



EAGLE-4: One Flexible Drafter for All Batch Sizes and Hardware

EAGLE-4 Team

Eagle AI

hongyang@eagle.ai

<https://github.com/eagle-hq/EAGLE-4>

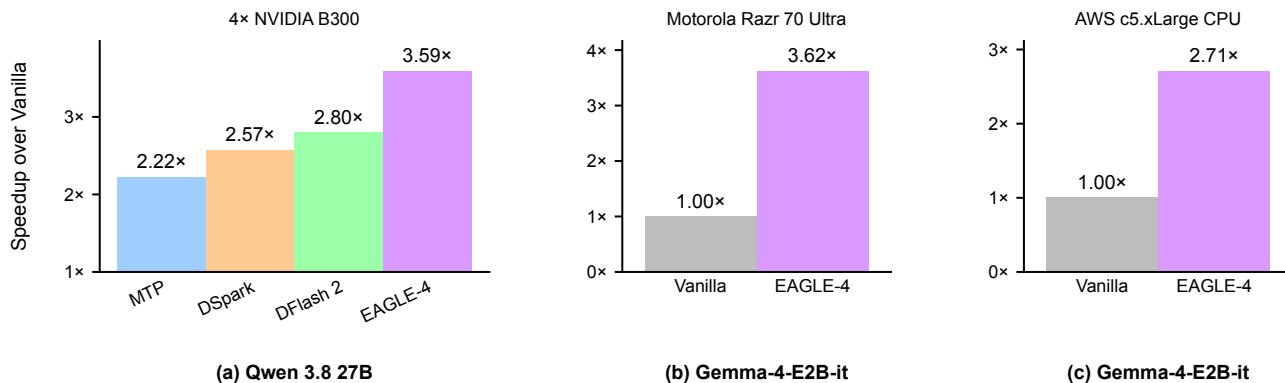


Figure 1. Single-request throughput speedup over vanilla decoding at $T = 0$. (a) Qwen 3.8 27B on the SPEED-Bench qualitative split using $4\times$ NVIDIA B300 GPUs. (b, c) 4-bit Gemma-4-E2B-it: (b) HumanEval on the CPU of a Motorola Razr 70 Ultra, measured before thermal throttling; (c) the arithmetic mean of speedups across GSM8K, MATH-500, HumanEval, and MT-Bench on an AWS c5.xlarge CPU instance.

Abstract

Speculative decoding has emerged as a key technique for accelerating large language model inference. In this work, we advocate for autoregressive drafting and argue that its sequential nature provides greater flexibility across diverse inference regimes. We introduce EAGLE-4, the latest iteration of the EAGLE family, which achieves 1.28x the decoding throughput of the previous state of the art and 1.62x that of the target model’s built-in multi-token prediction (MTP) head at batch size 1, while using a single drafter that dynamically adapts to different batch sizes and hardware platforms. In contrast to parallel drafting methods, which typically rely on fixed block sizes and may require separately optimized drafters for different inference settings, EAGLE-4 flexibly adjusts its draft structure according to the batch size and hardware-specific inference costs. Building on the EAGLE Efficient LM HeadTM, which reduces draft-model overhead by more than 40%, and EAGLE-HarnessTM, which enables hardware-aware draft adaptation, EAGLE-4 delivers 1.70x and 1.99x the throughput of DFlash 2 and DSpark

at 512 concurrent requests, and it is faster than vanilla decoding in every end-to-end GPU, CPU and mobile measurement we report. These results demonstrate that autoregressive drafting provides a flexible foundation for efficient speculative decoding across diverse inference regimes, while achieving state-of-the-art performance at batch size 1. The code is available here: <https://github.com/eagle-hq/EAGLE-4>.

1. Introduction

Speculative decoding has been widely adopted in production-grade inference engines. It was originally developed primarily for small-batch inference, where large language model (LLM) inference is often memory-bound on modern AI accelerators. Because autoregressive models generate tokens sequentially, processing only a small number of tokens at each decoding step leads to low hardware utilization and makes memory movement a major contributor to inference latency. Speculative decoding addresses this inefficiency by using a lightweight draft model to predict several future tokens in advance, which are then verified in parallel by the target model. By increasing the number of tokens pro-

cessed per target-model invocation, speculative decoding can reduce the effective cost of autoregressive generation.

In real-world LLM serving, batch size can vary substantially over time, leading to different compute and memory constraints. Under small-batch workloads, longer draft lengths are often preferable because the system has more available compute capacity, whereas large-batch workloads are more compute-intensive and may benefit from shorter draft lengths. Autoregressive draft models naturally provide the flexibility to adapt draft length to changing serving conditions. In contrast, existing block-drafting methods, such as DFlash (Chen et al., 2026; Inco AI, 2026) and DSpark (Cheng et al., 2026), require the block size to be specified in advance as a predefined hyperparameter, limiting their ability to dynamically adjust drafting computation to the current workload. This limitation becomes particularly pronounced at extremely large batch sizes, where the optimal block size may shrink to only one or two tokens. In this regime, the computational advantage of parallel block drafting diminishes, and block drafting increasingly approaches autoregressive drafting in terms of its effective speculation granularity.

This work represents the latest iteration of the EAGLE series (Li et al., 2024a;b; 2025; Eldenk et al., 2026), which has become a widely adopted architecture for autoregressive drafting. Multi-Token Prediction (MTP) has followed the same architectural paradigm as EAGLE since DeepSeek-V3 (Liu et al., 2024), and this design has subsequently been adopted by the MTPs of mainstream large language models, including DeepSeek-V4 (Xu et al., 2026), GLM-5 (Zeng et al., 2026), Kimi K3 (Kimi Team et al., 2026), Qwen 3 (Yang et al., 2025), and Gemma 4 (Gemma Team et al., 2026). Specifically, existing MTP implementations follow the core EAGLE design: they concatenate the target model’s hidden state with the corresponding token embedding and feed the resulting representation into a lightweight draft model to predict subsequent tokens. Building on this established architecture, EAGLE-4 offers the following additional advantages:

