# SSSD: Simply-Scalable Speculative Decoding

**Michele Marzollo[1], Jiawei Zhuang[1], Niklas Roemer[1,2], Niklas Zwingenberger[1], Lorenz K. Muller[1], Lukas Cavigelli[1]**

[1]Huawei, [2]ETH Zurich

**Correspondence:** michele.marzollo@huawei.com

## Abstract

Speculative decoding has emerged as a popular technique for accelerating inference in Large Language Models. However, most existing approaches yield only modest improvements in production serving systems. Methods that achieve substantial speedups typically rely on an additional trained draft model or auxiliary model components, increasing deployment and maintenance complexity. This added complexity reduces flexibility, particularly when serving workloads shift to tasks, domains, or languages that are not well represented in the draft model's training data.

We introduce Simply-Scalable Speculative Decoding (SSSD), a training-free method that combines lightweight n-gram matching with hardware-aware speculation. Relative to standard autoregressive decoding, SSSD reduces latency by up to 2.9×. It achieves performance on par with leading training-based approaches across a broad range of benchmarks, while requiring substantially lower adoption effort—no data preparation, training or tuning are needed—and exhibiting superior robustness under language and domain shift, as well as in long-context settings.

## 1 Introduction

Large Language Models (LLMs) have become ubiquitous across a wide variety of applications, yet their high memory and compute requirements continue to pose substantial challenges for delivering low-latency, cost-efficient user experiences. LLM inference consists of two main phases: the prefill phase, where the input sequence is processed in parallel to compute and store the KV-cache and generate the first token, and the autoregressive decoding phase, where the model generates one token at a time through a full pass over the model weights (Yuan et al., 2024).

Speculative decoding (SD) (Xia et al., 2023; Leviathan et al., 2023; Chen et al., 2023a) breaks the autoregressive loop by using a smaller model to "guess" future tokens (the *drafting* phase), which are then verified in a single forward pass of the target model. If some guesses are correct, a single iteration produces multiple tokens and the overall latency is reduced while maintaining the exact output distribution of the original model. In practice, real-world adoption has remained limited: suitable drafting models are often unavailable and the cost of training a well-aligned draft model can be prohibitive (Li et al., 2024b). Despite the emergence of numerous SD variants, many approaches fail to deliver speedups at practical batch sizes, while introducing substantial implementation and deployment complexity. In production systems, where ease of integration and cost efficiency matter as much as raw latency improvements, these limitations often outweigh the gains.

In this paper, we present a new approach explicitly designed for practical deployment. Our method uses a lightweight, custom-built n-gram model that runs entirely on CPU. It keeps adoption simple by avoiding the training or deployment of any additional models, enabling simpler integration into existing systems and better generalization to new domains, tasks, or languages without manual intervention or additional engineering effort. The source code is available at `https://github.com/huawei-csl/sssd_speculator`.

## 2 Background and Related Work

Since the introduction of SD, numerous variants have been proposed. While a detailed overview of all existing methods is beyond the scope of this work, Xia et al. (2024) provide an extensive survey of the field.

### 2.1 Main speculative decoding methods

The standard approach to SD uses a smaller, faster draft model to autoregressively generate candidate



*Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33123–33139

tokens for verification. For this approach to be effective, the draft model must be closely aligned with the target model, as generating candidates incurs nontrivial computational cost. Typically, effective draft models are drawn from the same model family as the target model, sharing training data and recipes. However, such models are often unavailable, and training new ones can be prohibitively expensive. To address this limitation, Spector and Re (2023) propose tree-based speculation, which generates multiple possible continuations rather than a single sequence, thereby increasing the number of accepted tokens, as well as staged speculation, which applies speculative decoding recursively to the draft model itself. Medusa (Cai et al., 2024) replaces the draft model with multiple decoding heads attached to the target model's final layer. These heads can be trained through self-distillation using substantially less data, preserving alignment with the target distribution while reducing training costs. Similarly, EAGLE (Li et al., 2024b,a) trains a single autoregressive head to generate candidate trees. EAGLE-3 (Li et al., 2025) further advances this method by leveraging hidden states from multiple layers of the target model as input for the speculative head, achieving state-of-the-art inference speedups.

All these methods require training draft models or heads, deploying them on accelerators—which can be challenging in distributed settings (Chen et al., 2023a)—and retraining whenever the target model or application domain changes (Hong et al., 2025). Multi-token prediction (MTP) (Gloeckle et al., 2024) introduces additional heads into the target model at training time, modifying the classical next-token prediction (NTP) training objective. Although MTP is primarily designed to improve model quality, these auxiliary heads can also be leveraged for SD. Nevertheless, aside from a few notable examples (DeepSeek-AI et al., 2025), this paradigm has not yet seen widespread adoption.

### 2.2 Multilinguality

Prior work has shown that the speculation accuracy of draft models varies substantially across languages. Sandler et al. (2025) propose a fine-tuning strategy to mitigate speedup disparities across underrepresented tasks and languages. Yi et al. (2024) observe that small draft models, such as Medusa and EAGLE, perform well in English but generalize poorly to other languages due to their limited capacity, and propose training language-specific drafters. While these approaches effectively address important limitations of speculative decoding, they do so at the cost of increased deployment and maintenance complexity, which further motivates a training-free, self-adaptable alternative.

### 2.3 Model-free methods

Model-free approaches eliminate the need for additional trainable parameters for drafting and focus instead on lightweight, easily deployable alternatives. Some approaches leverage the target model itself to generate candidate tokens, such as by skipping intermediate layers during the drafting phase (Zhang et al., 2024b) or by generating a pool of n-grams for use in subsequent speculative iterations (Fu et al., 2024). While simplifying deployment, these approaches come with major computational costs for the drafting phase, which overshadow the speedups given by accepting more tokens. Other model-free methods generate candidate tokens through token-level pattern matching or external retrieval, avoiding reliance on the model itself. Prompt Lookup Decoding (PLD) (Saxena, 2023), LLMA (Yang et al., 2023), and Lookahead (Zhao et al., 2024b) look for recent matches in the prompt or across previous generations to propose likely continuations. REST (He et al., 2024) and ANPD (Ou et al., 2024) use large external corpora to statistically estimate probable next tokens.

These retrieval-based methods are attractive due to their ease of deployment; however, this simplicity comes at a cost: they produce candidate tokens with lower acceptance rates and often achieve good speedups only on specific tasks.

### 2.4 Large-scale speculative decoding

Most work on SD focuses on accelerating inference for single-prompt scenarios. However, in practical deployments, batching is the primary strategy for reducing cost, which often outweighs latency concerns. While batching limits the effectiveness of speculative techniques due to shared resource constraints (Su et al., 2023), it remains central to LLM inference systems, which typically employ continuous batching (Yu et al., 2022) to optimize resource utilization and user experience. Liu et al. (2025) show that, in this context, standard autoregressive decoding can outperform SD at moderate request rates due to the overhead of running the draft model. In contrast, lightweight methods like Prompt Lookup Decoding perform well at high request rates in tasks where simple retrieval

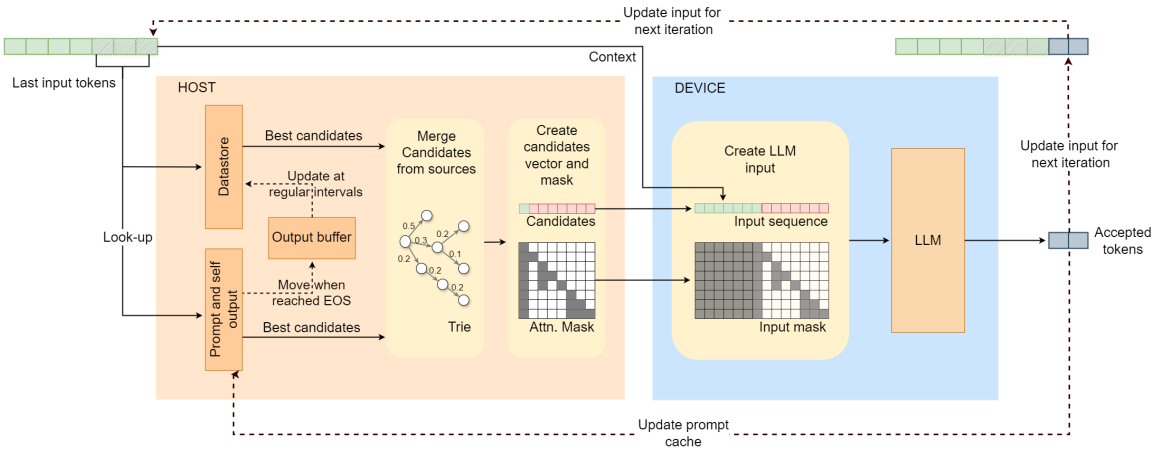



Figure 1: A representation of the system with the main steps of the SSSD method.

is effective. Nevertheless, a number of model-based approaches report consistent speedups even in batched settings (Miao et al., 2024; Zhong et al., 2025; Li et al., 2025; Zhang et al., 2024a), in some cases up to batch size 128 in long-context regimes (Sadhukhan et al., 2025).

