Aho–Corasick (AC) automaton (Aho and Corasick, 1975) to maintain an n-gram-based Retrieval Tree. Unlike a plain n-gram trie, the AC automaton supports failure links that facilitate fast state transitions and naturally enrich draft diversity. See Appendix E.2 for a brief introduction to the AC automaton.

Transition Rule As shown in Figure 3, we incorporate an **LRU-style (Least Recently Used) eviction mechanism** into the AC automaton so that infrequent n-gram patterns are pruned while new ones from the incoming context are continually incorporated. Each time a state is visited – either through a valid transition or via a failure-link fallback – that state is marked as "touched". Importantly, when backtracking with failure links, all of its ancestor (prefix) states are also necessarily touched. This ensures that prefix nodes always remain equal or more recent than their descendants. For example, when the token `Yes` follows `[<think>, </think>]` and there is no direct transition, the automaton backtracks along its failure link to state `[</think>]` and then transitions to `[</think>, Yes]`; all states along this fallback and transition path are touched.

Update and Eviction Rule During decoding, newly observed n-grams are incrementally inserted into the automaton. Insertion follows the

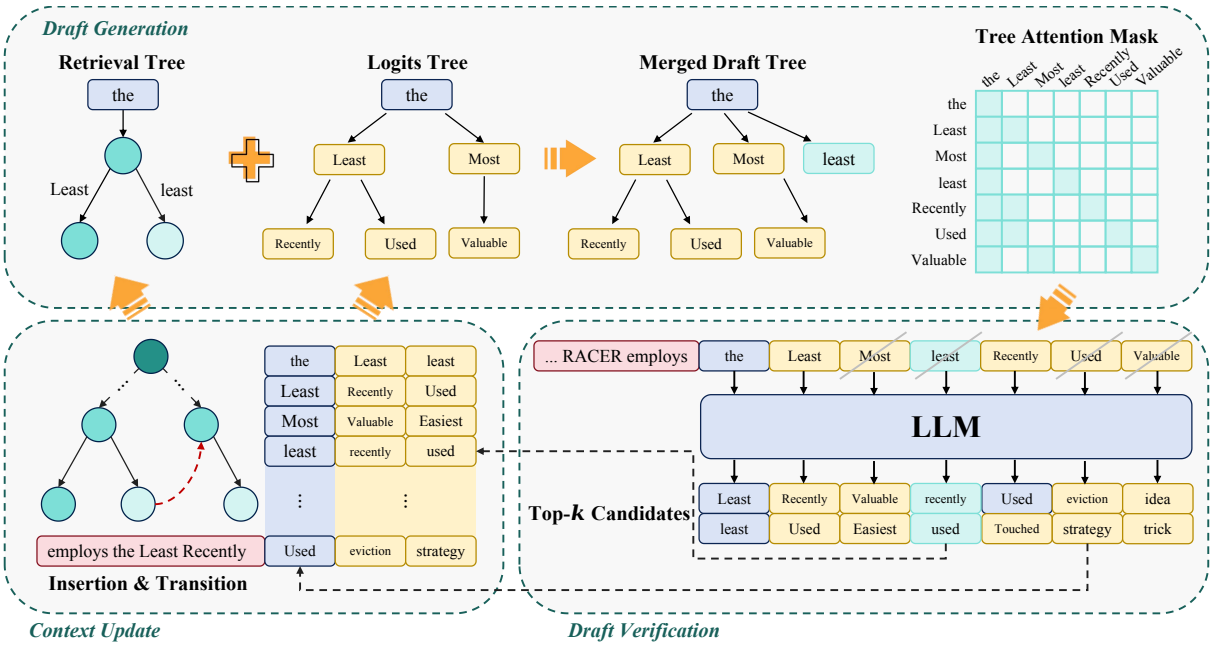

Figure 4: Overview of RACER. At each decoding step, the AC automaton accepts the next token and identifies border nodes with depth ≥ 2, from which the globally most frequent children are selected as retrieval candidates. If retrieval nodes do not fill the draft capacity C, the remaining slots are assigned to Logits Tree expansion (Eq. 3). Verified n-grams are inserted into the automaton, while the logits adjacency matrix is refreshed with the newly generated logits.

transition path whenever possible; if a transition does not exist, a new node is allocated either from an empty slot or by reusing an LRU leaf node. When the automaton reaches its predefined capacity, the LRU *leaf* node is evicted (e.g., the leaf `Okay` in Figure 3). All nodes are managed using a hash table and a doubly linked list, enabling $\mathcal{O}(1)$ updates and eviction. Failure links are updated lazily: they are rebuilt only once at the end of the prefilling phase. Before the rebuild, the newly added portion of the structure temporarily behaves as a standard trie without failure links.

Expansion Rule We consider all match states (borders) whose matched depth is at least 2. For each border, we take the sub-trie rooted at that state, collect all outgoing continuations, pool them across borders, and select the globally most frequent top-k continuation states to expand the Retrieval Tree. Detailed examples of the expansion procedure based on the frequencies are presented in Appendix E.4.

3.3 Integration Strategy

Given a fixed speculative capacity C, retrieval-based candidates are first generated according to the expansion rule in Section 3.2, while the remaining capacity is allocated to the Logits Tree via breadth-first expansion (Eq. 3). Since retrieval candidates are sparse but structurally reliable, we retain only the most confident ones and leave the remaining budget for logits-based exploration through the top-k adjacency matrix.

Importantly, retrieval not only complements the current speculation, but also provides stronger cues for upcoming tokens than logits alone. Because retrieval captures repeated patterns in closer contexts, it guides the logits distribution toward sharper predictions and mitigates error accumulation in speculative expansions.

The two candidate sets are finally merged into a unified draft tree through a trie-based union, and verified by the target model under the *guess-and-verify* scheme. This hybrid design enables RACER to exploit both *seen information* from retrieval and *unseen speculation* from logits, achieving higher acceptance rates with controlled memory usage. The overall workflow is presented in Figure 4; further implementation details can be found in Section E.

4 Experiments

4.1 Experimental Setup

Following prior work (Luo et al., 2025), we focus on greedy decoding with batch size 1 and maxi-

Table 1: Results on Spec-Bench, HumanEval, and MGSM-ZH with **Vicuna** and **LLaMA3.1** target models. Reported in MAT and speedup ratio. Best results are in **bold**, suboptimal results are underlined.

Model	Method	Spec-Bench		HumanEval		MGSM-ZH		Average	
		MAT	Speedup	MAT	Speedup	MAT	Speedup	MAT	Speedup
Vicuna 7B	PLD	1.71	1.50	1.58	1.40	2.57	2.27	1.95	1.87
	REST	1.82	1.45	2.06	1.71	1.29	1.06	1.72	1.41
	LogitSpec	2.34	1.77	2.22	1.66	3.55	2.67	2.70	2.03
	TR	2.76	2.06	2.83	2.17	3.00	2.30	2.86	2.18
	RACER	**3.00**	**2.21**	**3.11**	**2.29**	**3.71**	**2.77**	**3.27**	**2.42**
Vicuna 13B	PLD	1.65	1.41	1.59	1.43	2.45	2.11	1.90	1.65
	REST	1.82	1.44	2.07	1.71	1.31	1.07	1.73	1.41
	LogitSpec	2.32	1.73	2.23	1.77	3.44	2.72	2.66	2.07
	TR	2.79	1.99	2.83	2.08	3.05	2.22	2.89	2.10
	RACER	**2.95**	**2.25**	**3.09**	**2.42**	**3.64**	**2.83**	**3.23**	**2.50**
Vicuna 33B	PLD	1.33	1.03	1.64	1.48	2.18	1.97	1.72	1.49
	REST	1.81	1.54	1.98	1.72	1.32	1.17	1.70	1.48
	LogitSpec	2.32	1.73	2.35	1.92	2.96	2.44	2.54	2.03
	TR	2.63	1.83	2.79	2.05	2.83	2.10	2.75	1.99
	RACER	**2.74**	**2.20**	**3.16**	**2.58**	**3.36**	**2.77**	**3.09**	**2.52**
LLaMA3.1 8B	EAGLE-3	**3.76**	**2.51**	**4.41**	**3.06**	1.54	1.06	**3.24**	2.21
	RACER	2.82	2.41	3.22	2.87	**3.33**	**2.89**	3.12	**2.72**

mum output length 1024. We report the following metrics: **Mean Accepted Tokens (MAT)**: the average number of tokens confirmed in a single speculative decoding step; **Speedup Ratio**: relative performance compared with HuggingFace's implementation of autoregressive decoding.

For all experiments, we use the following default hyperparameters unless otherwise specified. For the Logits Tree, the maximum breadth is set to 8. For the Retrieval Tree, we maintain up to 10,000 nodes with an n-gram length of 10. The draft size of each decoding step is 64 as suggested in Medusa (Cai et al., 2024). These values are chosen based on preliminary analyses, and we further demonstrate in Appendix D.4 that our method is robust with respect to these hyperparameters. Further details can be found in Appendix A.

Target Models and Datasets We utilize instruct model Vicuna (Chiang et al., 2023) at three scales: 7B, 13B, and 33B, where the 7B and 13B models use version 1.5 and the 33B model uses version 1.3. In addition, we include LLaMA3.1-8B. We further incorporate reasoning models: OpenPangu (Chen et al., 2025) at 7B and Qwen3 (Team, 2025) at the corresponding 8B, 14B, and 32B scales. We evaluate on three benchmarks: **Spec-Bench** (Xia et al., 2024), **HumanEval** (Chen et al., 2021), and **MGSM-ZH** (Cobbe et al., 2021; Shi et al., 2022). Spec-Bench covers diverse scenarios including Multi-turn Conversation (MT), Translation (Trans), Summarization (Sum), Question Answering (QA), Mathematical Reasoning (Math), and Retrieval-Augmented Generation (RAG). HumanEval is a widely used benchmark for code generation. MGSM-ZH is the Chinese counterpart of GSM8K. Following the multilingual evaluation setup used in PEARL (Liu et al., 2025a), we adopt MGSM-ZH to assess non-English mathematical reasoning in a language where OpenPangu and Qwen3 exhibit stable performance, while avoiding languages for which Vicuna generates unstable outputs. Together, these benchmarks cover general-purpose, domain-specific, and cross-lingual reasoning tasks.