- Compared with MTPs (Liu et al., 2024; Xu et al., 2026; Zeng et al., 2026; Kimi Team et al., 2026; Yang et al., 2025; Gemma Team et al., 2026), EAGLE-4 achieves substantially higher throughput in small-batch settings, enabled by its hardware-aware tree drafting strategy. The strategy first characterizes the performance of the target hardware through a lightweight warm-up run and then dynamically adapts the draft tree size to the hardware and inference environment. The strategy automatically determines when to stop drafting based on the trade-off between draft cost and the confidence of the draft model.
- Compared with official checkpoints from DFlash 2 (Inco AI, 2026) and DSpark (Cheng et al., 2026), EAGLE-4 achieves $1.28\times$ and $1.40\times$ the throughput, respectively, at concurrency 1 and temperature $T = 0$. On the SPEED-Bench qualitative split with Qwen 3.8 27B, EAGLE-4 achieves 863.8 tokens/s, compared with 672.6 tokens/s for DFlash 2 and 618.6 tokens/s for DSpark (Table 2). At large batch sizes, fixed-size block drafting can incur redundant computation. DFlash 2 and DSpark use a fixed block size, whereas the optimal speculation length tends to decrease as the batch size increases. As a result, these methods may generate more draft tokens than can be effectively utilized, with excess tokens ultimately discarded and their drafting computation wasted. In contrast, EAGLE-4 dynamically adjusts its draft length based on the target hardware and current inference workload, avoiding unnecessary drafting computation.
- EAGLE-4 automatically adapts its speculation budget to different inference hardware and workloads. On $4\times$ NVIDIA B300 GPUs, EAGLE-4 delivers $3.59\times$ the throughput of vanilla decoding at concurrency 1 and temperature $T = 0$ (Table 2). For a single request at temperature $T = 0$, EAGLE-4 achieves an average $2.71\times$ speedup over vanilla decoding across four benchmarks on an AWS c5.xlarge instance with 4 vCPUs and 8 GB of RAM. On a Motorola Razr 70 Ultra, a commercially available smartphone powered by the Snapdragon 8 Elite with 8 CPU cores and 16 GB of RAM, EAGLE-4 achieves a $3.62\times$ speedup over vanilla decoding on HumanEval for a single request at temperature $T = 0$.

EAGLE-4 also retains the lightweight single-layer Transformer of the EAGLE series. The public DFlash 2 and DSpark drafters for Qwen 3.8 27B, in contrast, each stack five Transformer layers, so every drafted position passes through roughly 1.66B layer parameters compared with 0.44B for EAGLE-4 (Table 1). **EAGLE-4 therefore spends about $3.8\times$ fewer FLOPs per drafted token**, which matters most at large batch sizes, where the drafter’s matrix multiplications become compute-bound. The same compact design suits edge devices with tight memory budgets, and the EAGLE-4 head for Gemma 4 occupies only 134 MB.

1.1. Observations

EAGLE-4 is motivated by three observations that enable flexible speculation across different batch sizes and hardware.

Firstly, parallel drafting is constrained by a fixed block size, whereas autoregressive drafting naturally supports adaptive draft lengths through test-time tree-size ad-

Table 1. Drafter backbone size for Qwen 3.8 27B. Transformer-layer parameters exclude the feature-fusion projection, token embeddings and output-vocabulary layers, and are read from the released checkpoints.

Drafter	Layers	Layer params per drafted position
EAGLE-4 (ours)	1	0.44B
MTP (built-in)	1	0.37B
DFlash 2	5	1.66B
DSpark	5	1.65B

justment. DFlash predicts a block of tokens in a single diffusion-model pass, while DSpark combines a parallel backbone with a lightweight sequential head. Although these approaches reduce sequential drafting latency, their multi-layer backbones still introduce substantial computational overhead. More importantly, the optimal trade-off between drafting and verification costs varies across serving workloads and batch sizes: small batches may have sufficient compute capacity to accommodate longer drafts, whereas large batches make both drafting and verification more expensive. Maximizing throughput therefore requires dynamically balancing the potential benefit of longer speculation against its associated computational cost, rather than using a fixed draft size across different serving conditions.

Autoregressive drafting provides a natural mechanism for making this trade-off during proposal construction. After each drafting step, the drafter can decide whether to extend the current chain or stop, thereby avoiding computation on an unpromising suffix. Hybrid parallel-autoregressive methods can also adapt the amount of verification; for example, DSpark selects a verification prefix based on confidence estimates and serving-load information. However, once a block has been generated, shortening the verification prefix cannot recover the computation already spent generating discarded tokens. In contrast, a fully autoregressive drafter makes the stop-or-continue decision *before* executing the next drafting step, using the predictions obtained so far to avoid unnecessary computation. This enables draft length to adapt dynamically to the expected benefit of further speculation and the prevailing serving cost.

Although DSpark supports dynamic chain drafting (Cheng et al., 2026) and DFlash can be extended to support DDTree (Ringel & Romano, 2026), we argue that the benefits of dynamic drafting are more limited in these block-based approaches than in fully autoregressive drafting. The key difference lies in how dependencies among drafted token positions are modeled. Autoregressive drafting explicitly conditions each newly drafted token on the previously generated tokens, i.e., $p(x_{t+1} \mid x_{\leq t})$, thereby naturally capturing the causal dependencies along each draft branch. This allows subsequent drafting decisions to incorporate the actual tokens generated so far, making adaptive tree expansion par-

ticularly effective. In contrast, block-based methods such as DFlash generate multiple token positions jointly, while hybrid methods such as DSpark introduce sequential conditioning only through a lightweight autoregressive head after parallel generation. Consequently, the dependencies among token positions within the parallel block are either jointly modeled or only weakly conditioned on preceding draft tokens. Dynamic pruning or expansion can still adjust the amount of computation, but has less opportunity to exploit the branch-specific information revealed by earlier draft tokens, making the benefit of dynamic drafting inherently smaller than in fully autoregressive drafting.

Secondly, existing dynamic tree strategies are hardware-agnostic. Autoregressive drafting also enables flexible tree expansion. As in EAGLE-3 (Li et al., 2025), each branch can be extended conditioned on its own drafted prefix, allowing computation to be preferentially allocated to promising continuations. The tree can thus be widened, deepened, or truncated as drafting proceeds, adapting its structure to the current prediction confidence. EAGLE-4 builds on EAGLE-3 by further introducing *hardware-aware drafting*, which explicitly accounts for the computational cost of speculation. Hardware-specific cost measurements are collected once during a warm-up phase and cached for subsequent inference. Together with the current workload and the estimated benefit of further speculation, these measurements determine whether and how far to extend each draft chain or tree (Section 3.1). This design dynamically adjusts the speculation budget to the serving environment and workload, retaining more speculation when its expected benefit justifies the cost while limiting unnecessary drafting and verification overhead under heavier workloads.

EAGLE-4 and DSpark (Cheng et al., 2026) both adapt speculative decoding to the runtime inference cost, but they operate at different levels of the decoding process. DSpark generates a sequential draft block and dynamically determines the verification prefix length for each request based on prefix survival probabilities and the target engine’s throughput under the current load. In contrast, EAGLE-4 adopts a hardware-aware dynamic tree construction mechanism that jointly accounts for the costs of both drafting and verification, and dynamically determines how much speculative computation to perform in the first place. Specifically, EAGLE-4 can adapt the draft depth and per-layer width, as well as the number of tokens ultimately verified, according to the GPU, batch size, and runtime cost profile. Thus, while DSpark dynamically allocates the verification budget over a fixed sequential draft, EAGLE-4 dynamically adapts the structure and amount of speculative computation itself.