### Contributions

1. We propose SSSD, an SD method that requires no data preparation, training, or fine-tuning;
2. We evaluate and compare the end-to-end latency–throughput trade-off in SGLang;
3. We demonstrate SSSD's suitability for task, domain, and language adaptation, including in cold-start settings;
4. We provide insights into parameter selection, including datastore size, data sources, and hardware-aware speculation length adjustments.

## 3 Method

Designing an efficient and scalable speculative decoding method involves several critical challenges. Most existing work focuses on prediction quality, but two additional aspects are crucial: standard approaches introduce substantial overhead, as the cost of drafting can offset the benefits of correct predictions (Liu et al., 2025), and the verification phase must be optimized to make effective use of hardware resources.

### 3.1 Algorithm design

The first objective of SSSD is to remove the complexity and cost of running the drafting phase on-device (where *device* refers to the GPU or any alternative accelerator). Although skipping model identification, modification, training, and deployment is clearly attractive, existing parameter-free approaches suffer from limited speculation quality.

The core idea of our method is to treat the prompt and self-output as a unified n-gram source and to integrate it with a large text datastore. This datastore can be built from any available corpus or populated automatically during serving by storing outputs from past conversations. Our experiments show that the two sources provide complementary candidates, and that combining them substantially improves draft-token quality. Figure 1 illustrates the process: the final tokens of the input sequence (the draft *prefix*) are matched against both sources, and the continuations of the matches are used to choose the candidates for tree-based verification, as described in the following sections.

#### 3.1.1 Prompt and self-output

We treat the prompt and the model's previously generated tokens (*self-output*) as a single sequence, which we refer to as the *input*. This input is stored in a trie-like structure that is continuously updated as new tokens are generated, allowing efficient n-gram retrieval.

To select candidate continuations for a prefix of length $P$ (we use $P = 4$), we examine the continuations of all prefix lengths from 1 to $P$ (see Algorithm 1 in the Appendix). Each of these trees of continuations will be considered when choosing the best tokens to verify.

#### 3.1.2 Datastore

The large datastore is constructed as in REST (He et al., 2024), using a set of suffix arrays over tokenized text to efficiently retrieve continuations of a given prefix. Unlike REST, our datastore can be

continuously updated with the model's own outputs. This provides two advantages: (1) the data is aligned with the model's output distribution, improving speculation quality, and (2) the system can start from an empty datastore, eliminating any need for precomputation. We also modify the datastore query algorithm to improve speculation quality and reduce retrieval time.

For each prefix, we locate its continuations using binary search on each suffix array and then sample a small, fixed number of them at regular intervals. Because the suffix arrays are sorted, interval sampling approximates the underlying distribution. We then build a weighted tree of these continuations, where edge weights represent occurrence counts. If too few continuations are found, we progressively shorten the prefix and repeat the search until reaching a minimum threshold. Continuations are merged at each iteration by summing edge weights (see Algorithm 2 for details).

Lookahead (Zhao et al., 2024b) also combines prompt information with continually updated statistics of the model's output, stored in a tree with aggressive pruning to control memory and retrieval latency. However, pruning degrades candidate quality: rare sequences, though infrequent, collectively account for many useful n-grams. In our experiments, all pruning strategies harmed acceptance rates. Accordingly, for the live datastore we limit size by discarding the oldest entries, avoiding internal pruning. By combining a suffix-array design with an efficient sampling strategy, our approach accommodates significantly more data than previous speculative methods, resulting in higher speculation accuracy while maintaining predictable retrieval cost.

### 3.1.3 Datastore management

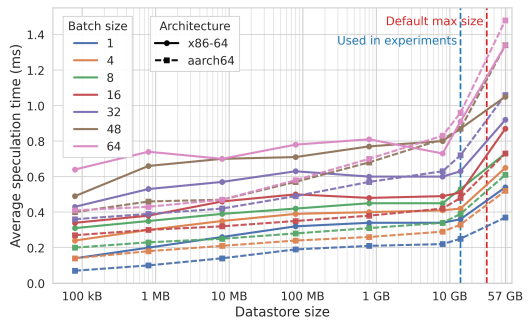



Figure 2: Retrieval cost at increasing datastore sizes and varying batch sizes on x86-64 and aarch64 hosts. The speculation length is fixed at 16 candidates, which is higher than what is typically used at large batch sizes.

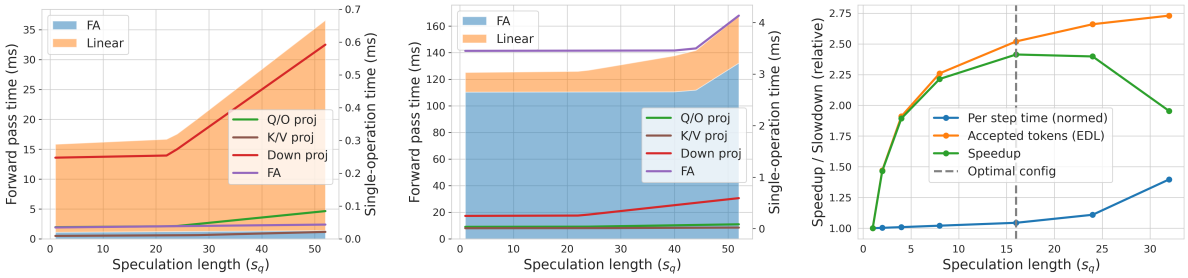
As discussed above, the datastore update policy operates over multiple suffix arrays (*sub-indices*) rather than a single index. In the default configuration, each sub-index stores 512M tokens, with 4 bytes for both the value and the index per token (≈4 GB per sub-index). Retrieval samples a fixed number of continuations (100 by default), independent of datastore size, such that adding more sub-indices reduces the number of samples drawn from each. The number of sub-indices is capped at 8 (typically 4 in our experiments). In the live-datastore setting, when incoming data exceeds this capacity, the oldest sub-index is discarded and replaced with one built from recent data. This design keeps speculation cost bounded and preserves prediction quality by adapting to distribution shifts.

Rebuilding proceeds at regular intervals. If the accumulated data does not fill a sub-index, the last sub-index is rebuilt using its previous content combined with the new data. Otherwise, the last sub-index is filled, a new one is created, and, if the total capacity is exceeded, the oldest is removed once the new one is ready. Rebuilding takes several minutes but runs asynchronously and does not affect retrieval latency; memory pooling avoids allocation and deallocation overhead.

Figure 2 shows that logarithmic search over the suffix arrays keeps retrieval latency low even for datastores larger than those typically deployed in high-throughput settings. Despite substantial differences in raw CPU performance (x86-64: 96 cores / 192 threads @ 2.4 GHz; aarch64: 192 cores / 192 threads @ 2.6 GHz), retrieval cost remains low and comparable across both architectures.

### 3.1.4 Fusing multiple data sources

After retrieving candidates from the two sources, we must combine them to produce high-quality next-token proposals. Each source provides an estimate of the likelihood that a candidate will be accepted during verification, which we refer to as its *probability*. This pseudo-probability is computed as the number of occurrences of the full sequence (prefix plus candidate tokens) divided by the number of occurrences of the prefix alone. However, these estimates differ in reliability: probabilities derived from the input are typically overconfident compared to those from the datastore. To correct this, we apply a scaling factor to input-based probabilities. Within the input itself, we further dampen probabilities based on the length of the prefix match and the candidate's depth in the tree. We select




(a) Context length: 1000 (b) Context length: 100,000 (c) Context length: 1000

Figure 3: (a,b) Roofline-based forward-pass time vs. speculation length, with contributions from linear operations and FlashAttention for Llama-3.1-8B at batch size 8 (same hardware as in Figure 6a). Curves correspond to single operations from Table 2. (c) Accepted tokens, normalized cost, and theoretical speedup vs. speculation length.

these weights once via a simple grid search, and they transfer well across models and tasks.