Baselines We compare RACER against two retrieval-based methods (PLD and REST), two logits-involved methods (Token Recycling and LogitSpec), and the state-of-the-art model-based method EAGLE-3. **PLD** (Saxena, 2023) stores past n-grams as sequential keys and their succeeding m-grams as predicted values. **REST** (He et al., 2023) builds a suffix array over the training set to locate the longest suffix match, then expands the matched continuation into a trie. **Token Recycling (TR)** (Luo et al., 2025) maintains a top-k adjacency matrix from token logits and extends it

Table 2: Results on Spec-Bench, HumanEval, and MGSM-ZH with **OpenPangu** and **Qwen3** target models. Reported in MAT and speedup ratio. Best results are in **bold**, suboptimal results are underlined.

Model	Method	Spec-Bench		HumanEval		MGSM-ZH		Average	
		MAT	Speedup	MAT	Speedup	MAT	Speedup	MAT	Speedup
OpenPangu 7B	PLD	1.48	1.37	1.44	1.27	1.53	1.41	1.47	1.35
	RACER	**2.47**	**1.99**	**2.65**	**2.12**	**2.77**	**2.26**	**2.63**	**2.12**
Qwen3 8B	PLD	1.52	1.35	1.52	1.41	1.69	1.52	1.58	1.43
	EAGLE-3†	**3.46**	**2.14**	**3.84**	**2.44**	1.41	0.86	**2.90**	1.81
	RACER	2.73	2.13	2.79	2.24	**2.95**	**2.26**	2.82	**2.21**
Qwen3 14B	PLD	1.45	1.34	1.43	1.27	1.59	1.49	1.49	1.37
	EAGLE-3†	**2.72**	1.87	**3.03**	2.05	1.56	1.12	2.44	1.68
	RACER	2.67	**2.23**	2.77	**2.29**	**2.88**	**2.30**	**2.77**	**2.27**
Qwen3 32B	PLD	1.45	1.34	1.34	1.23	1.56	1.46	1.45	1.34
	EAGLE-3†	**2.88**	2.12	**2.97**	**2.17**	1.60	1.18	2.48	1.82
	RACER	2.66	**2.17**	2.55	2.08	**2.78**	**2.28**	**2.66**	**2.18**

† EAGLE-3 models and weights from AngelSlim's re-implementation.

into a draft tree using a predefined template. **LogitSpec** (Liu et al., 2025b) speculates the first draft token from the top-k candidates of the last-step logits, then augments expansion with retrieval tokens drawn from the context. **EAGLE-3** (Li et al., 2025) incorporates low-, mid-, and high-level features of the target model, with a Transformer decoder layer as its core. Since the official Qwen3 draft model weights for EAGLE-3 have not been released, we use the re-implementation by AngelSlim[1]. All baselines are run with their default hyperparameters.

4.2 Main Results

Table 1 and 2 reports the performance of RACER compared to baseline methods on different datasets and different target models. Among retrieval-based methods, PLD relies solely on the context and REST leverages an external training set. Their speedup ratios remain below $2\times$, highlighting the inherent limitations of retrieval-only approaches. In contrast, methods involving logits – whether model-free or model-based – can readily surpass $2\times$ speedup, demonstrating the advantage of exploiting predictive distributions from the target model.

RACER achieves the best speedup across most benchmarks and consistently delivers the highest overall speedup across different target models. Notably, both OpenPangu and Qwen3 are reasoning models that typically produce much longer outputs than Vicuna. The stable speedup observed across all tasks, despite their differing model architectures, suggests that RACER can be reliably integrated into long-context applications and remains robust to architectural variations. For LLaMA3.1 and Qwen3-series target models, EAGLE-3 achieves the highest MAT on most tasks except the Chinese reasoning benchmark MGSM-ZH. However, its advantage in MAT does not translate into end-to-end efficiency, as RACER still outperforms it in terms of speedup ratio. This is because EAGLE-3 requires an additional draft model, incurring extra inference cost, whereas RACER remains lightweight.

Moreover, the weaker performance of EAGLE-3 on MGSM-ZH highlights a broader limitation of model-based approaches: their effectiveness is sensitive to the distribution and coverage of the draft model's training data. It is plausible that EAGLE-3, trained primarily on English corpora, fails to simulate the target model's distribution accurately in Chinese reasoning tasks. Such data distribution mismatches, rooted in differences in post-training procedures or training corpora, generally constrain the robustness of model-based SD methods. For completeness, we additionally report results on English reasoning benchmarks GSM8K (Cobbe et al., 2021), AIME (Veeraboina, 2023) and MATH (Hendrycks et al., 2021) in Appendix G.

Compared with the other two logits-involved methods, TR and LogitSpec, RACER consistently outperforms them on both MAT and speedup.

[1] https://github.com/Tencent/AngelSlim

Table 3: Ablation experiments on Spec-Bench, HumanEval and MGSM-ZH.

Model	Method	Spec-Bench		HumanEval		MGSM-ZH	
		MAT	Speedup	MAT	Speedup	MAT	Speedup
Vicuna 7B	w/o logits	1.59↓1.41	1.43↓0.78	1.67↓1.44	1.52↓0.77	2.39↓1.32	2.11↓0.66
	w/o retrieval	2.72↓0.28	2.01↓0.20	2.68↓0.43	2.04↓0.25	2.95↓0.76	2.23↓0.54
	RACER	**3.00**	**2.21**	**3.11**	**2.29**	**3.71**	**2.77**
Vicuna 13B	w/o logits	1.56↓1.39	1.38↓0.87	1.65↓1.44	1.43↓0.99	2.29↓1.35	1.95↓0.88
	w/o retrieval	2.68↓0.27	2.06↓0.19	2.70↓0.39	2.14↓0.28	2.98↓0.66	2.34↓0.59
	RACER	**2.95**	**2.25**	**3.09**	**2.42**	**3.64**	**2.83**
Vicuna 33B	w/o logits	1.46↓1.28	1.38↓0.82	1.76↓1.40	1.66↓0.92	2.10↓1.26	1.97↓0.80
	w/o retrieval	2.55↓0.19	2.05↓0.15	2.66↓0.50	2.19↓0.39	2.74↓0.62	2.27↓0.50
	RACER	**2.74**	**2.20**	**3.16**	**2.58**	**3.36**	**2.77**

Overall, TR achieves better performance than LogitSpec, except on the reasoning task MGSM-ZH. This suggests that in general tasks, TR is able to exploit logits more effectively. However, in reasoning tasks where repeated patterns from previous context are frequent, retrieval provides crucial guidance and brings substantial benefits. Therefore, an effective strategy must integrate both logits and retrieval. RACER achieves this integration successfully, yielding consistently superior MAT and speedup across benchmarks.

In summary, RACER consistently delivers the best trade-off between acceptance and efficiency, achieving stable improvements across model sizes, domains, and languages, thus demonstrating its robustness and generality compared to retrieval-only, logits-only, and model-based baselines. Additional results under different hardware configurations and sampling settings are provided in Appendix C and D.

4.3 Ablation Study

To better verify RACER, we conduct ablation experiments with Vicuna. Table 3 reports ablation results by removing either the logits or retrieval component. We observe that removing logits causes the most severe degradation: MAT drops by more than one token on average and speedup decreases by 0.8-1.0×, confirming that logits form the backbone of speculative expansion. In contrast, removing retrieval leads to smaller but still notable drops, especially on MGSM-ZH where MAT and speedup decrease by up to 0.7 and 0.6, respectively. This highlights the complementary role of retrieval in reasoning tasks, where repeated patterns provide strong predictive cues. Across all three model scales, RACER consistently benefits from both components, validating our integration strategy that balances generalization from logits with structural guidance from retrieval.

To isolate the contributions of each component, we conduct ablations on retrieval and logits separately. For **retrieval**, we evaluate retrieval-only RACER with a fixed-interval automaton update to ensure independence from the logits component. As shown in Table 4, RACER consistently achieves the best performance across all benchmarks, with MAT improving by 0.20-0.70 over the strongest baseline and reaching an average of 2.53 (vs. 1.98 for SAMD and 1.95 for PLD), along with the highest average speedup (2.05×). Gains are especially significant on MGSM-ZH (3.32 vs. 1.29-2.65), indicating stronger robustness in multi-step reasoning tasks, while consistent improvements on Spec-Bench and HumanEval further demonstrate its general effectiveness.

For **logits**, we compare the retrieval-free variant (Table 3) with TR (Table 1). While TR benefits from a Vicuna-7B-calibrated static tree, its performance degrades at larger scales. In contrast, our logits component does not rely on model-specific tuning and performs better at larger scales.

We further study the integration strategy via **Half** (fixed budget split) and **Hard** (fallback switching). As shown in Table 5, **Merge** consistently performs best, achieving the highest MAT and speedup by effectively coordinating retrieval and logits.

Taken together, these experiments isolate the retrieval component, the logits component, and the integration strategy, and demonstrate that each contributes independently to the overall improvement.

Table 4: Evaluation results on Spec-Bench, HumanEval and MGSM-ZH using an RTX 3090 GPU. $\star$ indicates that RACER here only employs the retrieval automaton, configured with 10,000 nodes, an n-gram length of 8, and updating AC automaton failure links every 20 steps.

Model	Method	Spec-Bench		HumanEval		MGSM-ZH		Average	
		MAT	Speedup	MAT	Speedup	MAT	Speedup	MAT	Speedup
Vicuna 7B	PLD	1.72	1.57	1.57	1.45	2.57	2.35	1.95	1.79
	REST	1.82	1.37	2.06	1.65	1.29	1.06	1.49	1.36
	SAMD	1.70	1.65	1.59	1.54	2.65	2.49	1.98	1.89
	RACER*	**2.02**	**1.69**	**2.25**	**1.86**	**3.32**	**2.61**	**2.53**	**2.05**

Table 5: Ablation on integration strategies.

Integration	Merge (Ours)	Half	Hard
MAT	**3.00**	2.69	2.77
Speedup	**2.18**	1.97	2.11

5 Related Work

Efficient inference is crucial for real-time applications and resource-constrained scenarios. Various strategies, including KV cache compression (Luohe et al., 2024) and model weight quantization (Ma et al., 2025), have been developed to reduce latency. Among these, speculative decoding (SD) (Leviathan et al., 2023; Chen et al., 2023) stands out as a promising technique that predicts multiple continuations simultaneously, reducing decoding steps while maintaining accuracy.

SD methods can be broadly categorized into **draft-model-based** and **draft-model-free** approaches. Draft-model-based methods use additional models to predict draft tokens. These models are typically (i) separately trained or (ii) derived from smaller variants of the same model family. Some methods reuse hidden states to predict multiple future tokens (Cai et al., 2024; Li et al., 2024a,b), employing different layer selection and draft tree expansion strategies. Beyond post-training draft models, multi-token prediction (MTP) integrates draft generation during pre-training. Gloeckle et al. (2024) proposes parallel prediction of multiple tokens with independent output heads, while DeepSeek-AI (2024) introduces sequential multi-token prediction to preserve the full causal chain at each prediction depth.

