SPECULATIVE DECODING: PERFORMANCE OR ILLUSION?

Xiaoxuan Liu [*1] **Jiaxiang Yu** [*1] **Jongseok Park** [1] **Ion Stoica** [1] **Alvin Cheung** [1]

ABSTRACT

Speculative decoding (SD) has become a popular technique to accelerate Large Language Model (LLM) inference, yet its real-world effectiveness remains unclear as prior evaluations rely on research prototypes and unrealistically small batch sizes. We present, to our knowledge, the first systematic study of SD on a production-grade and widely deployed inference engine (vLLM), covering multiple SD variants (n-gram, EAGLE/EAGLE-3, Draft-Model, Multi-Token Prediction) across diverse workloads, model scales, and batch sizes. We analyze key factors governing SD performance, and quantify a theoretical upper bound on SD speedup. Our results show that verification by the target model dominates the execution, while acceptance length varies markedly across output token positions, requests, and datasets. Comparing measured performance with theoretical bounds reveals substantial gaps between observed and theoretical upper bounds, and we leverage this observation to highlight new research opportunities that our study opens up in improving SD. Our code for profiling and simulator is available at https://github.com/orgs/SpecDecode-Bench/repositories.

1 INTRODUCTION

Speculative decoding (SD) has emerged as a highly effective approach to speed up inference in large language models (LLMs). It has been widely used in real workloads, ranging from chat completion (OpenAI, 2022), question answering (Cobbe et al., 2021; Rein et al., 2023) to coding tasks (Jimenez et al., 2024).

Despite the substantial progress in SD, there has not been a systematic study on its effectiveness. Prior studies only used prototype implementations rather than production-level systems, which greatly undermines the validity of their conclusions. Worse yet, these evaluations are often conducted with a batch size of one (Xia et al., 2024; Cai et al., 2024; Leviathan et al., 2023; Li et al., 2024b;c; 2025)—an unrealistic configuration that fails to reflect real-world deployment. Second, since the introduction of SD, numerous variants have emerged, including draft-model-based (Miao et al., 2023; Liu et al., 2023; Zhou et al., 2023; Ye et al., 2024; Chen et al., 2024), n-gram-based (Saxena, 2023; Somasundaram et al., 2024), and tree-based (Cai et al., 2024; Li et al., 2024b; Lin et al., 2024; Fu et al., 2024; Li et al., 2024c; 2025) approaches. Yet, we are unaware of any systematic analysis comparing these variants and understanding their respective use cases. This gap poses practical challenges: when deploying SD in production, practitioners are often left without clear guidance on which variant to use for different model architectures or workload conditions.

We aim to study SD in actual deployment settings using real-world workloads. Towards that end, we systematically benchmark SD in vLLM (Kwon et al., 2023), a production-ready inference system, and evaluate multiple SD variants across diverse datasets. Although this setup may appear straightforward, several subtle yet important caveats arise in practice. First, while SD is theoretically guaranteed to preserve the same token distribution as standard decoding, we observe that the generated outputs are not always identical. This discrepancy stems from the inherent nondeterminism (He, 2025) in LLM inference—caused by factors such as kernel parallelism and floating-point variation. Furthermore, we study the SD performance on two increasingly prominent setups: reasoning workloads, which involve longer and more structured generations, and multi-token prediction (MTP) (Liu et al., 2025; Zeng et al., 2025), a recently emerging acceleration technique that complements SD. Across all workloads, every SD variant outperforms the no-SD baseline. The speedup decreases as the batch size increases, consistent with the reduced opportunities for SD on computation-bound scenarios (Liu et al., 2024). Notably, reasoning workloads that require long chains of thought—and therefore generate longer outputs—exhibit speedups comparable to those observed on standard language-modeling tasks.

Next, we conduct a detailed analysis to understand the performance of speculative decoding (SD) across different variants. The overall speedup of SD is governed by two key

[*]Equal contribution [1]UC Berkeley. Correspondence to: Xiaoxuan Liu <xiaoxuan_liu@berkeley.edu>, Jiaxiang Yu <jxyu@berkeley.edu>.

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

factors: the execution efficiency of its constituent stages and the token acceptance rate.

We first examine the execution efficiency of the different SD stages. In general, SD consists of three stages—proposing tokens, verifying tokens, and system overhead such as scheduling and rejection sampling. By examining the time breakdown of the three stages, we find that the proposing stage accounts for only a small portion of the total execution time, and the execution of the large model during verification remains the dominant cost. Thus, when the acceptance rate is low, repeatedly running the large model on rejected tokens incurs substantial runtime cost. As shown in prior work (Liu et al., 2024), this extra cost becomes prohibitive under high system load, and can even cause SD to be slower than standard decoding.

We next conduct a detailed analysis of acceptance behavior in SD, examining when proposed tokens are accepted or rejected. We observe that acceptance patterns exhibit three levels of variability: within a request, across different requests, and across different datasets. Furthermore, different SD methods demonstrate entirely different behavior. For example, learned SD methods such as EAGLE maintain stable and consistently high acceptance across diverse workloads, whereas training-free method like n-gram produces more variable but occasionally much longer accepted spans in tasks like code-editing, where frequent local repetitions allow it to reuse previously seen patterns.

Based on the above observations, we ask a fundamental question: *how far are current SD approaches from the optimal speedup, and what is the theoretical upper bound of speculative decoding?*

First, motivated by the observation that a large fraction of proposed tokens are ultimately rejected, we identify a promising direction for improving SD performance: **verifying only those tokens that are likely to be accepted**. In the ideal case, a proposal method that perfectly predicts the large model's outputs would eliminate the need to invoke the large model altogether. To quantify the gap between observed and theoretical speedups, we develop a simulator grounded in real-world benchmarking data. This simulator evaluates performance under an idealized setting in which all proposed tokens are accepted, thereby minimizing verification cost and achieving the optimal speedup.

Second, we observe that different SD methods achieve higher acceptance at different token positions. This complementarity suggests that adaptively combining multiple SD methods can unlock additional speedup beyond what any single method can achieve in isolation. Our results show that such adaptive combinations can improve end-to-end speedup to 4.9× relative to standard (no-SD) decoding.

In summary, the paper makes the following contributions:

- **Production-grade evaluation of speculative decoding.** We present the first systematic study of SD within a widely adopted, highly optimized inference engine (vLLM), thereby bridging the gap between research prototypes and real-world deployments. We evaluate mainstream SD variants across various real-world workloads.
- **Decomposition of speedup and performance bottlenecks.** We analyze key factors that affect SD performance and find that: (1) verification cost remains dominant, and (2) acceptance behavior exhibits variability across positions, requests, and datasets. To the best of our knowledge, this is the first work to thoroughly examine both the time/memory breakdown and acceptance dynamics of SD across diverse workloads.
- **Theoretical upper bound of SD speedup.** Based on these observations, we analyze the gap between measured and maximum achievable speedup. By leveraging position-specific acceptance statistics without modifying the proposing method, we compute the theoretical upper bound of SD performance, revealing an orthogonal and promising direction for future optimization.

2 Background and Motivation

Speculative Decoding. Large language models (LLMs) generate text auto-regressively: given a prompt $(x_1, \ldots, x_n)$, they produce an output sequence $(x_{n+1}, \ldots, x_{n+S})$, one token at a time. At each step, the model computes a probability distribution over the vocabulary and samples the next token. However, while LLMs *generate* tokens sequentially, they can also *evaluate* the probabilities of a *given* token sequence in parallel. That is, for candidate tokens $x_{n+1}, \ldots, x_{n+S}$, an LLM can compute $P(x_{n+1} \mid x_1, \ldots, x_n), \ldots, P(x_{n+S} \mid x_1, \ldots, x_{n+S-1})$ in a single forward pass. Speculative decoding (Leviathan et al., 2023; Chen et al., 2023) utilizes this property of LLMs as an evaluator on a string of candidate tokens in parallel, enabling higher hardware utilization and potentially generating multiple tokens. For example, a smaller model may generate k candidate tokens, and our target LLM model then predicts $k+1$ probability distributions for each token position in parallel. We then accept the first m tokens from the k candidate tokens that satisfy the sampling method.

SD Variations. When SD was first introduced, the standard approach was to use a smaller draft model with the same vocabulary as the target model, since rejection sampling must operate on a probability distribution in the same token embedding space. Subsequent work has proposed various ways to construct the draft model, such as quantization (Lin et al., 2023; Frantar et al., 2023; Kim et al., 2024) or distillation (Zhou et al., 2023) of the target LLM. However, in practice, an off-the-shelf draft model is not always available, requiring quantizing or pruning the original model, or even

pretraining a smaller model from scratch.

To overcome the limitations of requiring a separate draft model, *draft-model-free* SD methods have been proposed. One common approach is to add auxiliary prediction layers on top of the final transformer block (Li et al., 2024b;c; 2025). These additional "head" layers take the hidden states as input and directly propose tokens. However, practitioners still need to fine-tune these drafting heads separately. Recently, a new line of work has removed the need for post-hoc training. Multi-Token Prediction (MTP) (Liu et al., 2025; Zeng et al., 2025) co-trains auxiliary prediction heads jointly with the main model, rather than fine-tuning them afterward. As a result, these auxiliary heads are available out of the box and achieve substantially higher token-acceptance rates.

Another line of draft-model-free SD methods utilize a non-LLM proposer. A representative example is prompt lookup decoding, or n-*gram* (Saxena, 2023). The intuition is that LLM outputs often contain recurring phrases; therefore, we can extract n-gram snippets from previously generated text and reuse them as token proposals. This approach is particularly effective in workloads such as code or passage editing, where the input and output share substantial overlap. Another example is using retrieval-augmented generation (RAG) to retrieve candidate proposals from an external corpus (He et al., 2024). These non-LLM approaches demonstrate the potential for lightweight or task-specific drafting mechanisms.

Problems of Existing Benchmarks. Most prior works evaluate SD by building proof-of-concept research prototypes, as integration into a production-grade inference system is nontrivial and requires modifications across multiple components. However, this makes a fair comparison between SD methods difficult and the benefits in real-world deployment impossible to predict, due to the following limitations. First, many prototype implementations lack key optimizations in production systems, such as the use of CUDA graphs (NVIDIA, 2024a) to reduce system overhead, a common feature in LLM serving systems (Kwon et al., 2023; Zheng et al., 2024). Second, the implementations are typically evaluated with a batch size of one, which is unrepresentative of large-scale batched deployments of the real world. Finally, prior work lacks an in-depth analysis of SD speedup, as they primarily focus on high-level metrics such as average latency or dataset-level acceptance rate. (Li et al., 2024b; 2025; Xia et al., 2024) We believe that a deeper understanding, such as execution time breakdown and position-level acceptance behavior, is necessary to fully explain and optimize the end-to-end performance of SD.

3 End-to-End Performance of Speculative Decoding

In this section, we evaluate the performance of SD variants across various model sizes and datasets. The complete experimental setup is summarized in Tab. 5 of the Appendix.

3.1 Experiment Setup

Inference Engine and Hardware. All experiments are conducted using vLLM v0.10.1.1 unless otherwise stated. To closely approximate real-world serving scenarios, we enable all default vLLM optimizations, including KV cache management, continuous batching, chunked prefill, and CUDA Graphs. We run all experiments on NVIDIA H100 (NVIDIA, 2024b) with 80 GB of memory. For 8B models, we use a single GPU, while for the 70B/106B models, we use four GPUs with a tensor parallel size of four.

Models and SD variants We evaluate two models from the Llama-3 family (Dubey et al., 2024): Llama3.1-8B-Instruct and Llama3-70B-Instruct. We also include Qwen3-8B[1] (Yang et al., 2025) and GLM-4.5-Air-106B (Zeng et al., 2025) for reasoning workloads. We benchmark the following SD variants:

- **Draft-model-based** (Leviathan et al., 2023; Chen et al., 2023): Uses a smaller draft model to propose 3 draft tokens per step, which are later verified by the target model. Details of model pairing are in Appendix Tab. 6.
- **EAGLE** (Li et al., 2024b) and **EAGLE-3** (Li et al., 2025): Draft-model-free methods with fine-tuned auxiliary prediction heads, predicting 3 draft tokens per step for chain-based settings[2].
- **Multi-Token Prediction** (MTP) (Liu et al., 2025): Co-trained draft-model-free approach where auxiliary heads are jointly trained with the target model; we use 3 draft tokens per step.
- **n-gram** (Saxena, 2023; Somasundaram et al., 2024): Training-free method reusing recurring n-gram phrases from the prompt; we use 3 draft tokens per step with `prompt_lookup_max=7` and `prompt_lookup_min=3`. For `InstructCoder` with Llama3.1-8B, Llama3-70B and Qwen3-8B without reasoning, we additionally evaluate n-gram with 5 draft tokens per step to study the effect of a larger proposal length.

Unless otherwise specified, the maximum generation length

[1]Throughout the paper, Qwen3-8B-Thinking denotes the same model with thinking mode enabled, whereas Qwen3-8B refers to its non-thinking mode.

[2]Official EAGLE-3 weights are unavailable for Llama-3-70B, EAGLE weights are likewise unavailable for Qwen3-8B and EAGLE-3/EAGLE weights are also unavailable for GLM-4.5-Air (Li et al., 2024b).

is capped at 8192 tokens. For reasoning workloads AIME22-24 and GPQA-Main, we extend this limit to 32,768 tokens to prevent early truncation. Decoding is performed with temperature set to 0 and no sampling truncation (`top_p=1`, `top_k=-1`).

Workloads. We evaluate six datasets representing diverse real-world applications: *CNN/DailyMail* (Hermann et al., 2015; See et al., 2017) (summarization), *ShareGPT* (sha, 2024) (multi-turn chat)[3], *InstructCoder* (Li et al., 2024a) (code editting), *GSM8K* (Cobbe et al., 2021) (grade-school math), and two complex reasoning datasets: *AIME22-24* (AoPS, 2024; Project Numina, 2024) and *GPQA-Main* (Rein et al., 2023). We detailed the dataset information in Appendix Sec. A.1.

