

# Spexis: Speculative and Lookahead Scheduling for LLM Inference

Hyungyu Jung<sup>1</sup>, Jaehyeok Yu<sup>1</sup>, Hoonseo Choi<sup>2</sup>, Sungkyun Kim<sup>2</sup>, Jinho Lee<sup>1</sup>, Jiwon Seo<sup>1</sup>

<sup>1</sup>Seoul National University, <sup>2</sup>Hanyang University

## Abstract

Spexis is a multi-GPU LLM inference framework that improves the efficiency of pipeline and tensor parallelism through speculative parallelism. Rather than using speculative decoding only to accelerate token generation, Spexis runs speculation in parallel with normal execution, introducing a new parallelism axis without increasing KV-cache memory usage. This improves memory efficiency and helps mitigate the bottlenecks of multi-GPU inference.

Spexis further uses lookahead scheduling to predict speculation quality and future memory pressure, allowing it to reduce wasted speculation, KV-cache eviction, and recomputation. Built on top of vLLM, Spexis largely improves serving performance across a range of GPU configurations, achieving speedups of up to 34% over a baseline that uses the optimal combination of pipeline and tensor parallelism. Spexis’s source code is publicly available at <https://github.com/mlsys-seo/spexis>.

## 1 Introduction

LLMs are compute-intensive and are thus typically served on GPUs or other hardware accelerators. Recent models have also grown significantly in parameter size and KV cache memory demand, requiring multiple GPUs for their execution.

For multi-GPU inference, model parallelism is widely used, notably tensor parallelism (TP) and pipeline parallelism (PP). However, both face performance bottlenecks: TP incurs synchronization overhead, while PP suffers from reduced batch sizes. The problem becomes worse as more GPUs are used. In PP, adding more pipeline stages increases memory pressure from micro-batches, while in TP, using more GPUs incurs higher communication overhead.

We propose Spexis, a scheduling technique that extends speculative decoding for scalable LLM in-

ference. Spexis introduces speculative parallelism as a new scheduling primitive and executes speculation in parallel with normal execution. This design provides additional parallelism without increasing KV cache memory usage, making multi-GPU execution more memory-efficient.

To further improve memory efficiency, Spexis uses prediction-based scheduling to estimate future memory pressure and determine when to run new requests. This helps reduce eviction and recomputation overhead while improving batch efficiency.

Our evaluation shows that Spexis achieves up to a 34% speedup. Our contributions are as follows.

**Speculative-parallel scheduling.** We introduce speculative-parallel (SP) scheduling, a new parallel execution axis that complements pipeline and tensor parallelism. We further improve it with selective speculation and remaining-length prediction.

**Analytic cost model.** We develop cost models for PP, TP, and SP to efficiently combine these methods. We also develop an eviction-aware model that captures recomputation costs to determine when to schedule incoming requests for higher throughput.

**Implementation and evaluation.** We implement Spexis on top of vLLM, a widely used LLM inference system. Across NVIDIA A40, RTX PRO 6000, H100, A100 and L40S GPUs, Spexis achieves speedups of up to 34%.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 presents the motivation for our work. Section 4 introduces Spexis and speculative parallelism. Section 5 describes our lookahead scheduling optimization. Section 6 presents our prototype implementation. Section 7 evaluates Spexis and Section 8 concludes.

## 2 Related Work

**Scheduling for single-node LLM inference.** The unique execution characteristics of LLMs (particularly the distinct prefill and decode phases and

---

**Correspondence:** Jiwon Seo <[seojiwon@snu.ac.kr](mailto:seojiwon@snu.ac.kr)>

highly variable sequence lengths) make it challenging to fully utilize computational resources. Prior work has addressed these challenges, including low decode utilization (Yu et al., 2022), fluctuating prefill overhead (Agrawal et al., 2024), and KV cache memory inefficiency (Kwon et al., 2023). While these approaches significantly improve performance in single-node deployments, they do not address the additional bottlenecks on distributing LLM inference across multiple GPUs or nodes.

### Parallelism strategy for distributed inference.

When an LLM exceeds the memory capacity of a single GPU, it must be distributed across multiple GPUs. Tensor parallelism (TP) (Shoeybi et al., 2020) partitions the weight matrices of each Transformer layer across GPUs for concurrent execution. In multi-node deployments, pipeline parallelism (PP) (Huang et al., 2019) is commonly used to partition model layers into multiple stages. Because TP and PP suffer from scalability limitations due to synchronization overhead and memory bottlenecks, respectively, hybrid PP+TP configurations have become standard in practice (vLLM Team, 2025). However, it still does not fully eliminate the bottlenecks of PP and TP. Spexis mitigates these bottlenecks with its memory-efficient speculative-parallel execution.

**Scheduling with length prediction.** In LLMs, KV cache memory grows proportionally with sequence length, yet total available memory is fixed and output length is unknown, making memory-aware scheduling inherently challenging. To mitigate this, prior work has explored generation length prediction to improve scheduling efficiency. S3 (Jin et al., 2023) enables larger batch sizes, LTR and TRAIL (Fu et al., 2024; Shahout et al., 2024) reduce average latency by prioritizing shorter requests in the scheduling queue. In contrast, our work uses predicted length information to reduce KV-cache eviction overhead and improve serving throughput. When the predicted lengths indicate that admitting a new request would exceed memory capacity, its prefill can be deferred. To guide this decision, we develop a cost model that explicitly balances the delay of new requests against the recomputation cost of evicted ones.

**Speculative decoding.** Speculative decoding has been proposed and extensively studied to overcome the autoregressive nature of transformer-based models (Leviathan et al., 2023). In spec-

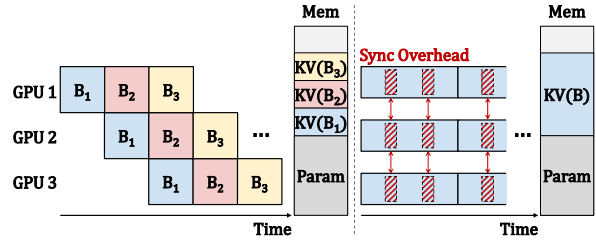


Figure 1: Execution timeline and memory usage of pipeline parallel execution (left) and tensor parallel execution (right) using three GPUs.

ulative decoding, a lightweight draft model predicts multiple tokens ahead, and the base model verifies them in parallel to accelerate generation. Various self-speculative approaches have been explored to eliminate the overhead of maintaining a separate draft model (Zhang et al., 2024; Elhoushi et al., 2024; Liu et al., 2024). For example, Kangaroo (Liu et al., 2024) reuses the early layers of the base model as a draft model while the remaining layers verify the drafted tokens. We adopt a similar self-speculative approach and apply speculative decoding to pipeline-parallel execution, effectively filling the pipeline idle time that causes the bubble problem in PP.