In contrast, **draft-model-free** methods eliminate the need for additional models. PLD (Saxena, 2023) builds libraries from past content, achieving speedup in tasks with high redundancy like summarization. REST (He et al., 2023) builds retrieval libraries from existing corpora, offering substantial speedup but facing challenges such as large memory requirements and retrieval inefficiencies. Token Recycling (Luo et al., 2025) requires no additional generation, covering a broader range of continuations using past logits, while minimizing storage and retrieval costs.

6 Conclusion

In this work, we introduced RACER, a trainingfree method that unifies retrievalbased and logitsbased signals for speculative decoding. By treating retrieved exact patterns as structural anchors and logits as dynamic future cues, RACER constructs richer speculative drafts while remaining lightweight and plugandplay. Extensive experiments across multiple model families and benchmarks demonstrate consistent acceleration, stable memory usage, and improved speculative efficiency measured by MAT and speedup ratio. We believe RACER establishes a general foundation for trainingfree speculative decoding, opening avenues for future work on integrating more advanced retrieval structures, multilingual retrieval cues, and harmonization with parallel or distributed decoding algorithms.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (No. 62306216), the Natural Science Foundation of Hubei Province of China (No. 2023AFB816), and the National Science and Technology Major Project (No. 2023ZD0121502). It was also partially supported by the Open Fund of the Interdisciplinary Center for Intelligent Systems, Nanjing University of Science and Technology (No. YB202401).

Limitations

While this work demonstrates significant performance improvements in accelerating language model inference, the applicability of RACER to multimodal tasks remains untested. The current experiments focus solely on text-based tasks, and further evaluation is needed to explore its potential in tasks involving other modalities, such as vision and speech. Future work could investigate how to incorporate vision/audio token representations into RACER to improve performance in multimodal tasks, thereby extending its benefits beyond text-based applications.

References

Alfred V Aho and Margaret J Corasick. 1975. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, and 29 others. 2023. Qwen technical report. *CoRR*, abs/2309.16609.

Anselm Blumer, Janet Blumer, Andrzej Ehrenfeucht, David Haussler, and Ross McConnell. 1984. Building the minimal dfa for the set of all subwords of a word on-line in linear time. In *International Colloquium on Automata, Languages, and Programming*, pages 109–118. Springer.

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 preprint arXiv: 2401.10774*.

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

Hanting Chen, Yasheng Wang, Kai Han, Dong Li, Lin Li, Zhenni Bi, Jinpeng Li, Haoyu Wang, Fei Mi, Mingjian Zhu, Bin Wang, Kaikai Song, Yifei Fu, Xu He, Yu Luo, Chong Zhu, Quan He, Xueyu Wu, Wei He, and 5 others. 2025. Pangu embedded: An efficient dual-system LLM reasoner with metacognition. *CoRR*, abs/2505.22375.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

DeepSeek-AI. 2024. Deepseek-v3 technical report. *CoRR*, abs/2412.19437.

Zhichen Dong, Zhanhui Zhou, Zhixuan Liu, Chao Yang, and Chaochao Lu. 2025. Emergent response planning in LLMs. In *Forty-second International Conference on Machine Learning*.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriel Synnaeve. 2024. Better & faster large language models via multi-token prediction. *arXiv preprint arXiv:2404.19737*.

Zhenyu He, Zexuan Zhong, Tianle Cai, Jason D Lee, and Di He. 2023. Rest: Retrieval-based speculative decoding. *Preprint*, arXiv:2311.08252.

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

Yuxuan Hu, Ke Wang, Xiaokang Zhang, Fanjin Zhang, Cuiping Li, Hong Chen, and Jing Zhang. 2025. SAM decoding: Speculative decoding via suffix automaton. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12187–12204, Vienna, Austria. Association for Computational Linguistics.

Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. 1977. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350.

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

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024a. EAGLE-2: Faster inference of language models with dynamic draft trees. In *Empirical Methods in Natural Language Processing*.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024b. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *International Conference on Machine Learning*.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2025. EAGLE-3: Scaling up inference acceleration of large language models via training-time test. *Preprint*, arXiv:2503.01840.

Tianyu Liu, Yun Li, Qitan Lv, Kai Liu, Jianchen Zhu, Winston Hu, and Xiao Sun. 2025a. PEARL: Parallel speculative decoding with adaptive draft length. In *The Thirteenth International Conference on Learning Representations*.

Tianyu Liu, Qitan Lv, Hao Li, Xing Gao, and Xiao Sun. 2025b. Logitspec: Accelerating retrieval-based speculative decoding via next next token speculation. *arXiv preprint arXiv:2507.01449*.

Xianzhen Luo, Yixuan Wang, Qingfu Zhu, Zhiming Zhang, Xuanyu Zhang, Qing Yang, and Dongliang Xu. 2025. Turning trash into treasure: Accelerating inference of large language models with token recycling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6816–6831, Vienna, Austria. Association for Computational Linguistics.

Shi Luohe, Hongyi Zhang, Yao Yao, Zuchao Li, and Hai Zhao. 2024. Keep the cost down: A review on methods to optimize LLM's KV-cache consumption. In *First Conference on Language Modeling*.

Ziyang Ma, Zuchao Li, Lefei Zhang, Gui-Song Xia, Bo Du, Liangpei Zhang, and Dacheng Tao. 2025. Model hemorrhage and the robustness limits of large language models. *CoRR*, abs/2503.23924.

Udi Manber and Gene Myers. 1993. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948.

Somesh Mehra, Javier Alonso Garcia, and Lukas Mauch. 2025. On multi-token prediction for efficient llm inference. *arXiv preprint arXiv:2502.09419*.

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*, ASPLOS '24, page 932949, New York, NY, USA. Association for Computing Machinery.

OpenAI. 2025. gpt-oss-120b & gpt-oss-20b model card. *Preprint*, arXiv:2508.10925.

Apoorv Saxena. 2023. Prompt lookup decoding.

Freda Shi, Mirac Suzgun, Markus Freitag, Xuezhi Wang, Suraj Srivats, Soroush Vosoughi, Hyung Won Chung, Yi Tay, Sebastian Ruder, Denny Zhou, Dipanjan Das, and Jason Wei. 2022. Language models are multilingual chain-of-thought reasoners. *Preprint*, arXiv:2210.03057.

Luohe Shi, Zuchao Li, Lefei Zhang, Baoyuan Qi, Guoming Liu, and Hai Zhao. 2026. Scaling llm speculative decoding: Non-autoregressive forecasting in large-batch scenarios. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40:32947–32955.

Llama Team. 2024. The llama 3 herd of models. *CoRR*, abs/2407.21783.

Qwen Team. 2025. Qwen3 technical report. *CoRR*, abs/2505.09388.

Hemish Veeraboina. 2023. Aime problem set 1983-2024.

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

Lefan Zhang, Xiaodan Wang, Yanhua Huang, and Ruiwen Xu. 2025. Learning harmonized representations for speculative sampling. In *The Thirteenth International Conference on Learning Representations*.

A Experimental Setup

Hardware Setup Experiments were conducted with all runs restricted to a single GPU to ensure fairness and reproducibility. We used an NVIDIA RTX 4090 (24GB) with 20 CPU cores for 7B/8B-scale models, and an NVIDIA A800 (80GB) with 64 CPU cores for 13B-scale and larger models, unless otherwise specified.

Software Setup Our implementation is based on PyTorch and HuggingFace Transformers. Experiments were run under the following environment:

- PyTorch `2.8.0` with CUDA `12.8` and cuDNN `91002`
- HuggingFace Transformers `4.37.1` for Vicuna experiments, and `4.52.3` for LLaMA3.1/OpenPangu/Qwen3 experiments

We enabled `fp16` inference on both GPUs. No further optimizations (e.g., quantization or specialized kernels) were applied, to ensure fair comparison with prior work.

Evaluation Instructions In our experiments, we employ different instructions for different evaluation tasks and models. For Vicuna, we use its standard instructions:

Chat Template for Vicuna on Spec-Bench and MGSM

A chat between a curious user and an artificial intelligence assistant. The assistant gives helpful, detailed, and polite answers to the user's questions.

USER: Question

ASSISTANT:

Chat Template for Vicuna on HumanEval

A chat between a curious user and an artificial intelligence assistant. The assistant gives helpful, detailed, and polite answers to the user's questions.

USER: Implement the following code. Code

ASSISTANT:

For other models, we use a common system prompt:

Default Chat Template on Spec-Bench and MGSM

You are a helpful assistant.

USER: Question

ASSISTANT:

Default Chat Template on HumanEval

You are a helpful assistant.

USER: Implement the following code. Code

ASSISTANT:

Below we present an example used in the following case study.

First Example of HumanEval

A chat between a curious user and an artificial intelligence assistant. The assistant gives helpful, detailed, and polite answers to the user's questions.

USER: Implement the following code.

```
from typing import List

def has_close_elements(
    numbers: List[float],
    threshold: float
) -> bool:
    """ Check if in given list of
    numbers, are any two numbers
    closer to each other than
    given threshold.
    >>> has_close_elements(
    ... [1.0, 2.0, 3.0], 0.5
    ... )
    False
    >>> has_close_elements(
    ... [1.0, 2.8, 3.0,
    ... 4.0, 5.0, 2.0],
    ... 0.3
    ... )
    True
    """
```

ASSISTANT:

B Case Study

We select the first example from HumanEval (Chen et al., 2021) to further illustrate how the Logits Tree and the Retrieval Tree operate independently, as well as how they are integrated within RACER. The prompt is provided at the end of Appendix A. For readability, we manually add indentation and line breaks. This example contains 178 tokens in the prefilling stage and 356 tokens in the decoding stage generated by Vicuna-7B.

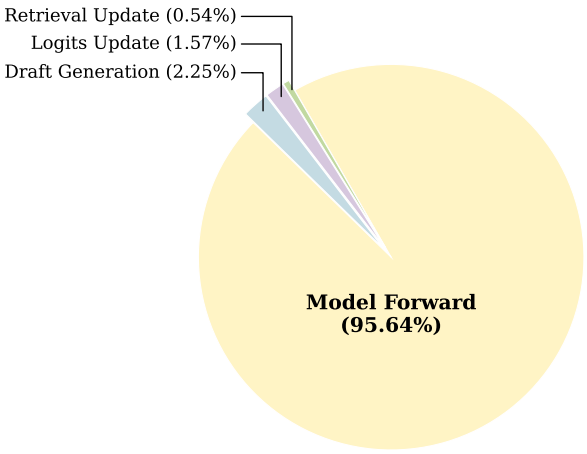

Figure 5: End-to-end inference latency breakdown by processing stage.

