LLM-42: Enabling Determinism in LLM Inference with Verified Speculation

Raja Gond[φ], Aditya K Kamath[ϑ], Ramachandran Ramjee[φ], and Ashish Panwar[φ]
[φ] Microsoft Research India [ϑ] University of Washington

Abstract

In LLM inference, the same prompt often returns different outputs across different runs. At the system level, this nondeterminism arises from floating-point non-associativity interacting with dynamic batching and GPU kernels whose reduction orders vary with batch size. A straightforward way to eliminate nondeterminism is to disable dynamic batching, but doing so severely degrades throughput. Another approach is to make the kernels batch-invariant; however, this tightly couples determinism to kernel design, requires new implementations, and imposes fixed overheads regardless of how much of the workload actually requires determinism.

Inspired by self-speculative decoding, we present LLM-42 — a scheduling-based approach to enable determinism in LLM inference. Our key observation is that a given sequence of tokens is likely to emit the same next token across multiple runs, even with dynamic batching. Moreover, most GPU kernels use the same reduction schedules for all inputs of a given shape. Leveraging these insights, LLM-42 decodes tokens using a nondeterministic fast path and enforces determinism through a verify-rollback protocol. The verifier replays fast path tokens under a fixed batch size to obtain a deterministic reference output, commits tokens that match the reference, and rolls back those that do not. LLM-42 reuses many existing kernels unchanged and incurs overhead only proportional to the fraction of traffic that requires determinism.

CCS Concepts: • **Computing methodologies → Machine learning**.

Keywords: Large language models; deterministic inference; speculative decoding; floating-point nondeterminism

ACM Reference Format:
Raja Gond, Aditya K Kamath, Ramachandran Ramjee, and Ashish Panwar. 2026. LLM-42: Enabling Determinism in LLM Inference with Verified Speculation. In *ACM SIGOPS 32nd Symposium on Operating Systems Principles (SOSP '26), September 29-October 02, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3830418.3843901

SOSP '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2585-2/2026/09
https://doi.org/10.1145/3830418.3843901

1 Introduction

Inference with large language models (LLMs) does not meet a basic system expectation: identical inputs do not reliably produce identical outputs, even with fixed decoding parameters, seeds, and hardware [15, 25, 48, 56]. Such nondeterminism undermines reproducibility [13, 48], complicates debugging and integration testing [53], and introduces variance into downstream pipelines such as reinforcement learning and evaluation [60]. In user-facing applications, nondeterministic outputs can also degrade user experience. As a result, there is a growing demand for determinism in LLM serving systems [14, 39, 45, 53] but a satisfactory solution to achieve it remains elusive.

He et al. [25] showed that, at the system level, nondeterminism in LLM inference stems from the non-associativity of floating-point arithmetic[1]. This effect manifests in practice because most of the core LLM operators—including matrix multiplications, attention, normalization and communication—rely on reduction operations, and GPU kernels for these operators choose different reduction schedules to maximize performance at different batch sizes. Hence, a token when computed under different batch sizes can produce numerically different results. The same study also proposed *batch-invariant computation* as a means to eliminate nondeterminism. In this approach, a kernel processes each input token using a single, universal reduction strategy independent of batching. Popular LLM serving systems such as vLLM and SGLang have recently adopted this approach [45, 54].

While batch-invariant computation guarantees determinism, we find that it is fundamentally over-constraining. Enforcing a fixed reduction strategy for every token—regardless of model phase or batch geometry—strips GPU kernels of the very parallelism they are built to exploit. For example, the batch-invariant GEMM kernels provided by He et al. do not use the split-K strategy that is otherwise commonly used to accelerate GEMMs at low batch sizes [27, 38]. Furthermore, most GPU kernels are not batch-invariant to begin with, so insisting on batch-invariant execution effectively demands new kernel implementations, increasing engineering and maintenance overhead. Finally, batch-invariant execution makes determinism the default for all requests, even when determinism is undesirable or even harmful [29, 31, 59].