Thirdly, a substantial portion of the cost of autoregressive drafting comes from the LM head. As the vocabulary size of LLMs continues to grow, the computational over-

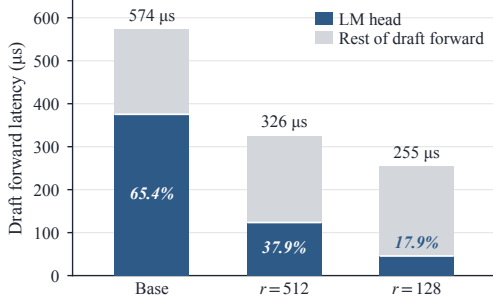


Figure 2. Latency breakdown of a single draft forward pass using the full LM head (Base) and EAGLE Efficient LM HeadTM with ranks $r = 512$ and $r = 128$. Blue and gray denote LM-head computation and the remaining draft-forward computation, respectively. Labels above the bars indicate total latency, while percentages show the fraction attributed to LM-head computation.

head of the LM head becomes increasingly significant for autoregressive drafters. This overhead is a key motivation behind parallel drafting methods. Although methods such as FR-Spec (Zhao et al., 2025), VocabTrim (Goel et al., 2025), BCL (Shoham, 2026), CORAL (Weng et al., 2025), and DynaSpec (Zhang et al., 2025) reduce LM-head computation by truncating the vocabulary, they can suffer from reduced acceptance length, particularly on more challenging tasks, because the truncated vocabulary may exclude relevant candidate tokens. We argue that the LM-head overhead can instead be substantially reduced with an efficient low-rank LM head that first narrows the candidate space and then performs exact scoring only on a small set of candidates, thereby reducing computation without sacrificing acceptance length (see Figure 2). In addition, lightweight attention mechanisms can further reduce the cost of processing long contexts. Specifically, our efficient LM head operates in two stages: it first generates a coarse candidate set and then selects the final draft tokens by exactly scoring only the candidates within this set. We describe these components in detail in Section 3.

We note an independent work on efficient LM heads, SlimSpec (Plaksin et al., 2026). Unlike SlimSpec, which uses a low-rank projection as the final LM head, EAGLE-4 uses the low-rank projection solely for candidate retrieval. Specifically, we first apply a low-rank head over the full vocabulary to identify a small set of candidate tokens, and then use the original full-rank LM head to compute exact logits for the selected candidates. This design decouples candidate retrieval from probability estimation: the low-rank head only needs to identify promising candidates, while their final scores remain exact. Thus, approximation errors in the low-rank head do not directly affect draft selection as long as the relevant candidates are successfully retrieved. This distinction also motivates our training objective, which focuses the limited capacity of the low-rank head on high-

probability candidates rather than requiring it to accurately model the full output distribution.

2. Preliminaries

2.1. Speculative Decoding

Speculative decoding alternates between inexpensive drafting and parallel target-model verification. Given a prefix $x_{1:t}$, a drafter samples m proposals $\hat{x}_{1:m}$. Let p_i and q_i denote the target and draft distributions conditioned on $x_{1:t}$ and $\hat{x}_{1:i-1}$. A single target forward pass evaluates the proposed continuation. Tokens are checked sequentially, with acceptance probability

$$\alpha_i = \min \left(1, \frac{p_i(\hat{x}_i)}{q_i(\hat{x}_i)} \right).$$

At the first rejection, at position j , the remaining proposals are discarded and a replacement is drawn from

$$p_j^{\text{corr}}(v) = \frac{[p_j(v) - q_j(v)]_+}{\sum_{u \in \mathcal{V}} [p_j(u) - q_j(u)]_+},$$

where $\mathcal{V}$ is the vocabulary and $[a]_+ = \max(a, 0)$. If all proposals are accepted, an additional token is sampled from the target distribution after the full draft. This procedure preserves the target model’s output distribution.

For greedy decoding, verification retains the longest draft prefix matching the target’s greedy predictions, then appends the target’s next token. Acceleration depends on both the number of tokens advanced per verification round and the combined cost of drafting and verification, motivating the hardware-aware design in Section 3.1.

2.2. The EAGLE Series

EAGLE (Li et al., 2024a) uses target hidden states to inform a lightweight autoregressive drafter. It predicts the next feature before the target LM head, which converts the prediction into token logits. The input includes token embeddings shifted one position ahead of the features, exposing the sampled token that determines the next feature. Training combines feature regression and token prediction. EAGLE also supports a fixed draft tree: tree attention lets each node attend to its own branch and the shared prefix, allowing the target to evaluate multiple candidate continuations in one forward pass.

EAGLE-2 (Li et al., 2024b) replaces the fixed tree with a context-dependent structure. It uses draft probabilities as estimates of acceptance likelihood and ranks nodes by the product of these probabilities along their paths from the root. Promising nodes are expanded during drafting, and the resulting candidates are reranked to select the tree for verification. This adapts the allocation of draft tokens to the current context without retraining the drafter.

EAGLE-3&3.1 (Li et al., 2025; Eldenk et al., 2026) removes the feature-regression objective and trains the drafter directly for token prediction. It fuses target hidden states from low, middle, and high layers into a single representation, which is combined with the embedding of the next token to generate a draft proposal. Since new target hidden states are unavailable until verification, subsequent draft steps recursively reuse the drafter’s own hidden states. To expose the model to these recurrent inputs during training, EAGLE-3 adopts a *training-time test* procedure that feeds the drafter’s outputs back into subsequent training steps. EAGLE-3 remains compatible with the dynamic tree construction introduced in EAGLE-2. EAGLE-3.1 (Eldenk et al., 2026) further improves the architecture with two modifications: post-normalization of the drafter hidden states and per-hidden-state RMSNorm applied after capturing target hidden states. EAGLE-4 builds upon this autoregressive drafting architecture.

3. EAGLE-4

In this section, we provide a detailed description of the implementation of EAGLE-4.