Time Allocation We divide the overall procedure into four parts: **Draft Generation**, **Model Forward** (Prefilling and Decoding Verification), **Logits Update**, and **Retrieval Update**. Figure 5 shows that most of the runtime is dominated by the model forward stage, which involves relatively heavy computation over model parameters. The remaining parts are lightweight and negligible compared to the cost of model forward.

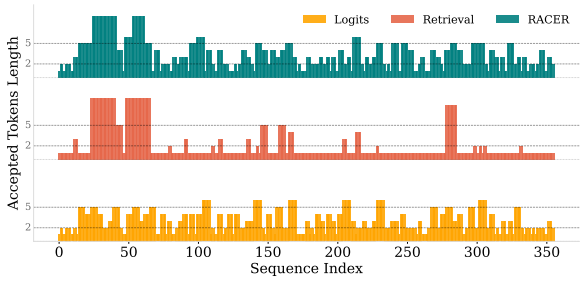

Figure 6: Case study on accepted tokens length. Accepted lengths are expanded to enable token-level alignment across methods.

Accepted Length To provide a token-level comparison, we expand each accepted segment by its length along the x-axis in Figure 6, such that an accepted length of l occupies l consecutive positions. Retrieval-only decoding yields an accepted length of one token in most cases, as no draft tokens are proposed or accepted. However, several pronounced peaks at an accepted length of 9 can be observed around sequence index 50, corresponding to rare but long matches. Logits-only decoding exhibits a more balanced acceptance pattern, with a large portion of accepted lengths exceeding 2 but remaining below 5. RACER combines the strengths of both, resulting in an acceptance trend that is generally above 2 due to the stability of logits-based speculation, while also exhibiting occasional peaks enabled by strict matches from retrieval.

Integration To further illustrate how logits-based speculation and retrieval operate independently and complement each other, we examine the token-level behavior in detail. In the first successful match produced by the Logits Tree, the last verified token is `Here`, and the best draft candidate is

`['', s, _question, <unk>, _the, <unk>]`.

This draft sequence is generated from logits and can be traced back to the system prompt fragment "`...polite answers to the user's questions...`". The next ground-truth token is `_one`; as a result, `_questions` and all subsequent draft tokens are rejected during verification.

For completeness, the top-8 logits corresponding to the tokens `'` (quotation mark) and `s` are listed below:

- `[s, t, \n, _s, <s>, hren, ']`
- `[_questions, _queries, _in, _question, </s>, _requests, questions, \n]`

In contrast, the retrieval-based method identifies only a match of length 1 (i.e., fewer than 2 tokens), since only the quotation mark appears in the history. Consequently, retrieval does not propose any draft tokens at this step.

In the first successful match produced by the Retrieval Tree, the last verified token is `_`, and the best draft candidate is

```
[close, _, elements,
(, numbers, :, List, [, float]
```

Notably, both close and _ appear among the **rejected tokens** of has since the mismatch of position. _ is later sampled following has, enabling retrieval to activate at this position and refresh the logits adjacency. The top-8 candidates for close and _ (old and new) are shown below:

- [_, ., ', Elements, -, </s>, (]
- [elements, element, </s>, _elements, ele, to, \n, Elements] (old, refreshed)
- [close, </s>, clos, _close, closed, \n, Close, open] (new)

The *copy-logit* mechanism adopts the refreshed logits for _, which now prioritize continuations starting with close. Although _ appears in the top-8 of close, the logits have been updated only with near-future context corresponding to close, and not with the farther continuation elements. As a result, logits-based candidates cannot extend to elements at this stage.

Retrieval, however, successfully captures this longer-range dependency from the user-provided context "def has_close_elements", thereby correcting the shift introduced by the logits-only approximation. Importantly, even when the logits-based candidates of close are rejected during verification, they still refresh the top-8 logits for subsequent tokens such as the quotation mark (') and _function. Consequently, the following logits-based candidate

['_function, _in, _Python, :, _List]

is almost accepted.

Below we show a portion of the response, where accepted draft tokens (excluding the sampled token) are highlighted in teal.

Example of Response

ASSISTANT: Here's one possible implementation of the `has_close_elements` function in Python:
```python
def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """Check if in given list of numbers, are any two numbers closer to each other than
    given threshold."""
    ... (849 characters)
```

In conclusion, both retrieval and logits contribute not only to the currently accepted tokens but also to anticipating future tokens for more effective self-speculation, making RACER both lightweight and accurate.

C Additional Experiment Results

Table 6 presents the results on individual Spec-Bench tasks, complementing the overall comparison in Table 1 and 2. RACER consistently outperforms other model-free methods in most cases, including all overall speedup ratios, with only a few exceptions in Translation (Trans), Question Answering (QA), and Mathematical Reasoning (Math). For Translation, the retrieval component contributes little to MAT and may even offset part of the logits-based advantage, leading to weaker performance on smaller models. However, with larger models such as Vicuna-33B, this effect becomes negligible, and RACER consistently outperforms TR. For QA, TR surpasses RACER on Vicuna-7B, suggesting that its predefined tree template may align better with the characteristics of this task. For Math, TR slightly outperforms RACER only on Vicuna-7B, but this advantage does not generalize to other model scales or hardware. In contrast, as model size increases, RACER shows a growing margin in overall speedup over TR, highlighting its robustness across diverse tasks and architectures.

D Additional Ablation Results

D.1 Component Contribution

Table 7 reports the ablation results on Spec-Bench across six tasks. The results clearly show that removing either logits or retrieval consistently harms performance, confirming that both components are essential for RACER. **Without logits**: Performance drops significantly across all tasks, often by more than $0.7\times$ in speedup. This highlights that logits are the dominant factor for efficient speculation. **Without retrieval**: The degradation is generally smaller but still noticeable, especially on tasks such as MT and Sum, where repeated patterns and structural cues play a larger role. **Full RACER**: By integrating both signals, RACER achieves balanced improvements and consistently outperforms the ablated variants, showing robustness across tasks, model scales, and hardware platforms. This task-dependent effect aligns with our main results and further validates

Table 6: Speedup ratios and overall MAT across different tasks of Spec-Bench evaluated on NVIDIA A800 (80GB).

Model	Method	MT	Trans	Sum	QA	Math	RAG	MAT	Speedup
Vicuna 7B	PLD	1.42	0.97	2.25	1.12	1.59	1.61	1.72	1.49
	REST	1.52	1.09	1.21	1.28	1.07	1.31	1.82	1.33
	LogitSpec	1.79	1.35	2.48	1.50	2.08	1.82	2.35	1.86
	TR	2.19	**1.84**	2.02	**2.02**	**2.58**	1.85	2.76	2.15
	RACER	**2.23**	1.61	**2.61**	1.82	2.46	**2.12**	**3.01**	**2.22**
Vicuna 13B	PLD	1.36	0.98	1.93	1.09	1.53	1.44	1.65	1.41
	REST	1.61	1.14	1.30	1.56	1.21	1.42	1.82	1.44
	LogitSpec	1.67	1.33	2.13	1.40	2.00	1.72	2.32	1.73
	TR	2.00	**1.74**	1.89	1.82	2.31	1.87	2.79	1.99
	RACER	**2.23**	1.70	**2.49**	**1.85**	**2.55**	**2.21**	**2.95**	**2.25**
Vicuna 33B	PLD	1.33	1.03	1.84	1.11	1.57	1.23	1.54	1.37
	REST	1.70	1.24	1.41	1.57	1.31	1.61	1.81	1.54
	LogitSpec	1.66	1.36	2.11	1.38	2.02	1.50	2.11	1.71
	TR	1.90	1.67	1.85	1.76	2.20	1.72	2.63	1.83
	RACER	**2.22**	**1.73**	**2.41**	**1.87**	**2.51**	**1.91**	**2.74**	**2.20**
OpenPangu 7B	PLD	1.40	1.54	1.47	1.46	1.36	1.47	1.49	1.44
	RACER	**2.07**	**2.46**	**2.15**	**1.89**	**2.37**	**2.31**	**2.48**	**2.17**
Qwen3 8B	PLD	1.37	1.67	1.35	1.41	1.72	1.44	1.52	1.47
	EAGLE-3	**2.43**	2.07	2.12	2.36	2.63	**2.45**	**3.47**	2.37
	RACER	2.41	**2.69**	**2.35**	**2.41**	**2.72**	2.42	2.73	**2.48**
Qwen3 14B	PLD	1.27	1.56	1.18	1.31	1.54	1.29	1.45	1.34
	EAGLE-3	1.93	1.86	1.60	1.82	2.12	1.76	**2.72**	1.87
	RACER	**2.12**	**2.55**	**2.10**	**2.19**	**2.47**	**2.11**	2.67	**2.23**
Qwen3 32B	PLD	1.26	1.54	1.13	1.35	1.48	1.29	1.44	1.33
	EAGLE-3	**2.20**	2.03	1.84	2.02	**2.47**	2.04	**2.88**	2.12
	RACER	2.10	**2.40**	**2.04**	**2.06**	2.38	**2.16**	2.66	**2.17**

Table 7: Ablation experiments on multiple tasks of Spec-Bench (speedup only).

Model	Method	MT	Trans	Sum	QA	Math	RAG
Vicuna 7B	w/o logits	1.42↓0.79	0.94↓0.71	1.85↓0.69	1.11↓0.71	1.51↓0.94	1.50↓0.59
	w/o retrieval	1.98↓0.23	1.68↑0.03	2.14↓0.40	1.81↓0.01	2.31↓0.14	1.90↓0.19
	RACER	**2.21**	**1.65**	**2.54**	**1.82**	**2.45**	**2.09**
Vicuna 13B	w/o logits	1.35↓0.88	0.90↓0.80	1.69↓0.80	1.04↓0.71	1.51↓1.04	1.56↓0.65
	w/o retrieval	2.04↓0.19	1.69↓0.01	2.17↓0.32	1.81↓0.04	2.37↓0.18	2.00↓0.21
	RACER	**2.23**	**1.70**	**2.49**	**1.85**	**2.55**	**2.21**
Vicuna 33B	w/o logits	1.39↓0.83	0.97↓0.76	1.63↓0.78	1.09↓0.78	1.56↓0.95	1.26↓0.65
	w/o retrieval	2.03↓0.19	1.72↓0.01	2.12↓0.29	1.84↓0.03	2.36↓0.15	1.85↓0.06
	RACER	**2.22**	**1.73**	**2.41**	**1.87**	**2.51**	**1.91**

Table 8: Ablation experiments with Qwen2.5 series on NVIDIA A800 (80GB). Overall MAT and speedup ratios on general dataset Spec-Bench are reported.