[1]Nondeterminism also comes from differences in sampling strategies (e.g., greedy vs top-k). Our goal is to ensure that output is deterministic for fixed sampling hyper-parameters; different hyper-parameters can result in different output and this behavior is intentional.

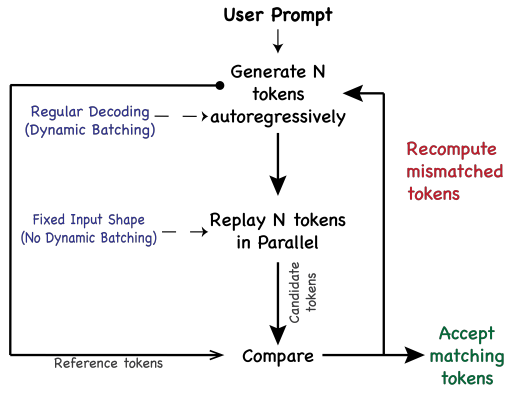


Figure 1. Overview of LLM-42.

In this paper, we introduce an alternate approach, named LLM-42, to enable determinism in LLM inference. We define a *run* as a model execution on the GPU that takes an input and produces a sequence of output tokens. Our approach is inspired by the following observations. First, if two runs of a request have generated the same sequence of output tokens through step t, they are likely to generate the same next token at step $t + 1$. However, if they produce different tokens, autoregressive decoding amplifies this initial divergence, causing subsequent tokens to increasingly diverge. Second, many GPU kernels use the same reduction schedule for all elements of a batch and, importantly, use the same schedule across batches with the same shape. Consequently, executing the same tokens with a fixed batch size provides a deterministic reference.

Using these observations and leveraging ideas from self-speculative decoding [43, 61], LLM-42 achieves determinism through a scheduling-based approach. Similar to self-speculative decoding, LLM-42 uses the *same model* for both speculation and verification. However, unlike self-speculative decoding which uses different execution paths for speculation and verification, LLM-42 uses the same model execution path but switches between nondeterministic execution for speculation and fixed-batch, deterministic execution for verification. Table 1 highlights the key differences between LLM-42 and self-speculative decoding.

By decoupling determinism from token generation, LLM-42 enables determinism to be enforced selectively, preserving the natural variability and creativity of LLM outputs where appropriate. This separation also helps performance: the fast path can use whatever batch sizes and reduction schedules are most efficient, prefill computation can follow a different reduction strategy than decode, and verification can be skipped entirely for requests that do not need determinism.

Efficient verification and infrequent rollbacks are central to the overall performance of LLM-42. Achieving both involves a trade-off governed by the verification window size, i.e., the number of tokens verified together. Smaller windows incur high verification overhead since their computation is largely memory-bound but require less recomputation due to reduced verification failures. In contrast, larger windows incur lower verification cost due to being compute-bound, but increase recomputation cost by triggering longer rollbacks on mismatches. To balance this trade-off, we introduce *grouped verification* (§4.3): instead of verifying a large window of a single request, we verify smaller fixed-size windows of multiple requests together. This design achieves the best of both: low rollback cost of small windows and high verification efficiency of large windows.

Self-Speculative Decoding	LLM-42
Fast path generates tokens using some form of approximation	No approximation; speculation may only have floating-point rounding errors
Low acceptance rate and limited speculation depth (2-8 tokens)	High acceptance rate and hence longer speculation (32-64 tokens)
All requests go through speculation and verification	Only specific requests tagged as deterministic

Table 1. Self-Speculative Decoding vs. LLM-42.

We implement LLM-42 in SGLang and evaluate it with Llama-3-8B and Llama-3-70B models on H100 GPUs. Overall, our key contributions are as follows:

- We highlight performance and engineering challenges of achieving determinism via batch-invariant computation.
- We present LLM-42: an approach that enables determinism in LLM inference by framing it as a scheduling problem.
- We show that LLM-42 retains near-peak performance (typically within 10% of optimal) when deterministic traffic is low, whereas SGLang incurs up to 52% overhead to enforce determinism.