Generation Length and Evaluation Metrics. Although in theory SD should preserve the same token distribution as standard decoding, in practice we observe discrepancies in the generated outputs—-*even under greedy decoding*. Tab. 1 illustrates this by showing variations in generation length when using the same set of prompts while changing the batch size and SD variant. For each configuration, we replicate each request multiple times to match the target batch size. This creates a steady-load setting by keeping request lengths uniform within each batch, so requests finish at similar times, and the effective batch size remains stable. To account for prompt heterogeneity, we repeat the process for 500 requests, average the results, and also record the generation length for each request.

Table 1. Average generation length (mean ± standard deviation) of the baseline (w/o SD) and different SD variants on the ShareGPT dataset using Llama3.1-8B across varying batch sizes.

Batch Size	w/o SD	ngram	Eagle	Eagle3
1	737±1048	750±1100	731±1010	736±1046
16	708±932	675±818	686±853	686±853
32	674±784	697±887	693±857	706±917
64	687±852	696±911	709±924	709±924
128	714±930	739±1050	709±924	709±924

As shown in Tab. 1, even without speculative decoding (the w/o SD column), generation length differs across different batch sizes, and similarly when SD is used. Importantly, these differences are not consistently longer or shorter than the baseline, suggesting that they stem from numerical nondeterminism rather than implementation bugs. Recent work (He, 2025) attributes such variability to kernel-level nondeterminism and floating-point variation.

To account for fluctuations in output length, we use token **throughput**, i.e., the number of generated tokens per second, as the performance metric. This measure removes the confounding effect of varying generation lengths. **Speedup** is then defined as the ratio between the throughput with speculative decoding (SD) and the throughput without it.

[3]ShareGPT is used here as a representative chat workload rather than as a held-out benchmark. Because the LLaMA-based EAGLE/EAGLE-3 weights used in our study are trained on training mixtures that include ShareGPT, results on ShareGPT may be optimistic and should be interpreted primarily as end-to-end performance on a realistic chat workload rather than as a clean measure of generalization.

3.2 Results

End-to-end speedup. We first present the end-to-end throughput comparison in Fig. 1. Due to page limits, we present the results for tree-style verification in Appendix A.2.2. Across most configurations, SD improves throughput over the baseline without SD. The improvement is most pronounced with small/medium batch sizes, when the system is memory-bound, so lots of compute can be used for proposing and verification.

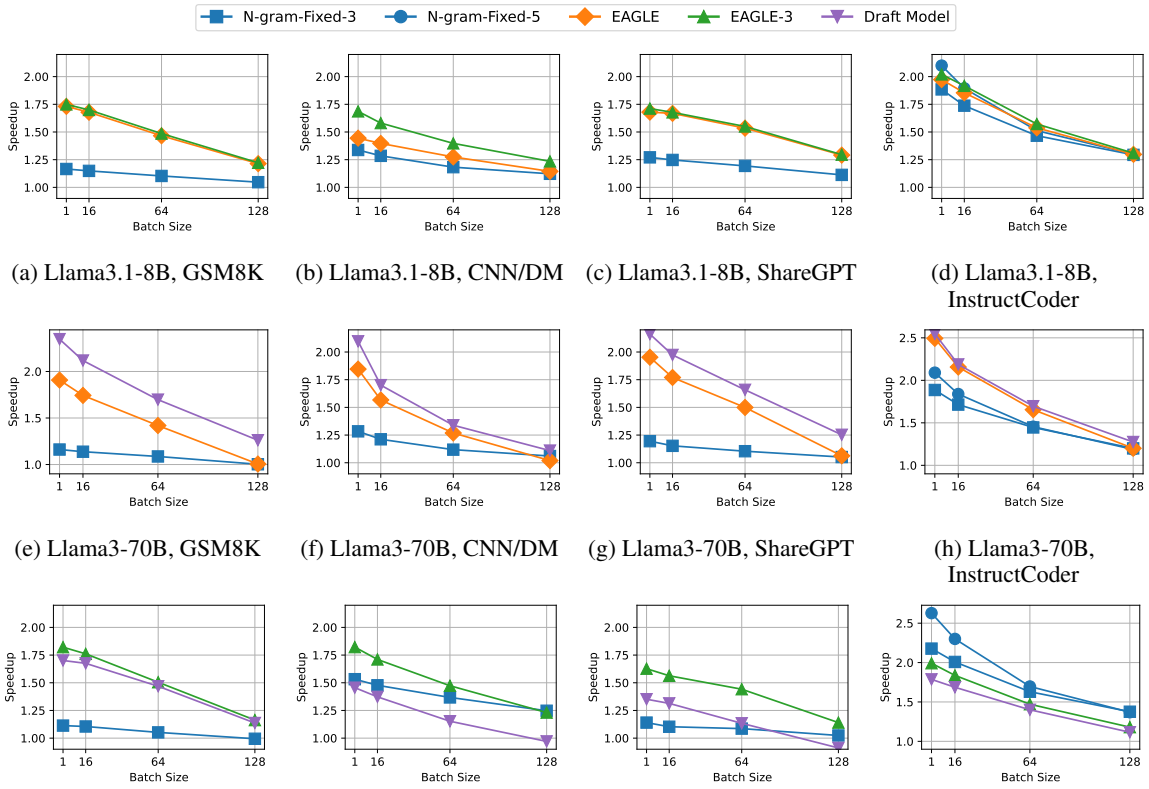
> **Batch Size.** *Increasing batch size improves absolute throughput but systematically reduces the relative speedup of speculative decoding, an effect that is amplified for larger models.*

Consistent with prior work (Liu et al., 2024), larger batch sizes yield higher absolute throughput but lower relative speedup. For example, for the Llama3.1-8B model on the GSM8K dataset, the speedup of EAGLE over the non-SD baseline decreases from 1.73× to 1.21× as the batch size increases from 1 to 128.

At a high level, SD trades additional computation for increased throughput or reduced latency. When the batch size is small, the system has sufficient idle compute resources, so the extra computation-spent on proposing and verifying tokens that are ultimately rejected—has limited impact. However, as the batch size grows, the system becomes increasingly compute-bound, making the overhead spent on proposing and verifying rejected tokens more costly, and thus leading to smaller speedups. As shown in Sec. 5, the verification stage dominates the overall execution time under such conditions.

We further observe that this trade-off between computation and speedup becomes more obvious as the model size increases. For instance, with EAGLE, on the ShareGPT dataset, increasing the batch size from 1 to 32 reduces the speedup by 4.3% (1.68× to 1.61×) for Llama3.1-8B, whereas for Llama3-70B, the speedup drops by 14.0% (1.96× to 1.72×). This trend arises because the 70B model is executed on four GPUs, so even with small or medium batch sizes, the system is already compute-bound, leaving limited spare compute to verify tokens that are ultimately rejected.

> **SD Variants and dataset.** *The n-gram method is generally less effective than other SD approaches across most workloads, with the notable exception of code-editing tasks. Moreover, the draft-model-based method achieves the best performance on the 70B target model; however, the effectiveness diminishes*



(a) Llama3.1-8B, GSM8K (b) Llama3.1-8B, CNN/DM (c) Llama3.1-8B, ShareGPT (d) Llama3.1-8B, InstructCoder

(e) Llama3-70B, GSM8K (f) Llama3-70B, CNN/DM (g) Llama3-70B, ShareGPT (h) Llama3-70B, InstructCoder

(i) Qwen3-8B, GSM8K (j) Qwen3-8B, CNN/DM (k) Qwen3-8B, ShareGPT (l) Qwen3-8B, InstructCoder

Figure 1. End-to-end performance on non-reasoning workloads. For the n-gram method, we report results for both three-token and five-token proposals.

as the target model size decreases (e.g., 8B).

GSM8K, CNN/DailyMail, and ShareGPT all show similar trends, with EAGLE-3 or draft-model-based method achieving the highest speedup across batch sizes. In contrast, on InstructCoder, the performance gap among SD variants narrows. For both Llama3.1-8B and Qwen3-8B, the n-gram method even outperforms EAGLE and EAGLE-3. This behavior arises because code-editing workloads such as InstructCoder exhibit strong token reuse, which can be effectively exploited by simple drafting methods like n-gram. We further analyze the performance of n-gram in Sec. 7.

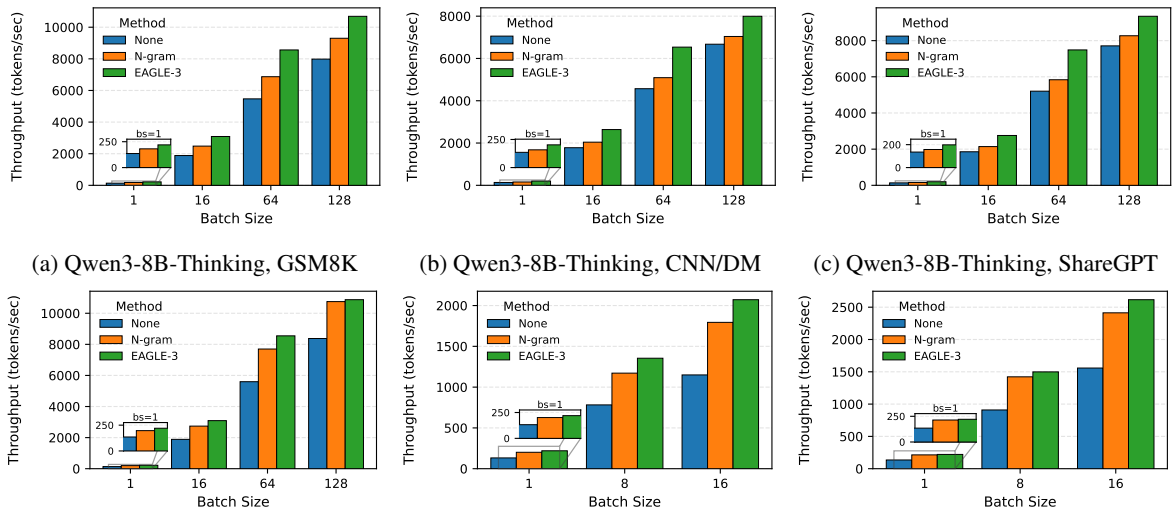
Next, we observe that draft-model–based methods outperform EAGLE on the Llama3-70B model in most settings. As shown in Fig. 6, draft-model–based approaches achieve higher token acceptance rates, while their proposing overhead is comparable to that of EAGLE (see Fig. 3). This behavior is expected: draft models are pretrained on large-scale corpora, whereas the EAGLE head is only fine-tuned on a relatively limited dataset, which naturally constrains its acceptance rate.

In contrast, on Qwen3-8B, draft-model–based speculative decoding consistently underperforms EAGLE-3 and, in some cases, even the training-free n-gram method. Specifically, for Llama3-70B, we use Llama3.2-1B as the draft model, whereas for Qwen3-8B we use Qwen3-0.6B. Although acceptance rates can be similar across the 70B and 8B setups on the same dataset—or even higher for the 8B case sometimes—draft-model–based SD is nonetheless noticeably less effective on the smaller model. For example, on GSM8K, the acceptance rate is around 76% for the 70B setup and around 81% for the 8B setup, yet the overall performance gain is substantially lower for Qwen3-8B.

This discrepancy arises because the proposing overhead is substantially larger for the 8B model. Specifically, the execution-time ratio between the draft and target model's single forward pass is around 12.5% for the 70B setup, compared to an estimated 37.5% for the 8B setup. As a result, drafting becomes significantly more expensive relative to verification on smaller models, which diminishes the overall benefit of draft-model-based SD.

Reasoning Workloads. *Models and drafting methods that sustain high acceptance rates over long contexts—such as EAGLE-3 and, in some cases, n-gram—derive the greatest benefit on reasoning tasks. In contrast, MTP performance is limited by the reuse of a single MTP head across drafted tokens.*

As reasoning workloads become increasingly prevalent, understanding the effectiveness of speculative decoding



(a) Qwen3-8B-Thinking, GSM8K (b) Qwen3-8B-Thinking, CNN/DM (c) Qwen3-8B-Thinking, ShareGPT

(d) Qwen3-8B-Thinking, InstructCoder (e) Qwen3-8B-Thinking, AIME22-24 (f) Qwen3-8B-Thinking, GPQA-Main

Figure 2. End-to-end performance on reasoning workloads.

(SD) in this regime becomes important. These workloads feature short prompts but long generation sequences. We evaluate SD on two representative reasoning benchmarks, AIME22–24 (Project Numina, 2024) and GPQA-Main (Rein et al., 2023).

As shown in Fig. 2, we report the absolute generation throughput across different batch sizes. To ensure fair and consistent comparisons, we restrict our evaluation to medium batch sizes. Reasoning tasks typically involve long generation sequences, which can trigger request preemption once the KV cache becomes full. By avoiding such preemption, the reported results more accurately reflect the inherent efficiency of speculative decoding rather than performance artifacts caused by memory pressure.

For Qwen3-8B-Thinking, EAGLE-3 consistently leads across datasets, achieving 1.64×–1.80× speedup over on GPQA-Main and AIME22–24, whereas n-gram performs comparably at 1.50×–1.58×. This trend likely arises because the model produces longer contexts and repetitive symbolic patterns, increasing opportunities for reuse. As shown later in Sec. 6, the average acceptance length in GPQA-Main generally grows with sequence length, with n-gram's acceptance rising faster than EAGLE-3's.

We defer the detailed MTP results to Appendix A.2.1, since the currently measured performance is less representative of MTP's full potential.

4 Understanding the Performance

In this section, we aim to have a better understanding of the end-to-end speedups reported in Sec. 3. We start by breaking down where time and memory are spent on different SD stages – drafting, verification, rejection sampling, and others, and how these keep us from reaching the theoretical upper bound. We then examine how acceptance patterns within a request, across requests, and across datasets.