Appendix A provides pseudocode for building the candidate trees and for merging data from multiple sources. From the merged tree, we then construct the candidate vector for verification and the corresponding attention mask.

### 3.2 Hardware-aware speculation

Having designed an algorithm that reduces drafting costs and complexity while preserving high speculation accuracy, the final optimization focus is on minimizing the latency overhead of the verification forward pass of the last generated token and the speculated tokens. While many speculative decoding studies highlight the underuse of accelerator resources as an important factor enabling SD, few explore how memory bandwidth and compute resources interact during SD, especially under batching. We propose a simple rule to dynamically choose a near-optimal speculation length based on the current batch size, significantly simplifying serving with SD.

We define the speculation length $s_q$ as the total number of nodes in the speculation tree, including the root (the latest accepted token) and the candidate tokens to be verified. We denote batch size as $b$ and group size for GQA (Ainslie et al., 2023) as $g$. Figure 3c illustrates the interaction between speculation length and forward pass time in determining end-to-end acceleration. The orange line represents the average number of tokens accepted by the LLM as a function of $s_q$. The blue line shows the corresponding relative increase in forward step time compared to standard autoregressive decoding ($s_q = 1$). Neglecting the drafting time overhead, which remains low and near-constant in our lookup-based approach, the optimal speculation length is the one maximizing the ratio between the algorithmic acceleration (average accepted tokens) and the relative increase in per-step latency.

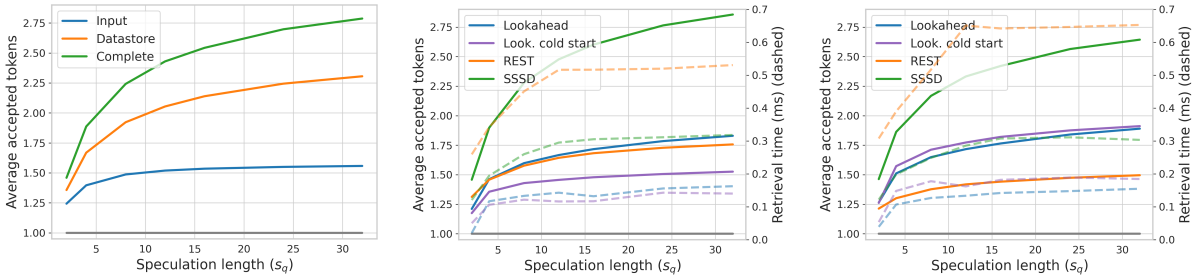
Appendix B presents a detailed analysis of the primary device-side operations during a decoding step; here we provide only a simplified overview. Figure 3a,b illustrates the cost of a forward pass, where the orange region corresponds to the linear layers and the blue region to the attention layers. In plot (a), where the context is short, the forward-pass cost is dominated by the linear layers. Their cost remains nearly constant as $s_q$ increases, until the computation reaches the compute-bound region of the roofline model at approximately $s_q \approx 22$. Up to this point, speculating 22 tokens per batch element is almost free. In Figure 3b, where the context is long, the dominant cost becomes loading the KV-cache in the attention layers. Although the linear layers follow the same trend as in the short-context case, their relative contribution is smaller, allowing even larger "free" speculation budgets.

As described in the appendix, all operations except attention exhibit a FLOPs-to-IO ratio of approximately $b \times s_q$. Following the roofline performance model (Williams et al., 2009), to operate near the roofline ridge point (i.e., the transition from memory- to compute-bound execution), we set

$$s_q = \frac{I_{\text{knee}}}{b},$$

where $I_{\text{knee}}$ denotes the arithmetic intensity at the ridge point, defined as the ratio of the accelerator's peak FLOPS to its peak memory bandwidth.

For small batch sizes, the speculation budget implied by this rule can become impractically large for efficient kernel execution, while longer speculation lengths yield diminishing returns. We there-




(a) Decomposed components (b) MT-Bench (c) GSM8k

Figure 4: (a) Speculation quality of SSSD data sources on 160 MT-Bench and GSM8K prompts. (b,c) Comparison with parameter-free baselines. REST and SSSD use the same datastore; the Lookahead cache is evaluated both warm (identical data as SSSD) and cold. Solid lines show accepted tokens, dashed lines show candidate retrieval and mask construction time. All experiments use Llama-3.1-8B.

fore cap the speculation length to fixed values (e.g., $s_q = 32$ for $b = 1$). This also handles the case where, for $g > b$, the attention operator becomes compute-bound at smaller $s_q$ than the linear layers.

## 4 Evaluation

To evaluate SSSD in a realistic inference setting, we integrate it into SGLang (Zheng et al., 2024), a high-performance LLM inference system. SGLang provides an efficient implementation of EAGLE-2, EAGLE-3 and Lookahead. We further integrate PLD and REST to enable comparisons with leading model-free methods.

Our main experiments focus on models from the Llama-3 family (Grattafiori et al., 2024), which are widely used in speculative decoding research. For other model families, our choice fell on those with publicly available EAGLE-3 heads.

For EAGLE and Lookahead, we use SGLang's default parameters for speculation length and tree structure. For other methods, we follow Section 3.2. Using SSSD-tuned parameters for EAGLE methods often degrades performance, and exhaustive tuning is computationally challenging and provided limited benefit in preliminary experiments.

### 4.1 Algorithm effectiveness

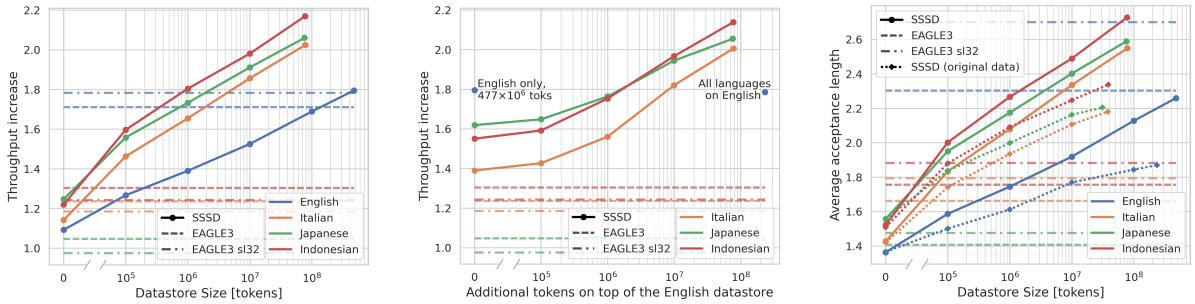
First, we show why our method is particularly effective by breaking down the different components that contribute to selecting candidates for speculation. The method's strength lies in the complementarity of candidates coming from the input and the datastore. This can be seen in Figure 4a, where the contribution of the two main components (the input, which includes prompt and self-output, and the datastore) almost adds up perfectly for large enough $s_q$, meaning that the algorithm balances candidates from the two sources effectively.

We also compare our implementation with the main alternative parameter-free methods, both in terms of speculation quality and retrieval time (see Figure 4b,c). We use the MT-Bench dataset used in (He et al., 2024) (the first prompt of each conversation), and the first 80 prompts from the GSM8k dataset (Cobbe et al., 2021), used in (Zhao et al., 2024b), showing that our method consistently outperforms both methods on both datasets. For SSSD and REST, we construct datastores using publicly available data generated either by the same model or models of the same family following the Magpie procedure (Xu et al., 2025), and we use the same data to simulate the Lookahead warm-start scenario.

### 4.2 Datastore cold-start and multilinguality

As emphasized earlier, the key strengths of our method lie in its ease of adoption and flexibility across new domains and languages. Accordingly, we investigate its behavior when applied directly to an inference system in a cold-start setting (without a pre-built datastore), examine the differences between using model-generated data and general-domain language data, and evaluate how rapidly SSSD adapts to new languages.