### 3 Motivation: Bottleneck in LLM Serving

LLM serving often requires multiple GPUs due to large parameters and KV cache memory demand. Multi-GPU inference commonly uses pipeline parallelism (PP) or tensor parallelism (TP), but both face scalability bottlenecks: PP from memory pressure and TP from communication overhead.

**Memory capacity bottleneck.** In pipeline-parallel execution, consecutive layers are grouped into stages and assigned across GPUs. To reduce pipeline bubbles caused by inter-stage dependencies, PP uses micro-batching, as illustrated in Figure 1. The main bottleneck in PP is the KV cache memory used by the micro-batches. Because different micro-batches process different sequences, each micro-batch requires its own KV cache entries, as shown in the figure. As a result, the size of each micro-batch must be much smaller than the batch size achievable without micro-batching. These small micro-batch sizes lead to low GPU utilization and, consequently, lower throughput.

**Communication bottleneck.** In tensor-parallel execution, each layer and its weight parameters are partitioned across multiple GPUs. After each partitioned computation, the partial outputs are syn-

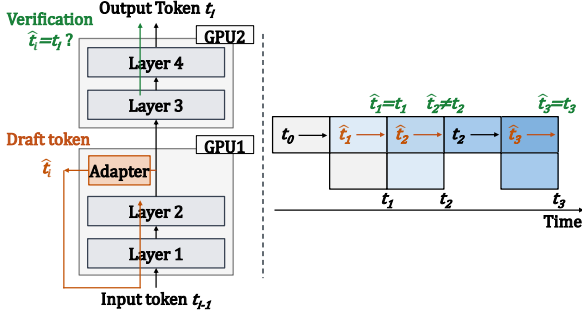


Figure 2: Overview of the self-speculative model and its execution timeline in Spexis under a two-stage pipeline. The timeline shows the execution of a single request: orange arrows indicate speculative execution, and black arrows indicate normal execution after speculation rejection.

chronized across the GPUs. For LLMs, this synchronization is required twice per transformer layer. As shown in Figure 1, GPUs remain idle during these synchronization steps, degrading overall resource utilization. High-bandwidth interconnects such as NVLink can mitigate this overhead, but the bottleneck is not fully eliminated. In particular, within a node, communication bandwidth is often non-uniform across GPU pairs, and all-reduce collectives across the local NVLink-connected GPU domain can still become a bottleneck.

These bottlenecks become worse as the degree of PP and TP increases. As a result, simply increasing model parallelism does not necessarily improve serving throughput and may even reduce it. This motivates a new parallel execution strategy that alleviates both bottlenecks while retaining the scalability benefits of multi-GPU execution.

#### 4 Spexis: Speculative-Parallel Execution

To mitigate the bottlenecks of pipeline and tensor parallelism, we introduce a new scheduling axis, speculative parallelism (SP). By jointly optimizing pipeline parallelism (PP), tensor parallelism (TP), and speculative parallelism (SP), Spexis effectively minimizes these bottlenecks.

**Self-speculation and parallel scheduling.** Self-speculative models use their earlier layers for drafting and the rest layers for verifying drafted tokens as part of their normal execution path. Similar to conventional speculative decoding, self-speculation drafts multiple tokens and then verify them jointly.

In Spexis, we execute speculation in parallel with normal model execution (i.e., verification) while running the entire model in a pipeline- and/or

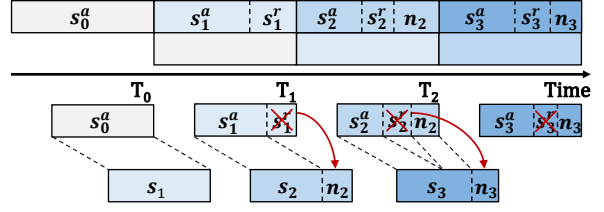


Figure 3: Example speculative-parallel execution over iterations 0–3. Here,  $s_t$  denotes the amount of speculative execution at iteration  $t$ ; the superscripts  $a$  and  $r$  denote accepted and rejected speculation, respectively; and  $n_t$  denotes the amount of normal execution at  $t$ . Below the timeline, we illustrate how tokens accepted at iteration  $i$  are used for speculative execution in iteration  $i+1$  ( $s_i^a \rightarrow s_{i+1}$ ), while rejected tokens are re-executed in iteration  $i+1$  ( $s_i^r \rightarrow n_{i+1}$ , denoted by red arrow).

tensor-parallel manner. As a simple example, Figure 2 shows a self-speculative model and its execution in Spexis using a two-stage pipeline on two GPUs. After generating draft tokens from the first stage, Spexis immediately begins speculative execution for those tokens in the first stage, in parallel with their verification in the second stage.

Because speculative execution does not require a separate KV cache and instead shares the cache with normal execution, it supports larger batch sizes, leading to higher throughput. In comparison, PP’s micro-batching uses separate KV caches for each micro-batch, which incurs the memory-capacity bottleneck discussed in Section 3. Furthermore, when SP is applied, the required degree of TP can be reduced, which in turn lowers its communication overhead.

The effectiveness of SP scheduling largely depends on the acceptance rate of draft tokens. In speculative decoding, the first few draft tokens, especially in self-speculative models, tend to have relatively high acceptance rates. For example, prior work reports a maximum acceptance rate of 85% for the first draft token in EAGLE (Li et al., 2024b), while our evaluation shows a maximum acceptance rate of 78% for the first draft token. These high early-token acceptance rates make SP practical and allow it to be combined effectively with TP and PP to optimize LLM inference.

**Cost model for PP, TP, and SP.** We develop a cost model for the steady-state decoding throughput of TP, PP, and SP, each using  $N$  GPUs. PP and SP both use  $N$  pipeline stages. For simplicity, we assume all sequences have equal length.

Let  $B$  denote the maximum number of sequences that can be processed in a batch under

| PP                   | TP                 | SP                 | SP <sub>α</sub>               |
|----------------------|--------------------|--------------------|-------------------------------|
| $B/N$                | $B$                | $B_{\text{eff}}^*$ | $B_{\text{eff},\alpha}^*$     |
| $T_{\text{pp}}(B/N)$ | $T_{\text{tp}}(B)$ | $T_{\text{sp}}(B)$ | $T_{\text{sp}}(B_{\alpha}^*)$ |

Table 1: Estimation of throughput for PP, TP, SP and SP<sub>α</sub>.  $N$  denotes the number of pipeline stages,  $B$  the maximum batch size, and  $T_*(\cdot)$  the single-stage decoding time for a given batch size. SP<sub>α</sub> incorporates selective speculation described in Section 5.1.