3.1. EAGLE-HarnessTM

Existing dynamic speculative decoding methods primarily optimize draft-tree construction according to predicted acceptance probability, without explicitly considering the inference cost of different tree structures (Li et al., 2024b; Cai et al., 2024). Let $\mathcal{T}$ denote a candidate draft tree and $A(\mathcal{T})$ its expected number of accepted tokens. While a wider or deeper tree can increase $A(\mathcal{T})$, it also introduces additional drafting and verification overhead, whose cost depends on the target and draft models, hardware platform, and batch size. **Thus, the optimal speculator should dynamically balance acceptance gain and inference cost, rather than optimize acceptance length alone.** The EAGLE-HarnessTM addresses this issue through inference-cost-aware dynamic tree construction. For each hardware platform and workload, it introduces an additional cost-estimation phase that profiles the inference latency of the draft and target models under different batch sizes, context lengths, and numbers of generated tokens. The measured costs are stored in lookup tables and cached for subsequent inference, providing hardware-specific estimates of the cost associated with different tree structures. During inference, these cached measurements are combined with the expected acceptance of candidate trees to dynamically determine the tree width, depth, and verification budget, allowing the draft tree to adapt to the underlying inference environment.

How EAGLE-Harness decides. The key insight is that one more draft step, or one more verified token, is worth taking only if the tokens it is expected to add outweigh the time it costs on the current hardware at the current load. The drafter already estimates the first quantity, since, following EAGLE-2, the product of draft probabilities along a branch approximates the probability that the branch is accepted. The profiling phase supplies the second. The harness therefore reduces every decision to a single comparison,

$$\frac{\Delta A}{\Delta C} \geq \tau,$$

where ΔA is the increase in the expected number of accepted tokens $A(\mathcal{T})$, ΔC is the profiled increase in drafting or verification time, and τ is a threshold. It applies this comparison at three points of every decoding round (Algorithm 1). First, after each draft step, it decides whether to draft one level deeper, with a threshold that rises with depth because estimates for deep branches are less reliable. Second, it decides how many of the drafted tokens to send for verification, adding the highest-scoring tokens one at a time for as long as **each additional token is worth its measured verification cost**. Third, it reads the current load from the profiled cost of verifying a fixed number of tokens, allowing up to eight branches per node when the system is lightly loaded and at most four under heavy load. The same drafter therefore builds deep, wide trees for a single request and short chains under heavy load, without any change to its weights. Because each decision uses only table lookups and scores the drafter already computes, the harness adds no work to the target model.

At $T > 0$, stopping on the tokens drawn in the current round would bias the output distribution. The harness therefore fixes the depth of each sampled round in advance, using the depth at which the previous round would have stopped, which preserves the target distribution while still tracking the cost-optimal depth.

3.2. EAGLE Efficient LM HeadTM

Computing full-vocabulary logits can account for a substantial fraction of the cost of a lightweight drafter. EAGLE-3 (Li et al., 2025) reduces this cost with a fixed, truncated draft vocabulary, which can limit draft accuracy. EAGLE-4 instead uses two stages: inexpensive candidate selection from the full vocabulary with a low-rank head, followed by exact scoring of the selected candidates with the full LM head.

Figure 2 illustrates the impact on drafting cost. In the profiled configuration, the full LM head accounts for 65.4% of the 574 μ s draft forward pass. The rank-512 and rank-128 efficient heads reduce total latency to 326 μ s and 255 μ s, respectively, corresponding to reductions of 43.2% and 55.6%. The LM head’s share falls to 37.9% and 17.9%, while the

Algorithm 1 One decoding round of EAGLE-Harness. A is the expected number of accepted tokens, $\hat{p}_i$ the estimated acceptance probability of draft node i (the product of draft probabilities along its branch), and τ_d and τ_v the drafting and verification thresholds, with τ_d rising with depth d .

Input: batch size B , average context length L , depth limit D , verification limit N
 Read the draft-step cost c_{draft} and verification costs $c_{\text{ver}}(n)$ at (B, L) from the profiled tables
 $\mathcal{T} \leftarrow$ root token
for $d = 1$ **to** D **do**
 Expand the most promising branches of $\mathcal{T}$ by one level
 if $\Delta A_d / c_{\text{draft}} < \tau_d$ **then**
 break {the next level is not worth its cost}
 end if
end for
 Sort the nodes of $\mathcal{T}$ so that $\hat{p}_1 \geq \hat{p}_2 \geq \dots$
 $n \leftarrow 1$
while $n < \min(N, |\mathcal{T}|)$ **and** $\hat{p}_{n+1} / (c_{\text{ver}}(n+1) - c_{\text{ver}}(n)) \geq \tau_v$ **do**
 $n \leftarrow n + 1$ {the next token is worth its cost}
end while
 Verify the top- n nodes with the target model in one forward pass

cost of the remaining draft computation stays approximately constant. At rank 128, the LM head is therefore no longer the dominant component of drafting latency.

3.2.1. ARCHITECTURE

Let $h \in \mathbb{R}^d$ be the drafter’s output hidden state and $W \in \mathbb{R}^{V \times d}$ its full LM head, where V is the vocabulary size and d is the dimension of the hidden state. A standard head computes $z = Wh$. We introduce a rank- r approximation $\tilde{W} = AB$, with $A \in \mathbb{R}^{V \times r}$ and $B \in \mathbb{R}^{r \times d}$, and first compute

$$\tilde{z} = A(Bh), \quad \mathcal{C} = \text{TopK}(\tilde{z}, K),$$

where TopK returns the indices of the K largest entries. We use $K \in \{128, 256, 512\}$. We then gather the corresponding rows of W within the set $\mathcal{C}$ and compute

$$z_{\mathcal{C}} = W_{\mathcal{C}}h.$$

Subsequent draft selection uses these exact logits within $\mathcal{C}$. In this two-phase process, the low-rank head therefore determines which tokens are considered, while the full head determines their scores. Since candidate selection covers the full vocabulary, the candidate set can change with the current hidden state.

The projection arithmetic per hidden state decreases from $\mathcal{O}(Vd)$ to $\mathcal{O}(dr + Vr + Kd)$, excluding the overhead of

top- K selection and row gathering. The full matrix W is retained, but only its selected rows are used in the second stage.

A truncated SVD of W is a natural initialization for the low-rank head, but at rank 128 it reduces the acceptance length of one Qwen 3.8 27B drafter by 0.48 tokens per verification. Because the low-rank head only selects candidates and the full head scores them, we train the two factors for candidate retrieval alone, with the rest of the drafter frozen. **The 128 candidates of the trained head then contain the full head’s top-1 token in 98.2% of drafting steps, compared with 91.5% for its SVD initialization**, and the trained head shows no acceptance-length loss beyond run-to-run noise while halving the draft time per verification cycle from 9.1 to 4.8 ms at batch size 1.

3.3. Attention and Cache Management

3.3.1. SPARSE AND SLIDING-WINDOW ATTENTION

For target models that use sparse or linear attention, a drafter with full attention can become a throughput bottleneck at long context lengths. EAGLE-4 therefore considers lighter attention mechanisms, including DeepSeek Sparse Attention (DSA) (DeepSeek-AI, 2025) and sliding-window attention (SWA) (Jiang et al., 2023), to reduce drafting overhead. SWA restricts attention to a local window, limiting the amount of historical context processed at each draft step.