To answer these questions, we use Qwen3-14B (Yang et al., 2025), as it offers strong native multilingual support. We evaluate end-to-end speedups as a function of the total number of accumulated tokens in the datastore. Data are generated by the model using prompts from multilingual sources (Xu et al., 2024; ShareGPT, 2023; Chen et al., 2023b; Zhao et al., 2024a). This setup simulates a deployment scenario in which user queries ar-




(a) Cold start (b) Starting from English datastore (c) Acceptance lengths (cold start)

Figure 5: Experiments with Qwen3-14B on a 48 GB GPU (165.2 TFLOPS in bfloat16; 1008 GB/s memory bandwidth) on MT-Bench (and translations). Temperature = 0.7, top-p = 0.8, top-k = 20; averages over five runs. (a, b) Speedup over autoregressive decoding at batch size 1 versus datastore size (≈1,000 tokens per user conversation on average). (c) Corresponding acceptance length, comparing model-generated and dataset-derived entries.

rive continuously and model outputs are incrementally stored in the datastore. We measure speedups across a typologically diverse set of languages, chosen to span differences in script, morphological typology, and syntactic structure, including English and three lower-resource languages. We consider two initialization settings: an empty datastore and a large English-only datastore, allowing us to assess both within-language scaling and cross-lingual adaptation.

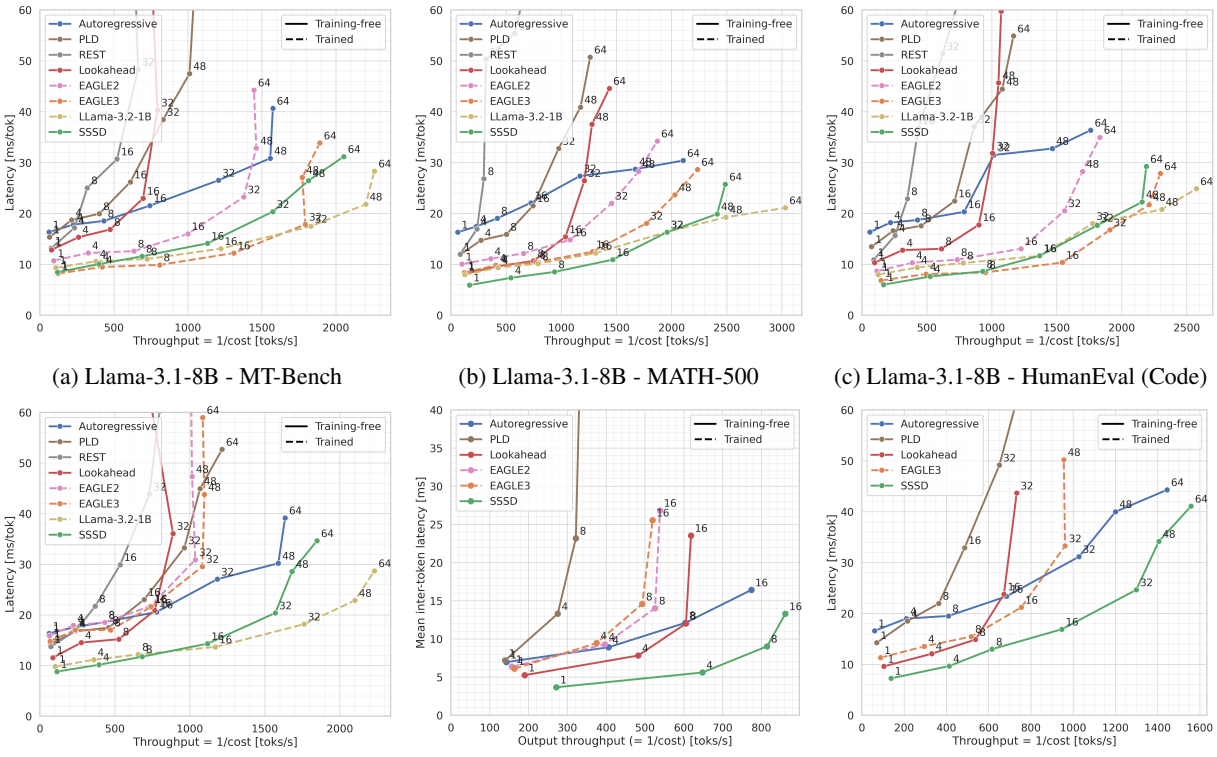
Our results show that incorporating multiple languages has a negligible negative impact: English performance remains essentially unchanged even when the datastore contains large amounts of non-English data. The method adapts rapidly: after only 1,000 conversations, speedups already reach 1.6×–1.8× for lower-resource languages and approximately 1.4× for English. Importantly, even in a cold-start setting, the model achieves a 1.1×–1.23× speedup, enabling immediate deployment without initial performance degradation. Starting from an English datastore when speculating new languages gives an initial benefit, and tends to converge to the behavior of the monolingual datastore when enough data is generated. Increasing the amount of data is consistently beneficial, yielding approximately logarithmic gains. With sufficient data, SSSD outperforms EAGLE-3 across all evaluated languages, with especially pronounced improvements for lower-resource languages.

In Figure 5c, we additionally compare a datastore populated with the model's own generated answers to one built from the original dataset answers, which follow a different distribution. We observe up to a 35% increase in correct token predictions when using model-generated data.

The larger speedups observed for non-English languages are encouraging, and we observe the same pattern on the other tested models as well. With current tokenizers, equivalent content often requires fewer tokens in English than in many other languages, making English generation cheaper and faster (Petrov et al., 2023; Ahia et al., 2023). From this perspective, the larger gains of SSSD for other languages help mitigate tokenization-driven disparities and make generation latency more uniform across languages. We do not find clear evidence that morphological typology or other broad linguistic properties are major drivers of cross-lingual variation in speculation quality. We hypothesize that this variation is driven primarily by language-specific factors and their interaction with tokenization, although a more systematic investigation is left to future work.

### 4.3 End-to-end evaluations with batching

For the experiments on Llama-3.1-8B, we use the same datastore for SSSD and REST, starting from the same data as in Section 4.1. To test cross-lingual adaptation, we add a small amount of German data (not generated by the model), consisting of translations of ShareGPT (Chen et al., 2023b) and Synthia (Meyer and Nagler, 2021) samples. We also add a subset of The Stack (Kocetkov et al., 2023) to improve performance on coding benchmarks. It is worth recalling that if all data were generated directly from the target model, speculation accuracy would be higher. The resulting datastore has a total size of 14 GB. We use the same datastore for all evaluated datasets, highlighting SSSD's ability to retrieve the relevant information for each task when the datastore contains hetero-




(a) Llama-3.1-8B - MT-Bench (b) Llama-3.1-8B - MATH-500 (c) Llama-3.1-8B - HumanEval (Code)

(d) Llama-3.1-8B - MT-Bench German (e) Llama-3.1-8B - SWE-Bench (f) DS-R1-Dist-Llama-8B - MATH-500

Figure 6: Evaluation of speculation methods on 8B models. (a–d) Llama-3.1-8B on a single GPU (same as in Figure 5) across multiple datasets. (e) Llama-3.1-8B in disaggregated prefill–decode mode on 4 GPUs (1 for decoding, 3 for prefill), each with 80 GB of VRAM, 989.5 TFLOPS, and 3.35 TB/s memory bandwidth. (f) DeepSeek-R1-Distill-Llama-8B on MATH-500 using the same hardware as in (a), averaged over five runs.

geneous text. For Lookahead, we do not perform cache warm-up, as it is not supported in SGLang and does not consistently yield performance improvements (see Figure 4c).

Results are shown in Figure 6, while results for Llama-3.3-70B are reported in Appendix C. We present them in a latency–throughput plot to illustrate the relationship between these two key metrics across batch sizes. We omit time-to-first-token (TTFT), as it is mostly determined by the prefill stage and is therefore not affected by SSSD. In contrast to model-based methods that require constructing (and potentially transmitting) a draft KV-cache, SSSD builds the input tree in a background thread during prefill and requires no additional KV-cache.

When running on a single device, we operate in offline mode (all prompts are available at the start). We measure the output-token throughput (output tokens/total time) and compute the average latency (batch size/throughput). The total time includes the prefill stage, which SD does not accelerate. Across all datasets and batch sizes, we find that SSSD consistently outperforms every n-gram method and EAGLE-2. It also surpasses EAGLE-3 on MATH-500 (Hendrycks et al., 2021) and on German-language questions. For general short-context conversational queries (MT-Bench (Zheng et al., 2023)) and short-context coding tasks (HumanEval (Chen et al., 2021)), EAGLE-3 maintains a slight advantage. Nonetheless, the performance gap remains small despite the simplicity of SSSD compared to EAGLE.