4.1 End to End Speedup Formula

As pointed in the original SD paper (Leviathan et al., 2023), Let c denote the execution-time ratio of a single forward pass between the drafting method and the target model, α is the token acceptance rate and k is the proposed length. The expected overall wall-time speedup achieved by speculative decoding, can be expressed as follows:[4]

$$E(speedup) = \frac{1-\alpha^{k+1}}{(1-\alpha)(kc+1)}. \tag{1}$$

As shown, the speedup depends on two factors: the proposed length (k), the execution-time ratio between the drafting method and the target model (c), and the token acceptance rate (α). Importantly, the absolute execution time of the drafting method is irrelevant; only its runtime relative to the target model affects the achievable speedup.

Employing a more sophisticated drafting method can improve the token acceptance rate α, but often at the cost of increased execution time, leading to a larger c. Consequently, optimal speedup is achieved by carefully choosing a drafting method that balances the trade-off between c and α. In the following sections, we evaluate these factors and additionally analyze the memory overhead of SD.

[4]This formulation assumes that the $k+1$ simultaneous evaluations of the target model take approximately the same amount of time as generating a single token in parallel. This assumption typically holds when the system is memory-bound, which is common at small batch sizes, but may break down once the system becomes compute-bound.

5 Execution Time and Memory Breakdown

5.1 Execution Time

In Fig. 3, we present the execution time breakdown of different stages in the baseline LLM execution and SD settings. For all experiments, we sample 500 requests from CNN/DailyMail. All other runtime settings are the same as Tab. 5. For each configuration, we report the fraction of total execution time attributable to four components: *(i)* **Drafting**, the time to generate speculated tokens, which corresponds to the lookup operation for n-gram and the autoregressive generation of the EAGLE heads for EAGLE and EAGLE-3; *(ii)* **Verification**, the time spent by the target model to validate proposed tokens; *(iii)* **Rejection Sampling**, time spent on generating final tokens based on verification logits; and *(iv)* **Other Overheads**, all other overheads in the vLLM execution system.

> **Execution Time.** *Verification time generally accounts for the largest share of runtime across speculative decoding variants, while sampling time is negligible. Drafting is negligible for n-gram, stays below 20% for EAGLE/EAGLE3, and stays under 40% for draft-model methods acorss all batch sizes.*

We observe that the verification stage generally takes the largest execution time, ranging from 42% to 95% in all execution methods, which increases with the model size and batch size.

Drafting costs, however, vary substantially by proposal mechanism. For n-gram, drafting time accounts for less than 2% across all executions, adding negligible overhead to the baseline execution. Both EAGLE and EAGLE3 show similar drafting overhead characteristics, accounting for a 12% to 20% of the execution time at batch size 1, and 3% to 7% at batch size of 512. Draft-model-based methods differ in that they must run a separate smaller model autoregressively to propose tokens, leading to a larger drafting overhead when the batch size is small. For Llama3-70B, drafting takes from 21% at batch size of 1 to 3% at batch size of 512. For Qwen3-8B, drafting is about 47% at batch size of 1 and decreases to 16% at batch size of 512. As a result, the verification fraction is correspondingly lower (roughly 72% for Llama3-70B and 42% for Qwen3-8B at batch size of 1). We leave a finer-grained decomposition of verification time to future work.

Furthermore, the sampling stage takes a negligible amount of time, accounting for less than 1.7% of total time across all executions. Lastly, vLLM overhead accounts for 3–12% of total execution time. This fraction decreases as model and batch size grow, since fixed overheads—such as scheduling—are independent of model scale and therefore become amortized at larger batch size or model size.

	No SD/n-gram	EAGLE	EAGLE 3
Static (GiB)	14.96	15.43	15.75
Per-token (KiB)	128	132	132

(a) Memory breakdown on Llama3.1-8B

	No SD/n-gram	EAGLE	D-M (1B)
Static (GiB)	131.4	133.3	133.7
Per-token (KiB)	320	324	352

(b) Memory breakdown on Llama3-70B

	No SD/n-gram	EAGLE3	D-M (0.6B)
Static (GiB)	15.25	16.00	16.37
Per-token (KiB)	144	148	256

(c) Memory breakdown on Qwen3-8B

Table 2. Static and per-token memory usage of the baseline model execution and different speculative decoding methods. DM stands for Draft-Model-based method.

Implication. Verification cost dominates the end-to-end execution of speculative decoding, indicating that running the large target model remains the primary computational bottleneck. When the proposed tokens are ultimately accepted, this cost is well justified. However, verifying tokens that are later rejected can incur substantial wasted computation. This observation motivates a natural question: how much of the verification compute is truly useful? Conversely, what speedup could be achieved if verification were performed only on "correct" tokens, eliminating wasted work? These questions form the basis of Sec. 8, where we estimate the theoretical upper bound for SD.

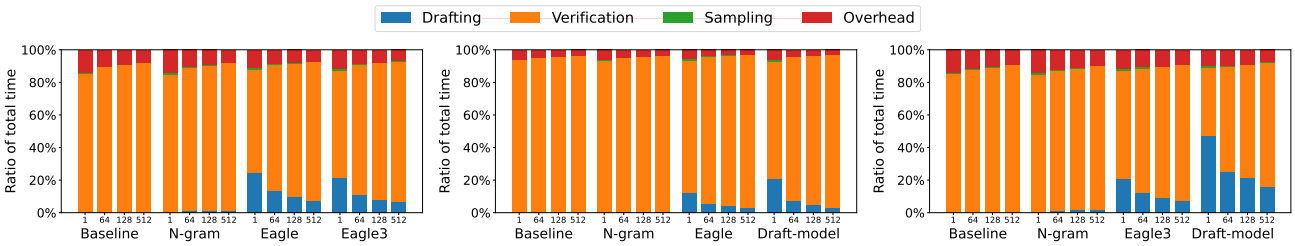
5.2 Memory

In Tab. 2, we present the GPU memory usage of each SD variant with Llama3.1-8B, Llama3-70B and Qwen3-8B. We use FP-16 precision and report the static memory overhead from the model weights and per-token KV cache calculated using model specs. We do not report intermediate memory overheads, such as the activation tensors during execution. The detailed calculations can be found in Appendix A.3.

> **Memory.** *Speculative decoding generally incurs minimal memory overhead, including both static parameters and KV cache: it is zero for n-gram, below 10% for EAGLE/EAGLE-3, while draft-model-based methods incur overhead that depends on the chosen draft model.*

We find that the memory overhead of speculative decoding (SD) is minimal. For n-gram SD, draft tokens are sampled from CPU-resident generation history, incurring no GPU memory overhead.

Static Memory. Static memory corresponds to the memory required to store model weights. Certain SD variants add extra layers or even an independent model, thereby incur-



(a) Llama3.1-8B (b) Llama3-70B (c) Qwen3-8B

Figure 3. Execution time breakdown across different models.

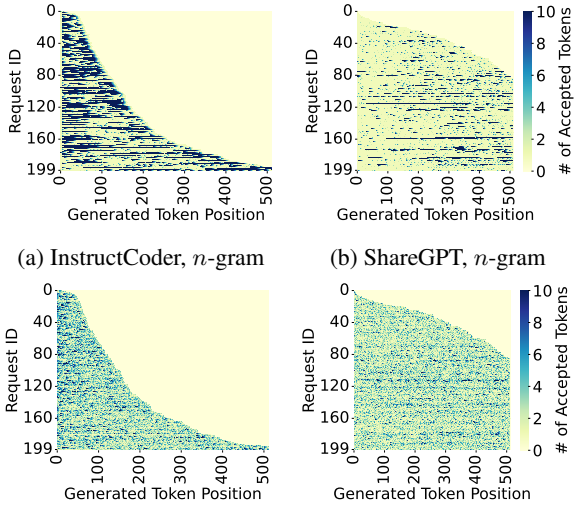
ring additional static memory overhead. EAGLE-based SD introduces an additional Transformer layer and thus requires extra static weights, but the overhead is small: 3.1% (EAGLE)/ 5.3% (EAGLE3) for Llama-3.1-8B, 1.4% (EAGLE) for Llama-3-70B, and 4.9% (EAGLE3) for Qwen3-8B. For draft-model-based SD, static memory overhead depends on the size of the draft model; in our setup, pairing Llama-3-70B with Llama-3.2-1B incurs a 1.8% increase while pairing Qwen3-8B with Qwen3-0.6B incurs a more significant 7.3% increase.

Per-token Memory. Per-token memory refers to the size of the KV cache allocated for each generated token. The total KV cache footprint grows with the number of generated but unfinished requests. Consequently, for workloads with long generation lengths such as reasoning workloads, memory cost introduced by KV cache becomes a dominant factor in overall memory consumption. Per-token overhead is modest for EAGLE-based methods, as each EAGLE layer requires the same KV cache as a single attention layer (4 KiB per token), resulting in a 3.1% overhead for Llama-3.1-8B and 1.3% for Llama-3-70B. In contrast, draft-model–based SD incurs substantially higher per-token memory overhead, as it relies on an additional multi-layer model for token proposal. In our configuration, pairing a 0.6B draft model with an 8B target model increases the per-token memory footprint from 144 KiB to 256 KiB, corresponding to a 1.77× increase.

6 Acceptance Behavior

To understand the acceptance rate, we measure the number of draft tokens generated[5] by the target model at each decoding step. For each dataset, we sample up to 200 requests. Each SD method proposes up to 20 tokens per step, and we record how many of these are generated at that step. After each generation step, even if multiple tokens were generated, we discard all but one and advance to the next position. Essentially, we approximate the maximum number of tokens that could be accepted at each generation step. We analyze results for two non-reasoning models (Llama-3-70B and Llama-3.1-8B; max 512 output tokens) and one reasoning model (Qwen-3-8B-Thinking; max 32K output tokens).

[5]Number of generated tokens = accepted tokens proposed by the drafting method + 1 bonus token. Similarly, for generation length, it is computed by adding the acceptance length by one.



(a) InstructCoder, n-gram (b) ShareGPT, n-gram

(c) InstructCoder, EAGLE (d) ShareGPT, EAGLE

Figure 4. Generation length per token position for Llama3.1-8B. Requests are sorted based on generation length. Darker colors indicate that more tokens are generated at the corresponding position.

Fig. 4 visualizes the number of tokens generated at each output position for InstructCoder and ShareGPT (notice SD generates at least one token at each position, even if all proposed tokens are rejected). Each row represents a sampled request and each column a generation position, with color indicating the number of tokens produced by SD.

> **Summary.** *We observe the variance of generated token lengths along three dimensions: within a single request, across requests, and across datasets.*

Fig. 4 gives a high level feeling of the generation length at each position across requests. n-grams and EAGLE show different patterns. n-grams show high variance, lots of very dark and very light positions, some positions generate a lot, while others only generate one. EAGLE show good average generation length (averge color is darker) and also less variance. But there are variance across all dimentions, which we will describe in more detail below.

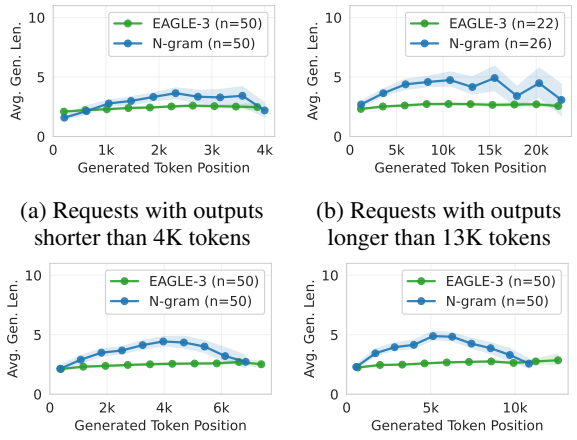
> **Within A Request.** *Longer generation workloads yield more accepted tokens per position, with n-gram benefiting from repetitive reasoning patterns*

but losing effectiveness near the end as the generation shifts toward conclusions.

Since different requests can have varying generation lengths, we sample 50 requests from each generation-length category: requests with generation length <4K, between 4–8K, between 8–13K, and over 13K. In Fig. 5, token positions are partitioned into ten intervals, and the mean generation length is computed and plotted for each interval.

For reasoning workloads, generation length generally increases as generation progresses for both methods, but the growth is substantially steeper for n-gram. As shown in Fig. 5, on GPQA-Main with Qwen-3-8B-Thinking, n-gram rises from roughly 1.6–3.7 generated tokens across the sequence in short outputs (<4K tokens) to 2.7–5 tokens in the longest responses (>13K tokens). EAGLE-3 grows much more gradually from around 2.1–2.5 among the shortest requests and to roughly 2.5–5.7 tokens among the longest requests. These results indicate that long scientific and technical generations accumulate locally repetitive structures, such as recurring variables, equations, and phrases, that n-gram can increasingly exploit. On the other hand, *EAGLE*-based methods remain less sensitive to such surface-level recurrence.

For n-gram, we additionally observe a decrease towards the last few token position intervals. This is likely because the model shifts from step-by-step reasoning, where phrases and patterns tend to repeat a lot, to writing the final answer and stopping, which is less repetitive and therefore harder for n-gram to reuse previous context. Thus, in the last few token intervals, more requests are in the final-answer stage, so the average generation length drops.