2 Background and Motivation

This section introduces LLMs, explains the source of nondeterminism in LLM inference and quantifies the cost of batch-invariant computation based deterministic inference.

2.1 Large Language Models

Transformer-based LLMs compose a stack of identical decoder blocks, each containing a self-attention module, a feed-forward network, normalization layers and communication primitives (Figure 2). The hidden state of every token flows through these blocks sequentially, but within each block the computation is highly parallel: attention performs matrix multiplications over the key–value (KV) cache, FFNs apply two large GEMMs surrounding a nonlinearity, and normalization applies a per-token reduction over the hidden dimension. Inference happens in two phases, namely prefill and decode. Prefill processes prompt tokens in parallel, generating KV cache for all input tokens. This phase is

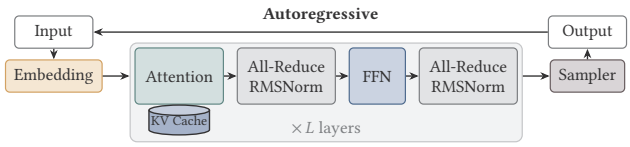
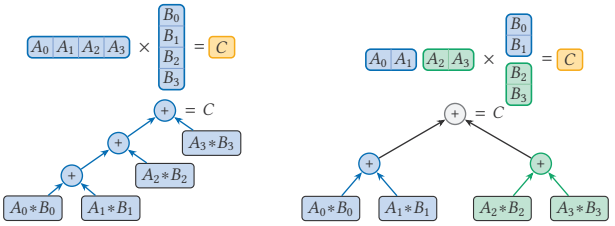


Figure 2. High-level architecture of LLMs.



(a) Without split-K. **(b)** With split-K.

Figure 3. GEMM kernels compute dot products using standard accumulation or split-K parallelization. While split-K increases parallelism, it alters the reduction tree based on K.

dominated by large parallel computation. Decode is autoregressive: each step consumes the most recent token, updates the KV cache with a single new key and value, and produces the next output token; decode phase is memory bound. In each forward pass, the LLM produces logits and the sampler converts logits into output tokens.

2.2 Nondeterminism in LLM Inference

In finite precision, arithmetic operations such as accumulation are non-associative, i.e., $(a+b)+c \neq a+(b+c)$. Nondeterminism in LLM inference stems from this non-associativity when combined with *dynamic batching*, a standard technique to achieve high throughput inference [11, 32, 55]. With dynamic batching, the same request gets co-located with different sets of requests across different runs. Further, GPU kernels adapt their parallelization—and consequently their reduction—strategies based on the input sizes. As a result, the same logical operation can be evaluated with different floating-point accumulation orders depending on its current batch, leading to inconsistent numerical results.

Reductions are common in LLM inference, appearing in matrix multiplications (GEMMs), attention, normalization and collective communication such as AllReduce. GEMM is the most common and time consuming operator. High-performance GEMM implementations on GPUs use hierarchical tiling and parallel reductions to improve occupancy and memory reuse. A common optimization is split-K parallelism [38], where the reduction dimension is partitioned across multiple thread blocks. Each block computes a partial result, which is then combined via an additional reduction step. Whether split-K is used—and how many splits are chosen—depends on the shape of input matrices. These choices determine the reduction tree used for computation and, consequently, its output, as shown in Figure 3. Similarly, attention kernels split work across the KV dimension to increase SM utilization [20, 21], followed by a reduction to combine partial results. Normalization operators reduce across feature dimensions. As inference progresses, batch-dependent reduction choices in these operators introduce small numerical drifts that propagate across kernels, layers and decoding steps, eventually influencing output tokens even when the sampling hyper-parameters are fixed.