We also report standard speculative decoding results using Llama-3.2-1B as the draft model. Trained under the same training regime as the target model on 9 trillion tokens with 916k GPU hours (Meta, 2024), it closely matches the target's output distribution and achieves the highest acceptance rates. While informative, this setting is impractical when similarly aligned draft models are unavailable; nevertheless, SSSD achieves comparable end-to-end speedups.

### 4.3.1 Long context

Long-context generation is increasingly important for LLM applications such as agentic AI and retrieval-augmented generation (RAG) (Lewis et al., 2020). In this regime, SSSD has two key

advantages over model-based speculative decoding methods. First, SSSD eliminates the need for a KV cache. Although draft models and heads are smaller than the target model, their computational cost still grows with context length. In contrast, SSSD maintains an approximately constant drafting cost as the context length increases. Second, SSSD's speculation quality remains stable independently of input size. This is because the context used for n-gram lookups consists of only a few tokens, whereas speculative models require long-context training to avoid accuracy degradation.

Speculative decoding is particularly beneficial in the long-context regime due to the memory-bandwidth bottleneck of attention layers (Section 3.2). While current SGLang overheads limit achievable gains in practice, SSSD still outperforms alternative methods on PG-19 (Rae et al., 2020) (Table 1). On agentic workloads (Figures 6e and 7f), SSSD achieves up to 1.9× throughput and latency improvements over autoregressive decoding and 1.66× over EAGLE-3.

| | Llama-3.1-8B | Llama-3.3-70B |
|---|---|---|
| Lookahead | 1.08× | 1.15× |
| EAGLE-3 | 0.80× | 1.09× |
| SSSD | **1.23×** | **1.26×** |

Table 1: Speedup over autoregressive decoding in long-context summarization on PG-19 (up to 40k tokens), using PD disaggregation.

### 4.4 Reasoning and speculative sampling

Another important recent application of LLMs is reasoning models: in order to answer more complex questions the model is prompted (Wei et al., 2022) or trained (Guo et al., 2025) to produce a reasoning trace before giving a final answer. This substantially increases generation length, making speculative decoding particularly important.

We evaluate one of the distilled DeepSeek-R1 models (Guo et al., 2025) on a subset of medium- and hard-difficulty questions from MATH-500, allowing generation of up to 4096 tokens. The datastore is constructed using publicly available reasoning traces from larger models within the same family, resulting in an 18 GB datastore. We follow the recommended generation settings (temperature = 0.6, top-p = 0.95), demonstrating that our method extends beyond greedy decoding. We adopt a simplified speculative sampling procedure (Leviathan et al., 2023), as implemented in SGLang, which preserves the target distribution without requiring draft probability estimates.

Results are shown in Figure 6f: SSSD achieves a 2.29× speedup over autoregressive decoding at batch size 1 and a 1.32× speedup over the strongest alternative method. Notably, it is the only method that achieves speedups at large batch sizes.

## 5 Conclusion

This work argues for a re-centering of speculative decoding around deployability rather than speculation quality alone. SSSD demonstrates that strong and reliable acceleration can be achieved without auxiliary training, specialized draft models, or task-specific tuning, showing state-of-the-art performance under distribution shift, multilingual inputs, and long-context workloads. By framing inference acceleration as a joint algorithm–system problem, SSSD highlights a path toward simpler and more scalable decoding strategies that better align with the operational constraints of modern LLM serving.

## Limitations

In Mixture of Experts (MoE) models (Jiang et al., 2024), increasing speculation length activates more experts and thus requires loading more of them from memory, unless all of them are already loaded due to a large batch size. In this case, SSSD is affected similarly to other speculative decoding methods, yielding smaller gains for MoE models. Speedups are also limited for models using MLA attention (DeepSeek-AI et al., 2024), which is typically run using the efficient, but compute-bound, FlashMLA kernels, which are less compatible for speculative decoding than standard MHA/GQA. In such scenarios, methods that prioritize higher speculation accuracy—at higher computational cost—may be more effective, as achievable speculation depth is inherently constrained.

Additionally, the drafting phase of SSSD depends on the CPU performance and requires some host memory capacity, which may limit its applicability in some deployment environments.

## Ethical considerations

Speculative decoding methods can introduce uneven performance across languages, tasks, and user populations. Prior work has shown that training-based draft models often generalize poorly beyond the domains and languages represented in

their training data, leading to substantially lower speculation accuracy–and consequently lower inference speedups–for underrepresented languages and tasks. In production systems, such disparities translate into higher latency and cost for some users, effectively creating unequal access to efficient LLM services.

SSSD exhibits substantially more uniform speedups across languages, including lower-resource languages, without requiring language-specific training, tuning, or maintenance. While this does not address broader issues of linguistic coverage or bias in the underlying language model, it reduces a specific source of disparity introduced by existing inference acceleration techniques.

Finally, as with other retrieval-based methods, using prior model outputs as a source of speculative candidates raises privacy considerations. In practice, SSSD operates on transient data already maintained by many serving systems, preserves the target model's output distribution, and does not introduce additional information leakage in generated outputs.

## References


Orevaoghene Ahia, Sachin Kumar, Hila Gonen, Jungo Kasai, David Mortensen, Noah Smith, and Yulia Tsvetkov. 2023. Do All Languages Cost the Same? Tokenization in the Era of Commercial Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 9904–9923, Singapore. Association for Computational Linguistics.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4895–4901, Singapore. Association for Computational Linguistics.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads. In *Forty-first International Conference on Machine Learning*.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023a. Accelerating Large Language Model Decoding with Speculative Sampling. *Preprint*, arXiv:2302.01318.

Lequn Chen. 2023. Dissecting Batching Effects in GPT Inference. `https://le.qun.ch/en/blog/2023/05/13/transformer-batching/`. Blog post.

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

Zhihong Chen, Shuo Yan, Juhao Liang, Feng Jiang, Xiangbo Wu, Fei Yu, Guiming Hardy Chen, Junying Chen, Hongbo Zhang, Li Jianquan, Wan Xiang, and Benyou Wang. 2023b. MultilingualSIFT: Multilingual Supervised Instruction Fine-tuning.

Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. 2024. Sequoia: Scalable and Robust Speculative Decoding. In *Advances in Neural Information Processing Systems*, volume 37, pages 129531–129563. Curran Associates, Inc.

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

Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *The Twelfth International Conference on Learning Representations*.

Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*.

DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, and 138 others. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. *Preprint*, arXiv:2405.04434.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, and 181 others. 2025. DeepSeek-V3 Technical Report. *Preprint*, arXiv:2412.19437.

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. 2024. Break the sequential dependency of LLM inference using LOOKAHEAD DECODING. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriel Synnaeve. 2024.

Better & faster large language models via multi-token prediction. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The Llama 3 Herd of Models. *Preprint*, arXiv:2407.21783.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, and 175 others. 2025. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638.

Zhenyu He, Zexuan Zhong, Tianle Cai, Jason Lee, and Di He. 2024. REST: Retrieval-Based Speculative Decoding. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 1582–1595, Mexico City, Mexico. Association for Computational Linguistics.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the MATH dataset. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Fenglu Hong, Ravi Shanker Raju, Jonathan Lingjie Li, Bo Li, Urmish Thakker, Avinash Ravichandran, Swayambhoo Jain, and Changran Hu. 2025. Training Domain Draft Models for Speculative Decoding: Best Practices and Insights. In *First Workshop on Scalable Optimization for Efficient and Adaptive Foundation Models*.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, and 7 others. 2024. Mixtral of Experts. *Preprint*, arXiv:2401.04088.

Denis Kocetkov, Raymond Li, Loubna Ben allal, Jia LI, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro Von Werra, and Harm de Vries. 2023. The Stack: 3 TB of permissively licensed source code. *Transactions on Machine Learning Research*.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NeurIPS '20, Red Hook, NY, USA. Curran Associates Inc.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024a. EAGLE-2: Faster inference of language models with dynamic draft trees. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 7421–7432, Miami, Florida, USA. Association for Computational Linguistics.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024b. EAGLE: speculative sampling requires rethinking feature uncertainty. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2025. EAGLE-3: Scaling up inference acceleration of large language models via training-time test. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Xiaoxuan Liu, Jongseok Park, Langxiang Hu, Woosuk Kwon, Zhuohan Li, Chen Zhang, Kuntai Du, Xiangxi Mo, Kaichao You, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. 2025. TurboSpec: Closed-loop Speculation Control System for Optimizing LLM Serving Goodput. *Preprint*, arXiv:2406.14066.