(a) Requests with outputs shorter than 4K tokens

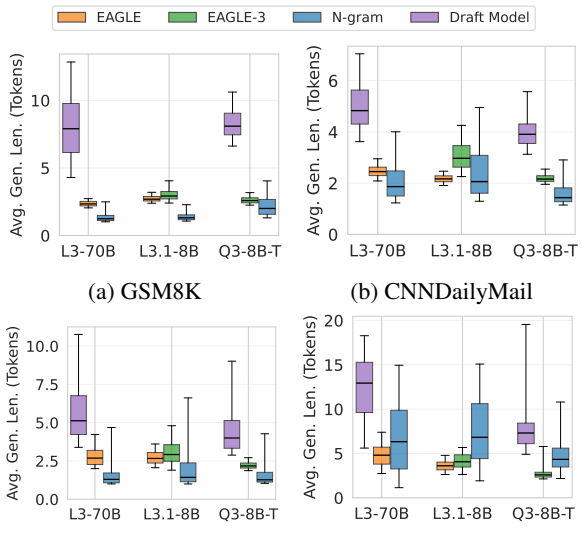
(b) Requests with outputs longer than 13K tokens

(c) Requests with outputs of 4K-8K tokens

(d) Requests with outputs of 8K-13K tokens

Figure 5. Average generation length (tokens) per output token position for Qwen3-8B-Thinking on GPQA-Main. Shaded bands show the 95% confidence interval.

Across Requests and Datasets. *Draft-model-based*



(a) GSM8K

(b) CNNDailyMail

(c) ShareGPT

(d) InstructCoder

Figure 6. Request-level generation length across datasets and models. Each box shows the distribution of per-request mean generation length, with the box spanning the 25th–75th percentiles and whiskers covering the 5th–95th percentiles. Models from left to right are Llama3.1-8B, Llama3-70B and Qwen3-8B-Thinking. Results for reasoning datasets (AIME22–24 and GPQA-Main) are shown in Fig. 14.

method typically achieves the longest median generation lengths across all workloads, EAGLE attains more stable per-request generation lengths, whereas n-gram shows higher variance and a heavy-tailed generation-length distribution.

We observe substantial variance in per-request generation length in Fig. 6. Even within the same workload, the efficiency of speculative decoding can differ drastically. For example, on InstructCoder with Llama3-70B, EAGLE produces relatively short request-level generation lengths (2.7-7.4 tokens), whereas n-gram and draft-model-based SD achieve longer lengths with wider spreads (1.1–15.0 tokens and 5.6-18.3 tokens, respectively). In other workloads (e.g. ShareGPT and CNN/DailyMail) and target models (e.g. Qwen3-8B), draft-model-based methods similarly shows higher generation lengths than n-gram and EAGLE.

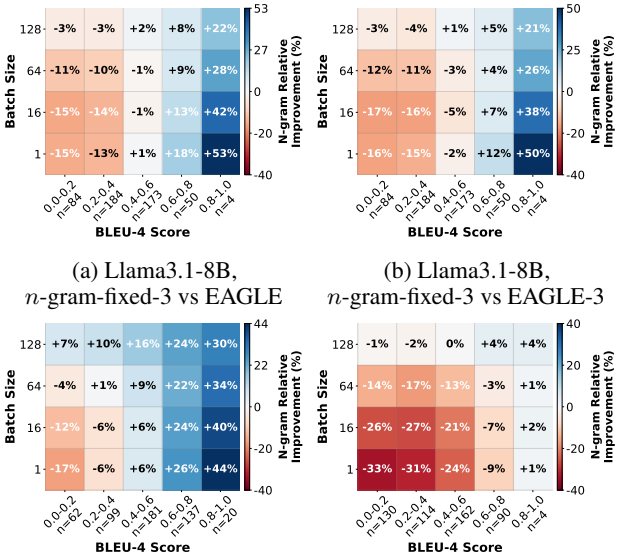
These distributions reveal distinct proposal behaviors for each SD variant. Draft-model-based methods often achieve substantially longest generation lengths, but their whiskers (5th-95th percentiles) are also broadly spread. The reason behind this may be request-dependent alignment: for some requests, the draft model can track the target model closely so that many proposed tokens are accepted, while for others, it diverges early and only a few draft tokens are accepted after verification.

For n-gram, it exhibits heavy tails and large variance across requests, with standard deviations roughly 2x-5x higher than those of EAGLE-based methods. While most n-gram

requests produce short accepted spans, a small subset yield exceptionally long bursts (often exceeding 15 tokens), as seen as long blue strips in Fig. 4. In code-editing tasks, these typically correspond to locally repetitive content (e.g., reused identifiers, function templates, or class definitions). We select and show an example in Fig. 15. Such bursty acceptance behavior reveals that n-gram speculation relies on discrete pattern matches that may occur infrequently but can produce large payoffs when present. Consequently, n-gram's performance is high when the local contexts are repetitive, but it remains unstable across requests and workloads.

In contrast, EAGLE and EAGLE-3 show compact, symmetric distributions centered near their medians (typically 2–4 tokens), indicating steadier acceptance driven by learned contextual representations rather than surface-level overlap.

These distributional behaviors are consistent with the end-to-end outcomes in Fig. 1. Draft-model-based methods deliver the largest speedups when the cost of running a separate draft model is relatively low. EAGLE/EAGLE3 follows and provides modest gains on most workloads. n-gram achieves its best speedups on highly repetitive code-editing tasks, where rare but very long bursts occur. Outside such settings, its heavy-tailed and unstable acceptance makes performance less predictable.



(a) Llama3.1-8B, n-gram-fixed-3 vs EAGLE

(b) Llama3.1-8B, n-gram-fixed-3 vs EAGLE-3

(c) Qwen3-8B, n-gram-fixed-3 vs EAGLE-3

(d) Llama3-70B, n-gram-fixed-3 vs EAGLE

Figure 7. Correlation between BLEU-4 and n-gram speedup on InstructCoder. Each heatmap shows n-gram's relative speedup (%) over EAGLE or EAGLE-3, grouped by BLEU-4 scores (x-axis) and batch sizes (y-axis). Blue means that n-gram outperforms the draft-model method, while red means the opposite. n=XYZ in the x-axis labels denotes the number of requests in each BLEU bucket.

7 Case Study: n-gram Performance on InstructCoder

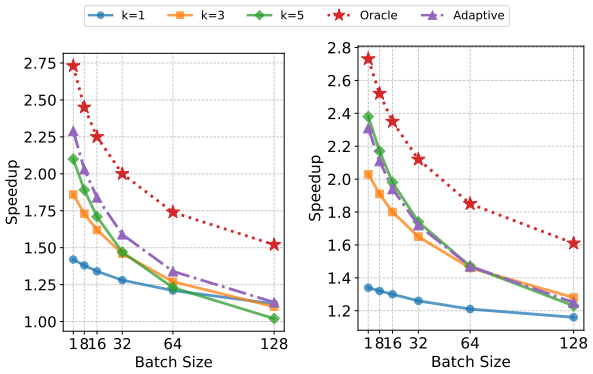
The n-gram SD is especially attractive for speculative decoding because it is training-free. We observe that n-gram achieves particularly strong speedups on the code-editing workload, such as InstructCoder, and we therefore conduct a focused case study to understand the underlying reasons.

We hypothesize that its advantage stems from the *local repetition* inherent in code-editing tasks: when upcoming tokens have already appeared in the prompt, n-gram lookup is more likely to propose correct continuations that are subsequently accepted by the target model during verification. To test this hypothesis, we quantify prompt–output overlap using BLEU-n (Papineni et al., 2002). BLEU-n measures the fraction of overlapping n-grams, with higher values indicating stronger local reuse or copying. For each request, we compute the BLEU score between the prompt and the output generated without speculative decoding, and then group them into 5 intervals: requests with BLEU score of 0-0.2, 0.2-0.4, 0.4-0.6, 0.6-0.8 and 0.8-1.0. Within each interval, we compare the speedups achieved by n-gram and by EAGLE/EAGLE-3. We report the results using BLEU-4 score by default (Papineni et al., 2002). In our experiments, BLEU-4 gives the clearest speedup heatmap and shows a clear cutoff where, above a certain overlap range, n-gram beats EAGLE/EAGLE-3 for every batch size. We have also repeated the analysis with BLEU-1 through BLEU-10 scores, and observe the same overall patterns.

> **Overlap Drives n-gram Speedups.** *n-gram works best on code editing because the output often reuses short spans that already appear in the prompt. As overlap increases, n-gram speedup rises and eventually surpasses EAGLE/EAGLE-3.*

We observe that higher BLEU-n scores strongly correlate with greater n-gram speedup (Fig. 7). For requests with lower BLEU-n scores (little overlap), n-gram underperforms due to inaccurate proposals. Once the BLEU-n score exceeds a certain threshold, (e.g. larger than 0.6 for Llama3.1-8B), n-gram consistently outperforms EAGLE and EAGLE-3 across all batch sizes. With a proposal length of 3, it achieves up to 53% higher speedup than EAGLE and EAGLE-3. In Fig. 16, when we further increase the proposal length of n-gram to 5, the boundary where n-gram performs better is clearer, and this brings an even higher performance of up to 100% higher speedup than EAGLE/EAGLE-3. This is likely because a larger proposal length allows n-gram to reuse longer repeated spans in the prompt or recent context when the overlap is high. Overall, these results confirm that n-gram speculation benefits directly from prompt-level repetition in code-editing workloads.

Finally, our BLEU analysis measures overlap only between



(a) n-gram, InstructCoder. (b) EAGLE3, InstructCoder.

Figure 8. Oracle, fixed-length, and adaptive proposed-length speedup on Llama3.1-8B.

the complete prompt and the complete generated output. In practice, n-gram can match and reuse tokens from the full in-context history, which includes both the prompt and the tokens generated so far. We leave an analysis that measures overlap with respect to this evolving context for future work.

8 Theoretical Speedup Upper Bound of Speculative Decoding

In this section, we aim to derive an upper bound on the speedup achievable by SD. Our goal is to understand how far current SD methods are from the theoretical optimum, assess their current performance limits, and highlight potential directions for future research.

As discussed in Sec. 5, verification time dominates the overall execution cost. This naturally raises the question: *if we could eliminate wasted verification by avoiding the verification of tokens that would ultimately be rejected, what is the maximum speedup that speculative decoding could achieve? In other words, assuming an oracle—or a sufficiently accurate mechanism—that proposes only tokens likely to be accepted, what is the upper bound on the achievable speedup?* In the following section, we try to derive this number and understand the gap between the current speedup and the theoretical upper bound.

Moreover, as observed in Sec. 6, different SD variants exhibit distinct acceptance behaviors. Specifically, they differ in the number of tokens accepted at each generation step. To better understand the true performance upper bound, we investigate whether combining different SD strategies could yield an even more optimal acceptance pattern and, consequently, higher overall efficiency.

8.1 Minimal Verification Cost

We begin by addressing the following question: if we could minimize the verification cost, what is the maximum speedup that speculative decoding could achieve? To explore this, we compare three proposal-length policies: a fixed proposed length, a simple adaptive heuristic (Joao Gante, 2023), and an oracle-based proposed length. The adaptive heuristic starts the proposal length from 5, increases by 2 when all drafted tokens are accepted, and otherwise decreases by 1, with a minimum of 1. It aims to use longer drafts when drafting is easy and back off when draft quality drops. The oracle setup assumes that the number of tokens accepted at each generation step is known in advance. At each step, we set the proposed length equal to the actual accepted length, ensuring that all proposed tokens are accepted. This setup effectively simulates the ideal case where verification incurs no rejection overhead.

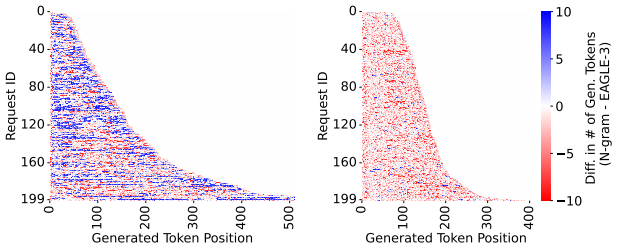
One might argue that if all proposed tokens can be perfectly accepted, the large model would no longer be necessary. While that is theoretically correct, it is practically unattainable: no proposal mechanism can perfectly predict future target tokens. Hence, the oracle configuration serves purely as an upper bound on the achievable speedup of speculative decoding, highlighting the performance ceiling that real methods can approximate.

We conduct this analysis using Llama 3.1-8B on the InstructCoder, ShareGPT, GSM8K, and CNN/DailyMail datasets. Fig. 8 shows the results on InstructCoder; results on the other datasets are provided in Appendix Fig. 17.

We observe a substantial gap between the oracle speedup and the speedup achieved with a fixed proposal length or the adaptive policy. As shown in Fig. 8, on the InstructCoder dataset with a batch size of one and n-gram as the SD method, the oracle setup achieves a speedup of approximately 2.75×, whereas the fixed proposed length configuration attains about 2.1× for the best proposed length of 5, and the adaptive heuristic improves this to about 2.3×. Similar patterns are observed for EAGLE3. Moreover, the gap generally widens as batch size increases: both the fixed-k curves and the adaptive curves drop much faster, while the oracle speedup degrades more gently. This trend is consistent with our observation in Fig. 3. As batch size grows, verification takes up a larger share of the decoding time, so fixed-k methods would incur larger overhead for verifying draft tokens that are ultimately rejected.

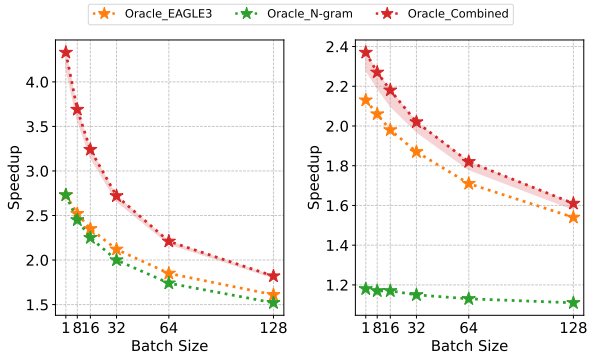
8.2 Potential of Combining Complementary SD Methods

Although EAGLE/EAGLE3 generally outperforms the n-gram method, a closer inspection of acceptance behavior reveals complementary strengths between the two. As illustrated in Fig. 9, the per-position differences in accepted tokens exhibit alternating regions of red (favoring EAGLE3) and blue (favoring n-gram), indicating that the two methods excel under different conditions. This behavior is intuitive: for example, in code snippets with strong local regularities or recurring grammatical patterns, n-gram can often find



(a) InstructCoder (b) GSM8K

Figure 9. Per-position accepted-length difference between n-gram and EAGLE3 on Llama 3.1-8B. Red indicates positions where EAGLE3 accepts longer spans, while blue indicates positions favoring n-gram.