TP. For PP, assuming  $N$  pipeline stages with  $N$  micro-batches, the effective batch size per stage becomes  $B/N$ . For SP, we estimate its effective batch size, defined as the expected number of accepted tokens in a batch, using the recurrence relation shown below and illustrated by the example in Figure 3.

$$\begin{aligned}
s_t^a &= (s_{t-1}^a + n_{t-1})\Theta_{t-1}, \\
s_t^r &= \sum_{i=1}^{N-1} (s_{t-i}^a + n_{t-i})(1 - \Theta_{t-i}), \\
n_t &= (s_{t-N}^a + n_{t-N})(1 - \Theta_{t-N}),
\end{aligned} \quad (1)$$

Here,  $s_t^*$  and  $n_t$  denote the numbers of sequences processed by speculative execution and normal execution, respectively, at iteration  $t$ , where  $s_t^* + n_t = B$ . Similarly,  $s_t^a$  and  $s_t^r$  denote the numbers of accepted and rejected speculative executions, respectively. Let  $\Theta_t$  denote the speculation accuracy at iteration  $t$ . For simplicity, we assume that the speculation accuracy is time-invariant, i.e.,  $\Theta_t = \Theta$ . The effective batch size at iteration  $t$  is therefore  $B_{\text{eff}}(t) = s_t^a + n_t$ , and the steady-state effective batch size  $B_{\text{eff}}^*$  is given as follows.

$$B_{\text{eff}}^* = \lim_{t \rightarrow \infty} B_{\text{eff}}(t) = \frac{B}{N - \Theta(N - 1)}. \quad (2)$$

We next estimate the decoding throughput of PP, TP, and SP using their effective batch sizes, which correspond to the numbers of decoded tokens, together with their execution times. To this end, we obtain the decoding-latency functions  $T_{\text{pp}}(\cdot)$ ,  $T_{\text{tp}}(\cdot)$ , and  $T_{\text{sp}}(\cdot)$  for PP, TP, and SP, respectively, through profiling. Table 1 summarizes the resulting throughput estimates, which we use to determine the optimal combination of PP, TP, and SP.

The estimation captures the memory overhead of PP from micro-batching and the synchronization overhead of TP, as illustrated in Figure 4 (left). The right plot compares estimated and measured throughput. The estimates are highly accurate, with only small errors, mostly due to batch-size variation, supporting its use for selecting the optimal parallel configuration.

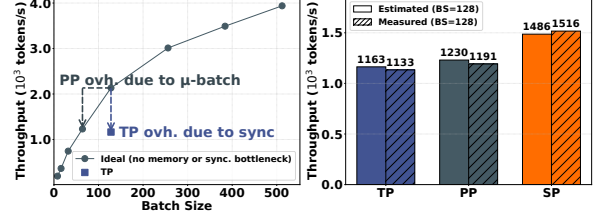


Figure 4: Throughput overheads of PP and TP (left), and comparison of the throughput estimated by our cost model (Table 1) and the measured throughput (right).

## 5 Lookahead Scheduling Optimization

We further improve the efficiency of speculative-parallel execution using two prediction-based techniques. The first predicts the acceptance probability of draft tokens and selects which tokens to speculate on to maximize throughput. The second predicts the remaining output length of each sequence and uses this information to delay the scheduling of incoming requests when necessary, thereby minimizing evictions of running requests.

### 5.1 Acceptance-Guided Selective Speculation

In self-speculative models, the confidence of a draft token is often used to estimate its acceptance probability (Liu et al., 2024; Li et al., 2024a). Spexis also uses this confidence, but rather than explicitly predicting acceptance, it ranks drafted tokens by confidence, discards low-ranked tokens, and speculates only on high-ranked ones.

More specifically, Spexis drops a fixed  $\alpha$  fraction of drafted tokens in a batch and performs speculation on the remaining tokens. Our analysis shows that a small fraction of low-confidence draft tokens in each batch has little chance of being accepted; thus, dropping them largely improves speculation accuracy. To determine  $\alpha$ , we profile the acceptance rate of the retained draft tokens. By jointly considering this acceptance rate and the latency of speculation, we identify the optimal  $\alpha$ . In our evaluation, we used an  $\alpha$  value in the range of 0.15–0.3.

We extend the cost model in the previous section to reflect selective speculation. Let  $\alpha \in [0, 1)$  denote the drop ratio,  $\Theta_\alpha$  the speculation accuracy under  $\alpha$  (with  $\Theta_0 = \Theta$ ), and  $B_t$  the total batch size at iteration  $t$ , including both speculative and normal executions. The steady-state batch size  $B_\alpha^*$  is then computed by the recurrence relation below (for simplicity, and since  $\alpha$  is small, we assume

that dropping affects only speculative execution):

$$B_{t+1} = \alpha B_{t+1-N} + (1 - \alpha) B_t, \quad (3)$$

$$B_\alpha^* = \lim_{t \rightarrow \infty} B_t = \frac{B}{1 + \alpha(N - 1)},$$

The expected effective batch size, counting only accepted tokens from speculative and normal executions, is then computed using Equations 2 and 3:

$$B_{\text{eff},\alpha}^* = \alpha B_\alpha^* + \frac{(1 - \alpha) B_\alpha^*}{N - \Theta_\alpha(N - 1)}. \quad (4)$$

Finally, the throughput of SP with selective speculation is computed as the effective batch size divided by the latency of processing the total batch size, i.e.,  $\frac{B_{\text{eff},\alpha}^*}{T_{\text{sp}}(B_\alpha^*)}$  as shown in Table 1 under  $\text{SP}_\alpha$ .

## 5.2 Remaining-Length-Aware Scheduling

SP is memory-efficient because speculative execution shares its KV cache with the normal execution. It further reduces memory pressure by predicting the remaining lengths of running sequences and using this information to delay the scheduling of new requests. Specifically, we estimate the approximate remaining lengths of running sequences to predict their KV cache usage in upcoming decoding iterations. Based on this estimate, we determine when to schedule new requests to prevent eviction of running sequences and their KV cache entries.