Udi Manber and Gene Myers. 1990. Suffix arrays: a new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, page 319–327, USA. Society for Industrial and Applied Mathematics.

Meta. 2024. Llama-3.2-1B Model Information. Accessed: 2025-01-05.

David Meyer and Thomas Nagler. 2021. Synthia: multidimensional synthetic data generation in python. *Journal of Open Source Software*, 6(65):2863.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page

932–949, New York, NY, USA. Association for Computing Machinery.

Jie Ou, Yueming Chen, and Prof. Tian. 2024. Lossless Acceleration of Large Language Model via Adaptive N-gram Parallel Decoding. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, pages 10–22, Mexico City, Mexico. Association for Computational Linguistics.

Aleksandar Petrov, Emanuele La Malfa, Philip H.S. Torr, and Adel Bibi. 2023. Language model tokenizers introduce unfairness between languages. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NeurIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. 2020. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations*.

Cody Rivera, Jieyang Chen, Nan Xiong, Jing Zhang, Shuaiwen Leon Song, and Dingwen Tao. 2021. TSM2X: High-performance tall-and-skinny matrix–matrix multiplication on GPUs. *Journal of Parallel and Distributed Computing*, 151:70–85.

Ranajoy Sadhukhan, Jian Chen, Zhuoming Chen, Vashisth Tiwari, Ruihang Lai, Jinyuan Shi, Ian En-Hsu Yen, Avner May, Tianqi Chen, and Beidi Chen. 2025. MagicDec: Breaking the Latency-Throughput Tradeoff for Long Context Generation with Speculative Decoding. In *The Thirteenth International Conference on Learning Representations*.

Jameson Sandler, Ahmet Üstün, Marco Romanelli, Sara Hooker, and Ferdinando Fioretto. 2025. The disparate impacts of speculative decoding. *Preprint*, arXiv:2510.02128.

Apoorv Saxena. 2023. Prompt Lookup Decoding.

ShareGPT. 2023. ShareGPT Vicuna Unfiltered. Accessed: 2026-01-01.

Benjamin Frederick Spector and Christopher Re. 2023. Accelerating LLM Inference with Staged Speculative Decoding. In *Workshop on Efficient Systems for Foundation Models @ ICML2023*.

Qidong Su, Christina Giannoula, and Gennady Pekhimenko. 2023. The Synergy of Speculative Decoding and Batching in Serving Large Language Models. *Preprint*, arXiv:2310.18813.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NeurIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76.

Heming Xia, Tao Ge, Peiyi Wang, Si-Qing Chen, Furu Wei, and Zhifang Sui. 2023. Speculative Decoding: Exploiting Speculative Execution for Accelerating Seq2seq Generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3909–3925, Singapore. Association for Computational Linguistics.

Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. 2024. Unlocking Efficiency in Large Language Model Inference: A Comprehensive Survey of Speculative Decoding. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 7655–7671, Bangkok, Thailand and virtual meeting. Association for Computational Linguistics.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. WizardLM: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*.

Zhangchen Xu, Fengqing Jiang, Luyao Niu, Yuntian Deng, Radha Poovendran, Yejin Choi, and Bill Yuchen Lin. 2025. Magpie: Alignment data synthesis from scratch by prompting aligned LLMs with nothing. In *The Thirteenth International Conference on Learning Representations*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Nan Yang, Tao Ge, Liang Wang, Binxing Jiao, Daxin Jiang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023. Inference with Reference: Lossless Acceleration of Large Language Models. *Preprint*, arXiv:2304.04487.

Euiin Yi, Taehyeon Kim, Hongseok Jeung, Du-Seong Chang, and Se-Young Yun. 2024. Towards fast multilingual LLM inference: Speculative decoding and specialized drafters. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 10789–10802, Miami, Florida, USA. Association for Computational Linguistics.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA. USENIX Association.

Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, Yan Yan, Beidi Chen, Guangyu Sun, and Kurt Keutzer. 2024. LLM Inference Unveiled: Survey and Roofline Model Insights. *Preprint*, arXiv:2402.16363.

Aonan Zhang, Chong Wang, Yi Wang, Xuanyu Zhang, and Yunfei Cheng. 2024a. Recurrent Drafter for Fast Speculative Decoding in Large Language Models. *arXiv:2403.09919*.

Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. 2024b. Draft & Verify: Lossless Large Language Model Acceleration via Self-Speculative Decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11263–11282, Bangkok, Thailand. Association for Computational Linguistics.

Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. 2024a. Wildchat: 1m chatgpt interaction logs in the wild. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Yao Zhao, Zhitian Xie, Chen Liang, Chenyi Zhuang, and Jinjie Gu. 2024b. Lookahead: An Inference Acceleration Framework for Large Language Model with Lossless Generation Accuracy. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '24, page 6344–6355. Association for Computing Machinery.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-bench and Chatbot Arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NeurIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NeurIPS '24, Red Hook, NY, USA. Curran Associates Inc.

Shuzhang Zhong, Zebin Yang, Ruihao Gong, Runsheng Wang, Ru Huang, and Meng Li. 2025. ProPD: Dynamic Token Tree Pruning and Generation for LLM Parallel Decoding. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '24, New York, NY, USA. Association for Computing Machinery.

# A Algorithms

This section describes the main algorithms used for candidate retrieval. The algorithms presented here are not exact replicas of the implementation but are logically equivalent; minor differences are introduced to improve clarity. We omit pseudocode for constructing the underlying data structures and denote the cardinality of data structures (e.g., the total number of nodes in a tree or the number of elements in a list) by $|\cdot|$.

## A.1 Input tree

To retrieve candidates from the prompt and self-output, we maintain a trie data structure whose nodes correspond to token prefixes. When a sequence of tokens is inserted into the trie, the count of every node along the corresponding path is incremented. Consequently, each node stores the number of times its associated prefix has been observed.

Based on these counts, each node is associated with probability values conditioned on different prefixes, denoted as node.*prob*. For a prefix represented by a node $u$ and a longer sequence represented by a descendant node $v$, this probability is defined as

$$\text{node.}prob = \frac{\text{count}(v)}{\text{count}(u)}.$$

This quantity represents the empirical probability that the prefix corresponding to $u$ is extended to the sequence corresponding to $v$; the extension need not be immediate.

When retrieving continuations for multiple prefixes of varying lengths from the input trie, we return references to the nodes corresponding to the ends of those prefixes. Due to the recursive structure of the trie, each such node implicitly defines a complete candidate tree, namely the subtree rooted at that node.

The input trie, together with its node counts, is constructed incrementally and reused across forward passes for the same prompt. At each iteration, the trees of candidates from the input are simply subtrees of the input trie. In contrast, the datastore tree is recomputed at each decoding step based on the candidate continuations identified by Algorithm 2.

## A.2 Datastore

As explained in Section 3.1.3, the datastore consists of multiple suffix arrays (capped at most at 8). The suffix arrays follow a standard implementation (Manber and Myers, 1990): given a prefix,

**Algorithm 1** Retrieving candidates from the input tree for a given prefix. The tree typically has a maximum depth below 10.

```
function MATCHINPUT(prefix)
    Input: T: Tree of prompt + current output
    results ← [ ]
    iter ← 0
    while iter < |prefix| do
        p ← prefix[iter..]                    ▷ sub-prefix
        node ← T.ROOT
        i ← 0
        while i < |p| and p[i]∈ node.children do
            node ← node.children[p[i]]
            i ← i + 1
        end while
        if i = |p| then                       ▷ sub-prefix found
            results.APPEND(⟨node, |p|⟩)
        end if
        iter ← iter+1
    end while
    return results
end function
```

**Algorithm 2** Retrieving candidate continuations for a given prefix from a suffix array.

```
function GETCANDS(prefix, branch_len)
    tree ← TREE(root_id←prefix[|prefix| − 1])
    iter ← 0
    while iter < |prefix| − 1 and |tree| < 50 do
        prefix ← prefix[iter..]
        conts ← index.BINARYSEARCH(
                              prefix, branch_len)
        step ← max(1, ⌊|conts| / 100⌋)
        i ← 0
        while i < |conts| do
            cont ← conts[i]
            tree.INSERT(cont)
            i ← i + step
        end while
        iter ← iter+1
    end while
    return tree
end function
```