2.3 Defeating Nondeterminism

He et al. introduced a new approach named batch-invariant computation to enable deterministic LLM inference [25]. In this approach, a GPU kernel is constrained to use a single, universal reduction strategy for all tokens, eliminating batch-dependent reductions. For example, batch-invariant GEMM kernels can be implemented by disabling dynamic split-K. Both SGLang and vLLM use this approach [45, 53, 54]: these systems either deploy the batch-invariant kernels provided by He et al. [35] or implement new kernels [44, 51].

Batch-invariant computation achieves determinism, but it is challenging to adopt this approach in real LLM serving systems. By enforcing a universal reduction strategy across all executions, it couples determinism to kernel design and sacrifices performance. It also turns determinism into a fixed tax paid by every request: dynamic batching aggregates requests, but batch-invariant kernels eliminate the very optimizations—such as split-K and shape-aware tiling—that make batching effective in the first place. Worse, because most kernels are not batch-invariant, adopting this approach requires maintaining a parallel kernel stack solely for determinism, compounding system complexity.

Figure 4 compares the throughput of cuBLAS-based non-batch-invariant GEMM kernels against two sets of batch-invariant kernels: Triton-based kernels developed by He et al. [35] and CUDA-based DeepGEMM kernels [8]. Both these kernels are used in SGLang. The matrix dimensions correspond to the QKV projection operation of the Llama-3-70B. On our H100 GPUs, cuBLAS kernels reach up to 801 TFLOPS, whereas the batch-invariant Triton kernel peaks at 533 TFLOPS, with a max slowdown of up to 7.21×. This gap arises because this Triton-based batch-invariant implementation does not use split-K or exploit newer hardware features such as Tensor Memory Accelerators [2] or advanced techniques like warp specialization [46]. DeepGEMM kernels are much faster than Triton kernels, reaching up to 785 TFLOPS, but, in most cases, are still significantly slower than cuBLAS kernels. In particular, at low batch sizes and higher TP dimensions that benefit from split-K, DeepGEMM kernels incur slowdown of up to 2.78× compared to cuBLAS.

We also compared three implementations of Fused RMSNorm–Residual Addition: a Python-based version that SGLang uses

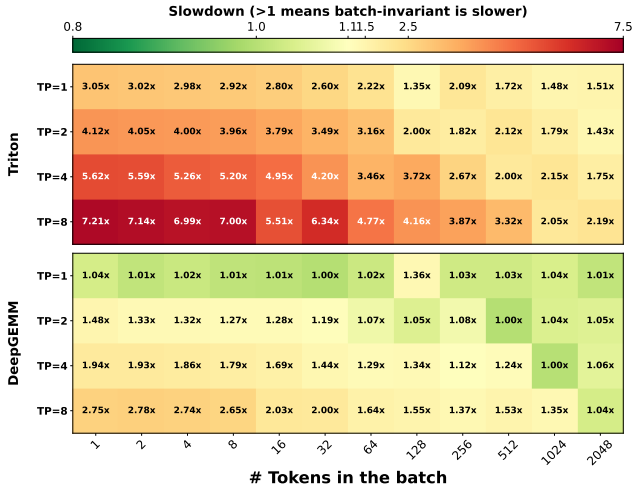


Figure 4. Slowdown of batch-invariant GEMM kernels compared to cuBLAS GEMM kernels (model: Llama-3-70B, operator: QKV projection).

in deterministic mode [7], a Triton-based batch-invariant kernel based on an implementation by He et al. [6], and a fused non-batch-invariant CUDA kernel from SGLang-Default implementation [10]. The Python version is up to 7× slower due to unfused operations and inefficient memory usage. The Triton kernel is faster than PyTorch kernel but still up to 50% slower than the fused CUDA implementation. Relying on such slower operators significantly degrades overall inference performance as shown below.