For remaining-length prediction, rather than predicting the exact length, we use ordinal-threshold prediction to estimate whether the remaining length falls below 4, 8, 16, 32, and 64 tokens, or exceeds 64 tokens. To make this prediction, we train a three-layer MLP that takes the hidden state of the draft layer as input and outputs five confidence scores corresponding to the five thresholds (4–64). For each threshold, we choose a confidence cutoff such that, when the model predicts that the remaining length falls below that threshold, the prediction attains high precision (83%, as shown in our evaluation) while maintaining reasonable recall.

With the predicted remaining lengths, we estimate KV cache memory usage over the next 64 decoding iterations and determine whether admitting a new request during that window would require eviction (Algorithm 1 describes this estimation process in detail). We jointly consider the throughput benefit of admitting the new request earlier and the recomputation cost incurred by evicting running sequences. Although the recomputation penalty is often larger in practice, the benefit of admitting a

### Algorithm 1 Look-ahead memory estimation over future iterations

---

**Input:**  $\mathcal{B}$ : set of running requests  $r$   
 $pp$ : pipeline depth;  $\theta$ : speculation accuracy  
 $\mathcal{E}(\cdot)$ : length prediction precision  
 $\ell^{(r)}, \tau^{(r)}$ : current and remaining length of request  $r$   
**Output:**  $\mathcal{M}$ : list of estimated memory usage (in tokens) in next 1..64 iterations  
**Notation:** Per request, let  $w$  count speculation rejections;  
 $t_{i,w} \triangleq i - ppw \triangleright \# \text{ accepted tokens at iteration } i$   
 $k_{i,w} \triangleq t_{i,w} + w \triangleright \# \text{ generated tokens at iteration } i$   
 $p_{i,w} \triangleq \binom{k_{i,w}}{t_{i,w}} \theta^{t_{i,w}} (1 - \theta)^w \triangleright \Pr[\# \text{ generated} = k_{i,w}]$   
 $\mathbf{1}[\cdot]$ : indicator function  $\triangleright 1 \text{ if true, } 0 \text{ otherwise}$

---

```

1:  $\mathcal{M} \leftarrow []$ ,  $L_{\text{total}} \leftarrow \sum_{r \in \mathcal{B}} \ell^{(r)}$ 
2: for all  $i \in \{1, \dots, 64\}$  do
3:    $w_{\text{max}} \leftarrow \lfloor i/pp \rfloor$ 
    $\triangleright \text{expected \# generated tokens per request}$ 
4:    $\mathbb{E}[L_{\text{gen}}] \leftarrow \sum_{w=0}^{w_{\text{max}}} k_{i,w} p_{i,w}$ 
    $\triangleright \text{completion probability of each request } r \in \mathcal{B}$ 
5:    $P_{\text{fin}}^{(r)} \leftarrow \mathcal{E}(\tau^{(r)}) \sum_{w=0}^{w_{\text{max}}} \mathbf{1}[k_{i,w} \geq \tau^{(r)}] p_{i,w}$ 
6:    $\mathbb{E}[N_{\text{fin}}] \leftarrow \sum_{r \in \mathcal{B}} P_{\text{fin}}^{(r)} \triangleright \text{expected \# finished requests}$ 
7:    $\mathbb{E}[L_{\text{fin}}] \leftarrow \sum_{r \in \mathcal{B}} \ell^{(r)} P_{\text{fin}}^{(r)} \triangleright \text{memory they release}$ 
    $\triangleright \text{memory estimate at look-ahead iteration } i$ 
8:    $M_{\text{est}} \leftarrow L_{\text{total}} + (|\mathcal{B}| - \mathbb{E}[N_{\text{fin}}]) \mathbb{E}[L_{\text{gen}}] - \mathbb{E}[L_{\text{fin}}]$ 
9:   Append  $M_{\text{est}}$  to  $\mathcal{M}$ 
10: end for
11: return  $\mathcal{M}$ 
```

---

new request earlier may sometimes outweigh the penalty. Hence we explicitly model this trade-off.

**Analytic model for length-aware scheduling.** We estimate the throughput over the period between two request arrivals. Let  $I$  denote the number of iterations in this period,  $bs$  the number of running decode requests before the new arrival, and  $t_d$  and  $t_p$  the latency of one decode iteration and one prefill, respectively. Then the prefill-to-decode (or recompute-to-decode) ratio is  $pdr := t_p/t_d$ .

We compare two scheduling policies: *aggr* (aggressive), which runs the new request immediately and may later incur eviction and recomputation, and *delay* (ours), which runs the request at the earliest expected iteration that does not trigger eviction.

We consider the case where  $n(>0)$  evictions occur under *aggr* during the period. Let  $i_{\text{evict}}^k$  and  $i_{\text{rec}}^k$  denote the eviction iteration and the subsequent recomputation iteration for the  $k$ -th eviction, respectively. We define  $i_{\text{wait}}^k := i_{\text{rec}}^k - i_{\text{evict}}^k$  as the number of iterations the  $k$ -th evicted sequence remains unscheduled before recomputation. Let  $i_{\text{delay}}$  denote the earliest scheduling iteration under *delay* that

avoids eviction. The throughput ratio, or gain,  $G$ , of *delay* over *aggr* is then computed as follows.

$$G = \frac{I + (n + 1)(pdr - 1)}{I + pdr - 1} \cdot \frac{bs + 1 - \frac{i_{\text{delay}}}{I}}{bs + 1 - \sum_{k=1}^n \frac{i_{\text{wait}}^k}{I}}$$

The details are provided in the appendix. The left term of the product captures the recomputation overhead, while the right term captures the cost of delaying scheduling in *delay* relative to waiting in *aggr*. Since  $bs$  is typically much larger than the subtracted terms, the right term is usually close to one, making the recomputation overhead the main factor in the gain of *delay*. However, if evictions are rare (e.g., only one) and  $i_{\text{delay}} \gg i_{\text{wait}}$ , the gain can be less than one. We account for both effects when scheduling new requests.

## 6 Implementation

We implemented our prototype on vLLM (v0.8.4) and Ray (v2.53.0) by extending GPURunner to add speculative-parallel execution. Our implementation remains compatible with existing optimizations in vLLM and Ray, particularly Ray’s compiled DAG optimization, which reduces RPC overhead. However, because the current DAG implementation exposes stage outputs only after the full model execution completes, we use a separate IPC mechanism to obtain the first-stage output for drafting.

Spexis places an LM head at the first and last pipeline stages. Because the additional LM head can imbalance the pipeline stages, we offload the input embedding, whose parameter size is comparable to that of the LM head, to host memory and perform the lookup on the CPU. This adds only a few hundred microseconds of overhead and has little impact on overall performance.