**Algorithm 3** Merging candidate tokens coming from different sources. The method INSERTTOK on the draft tree takes a token and a parent node, adds the token as a child of the parent if it does not exist yet, and returns the child node.

```
Input: prefix (vector of integers), dec_len (max
final number of candidates), branch_len (max
depth of each draft)
Output: draft_tree (the merged tokens)
input_trees ← input_cache.MATCHINPUT(
                              prefix)
datastore_tree ← datastore.GETCANDS(
                              prefix, branch_len)
pq ← ∅                                  ▷ priority queue
draft_tree ← TREE(root_id←prefix[|prefix|−1])
all_trees ← [⟨datastore_tree, 4⟩] ‖ input_trees
for each ⟨tree, match⟩ ∈ all_trees do
    if tree ≠ ∅ then
        node ← tree.ROOT
        for each child ∈ node.children do
            if tree = datastore_tree then
                discount ← 1
            else
                discount ← 0.6
            end if
            child_disc ← 0.6 + 0.1 · match
            prob ← node.prob · discount
            pq.PUSH(prob, ⟨node←child,
                    child_disc←child_disc,
                    parent←draft_tree.ROOT⟩)
        end for
    end if
end for
while pq ≠ ∅ and |draft_tree| < dec_len do
    prob, el ← pq.POP()
    new_candidate ← draft_tree.INSERTTOK(
            el.node.token_id, el.parent)
    for each child ∈ el.node.children do
        p ← child.prob · prob · el.child_disc
        pq.PUSH(p, ⟨node←child,
                child_disc←el.child_disc
                parent←new_candidate⟩)
    end for
end while
return draft_tree
```

its possible continuations are obtained by performing two binary searches to locate the first and last lexicographic occurrences of the prefix. Let $P$ be the prefix length and $N$ the number of tokens in a sub-index. For each sub-index, the cost of the two binary searches is $O(P \log N)$. The total search cost increases linearly with the number of sub-indices, which is capped and small.

The remaining work (merging continuations and selecting candidates) is independent of datastore size because the number of sampled continuations

is fixed. In practice, larger datastores often reduce retries (e.g., fewer fallbacks to $(P = 3)$), partially offsetting the cost increase.

The construction of the tree of candidates coming from the datastore is straightforward: inserting a selected continuation consists of adding any missing nodes along its path (each initialized with a count of 1) and incrementing the counts of all nodes that already exist along that path.

### A.3 Merging input and datastore candidates

After assigning weights to the candidate sources as described in Section 3.1.4, we select the final draft tokens using a priority queue, as detailed in Algorithm 3. The cost of this step is independent of the input and datastore size.

## B Impact of the key variables of the decoding phase

This section extends the analysis on the interactions between speculative decoding and hardware utilization, in particular under batching. Larger batch sizes lead to resource competition between batching and speculation, limiting SD speedup. Some studies (Su et al., 2023; Zhong et al., 2025) observe a linear relationship between forward pass overhead and speculation length, with steeper slopes at higher batch sizes. However, this empirical trend doesn't align with theoretical hardware predictions, likely due to unoptimized operator implementations. Sequoia (Chen et al., 2024) shows empirically that the cost of the forward pass remains almost constant up to $s_q < N$, where $N$ depends on the hardware and the model. Understanding these interactions in detail is crucial for deploying SD in an inference system: it enables quantifying the potential gains of SD and it simplifies determining optimal speculation parameters at deployment, relying solely on hardware features.

Table 2 shows the compute and memory costs of the main operations of an LLM, following Chen (2023). The equations are based on the Llama family (Grattafiori et al., 2024), but most dense models have similar architectures. We integrate the speculation length ($s_q$) into FLOP and memory usage formulas for different transformer model components. When $s_q = 1$, the forward pass matches standard autoregressive decoding.

Increasing the batch size $b$ nearly proportionally increases the compute-to-memory access ratio for all terms except FlashAttention. Since memory accesses for matrix multiplications only increases slightly with $b$, the batch size can be scaled up with minimal extra costs until reaching the compute-bound region of the roofline model (Williams et al., 2009). Similarly, increasing the speculation length $s_q$ almost proportionally increases the compute-to-memory access ratio for all matrix multiplications, including FlashAttention. Here as well, memory accesses only slightly rise with $s_q$, so this term can be increased with negligible cost until reaching the compute-bound threshold. Most importantly, for all matrix multiplications other than FlashAttention, the batch size and the speculation length have equivalent effects, always appearing as $b \times s_q$. Therefore, the theoretical free speculation budget can be computed from the hardware specifications by pushing $b \times s_q$ to the roofline model's compute-bound threshold. For instance, for the computation of the $Q$ matrix, assuming that there is no tensor parallelism, the speculation budget can be derived by solving

$$\frac{1}{(1/h + 1/bs_q)} = \frac{\text{Peak TFLOPS}}{\text{Memory Bandwidth (TB/s)}}.$$

If we consider the Llama-3-8B (Grattafiori et al., 2024) model ($h = 4096$) and GPU with a measured computing power of 165 TFLOP/s and a measured memory bandwidth of 0.95 TB/s, the (almost) free speculation budget can be theoretically obtained until $bs_q \approx 174$. This limit ($s_q = 22$ for $b = 8$) appears as a discontinuity in the slope of the green and purple lines in Figure 3a, where the initial segment reflects memory-bandwidth limitations, and the latter segment is determined by the available FLOPS. In practice, the available operators often under-utilize the hardware, making the empirical gains smaller than what can be seen in Figure 3. This is because of the complexity involved in creating efficient kernels, in particular for matrix multiplications where one matrix is "tall and skinny" (Rivera et al., 2021).

Finally, increasing the context length $s_{kv}$ has the main effect of increasing the FlashAttention's time. As a result, other matrix multiplications, unaffected by $s_{kv}$, have a lower overall weight in the forward pass (see Figure 3b). In this scenario, the cost of loading the KV-cache comprises the highest share, and is less impacted by a longer speculation length, unlike large matrix multiplications: the FLOPs:IO ratio ($\approx gs_q$), where $g$ denotes the group size in grouped-query attention (GQA), i.e., the number of

Table 2: Summary of FLOPs and memory IO for a speculative forward pass. We use $b$ for batch size, $s_q$ for speculation length, $s_{kv}$ for KV-cache length, $n_q$ and $n_{kv}$ for query and key/value heads, $g = n_q/n_{kv}$, $d$ for per-head dimension, $h$ for the model's hidden dimension ($h = n_q d$), $h_{mlp}$ for the MLP intermediate dimension, and $p$ for tensor parallelism. We assume fused scaled-dot-product attention (Dao et al., 2022; Dao, 2024). Input, local KV-cache, and mask tensors have shapes $X$: $(b, s_q, h)$, $K_{\text{cache}}, V_{\text{cache}}$: $(b, n_{kv}/p, s_{kv}, d)$, and $M$: $(b, s_q, s_{kv} + s_q)$. Per-rank local weight shapes are $W_q$: $(h, h/p)$, $W_k, W_v$: $(h, h/(gp))$, $W_o$: $(h/p, h)$, $W_1$: $(h, 2h_{mlp}/p)$, and $W_2$: $(h_{mlp}/p, h)$, with $W_1$ denoting the concatenated SwiGLU input projections. The remaining tensor shapes are $Q$: $(b, n_q/p, s_q, d)$, $K_{\text{new}}, V_{\text{new}}$: $(b, n_{kv}/p, s_q, d)$, $K, V$: $(b, n_{kv}/p, s_{kv} + s_q, d)$, $A_{out}$: $(b, n_q/p, s_q, d)$, $Y, Z$: $(b, s_q, h)$, and $H$: $(b, s_q, h_{mlp}/p)$. Memory IO is in bytes, with 2 bytes per weight, activation, and KV-cache element, and 1 byte per mask element. We assume identical $s_{kv}$ across sequences, count only GEMM FLOPs in attention, and exclude tensor-parallel communication.