	Qwen2.5-0.5B	Qwen2.5-1.5B	Qwen2.5-14B	Qwen2.5-32B
MAT	3.30	3.25	2.64	2.71
Speedup	2.64	2.64	2.33	2.19

Table 9: Ablation experiments with Qwen3 series on NVIDIA A800 (80GB). Overall MAT and speedup ratios on general dataset Spec-Bench are reported.

	Qwen3-0.6B	Qwen3-1.7B	Qwen3-4B	Qwen3-8B	Qwen3-14B	Qwen3-32B
MAT	2.89	2.78	2.78	2.73	2.67	2.66
Speedup	2.55	2.48	2.53	2.13	2.23	2.17

the complementary nature of logits- and retrieval-based speculation.

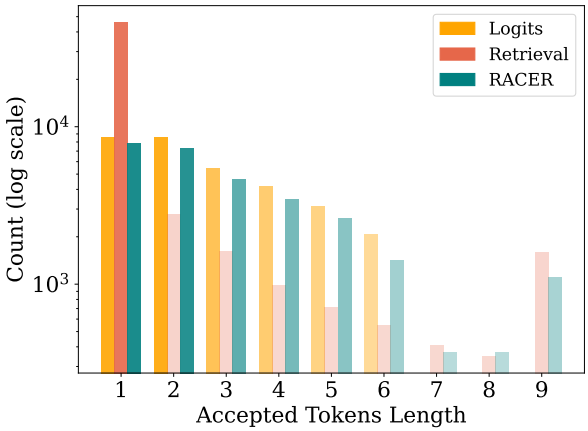

Figure 7: Comparison of accepted token lengths with Vicuna-7B across three settings: Logits-only decoding (w/o retrieval), Retrieval-only decoding (w/o logits), and RACER. Transparency indicates the relative frequency.

Figure 7 illustrates the distributions of accepted token lengths under three decoding strategies: Logits-only decoding (without retrieval), Retrieval-only decoding (without logits), and RACER. For **Retrieval-only**, the majority of decoding steps result in an accepted length of one token, indicating that no draft tokens are proposed or accepted in most cases. The remaining mass is concentrated on a few discrete values, with the top three being 2, 3, and 9. Notably, 9 corresponds to the longest possible draft given an n-gram length of 10, meaning that one context token and one sampled next token are matched, followed by nine draft tokens that have appeared previously and are accepted. This pattern reveals the sparse yet strict nature of retrieval: matches occur infrequently, but when they do, they can yield a large number of accepted tokens. In contrast, **Logits-only** decoding does not exhibit extreme values such as 9, as its expansion depth is restricted to 6 due to the overall budget of 64 candidates. Most of the probability mass is concentrated at an accepted length of 2, corresponding to one sampled next token and one accepted draft token. The remaining accepted lengths, ranging from 1 and 3 to 6, show a gradual decay. This behavior reflects the advantage of logits reuse, where near-future tokens are approximated through self-speculation within the target model. **RACER** combines the strengths of both approaches and produces a more balanced distribution of accepted token lengths. While extreme long matches are less frequent than in pure retrieval, RACER achieves higher acceptance than logits-only decoding. Note that the y-axis reports absolute counts rather than normalized frequencies. Since RACER requires fewer decoding steps overall while achieving a higher average accepted token length per step, the heights across different settings are not directly comparable.

D.2 Model Size Ablation

Tables 8 and 9 report ablation results across different model sizes and architectures on Spec-Bench. The Qwen2.5 series are dense instruct models, while the Qwen3 series evaluated here are dense thinking models.

For Qwen2.5, we observe that both MAT and speedup are higher for the smaller models (0.5B and 1.5B), and drop when moving to 14B and 32B. A key factor behind this gap is the output length: on Spec-Bench, Qwen2.5-14B and Qwen2.5-32B generate on average about 1.2× more tokens than the 0.5B and 1.5B models. Since later tokens rely on longer-range context and are more likely to be rejected, MAT naturally decreases.

Qwen3 provides a complementary perspective. As thinking models, all Qwen3 variants tend to produce much longer answers than Qwen2.5, so the difference in average output length across Qwen3-0.6B to Qwen3-32B is relatively small. In this regime, the dominant factor for MAT is no longer model size, but long-range dependency itself. This explains why the MAT for Qwen3 stays within a narrow band (2.66 - 2.89): RACER is operating under consistently longer effective sequence lengths, and its logits-based component is increasingly challenged by dependencies far from the current position due to the nature of Transform-

Table 10: Ablation experiments of sampling temperature across multiple tasks of Spec-Bench.

Model	**Temp.**	MT	Trans	Sum	QA	Math	RAG	MAT	Speedup
Vicuna 7B	Greedy	2.21	1.65	2.54	1.82	2.45	2.09	3.00	2.21
	$T = 0.5$	2.19	1.62	2.56	1.96	2.64	2.16	3.05	2.25
	$T = 1.0$	2.30	1.74	2.52	1.90	2.55	2.34	3.03	2.29
OpenPangu 7B	Greedy	1.89	2.27	1.96	1.73	2.20	2.15	2.47	1.99
	$T = 0.5$	1.87	2.16	1.90	1.70	2.14	1.79	2.51	1.91
Qwen3 8B	Greedy	2.05	2.44	1.96	2.11	2.37	2.02	2.73	2.13
	$T = 0.5$	2.01	2.20	1.88	1.94	2.23	2.03	2.78	2.04

ers. Nonetheless, even under these longer-context conditions, RACER is still able to maintain MAT close to 3 on average.

Given that MAT remains roughly stable within each model family, the slight degradation in speedup as model size increases is expected and does not undermine RACER's core advantage. Larger models incur higher verification cost per token, so the same MAT translates into a smaller relative speedup – a general phenomenon shared by speculative decoding methods. The key takeaway from these ablations is that RACER sustains strong and stable acceleration (consistently above $2\times$) across a wide range of model sizes and sequence lengths, even as longer outputs and long-range dependencies make speculation inherently more difficult.

Moreover, although our experiments are conducted with `batch_size=1`, RACER's lightweight, model-free design suggests that its performance degradation at larger batch sizes should be less severe than that of model-based speculative methods. In contrast to approaches that rely on additional draft models and incur batch-wise overhead in both forward passes and synchronization, RACER avoids extra model execution, making it amenable to scaling in batched decoding scenarios.

D.3 Sampling Temperature

To complement the greedy results reported in the main text, Table 10 presents the speedup ratios under different sampling temperatures using the standard rejection sampling scheme (Leviathan et al., 2023; Chen et al., 2023).

Overall, the results indicate that RACER exhibits strong robustness to decoding temperature, with only marginal variations observed when switching between greedy decoding and nucleus sampling. Across all evaluated backbone models, task-wise speedups under different temperature settings remain highly consistent. For Vicuna-7B, the average speedup varies narrowly from 2.21 (Greedy) to 2.29 ($T = 1.0$), while individual task fluctuations stay within a small range. Similar stability is observed for OpenPangu-7B and Qwen3-8B, where enabling sampling ($T = 0.5$) leads to changes that are minor and non-systematic, sometimes even slightly decreasing speedup, but without any clear degradation trend. One notable exception is observed on the RAG task with OpenPangu-7B, where the speedup drops from 2.15 under greedy decoding to 1.79 at $T = 0.5$. Compared to other tasks and models, this difference is relatively larger. We attribute this behavior to the higher sensitivity of retrieval-augmented generation to early-token variability. In RAG-style prompts, slight noise introduced by stochastic sampling may alter the alignment between retrieved context and subsequent generation, leading to less effective speculative validation and reduced acceptance rates.

Importantly, this degradation is task- and model-specific, rather than a general limitation of RACER. Taken together, these results demonstrate that RACER's acceleration benefits are largely robust to the choice of decoding temperature, indicating that its performance gains stem from structural properties of the speculative mechanism rather than temperature-specific token distributions. This robustness makes RACER particularly attractive for real-world deployment, where decoding strategies may vary across applications without requiring retuning of acceleration hyperparameters.

D.4 Parameter Robustness

We further study the robustness of RACER under different hyperparameter settings on Vicuna-7B-v1.5. Conceptually, these hyperparameters con-

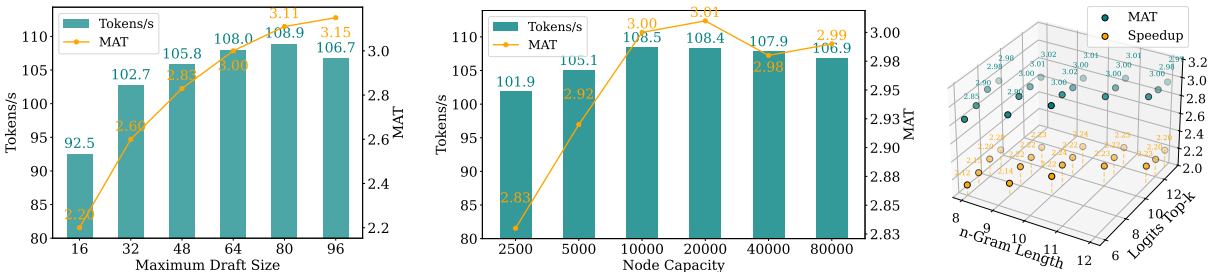

Figure 8: Ablation studies of RACER on key parameters: draft size, node capacity, n-gram depth (Retrieval Tree) and top-k breadth (Logits Tree).

trol complementary aspects of the unified speculative draft: the *draft size* specifies the total number of draft tokens per step, the *top-k breadth* of the Logits Tree controls how widely we explore model-predicted candidates, and the *capacity* and *n-gram depth* of the Retrieval Tree determine how many n-grams can be stored and how long they can be. A larger draft size or broader trees generally increases coverage and acceptance probability, but if they grow too large, the decoding regime may shift from memory-bound to compute-bound, leading to diminishing or even negative returns. Similarly, increasing retrieval capacity and n-gram depth allows matching more rare patterns, but over-emphasizing long-tail matches can dilute the draft budget and reduce MAT. In practice, RACER remains stable over a broad range of these settings.