For the drafting adapter, we follow prior work (Liu et al., 2024) and design it with a linear layer and an attention layer, followed by the (pre-trained) LM head. We trained the adapter on the ShareGPT dataset using the AdamW optimizer for five epochs. For Llama-3.3-70B, this training took about 12 hours on a single H200 GPU.

For the length prediction model, we use a three-layer MLP with two GELU activations. We train the model on ShareGPT to predict the confidence that the remaining length is below 4, 8, ..., and 64 tokens. For training, we first generate the full output sequence from each input and then use the

generated output to derive the remaining lengths. We train the model for 20 epochs, which took about 15 hours on a single H100 GPU.

## 7 Evaluation

We evaluate Spexis against PP and TP, including their optimal combinations. We conduct experiments using resources from both our private cluster and a public cloud platform. Across these environments, we evaluate two widely adopted model families, Llama 3.3 (Meta AI, 2024) and Qwen3 (Qwen Team, 2025). We use SpecBench (Xia et al., 2024), LMSYS-Chat (Zheng et al., 2023) and UltraChat (Ding et al., 2023) as evaluation datasets.

Our private cluster provides two configurations: (1) an inter-node configuration with two nodes, each equipped with one NVIDIA RTX PRO 6000 GPU and connected via 1 Gb Ethernet; and (2) an intra-node configuration with a single node equipped with eight NVIDIA A40 GPUs communicating over PCIe 4.0×16. We use these configurations to measure the overall performance gains of Spexis and provide detailed breakdowns.

To cover a broader range of GPU and interconnect configurations, we use four hardware configurations on Runpod (Runpod Inc., 2026), a GPU cloud provider offering access to a diverse range of accelerator instances. Each configuration consists of a single eight-GPU node: (1) H100 SXM with an NVSwitch-based NVLink fabric; (2) A100 SXM with an NVSwitch-based NVLink fabric; (3) RTX PRO 6000 with PCIe 5.0 × 16; and (4) L40S with PCIe 4.0 × 16. On these instances, we compare TP and SP to assess the effectiveness of SP on standard single-node, multi-GPU servers across diverse GPU and interconnect configurations.

### 7.1 Overall Performance Improvements

We evaluated the speedup of Spexis in both inter-node and intra-node settings on our private cluster. To evaluate end-to-end serving performance, we conducted a standard load-scaling experiment. Specifically, for each request rate, we submitted requests from SpecBench or LMSYS-Chat to the serving engine, with request arrivals following a Poisson process. We varied the request rate across runs and measured the latency metrics for each run. As the latency metric, we report mean normalized latency, i.e., the average end-to-end latency of each request normalized by its output length. This metric is widely used for evaluating LLM inference

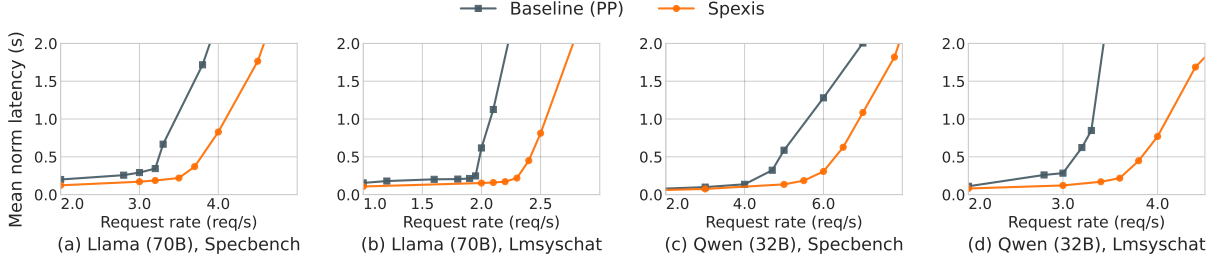


Figure 5: Mean normalized latency of Spexis and PP as the request rate increases in the inter-node setting.

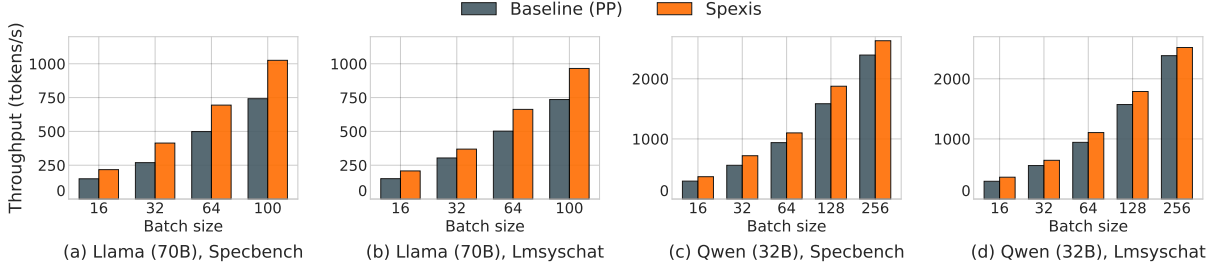


Figure 6: Decoding throughput of Spexis and PP as the batch size increases in the inter-node setting.

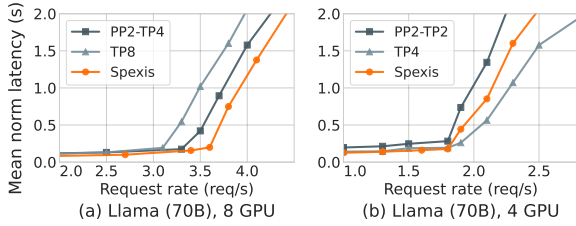


Figure 7: Mean normalized latency of Spexis, TP, and PP, including their best TP+PP configuration, as the request rate increases in the intra-node setting.

performance (Yu et al., 2022; Kwon et al., 2023).

We first report the results in the inter-node setting. Figure 5 shows the mean normalized latency of Llama and Qwen on the SpecBench and LMSYS-Chat datasets. The baseline uses a two-stage pipeline with two micro-batches. Spexis also uses a two-stage pipeline, with all optimizations in Section 5 enabled. We use a drop rate  $\alpha$  of 0.15 for Llama and 0.3 for Qwen. We see from the figure that Spexis can serve up to 34% higher request rates than the baseline. The largest gain is observed for Qwen on LMSYS-Chat, where Spexis serves 34% more requests per second.