(a) InstructCoder (b) GSM8K

Figure 10. Oracle speedup of EAGLE3, n-gram, and their oracle combination for optimal speedup on Llama3.1-8B. For Oracle_Combined, the shaded region shows the speedup after accounting for an estimated method-switching overhead.

exact matches, which are likely to be correct. In such cases, because n-gram incurs substantially lower proposing overhead, it can be the more efficient choice. At other positions where local repetition is weaker, EAGLE3 serves as a robust fallback, since it proposes correct tokens across diverse contexts more consistently.

This naturally brings up the next question: *can we combine their advantages to achieve even higher overall speedups?*

To explore the upper bound on achievable speedup, we assume a *perfect* predictor that can (1) determine which method performs best at each position, and (2) accurately predict the number of tokens that will be accepted at that position. Under this idealized assumption, we measure the maximum possible speedup attainable by speculative decoding.

It is important to note the simplifications in this setup, which represent an upper bound on achievable performance. First, we assume the predictor is flawless in determining the better method for each position. Second, we presume perfect knowledge of the number of tokens each method can accept at every position. Finally, we do not account for the overhead of maintaining the KV cache for EAGLE3's proposing head, which would otherwise require partial prefills if a request was paused and resumed.

Under these assumptions, we report the best achievable speedup in Fig. 10. **Oracle_Combine** represents a theoretical upper bound in which, at each generation position, the method (EAGLE3 or n-gram) that yields the longer accepted span is selected. The resulting accepted length is then used as the proposed length. Compared to the best fixed strategy, **Oracle_Combine** exposes substantial additional headroom, achieving up to a $1.6\times$ further speedup.

The size of this headroom depends strongly on the workload. InstructCoder shows the largest gap between existing methods and Oracle_Combine, matching Fig. 9 where red and blue regions alternate frequently, i.e., there are many positions where one method clearly outperforms the other. ShareGPT and CNN/DailyMail exhibit similar but smaller gains, indicating more limited but still meaningful room for the methods to complement each other. In contrast, GSM8K offers little additional benefit from combining because n-gram rarely achieves long acceptances, so **Oracle_Combine** stays close to **Oracle_Eagle**.

Taken together, these results point to a promising direction for future work: developing an accurate yet lightweight predictor capable of adapting to varying levels of acceptance behavior across workloads, requests and token positions.

9 Conclusion

This work presents the first systematic evaluation of speculative decoding (SD) in a real, optimized inference engine. By systematically dissecting end-to-end performance across workloads and SD variants, we identify verification as the dominant bottleneck and reveal strong variability in acceptance behavior across positions, requests, and datasets. Leveraging these insights, we quantify the theoretical upper bound of SD speedup and expose the gap between observed and ideal efficiency. Together, these findings deepen our understanding of SD's practical behavior and highlight opportunities that can further unlock its full potential in large-scale inference systems.

10 Acknowledgements

We thank all reviewers and our shepherd for their valuable feedback. We thank Tomas Ruiz for implementing draft-model-based speculative decoding in vLLM. We also thank everyone from the Sky Computing Lab for their discussions. This work was supported in part by the DGX server gifted by NVIDIA. Jiaxiang is supported by the Singapore National Science Scholarship and the NUS Development Grant. This work is supported by NSF grants IIS-1955488, IIS-2027575, DOE awards DE-SC0016260, AC02-05CH11231, and DARPA Agreement No. HR00112590131.

REFERENCES

ShareGPT dataset. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered, 2024. HuggingFace dataset.

AoPS. AIME Problems and Solutions. https://artofproblemsolving.com/wiki/index.php/AIME_Problems_and_Solutions, 2024.

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.

Cai, Y., Liang, X., Wang, X., Ma, J., Liang, H., Luo, J., Zuo, X., Duan, L., Yin, Y., and Chen, X. Fastmtp: Accelerating llm inference with enhanced multi-token prediction, 2025. URL https://arxiv.org/abs/2509.18362.

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

Chen, Z., May, A., Svirschevski, R., Huang, Y., Ryabinin, M., Jia, Z., and Chen, B. Sequoia: Scalable, robust, and hardware-aware speculative decoding. *arXiv preprint arXiv:2402.12374*, 2024.

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. *arXiv preprint arXiv:2110.14168*, 2021.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

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

He, H. Defeating nondeterminism in llm inference. *Thinking Machines Lab: Connectionism*, Sep 2025. doi: 10.64434/tml.20250910. https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

He, Z., Zhong, Z., Cai, T., Lee, J. D., and He, D. Rest: Retrieval-based speculative decoding, 2024. URL https://arxiv.org/abs/2311.08252.

Hermann, K. M., Kocisky, T., Grefenstette, E., Espeholt, L., Kay, W., Suleyman, M., and Blunsom, P. Teaching machines to read and comprehend. *Advances in neural information processing systems*, 28, 2015.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Joao Gante. Assisted generation: a new direction toward low-latency text generation, 2023. URL https://huggingface.co/blog/assisted-generation.

Kim, S., Hooper, C., Gholami, A., Dong, Z., Li, X., Shen, S., Mahoney, M. W., and Keutzer, K. Squeezellm: Dense-and-sparse quantization, 2024.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

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

Li, K., Hu, Q., Zhao, J., Chen, H., Xie, Y., Liu, T., Shieh, M., and He, J. Instructcoder: Instruction tuning large language models for code editing. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pp. 50–70, 2024a.

Li, Y., Wei, F., Zhang, C., and Zhang, H. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *International Conference on Machine Learning*, 2024b.

Li, Y., Wei, F., Zhang, C., and Zhang, H. Eagle-2: Faster inference of language models with dynamic draft trees, 2024c. URL https://arxiv.org/abs/2406.16858.

Li, Y., Wei, F., Zhang, C., and Zhang, H. EAGLE-3: Scaling up inference acceleration of large language models via training-time test. In *Annual Conference on Neural Information Processing Systems*, 2025.

Lin, F., Yi, H., Li, H., Yang, Y., Yu, X., Lu, G., and Xiao, R. Bita: Bi-directional tuning for lossless acceleration in large language models. *arXiv preprint arXiv:2401.12522*, 2024.

Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., Gan, C., and Han, S. Awq: Activation-aware weight quantization for llm compression and acceleration, 2023.

Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Bao, H., Xu, H., Wang, H., Zhang, H., Ding, H., Xin, H., Gao, H., Li, H., Qu, H., Cai, J. L., Liang, J., Guo, J., Ni, J., Li, J., Wang, J., Chen, J., Chen, J., Yuan, J., Qiu, J., Li, J., Song, J., Dong, K., Hu, K., Gao, K., Guan, K., Huang, K., Yu, K., Wang, L., Zhang, L., Xu, L., Xia, L., Zhao, L., Wang, L., Zhang, L., Li, M., Wang, M., Zhang, M., Zhang, M., Tang, M., Li, M., Tian, N., Huang, P., Wang, P., Zhang, P., Wang, Q., Zhu, Q., Chen, Q., Du, Q., Chen, R. J., Jin, R. L., Ge, R., Zhang, R., Pan, R., Wang, R., Xu, R., Zhang, R., Chen, R., Li, S. S., Lu, S., Zhou, S., Chen, S., Wu, S., Ye, S., Ye, S., Ma, S., Wang, S., Zhou, S., Yu, S., Zhou, S., Pan, S., Wang, T., Yun, T., Pei, T., Sun, T., Xiao, W. L., Zeng, W., Zhao, W., An, W., Liu, W., Liang, W., Gao, W., Yu, W., Zhang, W., Li, X. Q., Jin, X., Wang, X., Bi, X., Liu, X., Wang, X., Shen, X., Chen, X., Zhang, X., Chen, X., Nie, X., Sun, X., Wang, X., Cheng, X., Liu, X., Xie, X., Liu, X., Yu, X., Song, X., Shan, X., Zhou, X., Yang, X., Li, X., Su, X., Lin, X., Li, Y. K., Wang, Y. Q., Wei, Y. X., Zhu, Y. X., Zhang, Y., Xu, Y., Xu, Y., Huang, Y., Li, Y., Zhao, Y., Sun, Y., Li, Y., Wang, Y., Yu, Y., Zheng, Y., Zhang, Y., Shi, Y., Xiong, Y., He, Y., Tang, Y., Piao, Y., Wang, Y., Tan, Y., Ma, Y., Liu, Y., Guo, Y., Wu, Y., Ou, Y., Zhu, Y., Wang, Y., Gong, Y., Zou, Y., He, Y., Zha, Y., Xiong, Y., Ma, Y., Yan, Y., Luo, Y., You, Y., Liu, Y., Zhou, Y., Wu, Z. F., Ren, Z. Z., Ren, Z., Sha, Z., Fu, Z., Xu, Z., Huang, Z., Zhang, Z., Xie, Z., Zhang, Z., Hao, Z., Gou, Z., Ma, Z., Yan, Z., Shao, Z., Xu, Z., Wu, Z., Zhang, Z., Li, Z., Gu, Z., Zhu, Z., Liu, Z., Li, Z., Xie, Z., Song, Z., Gao, Z., and Pan, Z. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

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

Liu, X., Daniel, C., Hu, L., Kwon, W., Li, Z., Mo, X., Cheung, A., Deng, Z., Stoica, I., and Zhang, H. Optimizing speculative decoding for serving large language models using goodput, 2024. URL https://arxiv.org/abs/2406.14066v2.

Miao, X., Oliaro, G., Zhang, Z., Cheng, X., Wang, Z., Wong, R. Y. Y., Chen, Z., Arfeen, D., Abhyankar, R., and Jia, Z. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*, 2023.

NVIDIA. Getting started with cuda graphs. https://developer.nvidia.com/blog/cuda-graphs/, 2024a. Accessed: 2025-04-12.

NVIDIA. H100. https://www.nvidia.com/en-us/data-center/h100/, 2024b. Accessed: 2024-11-19.

OpenAI. Introducing ChatGPT. https://openai.com/index/chatgpt/, 2022.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.

Project Numina. AI-MO/aimo-validation-aime. https://huggingface.co/datasets/AI-MO/aimo-validation-aime, 2024.

Rein, D., Hou, B. L., Stickland, A. C., Petty, J., Pang, R. Y., Dirani, J., Michael, J., and Bowman, S. R. Gpqa: A graduate-level google-proof q&a benchmark, 2023. URL https://arxiv.org/abs/2311.12022.

Saxena, A. Prompt lookup decoding, November 2023. URL https://github.com/apoorvumang/prompt-lookup-decoding/.

See, A., Liu, P. J., and Manning, C. D. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1073–1083, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1099. URL https://www.aclweb.org/anthology/P17-1099.

Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., and Dao, T. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

Somasundaram, S., Phukan, A., and Saxena, A. Pld+: Accelerating llm inference by leveraging language model artifacts. *arXiv preprint arXiv:2412.01447*, 2024.

Xia, H., Yang, Z., Dong, Q., Wang, P., Li, Y., Ge, T., Liu, T., Li, W., and Sui, Z. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding. In Ku, L.-W., Martins, A., and Srikumar, V. (eds.), *Findings of the Association for Computational Linguistics ACL 2024*, pp. 7655–7671, Bangkok, Thailand and virtual meeting, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.456. URL https://aclanthology.org/2024.findings-acl.456.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., Zheng, C., Liu, D., Zhou, F., Huang, F., Hu, F., Ge, H., Wei, H., Lin, H., Tang, J., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Zhou, J., Lin, J., Dang, K., Bao, K., Yang, K., Yu, L., Deng, L., Li, M., Xue, M., Li, M., Zhang, P., Wang, P., Zhu, Q., Men, R., Gao, R., Liu, S., Luo, S., Li, T., Tang, T., Yin, W., Ren, X., Wang, X., Zhang, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Zhang, Y., Wan, Y., Liu, Y., Wang, Z., Cui, Z., Zhang, Z., Zhou, Z., and Qiu, Z. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Ye, Z., Lai, R., Lu, B.-R., Lin, C.-Y., Zheng, S., Chen, L., Chen, T., and Ceze, L. Cascade inference: Memory bandwidth efficient shared prefix batch decoding, February 2024. URL https://flashinfer.ai/2024/02/02/cascade-inference.html.

Ye, Z., Chen, L., Lai, R., Lin, W., Zhang, Y., Wang, S., Chen, T., Kasikci, B., Grover, V., Krishnamurthy, A., et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *Proceedings of Machine Learning and Systems*, 7, 2025.