3.3.2. APPEND-ONLY CACHE MANAGEMENT DURING DRAFTING

The drafter uses an attention context much smaller than the full sequence. During each drafting phase, newly generated cache entries are appended to this context without evicting its oldest entries. This append-only design simplifies cache management during drafting for attention mechanisms such as DSA and SWA.

3.4. MoE and Dense Feed-Forward Layers

We compare MoE and dense feed-forward layers in the drafter. In our experiments, MoE improves generalization across draft steps and context lengths, but increases memory requirements and training overhead. Balancing these trade-offs, we align the drafter’s feed-forward architecture with that of the target model.

4. Experiments

4.1. Inference on NVIDIA GPUs

Models. We conduct experiments with state-of-the-art open-source reasoning models, including Qwen 3.8 27B and Nemotron-3.5-Super-VL-120B-A12B.

Hardware. All results in this subsection are obtained on $4 \times$ NVIDIA B300 GPUs.

Tasks. We use the qualitative split of SPEED-Bench benchmark (Abramovich et al., 2026) to evaluate overall decoding performance. Table 2 compares overall throughput for EAGLE-4, DFlash 2,¹ DSpark,² MTP,³ and vanilla autoregressive decoding using Qwen 3.8 27B as the target model at decoding temperatures $T = 0$ and $T = 1$. We report end-to-end throughput at concurrency 1 and 512. The concurrency-512 runs use the qualitative split replicated four times (3,520 conversations) so that the batch stays full. EAGLE-4 uses its 12-4-16 EAGLE-Harness (depth 12, width 4, at most 16 verified tokens) at concurrency 1 and a 15-1-16 EAGLE-Harness at concurrency 512; MTP uses a 7-1-8 draft chain at concurrency 1 and a 1-1-2 draft chain at concurrency 512.

Table 3 compares overall throughput for EAGLE-4, MTP and vanilla autoregressive decoding using Nemotron-3.5-Super-VL-120B-A12B as the target model at decoding temperatures $T = 0$ and $T = 1$. EAGLE-4 uses its 15-1-16 EAGLE-Harness and MTP uses a 7-1-8 draft chain.

Metrics. We argue that acceptance length alone is insufficient to evaluate EAGLE-HarnessTM, as it does not capture the benefits of dynamic verification. Therefore, we focus on overall throughput rather than comparing acceptance lengths across methods.

Concurrency scaling. To complement the SPEED-Bench results, Figure 3 evaluates Qwen3.8-27B and Nemotron-3.5 on HumanEval (Chen et al., 2021) under the same closed-loop workload setting, with request concurrency C ranging from 1 to 512. All methods process the same requests at each concurrency for a given target model. For Qwen3.8-27B, EAGLE-4 uses EAGLE-Harness Tree for $C \in \{1, 2, 4\}$ and EAGLE-Harness Chain for higher concurrency. MTP uses a fine-tuned chain. The DFlash 2, DSpark, and vanilla decoding runs use the trtlm backend. For Nemotron, we compare EAGLE-4 with EAGLE-Harness Chain, MTP with a 7-1-8 fixed chain, and vanilla decoding. These results compare the complete serving configurations, including their respective attention backends and scheduling settings.

As shown in Figure 3(a), EAGLE-4 exceeds vanilla decoding at every plotted concurrency. At $C = 1$, EAGLE-4 achieves 929 tokens/s, compared with 759 for DFlash 2,

701 for DSpark, and 240 for vanilla decoding, corresponding to $1.22\times$, $1.33\times$, and $3.87\times$ their throughput, respectively. EAGLE-4 and DFlash 2 are nearly tied at $C = 4$ and $C = 32$, while EAGLE-4’s advantage grows at higher concurrency. At $C = 512$, EAGLE-4 reaches 32,935 tokens/s, delivering $1.67\times$ the throughput of DFlash 2, $1.95\times$ that of DSpark, and $1.43\times$ that of vanilla decoding. Both DFlash 2 and DSpark fall below vanilla decoding at $C = 256$ and $C = 512$. On Nemotron-3.5 (Figure 3(c)), EAGLE-4 remains ahead of both MTP and vanilla decoding at every concurrency we measure. At $C = 512$ it sustains about 40,800 tokens/s, $2.0\times$ **the throughput of vanilla decoding** and $1.17\times$ that of MTP. Each user still receives 87 tokens/s, compared with 45 under vanilla decoding.

Per-user throughput. Figure 3(b) presents the same runs in terms of aggregate and per-user throughput. Increasing concurrency raises aggregate throughput while reducing the generation rate experienced by each user. EAGLE-4 offers a more favorable trade-off at high concurrency: at $C = 128$, it achieves 23,063 tokens/s in aggregate at 201 tokens/s per user, compared with DSpark’s 11,334 tokens/s at a similar per-user rate of 197 tokens/s at $C = 64$. At $C = 512$, EAGLE-4 sustains 70 tokens/s per user, versus 48 for vanilla decoding, 41 for DFlash 2, and 35 for DSpark. Together, these measurements show that the EAGLE-4 serving configuration retains a throughput advantage under heavy load while sustaining higher per-user generation rates. EAGLE-4 also leads when throughput is compared at a fixed per-user speed. With every user served at 200 tokens/s or more, it delivers 23,063 tokens/s, $1.7\times$ **the throughput of DFlash 2**, and at 100 tokens/s per user it delivers about 28,500 tokens/s, $1.8\times$ DFlash 2 and $2.2\times$ vanilla decoding.

4.2. Inference on AMD GPUs

AMD team provides day-0 support and evaluates EAGLE-4 on a single AMD Instinct MI350X GPU (GFX950) with tensor parallelism 1. The workload comprises 117 turns from SPEED-Bench, selected with `every=10`, with a maximum of 4,096 newly generated tokens per request.

EAGLE-4 uses dynamic EAGLE-Harness tree with 12-4-32. At concurrency 1, this configuration reaches 262.89 tokens/s, compared with 260.79 tokens/s for 12-4-16, so we use it throughout the sweep. The maximum number of running requests equals client concurrency.