We report decoding performance separately in Figure 6. Under disaggregation, prefill and decoding run on separate devices, making overall performance largely dependent on decoding throughput. We compare the decoding throughput of the baseline and Spexis as the batch size increases from 16 to 256. Spexis consistently outperforms the

baseline across all batch sizes, with larger gains at higher batch sizes. At batch sizes of 64–100 for Llama and 128–256 for Qwen, Spexis achieves up to 54% higher throughput.

Now we report the serving performance for the intra-node setting. Figure 7 compares the normalized latency of Spexis with those of PP and TP, including their optimal combination, for Llama-3.3-70B. The left plot shows the results on eight GPUs, where Spexis outperforms the other two settings by 8% and 16%. On four GPUs, Spexis outperforms the PP+TP configuration with both degrees set to two, but remains slower than the TP-only setting. The cost model in Table 1 explains this small gap. For TP, communication latency is captured by the  $T_{TP}$  term, which is relatively small in the four-GPU configuration without inter-socket communication; using the profiled value of  $T_{TP}$ , the cost model estimates the throughput overhead as 21%. For SP, the speculation accuracy obtained from offline profiling determines  $B_{\text{eff}}^*$ , which translates into a 21.5% throughput overhead. This small difference in modeled overhead is consistent with the slight measured advantage of TP-4 over SP. Overall, these results show that, in intra-node environments, Spexis is particularly effective for larger models that require eight or more GPUs.

## 7.2 Performance Breakdown

We examine the effects of the two components of the lookahead scheduling optimization described

in Section 5: selective speculation and remaining-length-aware scheduling. For this breakdown, we measure performance as we incrementally add speculative-parallel execution, selective speculation, and remaining-length-aware scheduling to the baseline, using Llama-3.3-70B and Qwen3-32B on SpecBench.

Figure 8 shows the resulting performance breakdown. Speculative-parallel execution alone improves performance by 10–15%. Selective speculation provides additional gains of 3 pp and 12 pp for Llama and Qwen, respectively, by dropping the  $\alpha$  fraction of low-confidence draft tokens. It has a larger impact on Qwen, as the model has lower speculation accuracy. Remaining-length-aware scheduling provides a further 3 pp gain for both models by delaying admissions that would otherwise evict running requests and require KV-cache recomputation.

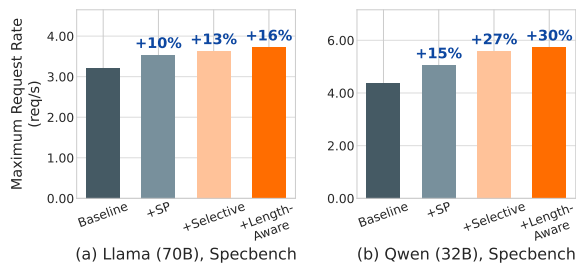


Figure 8: Performance breakdown of Spexis on SpecBench. From left to right, we incrementally add speculative-parallel execution (SP), selective speculation (Selective), and remaining-length-aware scheduling (Length-Aware), with percentages showing cumulative improvements over the baseline.

### 7.3 Performance on Cloud GPUs

We report experiments on cloud GPU instances to assess the effectiveness of Spexis across diverse intra-node GPU and interconnect configurations. Based on a survey of representative single-node, eight-GPU offerings from major cloud providers, we select four Runpod configurations—H100 SXM, A100 SXM, RTX PRO 6000, and L40S—covering both NVSwitch-based NVLink fabrics and PCIe interconnects. For each configuration, we compare Spexis against TP-only using Llama-3.3-70B and Qwen3-32B on SpecBench. Due to resource constraints, we report fixed-batch decoding throughput rather than the serving performance measured on our private cluster.

Table 2 reports the results. Spexis outperforms TP-only in five of the eight model–hardware pairs.

| GPU          | Interconnect  | Model | Thpt. (tokens/s) |             | Speedup |
|--------------|---------------|-------|------------------|-------------|---------|
|              |               |       | Spexis           | TP          |         |
| H100         | NVLink via    | L70   | 3587             | <b>3652</b> | 0.98×   |
| SXM          | NVSwitch      | Q32   | 2402             | <b>5068</b> | 0.47×   |
| A100         | NVLink via    | L70   | <b>1228</b>      | 966         | 1.27×   |
| SXM          | NVSwitch      | Q32   | 1454             | <b>2058</b> | 0.71×   |
| RTX PRO 6000 | PCIe 5.0 × 16 | L70   | <b>1492</b>      | 1236        | 1.21×   |
|              |               | Q32   | <b>2201</b>      | 2062        | 1.07×   |
| L40S         | PCIe 4.0 × 16 | L70   | <b>993</b>       | 930         | 1.07×   |
|              |               | Q32   | <b>1509</b>      | 1473        | 1.02×   |

Table 2: Fixed-batch decoding throughput (tokens/s) of Spexis and TP-only on single-node, eight-GPU Runpod configurations, measured at a fixed batch size of 64. Speedup is computed as Spexis throughput divided by TP throughput; bold indicates the higher throughput.

In particular, Spexis is faster in all four PCIe cases, achieving speedups of 1.02–1.21×, and achieves a 1.27× speedup for Llama-3.3-70B on A100 SXM. TP remains faster in the other three SXM cases because its communication overhead is lower than the speculation-failure overhead incurred by Spexis. Overall, these results demonstrate the effectiveness of Spexis across a range of intra-node GPU and interconnect configurations: Spexis consistently outperforms TP on the evaluated PCIe systems and can also outperform TP in a certain high-bandwidth SXM configuration.

### 7.4 Accuracy of Drafting Adapter

We evaluate the speculation accuracy of the self-speculative models used in our experiments. We fine-tune the models following prior work, while inserting the drafting adapter at the 1/2 point of the model for two-stage pipeline execution and at the 1/4 point for four-stage pipeline execution.

Table 3 reports the speculation accuracy across all SpecBench subtasks for Llama-70B and Qwen-32B. When the first half of the model is used for drafting, the accuracy is moderately high, ranging from 0.60 to 0.79. When only the first quarter is used, the accuracy decreases to 0.51–0.63.