Zeng, A., Lv, X., Zheng, Q., Hou, Z., Chen, B., Xie, C., Wang, C., Yin, D., Zeng, H., Zhang, J., Wang, K., Zhong, L., Liu, M., Lu, R., Cao, S., Zhang, X., Huang, X., Wei, Y., Cheng, Y., An, Y., Niu, Y., Wen, Y., Bai, Y., Du, Z., Wang, Z., Zhu, Z., Zhang, B., Wen, B., Wu, B., Xu, B., Huang, C., Zhao, C., Cai, C., Yu, C., Li, C., Ge, C., Huang, C., Zhang, C., Xu, C., Zhu, C., Li, C., Yin, C., Lin, D., Yang, D., Jiang, D., Ai, D., Zhu, E., Wang, F., Pan, G., Wang, G., Sun, H., Li, H., Li, H., Hu, H., Zhang, H., Peng, H., Tai, H., Zhang, H., Wang, H., Yang, H., Liu, H., Zhao, H., Liu, H., Yan, H., Liu, H., Chen, H., Li, J., Zhao, J., Ren, J., Jiao, J., Zhao, J., Yan, J., Wang, J., Gui, J., Zhao, J., Liu, J., Li, J., Li, J., Lu, J., Wang, J., Yuan, J., Li, J., Du, J., Du, J., Liu, J., Zhi, J., Gao, J., Wang, K., Yang, L., Xu, L., Fan, L., Wu, L., Ding, L., Wang, L., Zhang, M., Li, M., Xu, M., Zhao, M., Zhai, M., Du, P., Dong, Q., Lei, S., Tu, S., Yang, S., Lu, S., Li, S., Li, S., Shuang-Li, Yang, S., Yi, S., Yu, T., Tian, W., Wang, W., Yu, W., Tam, W. L., Liang, W., Liu, W., Wang, X., Jia, X., Gu, X., Ling, X., Wang, X., Fan, X., Pan, X., Zhang, X., Zhang, X., Fu, X., Zhang, X., Xu, Y., Wu, Y., Lu, Y., Wang, Y., Zhou, Y., Pan, Y., Zhang, Y., Wang, Y., Li, Y., Su, Y., Geng, Y., Zhu, Y., Yang, Y., Li, Y., Wu, Y., Li, Y., Liu, Y., Wang, Y., Li, Y., Zhang, Y., Liu, Z., Yang, Z., Zhou, Z., Qiao, Z., Feng, Z., Liu, Z., Zhang, Z., Wang, Z., Yao, Z., Wang, Z., Liu, Z., Chai, Z., Li, Z., Zhao, Z., Chen, W., Zhai, J., Xu, B., Huang, M., Wang, H., Li, J., Dong, Y., and Tang, J. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models, 2025. URL https://arxiv.org/abs/2508.06471.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., et al. Sglang: Efficient execution of structured language model programs. *arXiv preprint arXiv:2312.07104*, 2024.

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

A APPENDIX

A.1 Datasets Statistics

Fig. 11 shows the input and output length distributions across all datasets and models used in our evaluation.

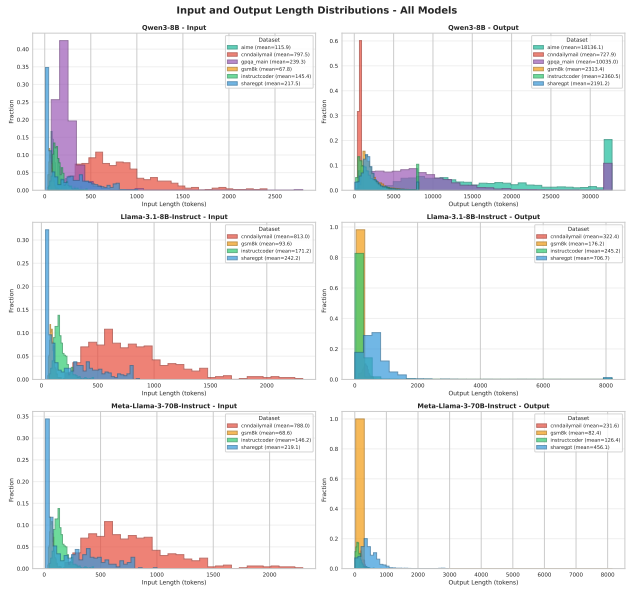
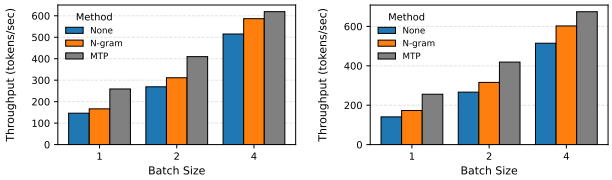


Figure 11. Dataset Input/Output Length distributions.

A.2 Extended Details on End-to-End Performance

A.2.1 MTP Performance



(a) GLM-4.5-Air, AIME22-24 (b) GLM-4.5-Air, GPQA-Main

Figure 12. End-to-end performance on reasoning workloads.

For GLM-4.5-Air, MTP outperforms n-gram but falls short of the theoretical upper bound, achieving 1.3×–1.8× speedup on GPQA-Main. This gap likely arises because, in our inference setup, the released open-source weights include only the first MTP module, and this MTP module is reused autoregressively to generate multiple draft tokens, even though vanilla MTP may not be explicitly trained for recursive multi-step prediction (Cai et al., 2025). As a result, its position-wise acceptance rate drops across drafted tokens (0.91 → 0.67 → 0.38 on GPQA-Main), reducing the accepted length and thereby limiting the achievable speedup.

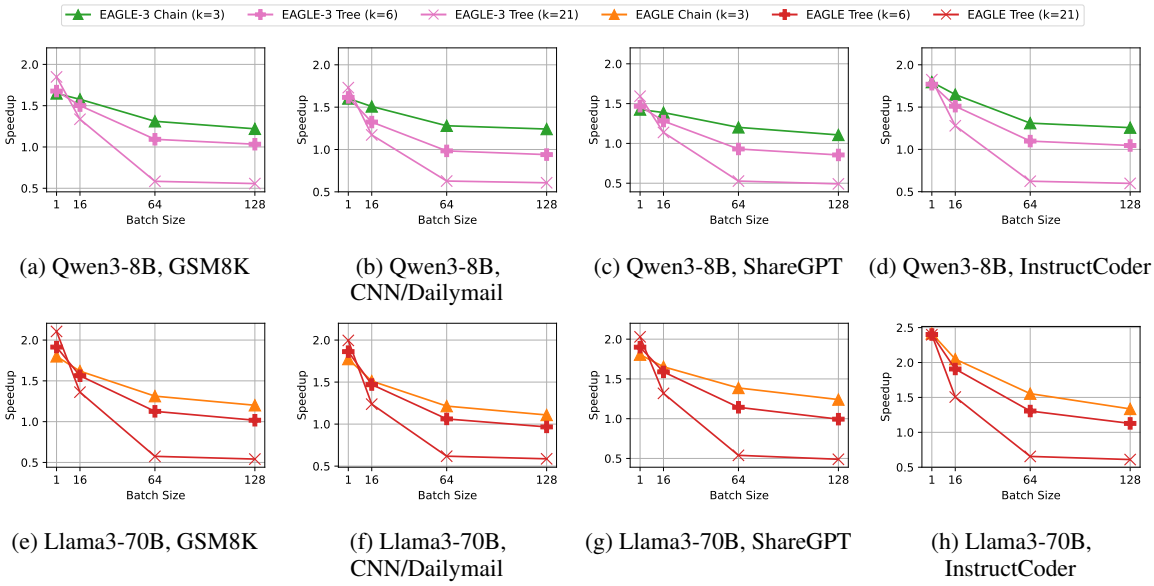
A.2.2 Tree-Style Verification

Tree-Style Verification. *Tree-based methods provide slightly higher speedup at batch size 1, but the advantage quickly disappears as batch size increases. Across batch sizes, the chain-style method often remains the more performant.*

We evaluate tree-style verification in SGLang v0.5.9 (Zheng et al., 2024) using Qwen3-8B and Llama3-70B on the non-reasoning workloads. We use SGLang for this study because the draft-tree path in the vLLM version we evaluated is not yet sufficiently optimized for a fair comparison across tree configurations. To control the comparison, we fix the tree depth to 3 in all runs. We use chain-style verification with k=3 as the baseline, evaluate a wider tree with k=21 and branching factor 4 to approximately match the tree structure used in EAGLE (Li et al., 2024b), and include an intermediate tree with k=6 and branching factor 2. Here, k denotes the number of draft tokens verified in parallel by the target model. Unless otherwise noted, all other settings follow those described earlier. By default, SGLang uses a dynamic draft-tree policy which optimises the tree structure during decoding (Li et al., 2024b). In this setup, FlashAttention-3 (Shah et al., 2024) is used for the chain configuration, whereas FlashInfer (Ye et al., 2025) is used for the tree configurations.

As shown in Fig. 13, tree-based EAGLE/EAGLE-3 achieves slightly higher speedup than the chain-based settings at batch size 1. For example, on Qwen3-8B with GSM8K, speedup increases from 1.65× for the chain to 1.68× for k=6 and 1.85× for k=21. On Llama3-70B with ShareGPT, it rises from 1.81× to 1.90× and 2.03×. However, this benefit quickly disappears as batch size increases. By batch size 64, the k=21 tree falls below 1× speedup on all workloads for both models, whereas the chain remains above 1× throughout.

This is likely because wider trees increase the accepted length, but also verify many more tokens that are later rejected. On Qwen3-8B with GSM8K, the accepted length increases from 2.25 for the chain to 2.51 and 2.92 for the k=6 and k=21 trees, respectively, while the acceptance rate decreases from 0.415 to 0.300 and 0.095. We observe the same trend on Llama3-70B with ShareGPT (accepted length: 2.29 → 2.55 → 2.93; acceptance rate: 0.429 → 0.310 → 0.097). As discussed in Sec. 5, verification dominates the execution cost, so the overhead of these additional rejected branches becomes increasingly expensive at larger batch sizes. Thus, tree-style verification provides only a narrow low-batch-size benefit, while chain-style verification remains the more robust setting overall.



(a) Qwen3-8B, GSM8K (b) Qwen3-8B, CNN/Dailymail (c) Qwen3-8B, ShareGPT (d) Qwen3-8B, InstructCoder

(e) Llama3-70B, GSM8K (f) Llama3-70B, CNN/Dailymail (g) Llama3-70B, ShareGPT (h) Llama3-70B, InstructCoder

Figure 13. End-to-end performance on non-reasoning workloads for tree-style verification. We compare a chain setting (k=3) against tree settings with k=6 and k=21, using a fixed depth of 3 for all runs.

A.3 Memory Breakdown Calculations

Static memory (GiB). Assuming FP16 weights (2 bytes/-param), the static GPU memory for model weights:

$$M_{\text{static,GiB}} = \frac{(P_{\text{target}} + P_{\text{draft}}) \cdot 2}{2^{30}}$$

Here, P_{target} is the target model's parameter size and P_{draft} includes any additional parameters introduced by the speculative decoding method, both in units of Billions.

For EAGLE/EAGLE3, P_{draft} refers to the parameters used in speculative decoding. For the EAGLE models we used, P_{draft} refers to the parameter size of the autoregressive head, which includes one decoding layer and one fully connected (FC) layer. They share the same embedding layer and language modeling (LM) head with the target model. For the EAGLE3 models we used, P_{draft} refers to the parameter size of the entire EAGLE3 model. It includes one decoding layer, the multi-layer feature fusion FC layer, a final normalization layer and its own LM head. In addition, the model checkpoints include two 1-dimensional token-id remapping tables (`t2d` and `d2t`) that translate between the target model's vocabulary and the draft LM head's vocabulary; they typically contribute negligibly to the parameter memory. When loading model weights in vLLM v0.10.1.1, the `t2d` tensors are skipped, and we exclude them when counting the parameter size. Similarly, these EAGLE3 models reuse the same embedding layer from the target models.

For Qwen3-0.6B, *tie_word_embeddings* defaults to True and its LM head shares the same weight matrix as the embedding layer; thus, we exclude the size of its language modeling (LM) head.

For example, the static memory for Llama3-70B-Instruct with Llama3.2-1B-Instruct as the draft model is $(70.55 + 1.23) * 10^9 * 2/2^{30} = 133.7GiB$. For Llama3.1-8B-Instruct with EAGLE-LLaMA3.1-Instruct-8B, it is $(8.03 + 0.25) * 10^9 * 2/2^{30} = 15.42GiB$.

Per-token KV cache (KiB). To calculate the KV cache size of each generated token:

$$M_{\text{KV/token,KiB}} = \frac{L_{\text{h}} \cdot 2 \cdot n_{\text{kv}} \cdot d_{\text{head}} \cdot 2}{2^{10}}.$$

L_{h} is the number of hidden layers, n_{kv} is the number of KV heads (since our models are all Grouped-Query-Attention-based), d_{head} is the head dimension, the first factor 2 is for both Key and Value, and the last factor 2 is bytes/element for FP16. For Llama3-70B-Instruct without speculative decoding or model-free SD methods like n-gram, it is $80 * 2 * 8 * 128 * 2/2^{10} = 320KiB$. Similarly, for Llama3.1-8B-Instruct, it is $32 * 2 * 8 * 128 * 2/2^{10} = 128KiB$.

For draft-model-based SD, the total per-token KV cache is the sum over each generated token by the target and draft model. For Llama3-70B-Instruct with Llama3.2-1B-Instruct as the draft model, the per-token KV cache memory size is $80 * 2 * 8 * 128 * 2/2^{10} + 16 * 2 * 8 * 64 * 2/2^{10} = 352KiB$.

Similarly, for EAGLE-based SD, the total per-token KV cache is the sum over each generated token by the target and EAGLE heads. For Llama3-70B-Instruct with Llama3.2-1B-Instruct as the draft model, the per-token KV cache memory size is $80 * 2 * 8 * 128 * 2/2^{10} + 1 * 2 * 8 * 128 * 2/2^{10}$

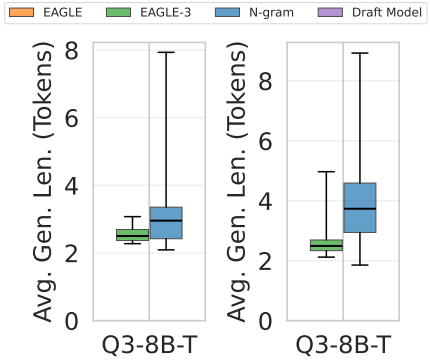
$= 324KiB$. The value 1 here means that EAGLE uses an additional transformer block for proposal generation, which effectively adds one extra layer.