¹DFlash 2 official checkpoint: <https://huggingface.co/incoai/Qwen3.8-27B-DFlash2>

²DSpark RedHatAI checkpoint: <https://huggingface.co/RedHatAI/Qwen3.8-27B-speculator.dspark>

³MTP official checkpoint: <https://huggingface.co/Qwen/Qwen3.8-27B>

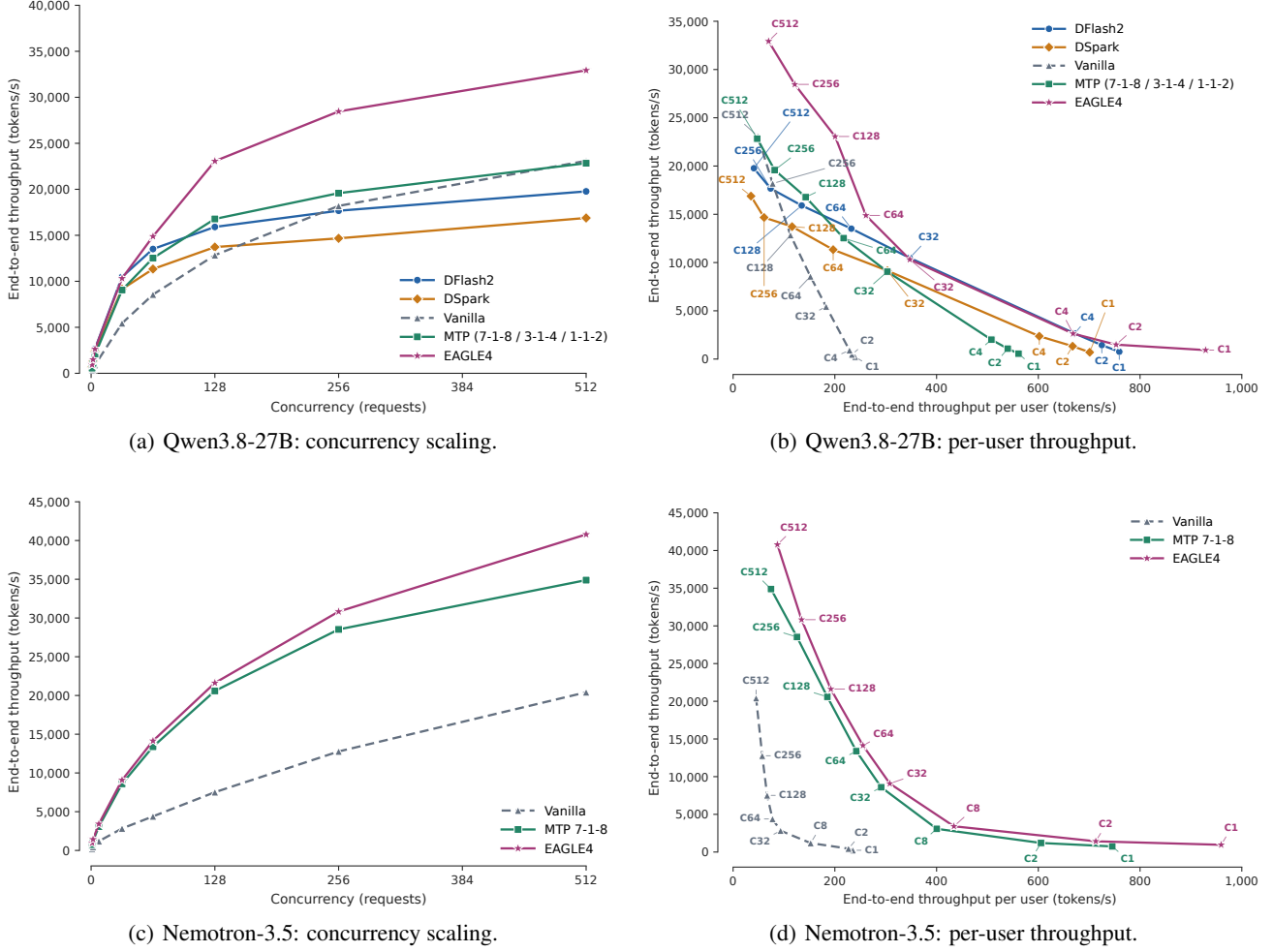


Figure 3. End-to-end throughput on HumanEval with Qwen3.8-27B (top) and Nemotron-3.5 (bottom), both on $4 \times$ NVIDIA B300 GPUs (tensor parallelism 4), with thinking enabled, temperature $T = 0$, and a 4,096-token output limit including reasoning tokens. (a, c) Aggregate throughput versus request concurrency. (b, d) Aggregate throughput versus per-user throughput; labels Ck denote concurrency k . Aggregate throughput is total output tokens divided by measured wall-clock time; per-user throughput divides the same token count by the sum of request latencies. Each row shows the same runs in both panels: $C \in \{1, 2, 4, 32, 64, 128, 256, 512\}$ for Qwen and $C \in \{1, 2, 8, 32, 64, 128, 256, 512\}$ for Nemotron. For Qwen MTP, we use 7-1-8 for $C1/2/4$, 3-1-4 for $C32-256$, 1-1-2 for $C512$. For Nemotron MTP, the target’s MoE sparsity and Mamba layers keep decoding memory-bound up to $C512$, so we keep the draft chain length fixed. HumanEval is replicated at large C so batches remain full.

Table 4. End-to-end throughput (tokens/s) on the 117-turn SPEED-Bench workload using one AMD Instinct MI350X GPU (TP1). Speedup is relative to vanilla decoding at the same concurrency.

Concurrency	Vanilla	EAGLE-4	Speedup
1	75.27	262.89	3.49$\times$
2	146.70	486.58	3.32$\times$
4	285.12	826.42	2.90$\times$
8	512.59	1,304.34	2.54$\times$
16	819.05	1,882.68	2.30$\times$
32	1,390.52	2,360.83	1.70$\times$

Table 4 shows that EAGLE-4 is faster than vanilla decoding at every tested concurrency. For a single request, throughput rises from 75.27 to 262.89 tokens/s, a $3.49\times$ speedup. The relative gain decreases with concurrency, while speedups remain $2.30\times$ at concurrency 16 and $1.70\times$ at concurrency 32. At concurrency 32, EAGLE-4 reaches 2,360.83 tokens/s. These measurements demonstrate throughput gains on AMD hardware across the evaluated loads.

4.3. Inference on CPUs

We evaluate EAGLE-4 on Gemma-4-E2B-it quantized to 4-bit, running entirely on CPU with `llama.cpp`. Vanilla

EAGLE-4: One Flexible Drafter for All Batch Sizes and Hardware

Table 2. Overall throughput on the qualitative split of SPEED-Bench with Qwen3.8-27B, evaluated on 4× NVIDIA B300 GPUs with tensor parallelism (TP) of 4. The maximum output length is capped at 4,096 tokens. All methods use the same inference engine, with MegaKernel disabled.