Spexis further improves speculation accuracy by selectively applying speculation to a subset of sequences, as described in Section 5.1. Specifically, we rank the sequences in a batch by their speculation confidence and drop the lowest  $\alpha$  fraction, speculating only on the higher-confidence ones. Figure 9 shows the acceptance ratio of tokens as a function of their confidence rank, along with the  $\alpha$  values selected for the two models. The results

show that dropping the lowest-confidence  $\alpha$  fraction and speculating only on the higher-ranked sequences improves speculation accuracy by 8–14%.

| Model | Exit | Task Accuracy |      |      |      |      |      | Avg.        |
|-------|------|---------------|------|------|------|------|------|-------------|
|       |      | Math          | QA   | Sum  | RAG  | Tran | Mult |             |
| L70   | 1/2  | .840          | .743 | .760 | .788 | .774 | .793 | <b>.785</b> |
|       | 1/4  | .680          | .629 | .581 | .622 | .570 | .656 | <b>.634</b> |
| L8    | 1/2  | .800          | .776 | .660 | .727 | .644 | .759 | <b>.759</b> |
|       | 1/4  | .665          | .609 | .487 | .547 | .461 | .625 | <b>.596</b> |
| Q32   | 1/2  | .682          | .535 | .565 | .619 | .609 | .603 | <b>.596</b> |
|       | 1/4  | .597          | .471 | .474 | .489 | .427 | .550 | <b>.511</b> |
| Q8    | 1/2  | .674          | .559 | .598 | .608 | .586 | .620 | <b>.637</b> |
|       | 1/4  | .601          | .494 | .515 | .521 | .469 | .557 | <b>.532</b> |

Table 3: Speculation accuracy on SpecBench for Llama-3.3-70B-Instruct (L70), Llama-3.1-8B-Instruct (L8), Qwen3-32B (Q32), and Qwen3-8B (Q8). The drafting adapter (Exit) is inserted at either the halfway point (1/2) or the one-quarter point (1/4) of the model.

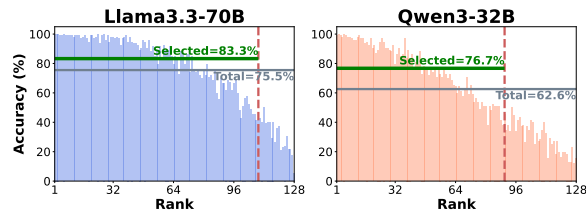


Figure 9: Speculation accuracy ranked by confidence score on SpecBench Dataset (both 1/2 exited). The value of cutoff  $\alpha$  is specified by dotted vertical lines.

## 7.5 Accuracy of Length Prediction

We evaluate the accuracy of our remaining-length prediction method. For the prediction, we trained a three-layer MLP model that outputs confidence scores for each sequence if its remaining length falls below 4, 8, 16, 32, and 64 tokens. Then we choose a confidence cutoff for each threshold (4, 8, ..., 64) based on profiling to predict if the remaining length is below the corresponding threshold.

Table 4 shows the precision and recall of the prediction method. The MLP is trained on a subset of ShareGPT, and the reported scores are measured on the remaining ShareGPT data as well as the full SpecBench and UltraChat datasets. The precision is generally high across datasets and length thresholds, mostly ranging from 70% to 90.7%. The recall is also moderately high ranging 40–71%.

| Model | Thresh.  | ShareGPT |        | SpecBench |        | UltraChat |        |
|-------|----------|----------|--------|-----------|--------|-----------|--------|
|       |          | Prec.    | Recall | Prec.     | Recall | Prec.     | Recall |
| L70   | $L < 4$  | 79.0%    | 53.4%  | 82.6%     | 59.7%  | 76.3%     | 56.0%  |
|       | $L < 8$  | 78.6%    | 55.2%  | 80.3%     | 59.5%  | 78.1%     | 57.6%  |
|       | $L < 16$ | 81.2%    | 58.1%  | 85.8%     | 59.4%  | 81.5%     | 61.9%  |
|       | $L < 32$ | 83.5%    | 64.0%  | 88.9%     | 61.3%  | 82.9%     | 68.6%  |
|       | $L < 64$ | 85.2%    | 67.6%  | 90.7%     | 67.5%  | 85.8%     | 71.1%  |
| Q32   | $L < 4$  | 76.1%    | 53.9%  | 70.0%     | 39.8%  | 63.5%     | 52.2%  |
|       | $L < 8$  | 78.4%    | 58.8%  | 73.9%     | 43.7%  | 67.5%     | 54.5%  |
|       | $L < 16$ | 82.6%    | 65.8%  | 77.6%     | 52.4%  | 73.0%     | 61.5%  |
|       | $L < 32$ | 87.0%    | 69.0%  | 83.5%     | 60.8%  | 80.1%     | 69.3%  |
|       | $L < 64$ | 85.7%    | 62.8%  | 83.1%     | 67.7%  | 82.3%     | 71.0%  |

Table 4: Precision (Prec.) and recall for the remaining-length prediction using thresholds of 4, ..., 64 tokens (L70: Llama-3.3-70B-Instruct, Q32: Qwen3-32B).

## 8 Conclusion

We present Spexis, a multi-GPU scheduling framework that improves the efficiency of LLM inference by integrating speculative decoding with distributed execution. Spexis introduces speculative parallelism, which overlaps speculative and normal execution to expose an additional axis of parallelism without increasing KV-cache memory usage. It further incorporates lookahead scheduling to make admission and execution decisions based on predicted speculation quality and near-future memory pressure, thereby reducing wasted speculation, KV-cache eviction, and recomputation.

Our evaluation shows that Spexis consistently improves serving performance across diverse GPU configurations, achieving up to 34% speedups over the baseline using the optimal combination of pipeline and tensor parallel execution. The results show that speculative decoding can be leveraged not only as an algorithmic acceleration technique, but also as an effective systems mechanism for improving the performance of distributed LLM inference.

## Limitations

Our exploration of Spexis leaves several areas for future investigation. First, the framework requires training auxiliary components—specifically, a drafting adapter and a length predictor. While this preliminary overhead is relatively small, it introduces an extra step in the deployment pipeline. Second, the performance of Spexis is inherently sensitive to speculation accuracy. Although we mitigate the impact of low-accuracy scenarios using the fallback mechanisms described in Section 5.1, significant domain shifts may still reduce the ex-

pected throughput gains. Finally, the relative advantage of Speculative Parallelism (SP) over Tensor Parallelism (TP) is highly dependent on hardware interconnects. In environments with exceptionally fast inter-GPU communication (e.g., pure NVLink setups), applying only TP might be sufficient. However, SP remains essential for scaling large models across diverse hardware configurations where communication latency is a bottleneck.