Model	P	L	n_{kv}	d_{head}
Llama3.1-8B-Instruct	8.03B	32	8	128
Llama3-70B-Instruct	70.55B	80	8	128
Llama3.2-1B-Instruct	1.23B	16	8	64
Qwen3-8B	8.19B	36	8	128
Qwen3-0.6B	0.596B	28	8	128
EAGLE-LLaMA3.1-Instruct-8B	0.25B	1	8	128
EAGLE3-LLaMA3.1-Instruct-8B	0.425B	1	8	128
EAGLE-LLaMA3-Instruct-70B	0.99B	1	8	128
EAGLE3-Qwen3-8B	0.40B	1	8	128

Table 3. Model specs used for memory calculations. P is the parameter size of the weights used, obtained by counting their parameters; L is the number of hidden layers, n_{kv} is the number of Key/Value heads, and d_{head} is the head dimension, obtained from the model's configuration files on Hugging Face.

A.4 Extended Results on Acceptance Behaviour

Fig. 14 shows the request-level distribution of mean accepted length for reasoning workloads on AIME and GPQA-Main, while Tab. 4 summarizes the dataset-level statistics across all methods.



(a) AIME22-24 (b) GPQA-Main

Figure 14. Request-level generated length across datasets and models. Each box shows the distribution of per-request mean accepted length, with the box spanning the 25th–75th percentiles and whiskers covering the 5th–95th percentiles. Model used is Qwen3-8B-Thinking.

A.5 Extended Results on Case Study: n-gram Performance on InstructCoder

Fig. 15 shows a representative example of prompt-output repetition in InstructCoder: the output preserves most of the prompt verbatim and applies only a localized edit, which n-gram can exploit to produce occasional long accepted spans. Fig. 16 quantifies this effect by correlating BLEU-4 scores with relative speedup.

Dataset	Method	Mean	Median	Std
InstructCoder	n-gram	7.27	6.57	4.19
InstructCoder	EAGLE	4.24	4.21	1.07
InstructCoder	EAGLE-3	4.18	4.06	0.99
CNN/DailyMail	EAGLE-3	3.08	2.97	0.61
CNN/DailyMail	EAGLE	2.32	2.31	0.21
CNN/DailyMail	n-gram	2.33	1.96	1.08
ShareGPT	EAGLE-3	3.06	2.92	0.87
ShareGPT	EAGLE	2.82	2.68	0.80
ShareGPT	n-gram	2.02	1.37	1.87
GSM8K	EAGLE-3	3.02	2.92	0.47
GSM8K	EAGLE	2.56	2.51	0.25
GSM8K	n-gram	1.41	1.28	0.45
GPQA_Main	n-gram	4.20	3.73	2.24
GPQA_Main	EAGLE-3	2.78	2.49	0.87
AIME22-24	n-gram	3.53	2.96	2.29
AIME22-24	EAGLE-3	2.62	2.50	0.54

Table 4. Dataset-level generated length by method, aggregated across models. For non-reasoning workloads, the average is aggregated over the results for Llama3.1-8B and Llama3-70B. For reasoning workloads, the average is aggregated over that of Qwen3-8B-Thinking.

Prompt

```
class Car:
    def __init__(self, make,
      model,
                 year, color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color

car1 = Car('Toyota', 'Camry',
           2018, 'Red')
print(car1.color)

# Replace the string data type
# used for car color with an
# enum data type. Just output
# the code, do not
# include any explanation.
```

Output

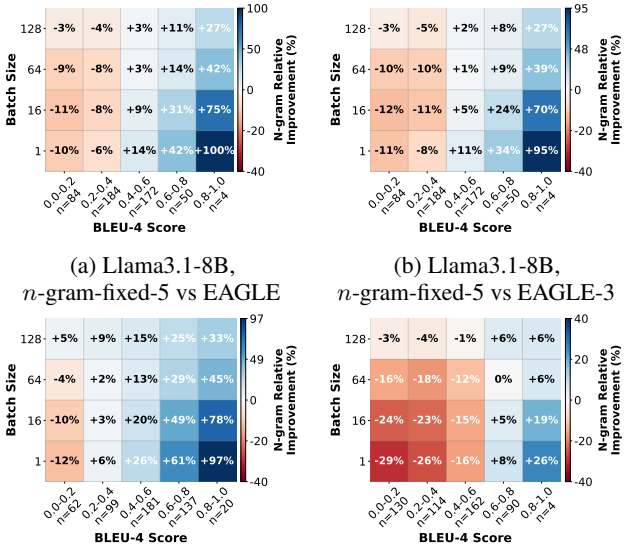
```
from enum import Enum

class Color(Enum):
    RED = 1
    BLUE = 2
    GREEN = 3
    YELLOW = 4
    BLACK = 5
    WHITE = 6

class Car:
    def __init__(self, make,
                 model, year,
                 color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color

car1 = Car('Toyota', 'Camry',
           2018, Color.RED)
print(car1.color)
```

Figure 15. **Example of prompt-output repetition in code editing from InstructCoder.** The output preserves most of the prompt and applies a localized edit, a structure that n-gram can exploit to produce occasional long accepted spans.



(a) Llama3.1-8B, n-gram-fixed-5 vs EAGLE

(b) Llama3.1-8B, n-gram-fixed-5 vs EAGLE-3

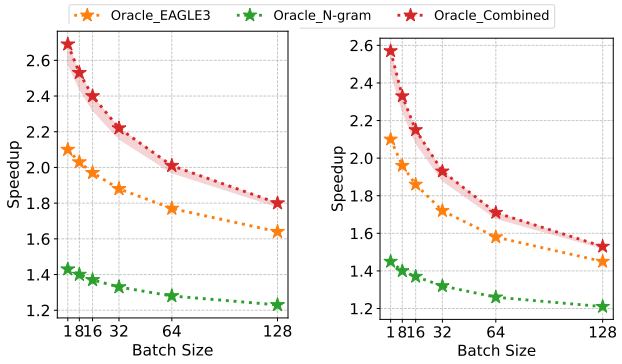
(c) Qwen3-8B, n-gram-fixed-5 vs EAGLE-3

(d) Llama3-70B, n-gram-fixed-5 vs EAGLE

Figure 16. Correlation between BLEU-4 and n-gram speedup on InstructCoder. Each heatmap shows n-gram's relative speedup (%) over EAGLE or EAGLE-3, grouped by BLEU-4 scores (x-axis) and batch sizes (y-axis). Blue means that n-gram outperforms the draft-model method, while red means the opposite. n=XYZ in the x-axis labels denotes the number of requests in each BLEU bucket. Figures show the results when that is set to 5.

A.6 Extended Results on Oracle Speedup

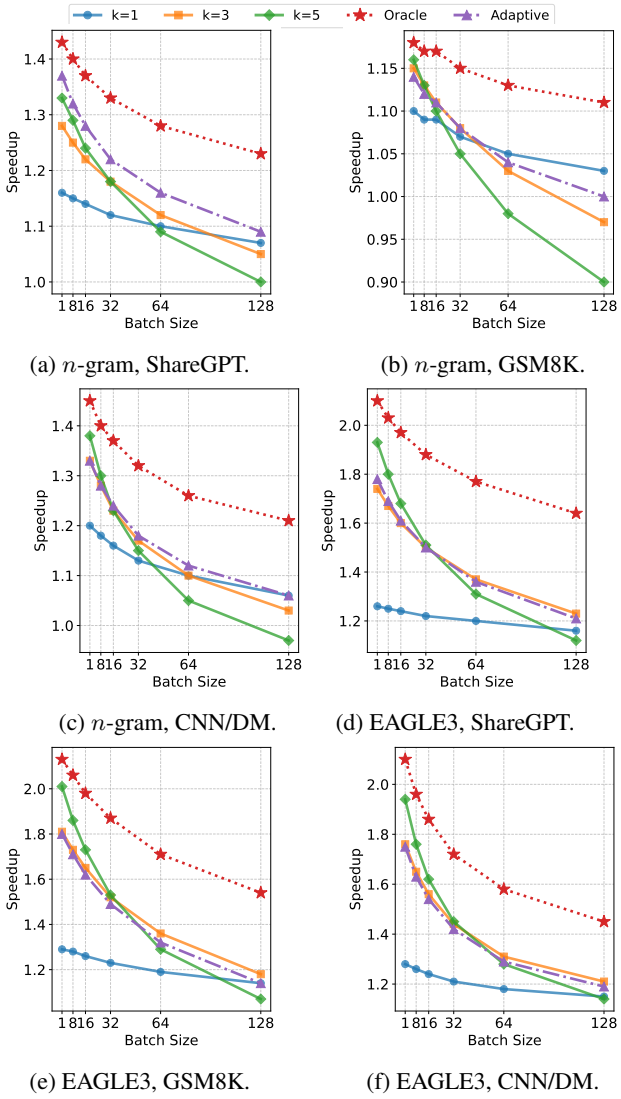
Fig. 17 compares oracle, adaptive and fixed proposed lengths for n-gram and EAGLE3 on Llama3.1-8B. Fig. 18 then shows per-position acceptance-length differences between the two methods, and Fig. 19 demonstrates the potential gain from combining them.



(a) ShareGPT

(b) CNN/DailyMail

Figure 19. Combining different SD methods for optimal speedup on Llama3.1-8B.



(a) n-gram, ShareGPT.

(b) n-gram, GSM8K.

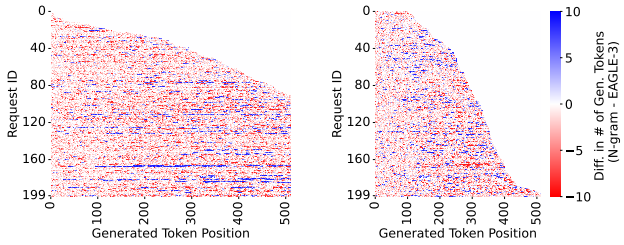
(c) n-gram, CNN/DM.

(d) EAGLE3, ShareGPT.

(e) EAGLE3, GSM8K.

(f) EAGLE3, CNN/DM.

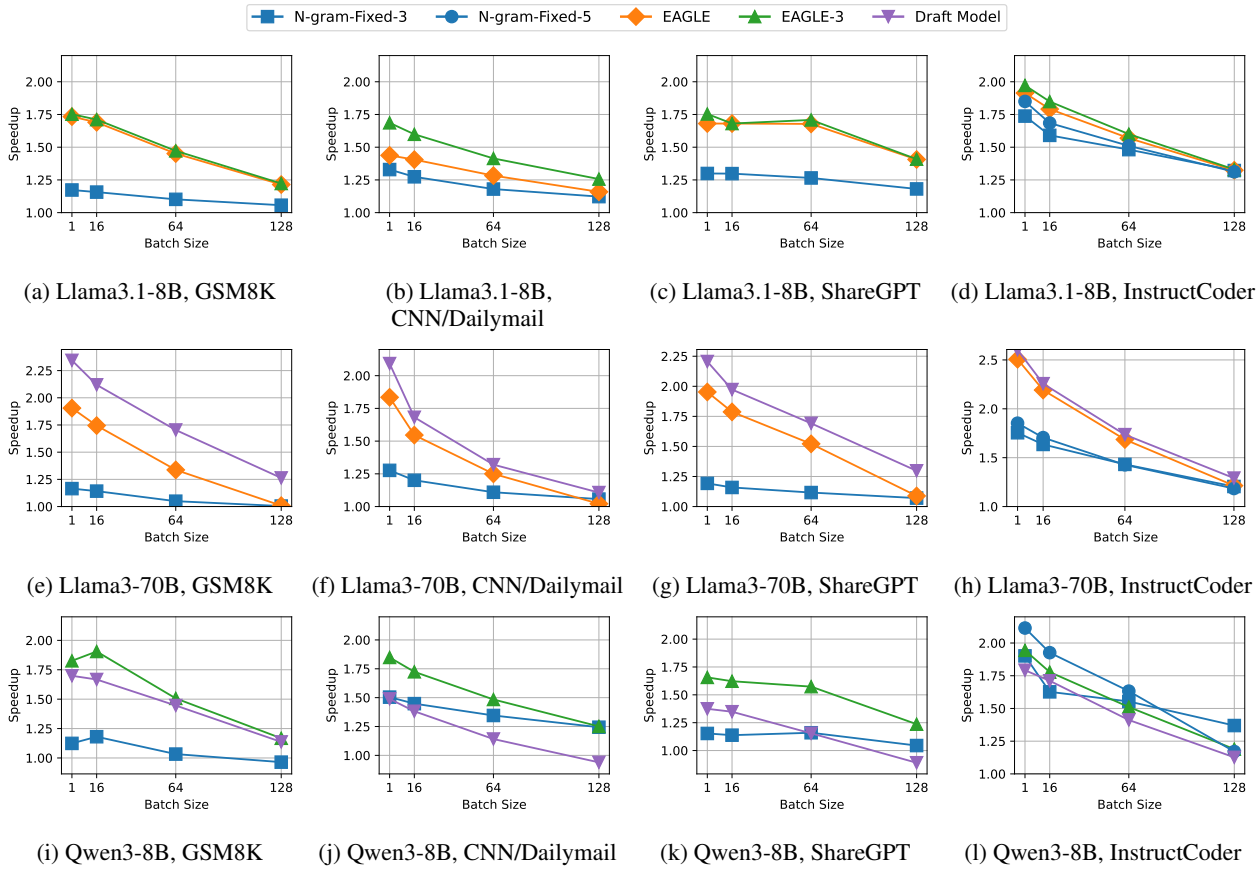
Figure 17. Oracle, fixed and adaptive proposed length speedup on Llama3.1-8B.



(a) ShareGPT

(b) CNN/DailyMail

Figure 18. Per-position accepted-length difference between n-gram and EAGLE3 on Llama 3.1-8B. Red indicates positions where EAGLE3 accepts longer spans, while blue indicates positions favoring n-gram. The figure illustrates that different speculative decoding methods exhibit distinct acceptance behaviors across decoding positions.