Method	Concurrency 1		Concurrency 512	
	Tokens/s	Speedup	Tokens/s	Speedup
Temperature $T = 0$				
EAGLE-4	863.8	(3.59×)	28,340	(1.34×)
DFlash 2	672.6	(2.80×)	16,641	(0.78×)
DSpark	618.6	(2.57×)	14,229	(0.67×)
MTP	534.5	(2.22×)	21,518	(1.01×)
Vanilla	240.5	(1.00×)	21,203	(1.00×)
Temperature $T = 1$				
EAGLE-4	608.6	(2.59×)	21,005	(1.03×)
DFlash 2	560.3	(2.39×)	14,217	(0.70×)
DSpark	519.4	(2.21×)	11,723	(0.58×)
MTP	369.1	(1.57×)	18,869	(0.93×)
Vanilla	234.9	(1.00×)	20,307	(1.00×)

Table 3. Overall throughput on the qualitative split of SPEED-Bench using Nemotron-3.5 on 4× NVIDIA B300 GPUs with tensor parallelism (TP) of 4. The maximum output length is capped at 4,096 tokens.

Method	Concurrency 1		Concurrency 512	
	Tokens/s	Speedup	Tokens/s	Speedup
Temperature $T = 0$				
EAGLE-4	647	(2.74×)	33,480	(1.69×)
MTP	568	(2.41×)	31,716	(1.60×)
Vanilla	236	(1.00×)	19,823	(1.00×)
Temperature $T = 1$				
EAGLE-4	563	(2.40×)	29,633	(1.51×)
MTP	500	(2.13×)	29,271	(1.49×)
Vanilla	235	(1.00×)	19,593	(1.00×)

autoregressive decoding and the earlier EAGLE-3 architecture serve as baselines. All measurements are collected on a small AWS c5.xlarge instance with 4 vCPUs and 8 GB of RAM.

We report output tokens per second for a single request, measured by the inference server over complete generations at temperature 0. Across the four benchmarks, EAGLE-4 achieves an average throughput of $2.71\times$ that of vanilla decoding on the same hardware and with the same target-model weights.

Table 5. Decoding throughput in tokens per second. Speedups are reported relative to vanilla decoding. Higher throughput is better.

Dataset	Vanilla	EAGLE-3	EAGLE-4
GSM8K	17.2	27.6 (1.60×)	49.3 (2.87×)
MATH-500	16.7	27.5 (1.65×)	48.8 (2.92×)
HumanEval	16.6	25.3 (1.52×)	45.8 (2.76×)
MT-Bench	17.2	21.5 (1.25×)	39.3 (2.28×)

The evaluation covers four representative workloads spanning common use cases for local LLM assistants. MT-Bench (Zheng et al., 2023) evaluates open-ended conversation and writing, and HumanEval (Chen et al., 2021) focuses on Python code generation. GSM8K (Cobbe et al., 2021) evaluates grade-school mathematical reasoning, while MATH-500, a subset of MATH (Hendrycks et al., 2021), evaluates competition-level mathematical reasoning. EAGLE-

4 consistently outperforms vanilla decoding across all four benchmarks, with speedups ranging from $2.28\times$ to $2.92\times$. For comparison, EAGLE-3 achieves $1.25\times$ – $1.65\times$ speedups over vanilla decoding. These results demonstrate that EAGLE-4 substantially improves upon both vanilla autoregressive decoding and the previous EAGLE-3 architecture on CPU inference.

EAGLE-4 introduces 0.27 GB of additional resident memory during generation. Of this overhead, 134 MB corresponds to the draft model, while the remainder is attributed to its KV cache and computation buffers. Because the draft model remains unchanged across target-model quantization settings, its memory footprint is largely independent of the target-model precision. Moreover, the acceptance behavior of speculative decoding is independent of target-model quantization when the target model’s outputs remain unchanged; therefore, higher target precision can make speculative decoding increasingly attractive as the computational cost of vanilla autoregressive decoding grows.

4.4. Inference on Mobiles

We further evaluate EAGLE-4 on a Motorola Razr 70 Ultra, a commercially available smartphone powered by the Snapdragon 8 Elite with 8 CPU cores and 16 GB of RAM. All experiments are conducted on an unmodified, non-rooted device. The target model and draft model are identical to those used in Section 4.3, and the entire inference stack runs on the CPU through `llama.cpp` using six threads. Vanilla decoding uses an upstream `llama.cpp` build without EAGLE.

Table 6. Decoding throughput on the Motorola Razr 70 Ultra in tokens per second. Results are measured for a single request at temperature 0 on a cool device. Higher throughput is better.

Workload	Vanilla	EAGLE-4	Speedup
HumanEval	22.8	82.6	3.62 $\times$
Long-form generation	23.3	74.7	3.21 $\times$

EAGLE-4 also reduces time to first token, from 1.2 s with vanilla decoding to 0.8 s on HumanEval, while adding only 0.26 GB of resident memory. On long-form generation, EAGLE-4 reduces energy consumption per generated token by more than half, from 356 mJ to 166 mJ, as measured at the USB input. The same generation completes in 14 s with EAGLE-4, compared with 43 s with vanilla decoding.

Deploying the EAGLE-4 stack on the phone requires several optimizations to the ARM CPU path. These include a persistent server-side thread pool, FP16 vector arithmetic, a small-batch attention kernel for verification, a NEON kernel for the model’s BF16 projection, and low-rank output heads

for both the draft and target models. The latter replaces the dense 262,144-way output projection with a two-stage shortlist-and-rescore procedure. Together, these optimizations reduce the draft-step latency from 8.7 ms to 3.9 ms. Because every drafted token is subsequently verified by the target model, EAGLE-4 produces the same final outputs as vanilla decoding.

A practical consideration for sustained on-device inference is thermal throttling. On this device, the vendor’s thermal policy reduces CPU frequency once the device shell reaches approximately 42°C, which occurs after roughly 30–45 s of continuous generation under our workload. A single long-form generation completes before thermal throttling is triggered.

4.5. EAGLE-4 on TokenSpeed

The TokenSpeed team (TokenSpeed Team, 2026) completed the integration of EAGLE-4 at day-0 and optimized its inference performance. In addition to evaluating EAGLE-4 on SPEED-Bench, the team measured its performance on long-context workloads.

Long context, single request. Table 7 reports single-request results using Qwen3.8-27B in BF16 with an EAGLE-4 drafter on $4\times$ NVIDIA B300 GPUs with tensor parallelism 4. Across context lengths of 32K, 128K, and 250K tokens, EAGLE-4 sustains decode throughput between 759 and 822 tokens/s. Time to first token (TTFT) increases from 0.55 to 5.7 seconds as the context grows from 32K to 250K tokens.