Figure 8 summarizes three ablation experiments. **Draft Size (Figure a).** The draft size controls the *total* number of speculative tokens proposed by both the Logits Tree and the Retrieval Tree in each step. Increasing the draft size from 16 to 64 steadily improves both MAT and throughput, after which the gains saturate. This shows that RACER benefits from moderately larger drafts while remaining stable even with further expansion. However, overly large drafts can push the system into a compute-bound regime, where the additional verification cost outweighs the benefit of higher acceptance. The optimal draft size is therefore hardware-dependent: on resource-constrained or edge devices, it is often preferable to cap the draft size to match the device's most efficient batch inference mode. **Retrieval Node Capacity (Figure b).** The Retrieval Tree is implemented as an AC automaton whose *capacity* specifies an upper bound on the number of n-gram states (e.g., 10K nodes). Expanding the automaton's storage from 2.5K to around 10K-20K nodes yields the best trade-off between MAT and throughput. Beyond this range, performance only fluctuates slightly, suggesting that RACER does not rely on excessively large retrieval buffers. The built-in LRU eviction policy exploits both *temporal* and *spatial* locality: frequently reused n-grams are retained, while rarely used ones are pruned. Since each node in an AC automaton has a unique parent and a failure link, the space complexity of the automaton $\mathcal{A}$ is $\mathcal{O}(|\mathcal{A}|)$, i.e., linear in the number of nodes. In practice, this overhead is modest and remains negligible even with node sizes up to 10K. **Joint Effect of n-gram Depth and Top-k Breadth (Figure c).** The n-gram depth controls the maximum height of the Retrieval Tree, and thus the longest context it can match. With an upper bound such as $n = 10$, the automaton can flexibly match keys of length 1 to 9 and retrieve the corresponding value sequences. Because candidates are selected according to empirical frequency, this design naturally balances exploration and exploitation: it covers diverse patterns while prioritizing high-yield matches. On the Logits Tree side, the top-k breadth specifies how many high-probability continuations are expanded at each level; increasing it extends coverage into the long tail but also competes for the finite draft budget. The 3D plot shows that both MAT and speedup improve smoothly as n-gram depth and top-k breadth increase, and the performance surface remains relatively flat near the optimal range (n-gram depth 9-11, top-k breadth 8-10), indicating that RACER is robust to small parameter deviations. This trend is consistent with our analysis in Section 3.1, where the 85th percentile rank for the *copy-logit* strategy is 9, suggesting that setting these parameters around this range is sufficient and provides a practical guideline when adapting RACER to new target models.

E More Implementation Details

E.1 Logits Tree Construction

The Logits Tree leverages the top-k adjacency matrix from past logits and expands breadth-first, following the breadth allocation rule in Eq. 3. The full procedure is described in Algorithm 1. Figure 9 illustrates how Logits Tree is pruned from dense to sparse, with the same capacity of 21.

E.2 Aho–Corasick Automaton Construction and Transition

The Aho–Corasick automaton could be simply described as trie with failure links. Each failure link at a node connects to the longest proper suffix of the string at that node, which also serves as a prefix for another pattern in the trie. If no such suffix exists, the link reverts to the root. This is analogous to the "failure function" in the Knuth–Morris–Pratt (KMP) string-matching algorithm (Knuth et al., 1977), but Aho–Corasick extends this idea to work efficiently for multiple patterns.

Figure 11 illustrates the AC automaton's structure, showcasing failure links in red and final states with double circles, though some transitions may be omitted for clarity. The process begins with an input sequence that progresses through the trie. If a mismatch occurs, such as when at the state "she" and the next input is "r" without a corresponding edge, the automaton utilizes failure links to backtrack until a valid node with the "r" edge is found or until it returns to the root. When the automaton reaches the state "her", not only is the pattern "her" itself recognized but the state of its failure pointer is also included. This forms part of a recursive process: matching a state involves sequentially matching the state of its failure pointer until it traces back to the root node, which represents the absence of further matches, denoted as ϵ.

For further clarification, Figure 10 illustrates the matching process for the string "sherd", identifying the substrings "she", "he", and "her". The elements highlighted in dark blue represent both the longest pattern prefix that the current state can match, and the minimal suffix information necessary for subsequent matches. This setup can be visualized as a "sliding window" that moves from left to right across each position. During normal transitions, this sliding window accordingly steps to the right. Conversely, during backtracking, the state transitions via the failure pointer, effectively discarding any irrelevant left-side components. Note that in actual implementations, the failure links in AC automata are primarily used during the construction phase and the match phase. Once the automaton is constructed, these failure links are often replaced by virtual transitions that directly lead to the correct states like Algorithm 2. This optimization streamlines the matching process, enhancing efficiency by reducing unnecessary transitions.

E.3 LRU Eviction Strategy

In RACER's retrieval automaton, we adopt an LRU (Least Recently Used) eviction mechanism to manage limited node capacity. When the maximum number of nodes is reached, the least recently accessed node is recycled and reassigned to represent a new state. This ensures that the automaton continuously adapts to the most relevant n-grams from the current decoding context while maintaining bounded memory.

The mechanism is shown in Algorithm 3 and works as follows: **Touch**: Every time a node is visited, it is moved to the front of the LRU list and its reference is updated in the hash map. This guarantees that the tail of the list always contains the least recently used node. **TouchPrefix**: When a failure transition occurs, not only the current node but also its ancestors along the prefix are "touched". This ensures that the entire matching path is marked as recently used. **TransTokens**: When processing a sequence of tokens, the automaton repeatedly performs *Touch* and failure transitions until either a matching child node is found or the traversal falls back to the root. **InsertTokens**: When inserting a new n-gram, if no free node is available, the node at the back of the LRU list (the least recently used one) is evicted and reset. It is then reassigned as the child for the new transition. Frequency counters along the insertion path are updated accordingly. This design ensures that: Only leaf nodes are evicted, preventing structural corruption of the automaton. The automaton remains adaptive to changing contexts, exploiting temporal and spatial locality during decoding. The eviction and update operations remain lightweight and efficient, keeping inference fast.

Time Complexity The Touch operation runs in constant time $\mathcal{O}(1)$, as it only updates the doubly linked list and hash map. TouchPrefix has

Algorithm 1 Construct Logits Tree (BFS) and Return Draft Candidates

function BUILDLOGITSTREE(next_token, C) ▷ C is the maximum number of draft nodes
  Initialize an empty queue Q
  Initialize an empty list candidates
  token(0) $\leftarrow$ next_token
  push $Q \leftarrow 0$
  $C \leftarrow C - 1$ ▷ Root consumes one draft slot
  while not Q.isEmpty() **do**
    $u \leftarrow Q$.dequeue()
    if $C > 0$ **then**
      next_breadth $\leftarrow \begin{cases} b(\mathrm{u}), & \text{if } u = \text{root} \\ \lfloor b(\mathrm{u})/2 \rfloor, & \text{otherwise} \end{cases}$ ▷ Breadth allocation follows Eq. 3
      for $j \leftarrow 0$ **to** $b(u)$ - 1 **do**
        if $C = 0$ **then**
          break
        end if
        $v \leftarrow k \times u + j$ ▷ The j-th child of u
        token(v) $\leftarrow$ top_k[token(u)][j]
        $b(v) \leftarrow$ next_breadth
        push $Q \leftarrow v$
        next_breadth $\leftarrow \lfloor$next_breadth$/2 \rfloor$ ▷ Breadth allocation follows Eq. 3
        $C \leftarrow C - 1$
      end for
    else ▷ Backtrack to get a draft candidate
      path $\leftarrow [\,]$; $v \leftarrow u$
      while $v \neq \epsilon$ **do**
        append token(v) to path
        $v \leftarrow$ parent(v)
      end while
      reverse(path) ▷ Leaf→root to root→leaf
      append path to candidates
    end if
  end while
  return candidates
end function

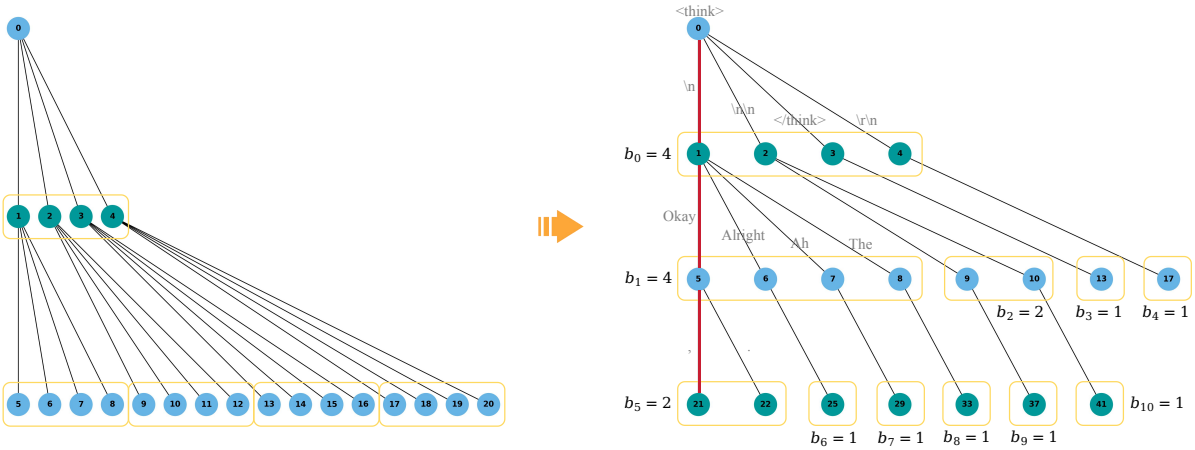

Figure 9: The tree on the left shows the expansion of an unpruned 4-ary tree with 21 nodes, while the tree on the right depicts the expansion of the pruned 4-ary tree with the same number of nodes. The path in red represent a possible candidate [<think>, <end_of_line>, Okay, <comma>].