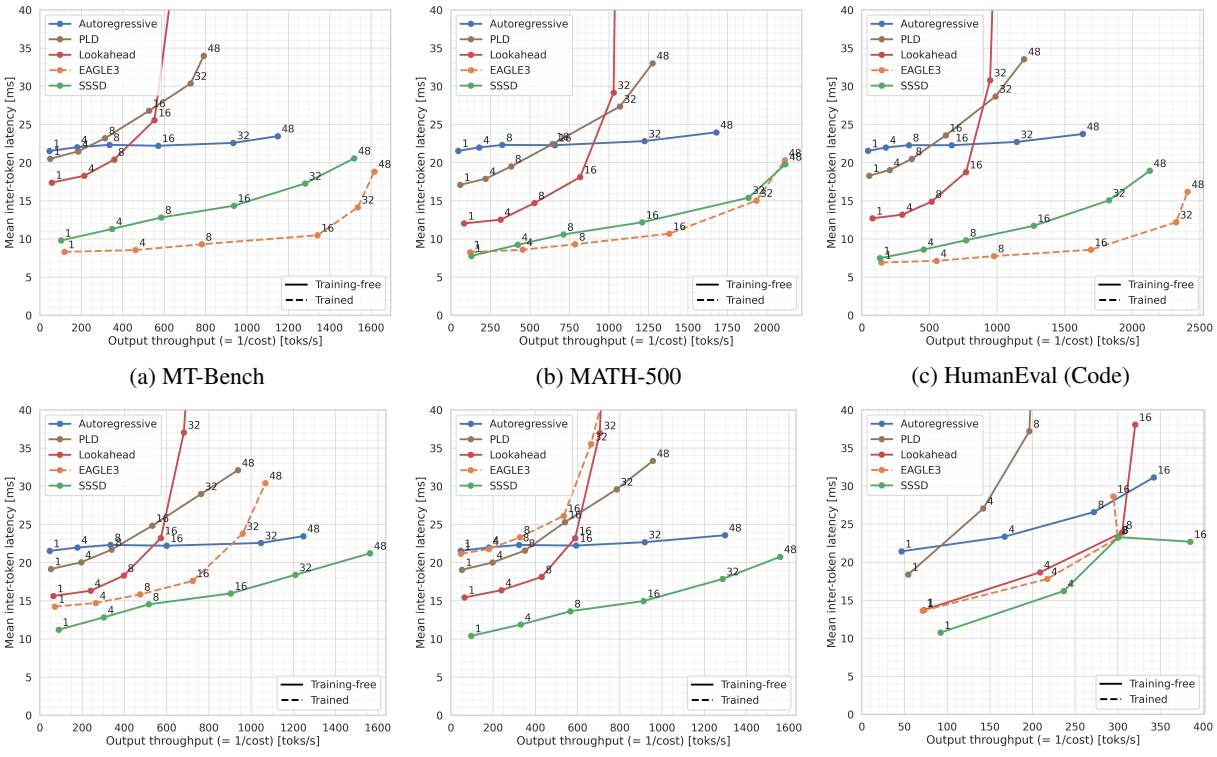
| Op | FLOPs | Read (bytes) | Write (bytes) | FLOPs:IO | FLOPs:IO appr. |
|---|---|---|---|---|---|
| $Q = XW_q$ | $\frac{2bs_qh^2}{p}$ | $2\left(bs_qh + \frac{h^2}{p}\right)$ | $2\frac{bs_qh}{p}$ | $\frac{1}{\frac{p+1}{h} + \frac{1}{bs_q}}$ | $\mathbf{bs_q}$ if $h \gg (p+1)bs_q$ |
| $K_{\text{new}} = XW_k$ | $\frac{2bs_qh^2}{gp}$ | $2\left(bs_qh + \frac{h^2}{gp}\right)$ | $2\frac{bs_qh}{gp}$ | $\frac{1}{\frac{gp+1}{h} + \frac{1}{bs_q}}$ | $\mathbf{bs_q}$ if $h \gg (gp+1)bs_q$ |
| $V_{\text{new}} = XW_v$ | same as $K_{\text{new}}$ | same | same | same | same |
| $A_{out} = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}} + M\right)V$ | $\frac{4bs_q(s_{kv}+s_q)h}{p}$ | $2\left(\frac{bh}{gp}(2(s_{kv}+s_q) + gs_q)\right) + bs_q(s_{kv}+s_q)$ | $2\frac{bs_qh}{p}$ | $\frac{1}{\frac{1}{gs_q} + \frac{1}{s_{kv}+s_q} + \frac{p}{4h}}$ | $\mathbf{gs_q}$ if $h \gg pgs_q/4$, $s_{kv}+s_q \gg gs_q$ |
| $Y = A_{out}W_o$ | $\frac{2bs_qh^2}{p}$ | $2\left(\frac{bs_qh}{p} + \frac{h^2}{p}\right)$ | $2bs_qh$ | $\frac{1}{\frac{p+1}{h} + \frac{1}{bs_q}}$ | $\mathbf{bs_q}$ if $h \gg (p+1)bs_q$ |
| $H = \text{SwiGLU}(YW_1)$ | $\frac{4bs_qhh_{mlp}}{p}$ | $2\left(bs_qh + 2\frac{hh_{mlp}}{p}\right)$ | $2\frac{bs_qh_{mlp}}{p}$ | $\frac{1}{\frac{1}{2h} + \frac{p}{2h_{mlp}} + \frac{1}{bs_q}}$ | $\mathbf{bs_q}$ if $h \gg bs_q/2$, $h_{mlp} \gg pbs_q/2$ |
| $Z = HW_2$ | $\frac{2bs_qhh_{mlp}}{p}$ | $2\left(\frac{bs_qh_{mlp}}{p} + \frac{hh_{mlp}}{p}\right)$ | $2bs_qh$ | $\frac{1}{\frac{1}{h} + \frac{p}{h_{mlp}} + \frac{1}{bs_q}}$ | $\mathbf{bs_q}$ if $h \gg bs_q$, $h_{mlp} \gg pbs_q$ |

query heads sharing a single key–value head, does not include the batch size, making the transition from memory-bandwidth bound to compute bound happen at much higher $s_q$ compared to the other operations if $g = 1$. Otherwise, for $b < g$, the attention layer gets in the compute-bound region for shorter speculation lengths. Depending on the dominating cost between linear layers and attention, the "free" compute budget can be given by the FLOPs:IO ratio divided by the batch size (for short context), or by the group size (for long context). In Figure 3b, you can see that for very long context, although the orange region turns steeper already at $s_q = 22$, the overall increase in cost is negligible until the attention becomes compute bound. Note that in Figure 3b the batch size $b = 8$ is larger than the group size $g = 4$.

For the tensor-parallel (TP) setting, we ignore communication costs, which are difficult to model accurately. Although this is a coarse simplification, our empirical evaluations show that the simplified speculation length selection remains effective in practice. We leave a more detailed treatment of communication costs to future work.

## C Large model results

We additionally report results on a larger model to demonstrate that our method effectively accelerates inference even for large-scale models that require tensor parallelism. To approximate a more realistic deployment setting, we employ prefill/decode disaggregation. For short-context tasks, we use tensor parallelism with TP = 2 for both prefill and decode. For long-context tasks, we introduce three data-parallel replicas for prefill (also with TP = 2), while the decode node remains unreplicated. We further enforce a substantially larger total batch size on the prefill workers than on the decode worker, simulating a scenario in which the decode worker never stalls after receiving the initial request. To




(a) MT-Bench (b) MATH-500 (c) HumanEval (Code)

(d) MT-Bench French (e) MT-Bench Russian (f) SWE-Bench

Figure 7: Evaluation of SD methods on Llama-3.3-70B with prefill-decode disaggregation. Short context uses 2 GPUs for prefill and 2 for decode; long context uses DP=3 for prefill (6 GPUs). Prefill is overprovisioned to keep the decode node busy, though at large batch sizes throughput can be limited by prefill waiting, especially for SD methods. All GPUs: 141 GB VRAM; 989.5 TFLOPS; 4.8 TB/s. All experiments run in bfloat16 with temperature 0.

maximize throughput, we disable chunked prefill. We report performance by plotting average inter-token latency against output token throughput.

All experiments are conducted on Llama-3.3-70B. For the SSSD datastore, we use several publicly available datasets of model-generated data, resulting in a total datastore size of 13 GB. One of these datasets contains a small amount of French and Russian data; we therefore select these two languages to evaluate multilingual performance.

Our method achieves strong results across all datasets, showing consistent speedups at any batch size. On math and coding tasks, we observe speedups of 2.8× and 2.9× at batch size 1, respectively. Even in the long-context setting and for languages with limited data coverage, the speedup remains at least 2×.

Finally, we observe that on English data, for this model, EAGLE-3 outperforms SSSD. This behavior is primarily attributable to the exceptionally high acceptance rate of EAGLE-3 on this model, together with the relatively smaller size of its head compared to the base model. Despite the exceptionally performant EAGLE-3 head, SSSD demonstrates stronger adaptability to new languages and better performance in long-context scenarios.