Table 7. Long-context, single-request performance of EAGLE-4, measured by the TokenSpeed team. Round latency is reported in milliseconds and TTFT in seconds.

Context	Decode (tokens/s)	Round (ms)	TTFT (s)
32K	759	5.57	0.55
128K	822	5.74	2.1
250K	761	6.04	5.7

5. Conclusion

EAGLE-4 combines lightweight autoregressive drafting with hardware-aware control of speculative computation. EAGLE-HarnessTM combines profiled execution costs with draft confidence to adjust tree width, depth, and verification budget, allowing a single drafter for each target model to adapt its speculation at inference time. Complementing this flexibility, the EAGLE Efficient LM HeadTM reduces drafting overhead through low-rank candidate retrieval over the full vocabulary followed by exact scoring of the selected candidates. Together, these components align the amount and cost of speculation with its expected benefit.

EAGLE-4 improves throughput in GPU serving, CPU in-

ference and on-device generation. On the SPEED-Bench qualitative split with Qwen 3.8 27B at $T = 0$, it delivers $1.28\times$ and $1.40\times$ the throughput of DFlash 2 and DSpark at concurrency 1, and $1.70\times$ and $1.99\times$ at concurrency 512, where both parallel drafters fall below vanilla decoding. On Nemotron-3.5 it reaches $2.0\times$ the throughput of vanilla decoding at 512 concurrent requests on HumanEval. With 4-bit Gemma-4-E2B-it, it achieves an average $2.71\times$ speedup across four CPU benchmarks and $3.62\times$ on a commercial smartphone. These results support autoregressive drafting, with speculation sized to the hardware and the load, as a flexible foundation for speculative decoding from phones to data-center GPUs.

References

- Abramovich, T., Ashkenazi, M., Putterman, I., Chislett, B., Mitra, T., Rouhani, B. D., Zilberstein, R., and Geifman, Y. SPEED-Bench: A unified and diverse benchmark for speculative decoding, 2026. URL <https://arxiv.org/abs/2604.09557>.
- Cai, T., Li, Y., Geng, Z., Peng, H., Lee, J. D., Chen, D., and Dao, T. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.
- Chen, J., Liang, Y., and Liu, Z. DFlash: Block diffusion for flash speculative decoding. *arXiv preprint arXiv:2602.06036*, 2026.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021. URL <https://arxiv.org/abs/2107.03374>.
- Cheng, X., Yu, X., Shao, C., Li, J., Xiong, Y., Qian, Y., Zhu, J., Ma, S., Zhang, X., Ye, J., et al. DSpark: Confidence-scheduled speculative decoding with semi-autoregressive generation. *arXiv preprint arXiv:2607.05147*, 2026.
- Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems, 2021. URL <https://arxiv.org/abs/2110.14168>.
- DeepSeek-AI. DeepSeek-V3.2-Exp: Boosting long-context efficiency with DeepSeek sparse attention, 2025.
- Eldenck, D., Mohapatra, P., Comlek, Y., Oktay, K., Zhang, H., and Xia, S. Attention drift: What autoregressive speculative decoding models learn. *arXiv preprint arXiv:2605.09992*, 2026.
- Gemma Team, Abd, S. E., Aggarwal, V., Algayres, R., Andreev, A., Bachem, O., Ballantyne, I., Brick, C., Cărbune, V., Casbon, M., et al. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026.
- Goel, R., Agrawal, S., Gagrani, M., Park, J., Zao, Y., Zhang, H., Liu, T., Yang, Y., Yuan, X., Lu, J., et al. VocabTrim: Vocabulary pruning for efficient speculative decoding in LLMs. *arXiv preprint arXiv:2506.22694*, 2025.
- Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the MATH dataset, 2021. URL <https://arxiv.org/abs/2103.03874>.
- Inco AI. DFlash 2: Keep Drafting Parallel, August 2026. URL <https://inco.ai/blog/dflash2/>.
- Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mistral 7B, 2023. URL <https://arxiv.org/abs/2310.06825>.
- Kimi Team, Bai, T., Bai, Y., Bao, Y., Cai, J., Cai, X., Cao, P., Cao, Y., Chai, Z., Charles, Y., et al. Kimi K3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026.
- Li, Y., Wei, F., Zhang, C., and Zhang, H. EAGLE: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077*, 2024a.
- Li, Y., Wei, F., Zhang, C., and Zhang, H. EAGLE-2: Faster inference of language models with dynamic draft trees. In *Proceedings of the 2024 conference on empirical methods in natural language processing*, pp. 7421–7432, 2024b.
- Li, Y., Wei, F., Zhang, C., and Zhang, H. EAGLE-3: Scaling up inference acceleration of large language models via training-time test. In *Advances in Neural Information Processing Systems*, volume 38, pp. 136737–136756, 2025.
- Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., et al. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

- Plaksin, A., Krutikov, S., Skvortsov, S., and Samarin, A. SlimSpec: Low-rank draft LM-head for accelerated speculative decoding. *arXiv preprint arXiv:2605.10453*, 2026.
- Ringel, L. and Romano, Y. Accelerating speculative decoding with block diffusion draft trees. *arXiv preprint arXiv:2604.12989*, 2026.
- Shoham, O. B. Balancing coverage and draft latency in vocabulary trimming for faster speculative decoding. *arXiv preprint arXiv:2603.05210*, 2026.
- TokenSpeed Team. TokenSpeed: A speed-of-light LLM inference engine. <https://github.com/lightseekorg/tokenspeed>, 2026.
- Weng, Y., Mei, D., Qiu, H., Chen, X., Liu, L., Tian, J., and Shi, Z. CORAL: Learning consistent representations across multi-step training with lighter speculative drafter. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5580–5593, 2025.
- Xu, A., Lin, B., Xue, B., Wang, B., Xu, B., Wu, B., Zhang, B., Lin, C., Dong, C., Ling, C., et al. DeepSeek-V4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.
- Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.
- Zeng, A., Lv, X., Hou, Z., Du, Z., Zheng, Q., Chen, B., Yin, D., Ge, C., Huang, C., Xie, C., et al. GLM-5: From vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.
- Zhang, J., Ullah, N., Schultheis, E., and Babbar, R. DynaSpec: Context-aware dynamic speculative sampling for large-vocabulary language models. *arXiv preprint arXiv:2510.13847*, 2025.
- Zhao, W., Pan, T., Han, X., Zhang, Y., Sun, A., Huang, Y., Zhang, K., Zhao, W., Li, Y., Zhou, J., et al. FR-Spec: Accelerating large-vocabulary language models via frequency-ranked speculative sampling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3909–3921, 2025.
- Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., and Stoica, I. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena, 2023. URL <https://arxiv.org/abs/2306.05685>.