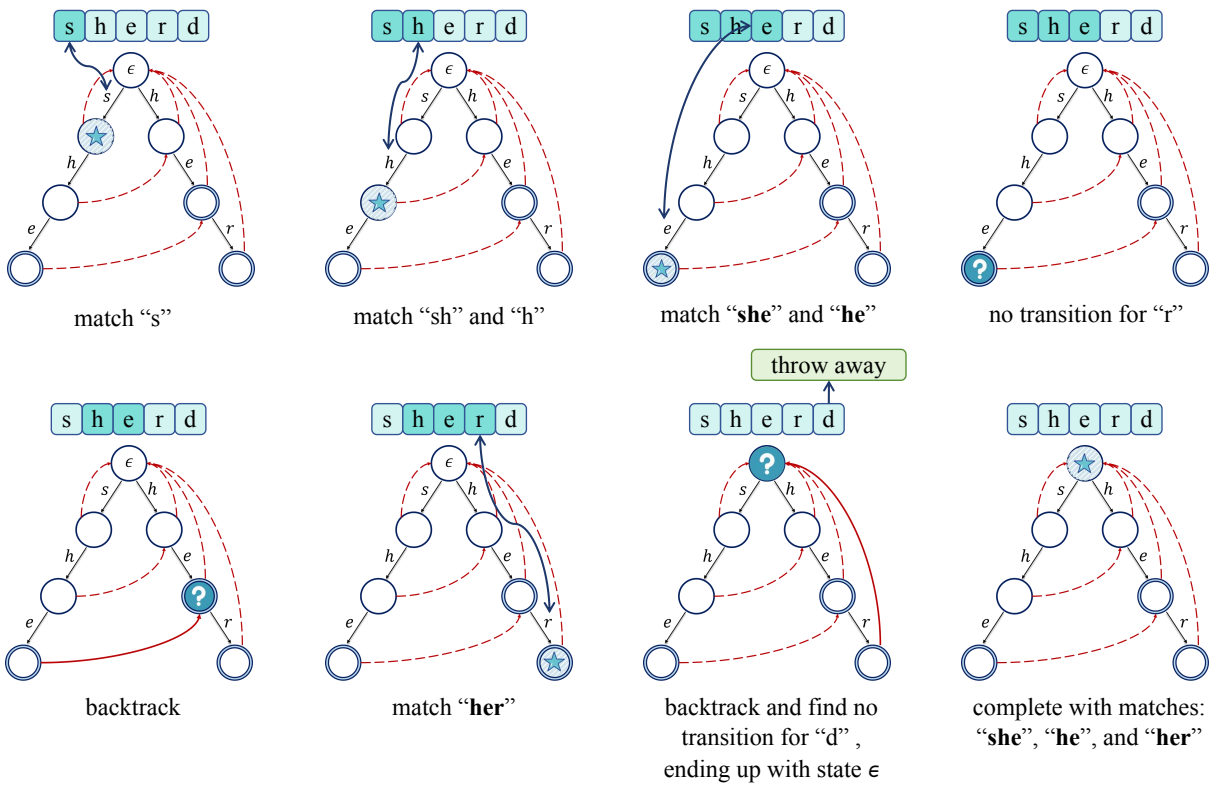

Figure 10: The process of how "sherd" matches patterns "she", "he", and "her" by transitions on an AC automaton.

Algorithm 2 Calculate failure links for nodes in an AC automaton

```
1:  function GET_TRANSITIONS
2:      Initialize an empty queue Q
3:      for v ← children of the root do
4:          fail(v) ← root                          ▷ Set initial failure state to root
5:          Enqueue v into Q
6:      end for
7:      while not Q.isEmpty() do
8:          u ← Q.dequeue()
9:          for i ← possible transitions from u do
10:             v ← child(u, i)
11:             if v ≠ ϵ then
12:                 fail(v) ← child(fail(u), i)      ▷ Update the failure pointer
13:                 Enqueue v into Q
14:             else
15:                 f ← fail(u)
16:                 while f ≠ root and child(f, i) = ϵ do
17:                     f ← fail(f)                  ▷ Backtrack through the failure pointer
18:                 end while
19:                 if child(f, i) ≠ ϵ then
20:                     fail(v) ← child(f, i)
21:                 else
22:                     fail(v) ← root
23:                 end if
24:             end if
25:         end for
26:     end while
27: end function
```

Algorithm 3 LRU Eviction Strategy in AC Automaton

function TOUCH(node)
    Move node to the front of LRU_LIST
    Update LRU_MAP[node] ← LRU_LIST.begin()
end function
function TOUCHPREFIX(node)
    while node $\neq \epsilon$ **do**
        fail(node) ← root ▷ Default failure link to the root before rebuild
        TOUCH(node)
        node ← parent(node)
    end while
end function
function TRANSTOKENS(tokens)
    $u \leftarrow$ curren_state
    for each $t \in$ tokens **do**
        TOUCH(u)
        while $u \neq$ root **and** child$(u, t) = \epsilon$ **do**
            $u \leftarrow$ fail(u) ▷ Failure transition
        end while
        TOUCHPREFIX(u) ▷ Update prefix after failure transition
        if child$(u, t) \neq \epsilon$ **then**
            $u \leftarrow$ child(u, t)
        else
            $u \leftarrow$ root
        end if
    end for
    TOUCH(u) ▷ Final state after processing tokens
    current_state ← u
end function
function INSERTTOKENS(tokens, frequency)
    $u \leftarrow$ root
    freq(u) ← freq(u) + frequency
    for each $t \in$ tokens **do**
        TOUCH(u)
        if child$(u, t) = \epsilon$ **then**
            new_node ← LRU_LIST.back()
            new_node.reset()
            child(u, t) ← new_node
        end if
        $u \leftarrow$ child(u, t)
        freq(u) ← freq(u) + frequency
    end for
    TOUCH(u) ▷ Touch the leaf node
end function

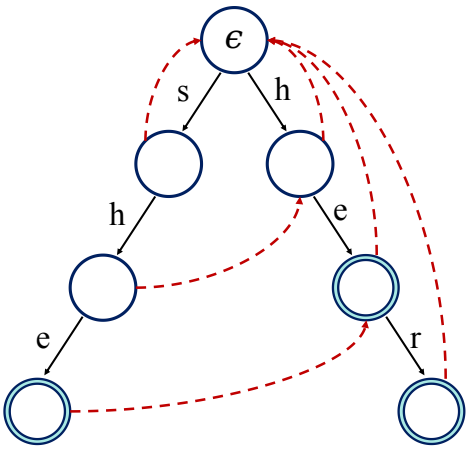

Figure 11: The illustration of an Aho–Corasick automaton with patterns “she”, “he”, and “her”, with final states in double circles and failure links in red.

a worst-case complexity of $O(d)$, where d is the maximum depth of the automaton (i.e., the n-gram length, typically a small constant). The TransTokens procedure processes each token in the input sequence and may backtrack up to depth d through failure links, giving a worst-case complexity of $\mathcal{O}(|\text{tokens}| \cdot d)$, though in practice it is close to linear $O(|\text{tokens}|)$. Finally, InsertTokens requires at most d steps for each token sequence, resulting in $O(|\text{tokens}|)$ complexity.

Space Complexity The storage requirement is linear in the number of automaton nodes. Trie nodes occupy $O(|\mathcal{A}|)$ space, bounded by the maximum capacity of the automaton. The LRU list and hash map also require $O(|\mathcal{A}|)$ space, as each node is tracked in both structures. Thus, total memory is $O(|\mathcal{A}|)$, capped in practice at 10^4-10^5 nodes, which corresponds to tens of megabytes – well within the budget of modern devices and suitable for memory-constrained scenarios.

E.4 Case Study of Retrieval Expansion

Suppose the current match state is `[<think>, </think>]`, and the automaton has observed:

- `[<think>, </think>, Okay]` with frequency 3,
- `[</think>, Yes, <space>]` with frequency 2,
- `[</think>, Yes, <comma>]` with frequency 1.

Pooling continuations over all prefixes ending in `</think>` yields:

$$\texttt{[Okay]}: 3, \quad \texttt{[Yes]}: 2{+}1 = 3,$$
$$\texttt{[Yes, <space>]}: 2, \quad \texttt{[Yes, <comma>]}: 1.$$

Selecting the top-3 continuations gives:

$$\texttt{[Okay]}: 3, \quad \texttt{[Yes]}: 2{+}1 = 3,$$
$$\texttt{[Yes, <space>]}: 2,$$

If a depth constraint is applied (i.e., only continuations consistent with the matched suffix of length 2 are allowed), then only the continuation `Okay` remains valid, because it is the only 3-gram whose prefix exactly matches the current border `[<think>, </think>]`. Other candidates such as `[Yes]` or `[Yes, <space>]` originate from shorter matches ending in `</think>`, and are therefore pruned under the depth restriction.

F Futher Explanation of Logits Tree

The tokenizer, with vocabulary $\mathcal{V}$, maps the text into discrete tokens. At each decoding step, one token is determined at a time, resulting in a token sequence that represents a specific forward path. For a path with k tokens, the LLM narrows the search space $\mathcal{V}^k$ using either greedy or nucleus sampling strategies. Speculative decoding with a draft tree can be viewed as a search with a scope broader than that of the LLM, but narrower than the entire exponential space.

It is reasonable to model the *seen information* using frequency as an indicator of the confidence in the likelihood of a continuation occurring in the future, given the long-tail property of natural language. Thus, we select n-grams for the Retrieval Tree. However, for the unseen portion, the hidden states of LLMs encode more than just the next token (Mehra et al., 2025; Dong et al., 2025). This implies that the top-k logits not only reflect the next token but also include some subsequent tokens. As a limiting case, setting k to the vocabulary size $|\mathcal{V}|$, which can range up to 32K or even 150K, would result in a *copy-logit* accuracy of 100%. However, due to the long-tail property of language, it is clear that covering the entire vocabulary is unnecessary. A significantly smaller k (e.g., 63, as used in our preliminary experiment) can still capture the advantages from the top spikes in the probability distribution. To further investigate this, we conduct ablation studies

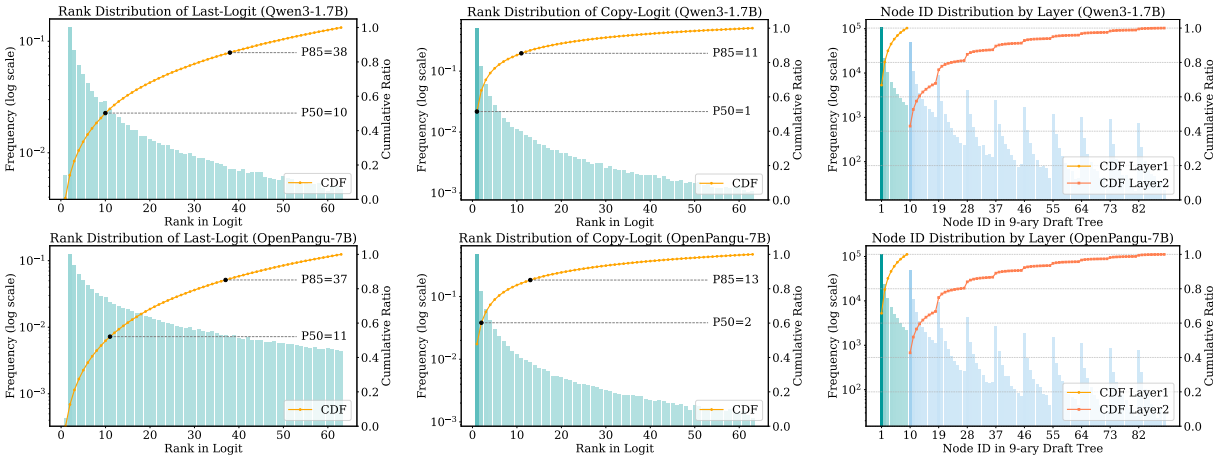

Figure 12: Accepted draft statistics of Qwen3-1.7B and OpenPangu-7B on Spec-Bench: *(Left)* Logits Tree with one layer beyond next-token expanded with *last-logit*. *(Middle)* Logits Tree with one layer beyond next-token expanded with *copy-logit*. *(Right)* Logits Tree with two layers beyond next-token expanded with *copy-logit* in 9-ary manner.

on the breadth and examine how different logit-reuse strategies influences the mean accepted tokens (MAT).