## Acknowledgments

This work was supported by the New Faculty Startup Fund from Seoul National University, by the research fund of Hanyang University (HY-201700000002388), and by Automation and System Research Institute at Seoul National University (No. 0418-20250030). This work was also supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2025-02214497, RS-2025-02263167, RS-2024-00438729, RS-2021-II211343, IITP-2026-RS-2021-II211817), and by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (RS-2026-25476387). We are grateful to VESSL AI for providing compute resources used in some of the early experiments in this work. Jiwon Seo is the corresponding author.

## References

- Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in llm inference with sarathi-serve. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI’24, USA. USENIX Association.
- Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023. [Enhancing chat language models by scaling high-quality instructional conversations](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3029–3051, Singapore. Association for Computational Linguistics.
- Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, and 1 others. 2024. Layerskip: Enabling early exit inference and self-speculative decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12622–12642.
- Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. 2024. Efficient llm scheduling by learning to rank. *Advances in Neural Information Processing Systems*, 37:59006–59029.
- Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, Hyoungho Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and 1 others. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32.
- Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. S<sup>3</sup>: Increasing gpu utilization during generative inference for higher throughput. *Advances in Neural Information Processing Systems*, 36:18015–18027.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.
- Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR.
- Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024a. Eagle-2: Faster inference of language models with dynamic draft trees. In *Proceedings of the 2024 conference on empirical methods in natural language processing*, pages 7421–7432.
- 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.
- Fangcheng Liu, Yehui Tang, Zhenhua Liu, Yunsheng Ni, Duyu Tang, Kai Han, and Yunhe Wang. 2024. Kangaroo: Lossless self-speculative decoding for accelerating llms via double early exiting. *Advances in Neural Information Processing Systems*, 37:11946–11965.
- Meta AI. 2024. Llama-3.3-70B-Instruct Model Card. <https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct/>.
- Qwen Team. 2025. [Qwen3 technical report](#). Preprint, arXiv:2505.09388.
- Runpod Inc. 2026. The AI developer cloud. <https://www.runpod.io/>. Accessed August 28, 2026.
- Rana Shahout, Eran Malach, Chunwei Liu, Weifan Jiang, Minlan Yu, and Michael Mitzenmacher. 2024. [Don’t stop me now: Embedding based scheduling for llms](#). Preprint, arXiv:2410.01035.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. [Megatron-lm: Training multi-billion parameter language models using model parallelism](#). *Preprint*, arXiv:1909.08053.

vLLM Team. 2025. [vllm documentation](#). [https://docs.vllm.ai/en/stable/serving/parallelism\\_scaling/](https://docs.vllm.ai/en/stable/serving/parallelism_scaling/). Accessed: 2026-03-17.

Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhi-fang 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.

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.

Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. 2024. [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.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zilong Lin, Eric P Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2023. [Lmsys-chat-1m: A large-scale real-world llm conversation dataset](#). *Preprint*, arXiv:2309.11998.

## A Appendix

We will now show the derivations of the steady-state batch-size limits and the throughput gain of remaining-length-aware scheduling.

### A.1. Saturated Limit of Equation (1)

Let the initial conditions be  $s_0^a = 0$ ,  $s_0^r = 0$ , and  $n_0 = B$ , with  $s_t^a = s_t^r = n_t = 0$  for  $t < 0$ . Let  $B_{\text{eff}}^* = \lim_{t \rightarrow \infty} (s_t^a + n_t) = s^a + n$  be the steady-state effective bound. By the conservation of the total quantity in the system,  $s_t^a + s_t^r + n_t = B$  for all  $t$ . Now:

$$B = B_{\text{eff}}^* + s^r$$

Where, by taking the limit  $t \rightarrow \infty$  on the recurrence relation for  $s_t^r$ :

$$s^r = \sum_{i=1}^{N-1} (s^a + n)(1 - \Theta) = (N - 1)B_{\text{eff}}^*(1 - \Theta)$$

Overall:

$$B_{\text{eff}}^* + (N - 1)B_{\text{eff}}^*(1 - \Theta) = B$$

$$B_{\text{eff}}^* = \frac{B}{N - \Theta(N - 1)}$$

As desired.

### A.2. Saturated Limit of Equation (3)

Let the initial conditions be  $B_i = B(1 - \alpha)^i$  for  $0 \leq i \leq N - 1$ . Let  $B_\alpha^* = \lim_{t \rightarrow \infty} B_t$ . Rearranging the recurrence relation yields  $B_{t+1} - B_t = \alpha(B_{t+1-N} - B_t)$ . Summing both sides from  $t = N - 1$  to  $T - 1$  creates a telescoping sum. Now:

$$B_T - B_{N-1} = \alpha \sum_{t=N-1}^{T-1} B_{t+1-N} - \alpha \sum_{t=N-1}^{T-1} B_t$$

Where, by shifting the index and canceling overlapping terms as  $T \rightarrow \infty$ :

$$B_\alpha^* - B_{N-1} = \alpha \sum_{j=0}^{N-2} B_j - \alpha(N - 1)B_\alpha^*$$

And, using the initial condition for the geometric sum:

$$\alpha \sum_{j=0}^{N-2} B(1 - \alpha)^j = B - B(1 - \alpha)^{N-1} = B - B_{N-1}$$

Overall:

$$B_\alpha^* - B_{N-1} = (B - B_{N-1}) - \alpha(N - 1)B_\alpha^*$$

$$B_\alpha^* + \alpha(N - 1)B_\alpha^* = B$$

$$B_\alpha^* = \frac{B}{1 + \alpha(N - 1)}$$

As desired.

### A.3. Derivation of Throughput Ratio G

We derive  $G$  over the period between two consecutive request arrivals. Following the notation in Section 5.2, we further let  $\text{Tok}_x$  and  $T_x$  denote the decoded-token count and execution time for  $x \in \{\text{aggr}, \text{delay}\}$ , respectively.

$$\begin{aligned}
\text{Tok}_{\text{aggr}} &= (bs + 1)I - \sum_{k=1}^n i_{\text{wait}}^k, \\
T_{\text{aggr}} &= (I - (n + 1))t_d + (n + 1)t_p, \\
\text{Thr}_{\text{aggr}} &= \frac{\text{Tok}_{\text{aggr}}}{T_{\text{aggr}}}, \\
\text{Tok}_{\text{delay}} &= (bs + 1)I - i_{\text{delay}}, \\
T_{\text{delay}} &= (I - 1)t_d + t_p, \\
\text{Thr}_{\text{delay}} &= \frac{\text{Tok}_{\text{delay}}}{T_{\text{delay}}}, \\
G &= \frac{\text{Thr}_{\text{delay}}}{\text{Thr}_{\text{aggr}}}.
\end{aligned}$$