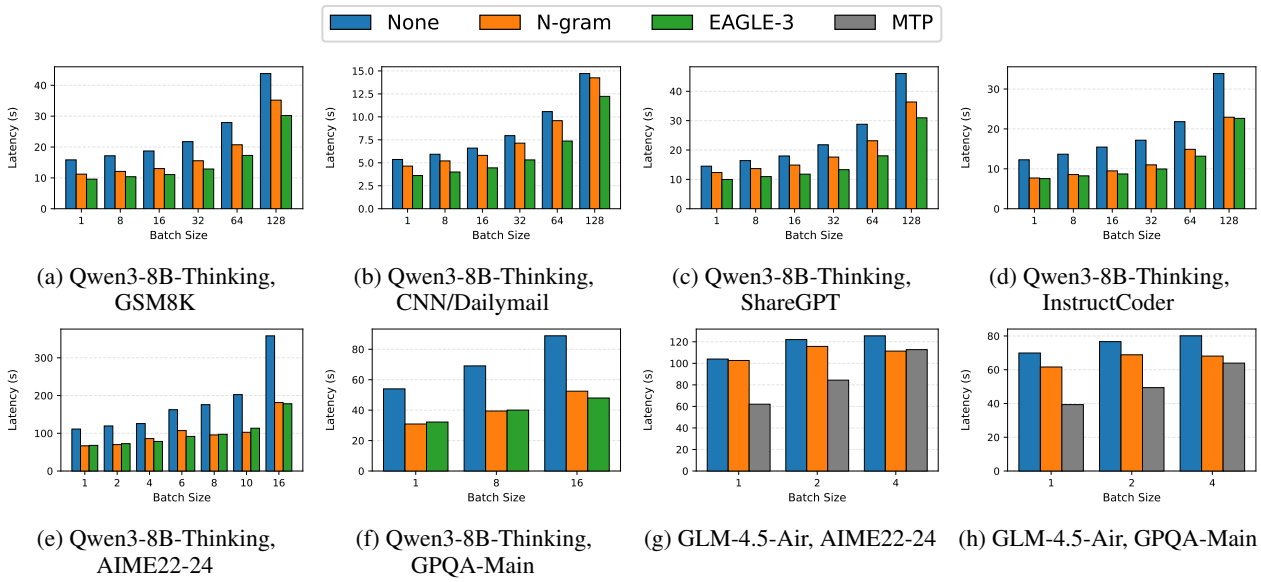


(a) Llama3.1-8B, GSM8K (b) Llama3.1-8B, CNN/Dailymail (c) Llama3.1-8B, ShareGPT (d) Llama3.1-8B, InstructCoder

(e) Llama3-70B, GSM8K (f) Llama3-70B, CNN/Dailymail (g) Llama3-70B, ShareGPT (h) Llama3-70B, InstructCoder

(i) Qwen3-8B, GSM8K (j) Qwen3-8B, CNN/Dailymail (k) Qwen3-8B, ShareGPT (l) Qwen3-8B, InstructCoder

Figure 20. End-to-end performance on non-reasoning workloads. Speedup is calculated between the average e2e request latency with SD and that without SD.



(a) Qwen3-8B-Thinking, GSM8K (b) Qwen3-8B-Thinking, CNN/Dailymail (c) Qwen3-8B-Thinking, ShareGPT (d) Qwen3-8B-Thinking, InstructCoder

(e) Qwen3-8B-Thinking, AIME22-24 (f) Qwen3-8B-Thinking, GPQA-Main (g) GLM-4.5-Air, AIME22-24 (h) GLM-4.5-Air, GPQA-Main

Figure 21. End-to-end performance on reasoning workloads in end-to-end request latencies.

Table 5. Main experiment configurations across datasets, models, and speculative decoding (SD) methods used in End-to-End Speedup Measurement. DM stands for Draft-Model-Based SD.

Dataset	Model	Hardware	SD Method	Max Output Len.	Batch Size
CNNDailyMail	Llama3.1-8B-Instruct	1×H100	n-gram, EAGLE, EAGLE-3	8K	1–128
ShareGPT	Llama3.1-8B-Instruct	1×H100	n-gram, EAGLE, EAGLE-3	8K	1–128
InstructCoder	Llama3.1-8B-Instruct	1×H100	n-gram, EAGLE, EAGLE-3	8K	1–128
GSM8K	Llama3.1-8B-Instruct	1×H100	n-gram, EAGLE, EAGLE-3	8K	1–128
CNNDailyMail	Llama3-70B-Instruct	4×H100	n-gram, EAGLE, DM	8K	1–128
ShareGPT	Llama3-70B-Instruct	4×H100	n-gram, EAGLE, DM	8K	1–128
InstructCoder	Llama3-70B-Instruct	4×H100	n-gram, EAGLE, DM	8K	1–128
GSM8K	Llama3-70B-Instruct	4×H100	n-gram, EAGLE, DM	8K	1–128
CNNDailyMail	Qwen3-8B	1×H100	n-gram, EAGLE-3, DM (v0.11.1rc1)	8K	1–128
ShareGPT	Qwen3-8B	1×H100	n-gram, EAGLE-3, DM (v0.11.1rc1)	8K	1–128
InstructCoder	Qwen3-8B	1×H100	n-gram, EAGLE-3, DM (v0.11.1rc1)	8K	1–128
GSM8K	Qwen3-8B	1×H100	n-gram, EAGLE-3, DM (v0.11.1rc1)	8K	1–128
CNNDailyMail	Qwen3-8B-Thinking	1×H100	n-gram, EAGLE-3	8K	1–128
ShareGPT	Qwen3-8B-Thinking	1×H100	n-gram, EAGLE-3	8K	1–128
InstructCoder	Qwen3-8B-Thinking	1×H100	n-gram, EAGLE-3	8K	1–128
GSM8K	Qwen3-8B-Thinking	1×H100	n-gram, EAGLE-3	8K	1–128
Reasoning workloads					
AIME(22--24)	Qwen3-8B-Thinking	1×H100	n-gram, EAGLE-3	32K	1–16
GPQA-Main	Qwen3-8B-Thinking	1×H100	n-gram, EAGLE-3	32K	1–16
AIME(22--24)	GLM-4.5-Air	4×H100	n-gram, MTP (v0.11.1rc1)	32K	1–4
GPQA-Main	GLM-4.5-Air	4×H100	n-gram, MTP (v0.11.1rc1)	32K	1–4

Table 6. Model configurations for EAGLE and Draft-Model-based SD. Each configuration specifies the target model, the associated draft (or auxiliary) model from Hugging Face, and the number of speculative tokens proposed per decoding step.

Variant	Target Model	# Draft Tokens	Draft / Auxiliary Model (Hugging Face)
EAGLE	Llama3.1-8B-Instruct	3	yuhuili/EAGLE-LLaMA3.1-Instruct-8B
	Llama3-70B-Instruct	3	yuhuili/EAGLE-LLaMA3-Instruct-70B
EAGLE-3	Llama3.1-8B-Instruct	3	yuhuili/EAGLE3-LLaMA3.1-Instruct-8B
	Qwen3-8B	3	AngelSlim/Qwen3-8B_eagle3
Draft-Model-Based	Llama3-70B-Instruct	3	meta-llama/Llama-3.2-1B-Instruct
	Qwen3-8B	3	Qwen/Qwen3-0.6B

B Artifact Appendix

B.1 Abstract

This artifact contains two components. The first is a vLLM-based profiling suite that includes implementation and benchmarking scripts for speculative decoding in vLLM, covering end-to-end throughput, time breakdown, and acceptance rate experiments across multiple models and methods (N-gram, EAGLE, EAGLE-3, Draft Model, MTP). Full reproduction of this suite requires Linux, Python 3.10, CUDA 12.8, conda/uv, HuggingFace access to gated Llama-3 models, a one-time ShareGPT download, and NVIDIA H100-80GB GPUs (1 GPU for 8B models and 4 GPUs for 70B/106B-scale models). The second component is a lightweight simulator that runs on any laptop with Python 3.10 and uses included pre-profiled acceptance traces to reproduce the paper's upper-bound and combined-proposer analyses. Evaluators can validate the artifact by running the profiling scripts to regenerate the main end-to-end results for Llama3.1-8B, or/and by running the simulator to reproduce the oracle upper-bound analyses. Expected outputs are PDF figures whose trends closely match the first row of Fig. 1 for Llama3.1-8B, Fig. 8, Fig. 17, Fig. 10 and Fig. 19.

B.2 Artifact check-list (meta-information)

- **Algorithm:** N-gram, EAGLE, EAGLE-3, Draft Model, MTP speculative decoding; simulation of speculative decoding performance.
- **Program:** (1) Python 3.10, vLLM (built from source per branch). (2) Python 3.10, `tqdm`, `pandas`, `matplotlib`.
- **Compilation:** (1) `uv pip install -e .` via `rebuild_env.sh`; CUDA 12.8. (2) `pip install -r requirements.txt`; no GPU required.
- **Data set:** (1) InstructCoder, CNN/DailyMail, GSM8K, AIME, GPQA (HuggingFace, auto-downloaded); ShareGPT (one-time download, see §B.4). (2) Pre-profiled acceptance data included in `simulator/data/`.
- **Run-time environment:** (1) Linux, CUDA 12.8, conda. (2) Any OS with Python 3.10.
- **Hardware:** (1) 1x-4x H100-80GB GPU (2) No GPU required.
- **Run-time state:** (1) No other process should occupy the target GPU(s). (2) N/A.
- **Execution:** (1) `run-l3-8b.sh` for Llama3.1-8B end-to-end performance profiling. (2) `./simulate_and_plot.sh`.
- **Metrics:** (1) Throughput speedup, acceptance rate, time breakdown x; (2) simulated speedup vs. batch size.
- **Output:** (1) `scripts/results/run_<timestamp>/` with `.jsonl` results and logs. (2) `results/{proposer}_{dataset}_speedup.csv` and PDF figures in `figures/llama3.1-8B/`.
- **Experiments:** (1). `run-l3-8b.sh` will take 24-36 hours to complete on 1x H100. (2). `simulate_and_plot.sh` will take 30-40 minutes to finish.
- **How much disk space required (approximately)?:** recommended to be in 500GB of free disk space.
- **How much time is needed to prepare workflow (approximately)?:** (1) ≈30-40 min for each environment building. (2) <5 min.
- **How much time is needed to complete experiments (approximately)?:** See Experiments above.
- **Publicly available?:** Yes.
- **Data licenses (if publicly available)?:** See HuggingFace model/dataset cards if applicable.

B.3 Description

B.3.1 How to access

(1) vLLM suite: https://github.com/SpecDecode-Bench/vllm; see `README.md` for more information. (2) Simulator: https://github.com/SpecDecode-Bench/simulator; see `README.md` for more information.

B.3.2 Hardware dependencies

(1) Profiling: 1x H100-80GB GPU for 8B models; 4x H100-80GB GPUs for larger models. (2) Simulation: No GPU required; any machine with Python 3.10 suffices.

B.3.3 Software dependencies

(1) `conda`, Python 3.10, `uv` (managed by `rebuild_env.sh`), `matplotlib`; HuggingFace access token for Llama-3 gated models. (2) Python 3.10, `tqdm`, `pandas`, `matplotlib` (installed via `pip install -r requirements.txt`).

B.3.4 Datasets

(1) All datasets except ShareGPT are downloaded automatically via HuggingFace `datasets`. ShareGPT requires a one-time download (see §B.4). (2) Pre-profiled JSONL acceptance data for Llama-3.1-8B-Instruct on H100 is included in `simulator/data/llama3.1-8B/` (gsm8k, instructcoder, sharegpt, cnn).

B.4 Installation

(1) vLLM suite: ShareGPT (one-time):

```
huggingface-cli download  \
anon8231489123/ShareGPT_Vicuna_unfiltered \
    ShareGPT_V3_unfiltered_cleaned_split.json \
  --repo-type dataset --local-dir /path/to/data/
export SHAREGPT_PATH=/path/to/dataset
```

(1) vLLM suite: environment (repeat per branch):

```
git checkout <branch>
ENV_DIR=/path/to/envs bash scripts/rebuild_env.sh
conda activate /path/to/envs
```

(2) Simulator:

```
cd simulator/
conda create -n specbench_simulator python=3.10 -y
conda activate specbench_simulator
pip install -r requirements.txt
```

B.5 Experiment workflow

(1) vLLM profiling. Set `CUDA_VISIBLE_DEVICES` at the top of the target script, then from `scripts/`:

```
export SHAREGPT_PATH=/path/to/sharegpt.json
bash run-l3-8b.sh    # 1 GPU
```

Each script runs warmup, all datasets $\times$ methods, and calls `vis_speedup.py` to produce PDF figures in `results/run_<timestamp>/figures/`.

(2) Simulator.

```
cd simulator/
./simulate_and_plot.sh
```

This runs EAGLE-3, N-gram, and combined proposers across all four datasets and saves CSV results to `results/` and PDF figures to `figures/llama3.1-8B/`.

B.6 Evaluation and expected result

(1) vLLM profiling. After running `run-l3-8b.sh`, we expect the same end-to-end performance results to be reproduced for Figure 1 (a)–(d).

(2) Simulator. We expect the same results to be reproduced for Figure 9 and Figure 11.

B.7 Experiment customization

(1) vLLM profiling. Set `num_reqs=5` and `batch_sizes="1 16"` for a $<$10 min smoke test (use a non-reasoning script). Individual datasets and methods can be toggled by editing the `for` loops.

(2) Simulator. Individual proposers, datasets, prediction methods, and batch sizes can be configured via CLI flags to `main.py`. Refer to the README on GitHub for more information.

B.8 Notes

For profiling, each script run writes to a fresh timestamped directory. Each branch requires its own virtual environment.