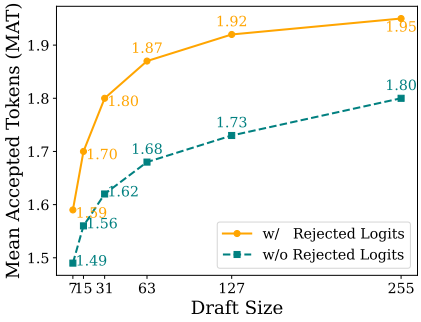

Figure 13: Ablation study on search breadth (top-k) for extending only one layer with *copy-logit* using Vicuna-7B.

Two settings will be discussed. **w/ Rejected Logits:** Reuse all k logits, even those rejected by the current verification. **w/o Rejected Logits:** Reuse at most two logits from the next token and one draft token (only if accepted). Both settings have the same limit when $k = |\mathcal{V}|$, where the MAT will be 2 (one sampled next token and one accepted draft token). Reusing rejected logits could provide updated context information for future reuse. For example, if AB and AC are proposed with prefix X, and only AB is accepted, the rejected logit of AC could still be used to approximate the logit of ABC. As a result, the first setting could accumulate advantages, narrowing the search scope more effectively than the second one.

Figure 13 demonstrates that reusing rejected logits consistently results in more accepted tokens, even when only 7 draft tokens are used (1.59 vs. 1.49). As the draft size increases, the gap between the two settings also widens. When using a draft size of 255 logits, the MAT reaches 1.95, which is very close to the theoretical maximum of 2. In contrast, reusing only 2 logits results in a MAT of 1.80. This suggests that the accumulative pruning of the draft scope is achieved with the first setting, and this attribute is crucial in the self-speculative procedure.

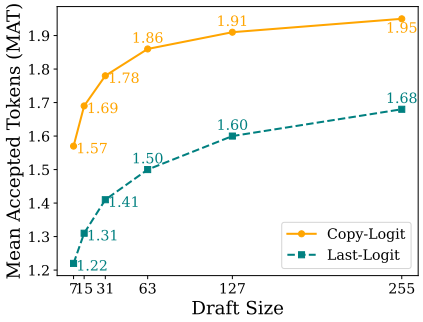

Figure 14: Ablation study on search breadth (top-k) for extending only one layer with *copy-logit* and *last-logit* using Qwen3-8B.

In Figure 14, *copy-logit* continues to improve MAT as k increases, approaching 2.0 when $k = 127/255$, whereas *last-logit* exhibits significantly slower growth and remains at 1.68 even at $k = 255$. This confirms that the heavy-tail advantage of copy-logit is not restricted to a single model or a single setting.

Supporting this observation, Figure 12 and

the supplementary experiments confirm the same trend seen in the main results (Figure 1 and Figure 2). Specifically, the *copy-logit* expansion consistently produces a sharper, heavy-tailed acceptance distribution compared to *last-logit*, with most accepted tokens concentrated in the top ranks. Furthermore, deeper k-ary expansions preserve this concentration and further validate the breadth allocation rule. These findings confirm that the phenomena discussed in the main text are robust across different model scales and settings, thereby reinforcing our choice of *copy-logit* as the default expansion strategy in RACER.

Actually, the expansion of the Logits Tree can be interpreted as a form of "beam search" in which the exact scores of individual paths are unavailable and are instead approximated via head dominance. Referring to Eq. 3, let the root breadth be $b_0 = k$, and define $m = \lfloor \log_2(k) \rfloor$. By binary decomposition, k can be written as

$$\begin{aligned} k &= \sum_{i=0}^{m} \alpha_i 2^i, \qquad \alpha_i \in \{0,1\},\ \alpha_m = 1, \\ &= 2^m + r, \quad r = \sum_{i=0}^{m-1} \alpha_i 2^i,\ 0 \le r < 2^m. \end{aligned} \tag{4}$$

Under this formulation, the maximum number of nodes in any single layer (i.e., the maximum layer breadth) can be expressed as

$$\mathcal{L}(k) = \sum_{i=0}^{m-1} \alpha_i (i+2) 2^i + 2^m \cdot m + 1. \tag{5}$$

We show that $\mathcal{L}(k)$ is upper bounded by $k\lceil \log_2(k) \rceil + 1$, and that the first occurrence of the maximum layer appears at depth $\lfloor \log_2(k) \rfloor + 1$.

Derivation of Layer Breadth To formalize the derivation, we introduce an auxiliary quantity $G(k')$, which denotes the maximum layer breadth of a subtree whose root node has maximum allowable breadth k' under the same expansion rule. Here, the bias term $[i \neq 0]$ in Eq. 3 is taken into account when considering deeper layers.

Let $m' = \lfloor \log_2(k') \rfloor$. The quantity $G(k')$ satisfies the following recursive characterization:

$$G(k') = \begin{cases} \displaystyle\sum_{i=1}^{m'} G\left(\left\lfloor \frac{k'}{2^i} \right\rfloor\right) + k' - m', & k' > 1, \\ 1, & k' = 1. \end{cases} \tag{6}$$

Starting from the leaves and aggregating contributions bottom-up, the maximum layer breadth of the entire Logits Tree with root breadth $b_0 = k$ can then be written as

$$\mathcal{L}(k) = \begin{cases} \displaystyle\sum_{i=0}^{m-1} G\left(\left\lfloor \frac{k}{2^i} \right\rfloor\right) + k - m, & k > 1, \\ 1, & k = 1. \end{cases} \tag{7}$$

For brevity, we omit the intermediate algebraic steps; the closed-form expression in Eq. 5 can be obtained by resolving the recursions in Eq. 6 and Eq. 7. It is also straightforward to see that the maximum layer breadth first occurs at depth $\lfloor \log_2(k) \rfloor + 1$. Beyond this depth, all node breadths reduce to 1 and the tree no longer expands, whereas before this depth, there always exists at least one node with breadth greater than 1, causing the layer size to continue increasing.

Upper Bound of Layer Breadth For $0 \le i \le m-1$, we have $i + 2 \le m + 1$, which yields

$$\begin{aligned} \sum_{i=0}^{m-1} \alpha_i (i+2) 2^i &\le (m+1) \sum_{i=0}^{m-1} \alpha_i \cdot 2^i \\ &= (m+1) \cdot r. \end{aligned}$$

Since $r = k - 2^m$, it follows that

$$\begin{aligned} \mathcal{L}(k) &\le (m+1) \cdot r + 2^m \cdot m + 1 \\ &= (m+1)k - 2^m + 1. \end{aligned}$$

When $k = 2^m$, we have $m = \lceil \log_2(k) \rceil$, and thus

$$\mathcal{L}(k) = 2^m \cdot m + 1 = k \cdot \lceil \log_2(k) \rceil + 1.$$

When $k > 2^m$, we have $m = \lceil \log_2(k) \rceil - 1$, which leads to

$$\begin{aligned} \mathcal{L}(k) &\le (m+1)k - 2^m + 1 \\ &= \lceil \log_2(k) \rceil \cdot k - 2^m + 1 \\ &< k\lceil \log_2(k) \rceil + 1. \end{aligned}$$

This completes the proof that $\mathcal{L}(k) \le k\lceil \log_2(k) \rceil + 1$, with equality holding if and only if k is a power of two.

G Comparison with EAGLE-3

As a model-free method, RACER is orthogonal to model-based methods like EAGLE-3. While EAGLE-3 benefits from extensive training and

Table 11: Results on reasoning tasks evaluated on NVIDIA A800 (80GB): GSM8K, AIME, and MATH, reported in mean accepted tokens (MAT) and speedup ratio. † denotes that the EAGLE-3 model weight is from AngelSlim's re-implementation.

Model	**Method**	**GSM8K**		**AIME**		**MATH**		**Average**	
		MAT	Speedup	MAT	Speedup	MAT	Speedup	MAT	Speedup
Qwen3 8B	EAGLE-3†	**3.86**	2.65	**3.44**	2.44	**3.55**	**2.60**	3.62	2.56
	RACER	3.01	**2.68**	2.91	**2.63**	2.88	2.58	2.93	**2.63**
Qwen3 14B	EAGLE-3†	**3.08**	2.23	**3.05**	2.24	**3.06**	2.23	**3.06**	2.23
	RACER	2.95	**2.72**	2.90	**2.64**	2.87	**2.55**	2.91	**2.64**
Qwen3 32B	EAGLE-3†	**3.32**	2.51	**3.26**	**2.34**	**3.33**	**2.42**	**3.30**	**2.42**
	RACER	2.87	**2.53**	2.84	2.30	2.82	2.32	2.84	2.38

model-based optimizations, RACER provides significant acceleration without requiring any additional training. Here, we explore in which scenarios RACER can complement EAGLE-3, potentially providing better acceleration in a hybrid setting with minimal modification to EAGLE-3.

In the supplementary results with Qwen3, we include several reasoning tasks: GSM8K (Cobbe et al., 2021), AIME (Veeraboina, 2023), and MATH (Hendrycks et al., 2021), using AngelSlim's re-implementation of EAGLE-3 model weights. We select 250 (+2) samples from each dataset: the first 250 samples for GSM8K, 250 random samples for AIME (from 1983 to 2024), and 50 random samples across levels 1 to 5, and 2 samples from level ? for MATH.

Table 11 shows that EAGLE-3 achieves higher MAT across all settings, which is expected given its task-specific training and model-based predictive capability. However, RACER attains comparable or even higher speedups in several cases (e.g., Qwen3-8B on GSM8K/AIME and Qwen3-14B across all tasks), despite having lower MAT. This highlights a key advantage of model-free decoding: RACER incurs nearly constant draft-generation cost, so its speedup does not degrade as sharply as computation-heavy model-based methods when model size increases. Across Qwen3-14B and Qwen3-32B, RACER maintains stable acceleration (2.38-2.72×), while EAGLE-3's speedup remains limited by verification overhead. These results suggest that RACER's speculative drafting is highly efficient even for long reasoning tasks, and that its acceleration stems from computational structure rather than model training. Most importantly, this comparison illustrates that RACER and EAGLE-3 offer orthogonal strengths: EAGLE-3 excels when high-quality next-step predictions are available via training. RACER excels when low overhead and robustness across tasks and model sizes are required.