End-to-end throughput. Figure 5 measures token generation throughput (tokens/s) under three scenarios for Llama-3-70B. On the left, 8 requests run on a nondeterministic server, achieving 1495 tokens/s. On the right, one new request enters the system, increasing batch size to 9; in the nondeterministic configuration, throughput increases to 1775 tokens/s. In SGLang, determinism is a property of the server configuration: all requests execute either in deterministic or nondeterministic mode. When the server is configured for determinism, it uses slower batch-invariant kernels, resulting in a throughput of 1192 tokens/s—33% lower than the nondeterministic configuration. This slowdown affects all requests, regardless of whether individual requests require determinism. LLM-42 enables selective determinism, avoiding this global trade-off. When one out of nine requests requires determinism, it achieves a decode throughput of 1733 tokens/s, which is 37% higher than SGLang-deterministic and within 3% of SGLang-nondeterministic.

Overall, these results show that batch-invariant execution incurs a substantial performance penalty. While it may be feasible to further improve the performance of batch-invariant kernels, doing so requires continued model- and hardware-specific kernel development, as in the case of DeepGEMM kernels that have been written in CUDA to improve upon Triton kernels. This engineering and maintenance burden

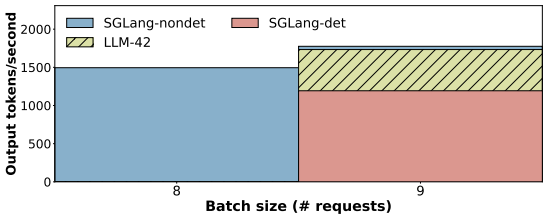


Figure 5. Decode throughput under different scenarios.

makes the approach difficult to sustain in practice. This is likely why deterministic inference is largely confined to debugging and verification today, rather than being deployed for real-world LLM serving.

2.4 Speculative Decoding

Speculative decoding [33, 34, 37] accelerates autoregressive LLM inference by first generating a sequence of candidate tokens using a lightweight draft model, and then verifying these candidates in parallel using the target model. Tokens that match the target model's predictions are accepted, while mismatches trigger rollback and regeneration from the point of divergence. In this scheme, generating draft tokens is cheap and verifying multiple tokens in parallel is efficient. Hence speculative decoding improves throughput without changing the final output distribution. Self-speculative decoding [22, 43, 57, 61] extends this idea by using the target model itself as the draft model, typically by skipping layers or performing early exits during drafting, thereby eliminating the need for a separate auxiliary model while retaining most of the performance benefits. Speculative decoding has emerged as one of the most effective approaches for accelerating LLM inference and we show that the same structure can be repurposed to enable determinism.

3 Observations

In this section, we distill a set of concrete observations about nondeterminism, GPU kernels and LLM use-cases. These observations expose why batch-invariant computation is too restrictive and motivate reasoning about determinism at the level of reduction schedules rather than individual operations.

> **Observation-1 (O1).** *If two runs of a request have generated the same sequence of output tokens through step t, they are likely to generate the same next token at step $t+1$. However, if they produce different tokens, autoregressive decoding amplifies this divergence over subsequent steps.*

This is because tokens become inconsistent only when floating-point drift is large enough to change the token picked by the sampler—e.g., by changing the relative ordering or acceptance of high-probability candidates under

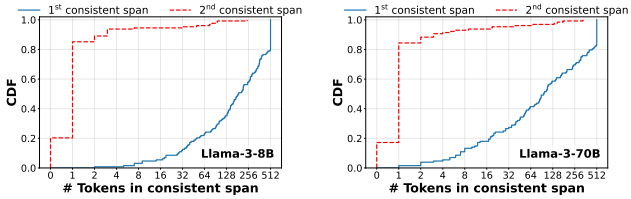


Figure 6. CDF of the first and second consistent spans under dynamic batching.

the decoder's sampling policy (e.g., greedy or stochastic sampling). In practice, such boundary-crossing events are rare, as numerical drift typically perturbs logits only slightly. However, autoregressive decoding amplifies even a single such deviation: once a token differs, all subsequent tokens may diverge. Since a single request typically produces hundreds to thousands of output tokens, two sequence-level outputs can look dramatically different even if the initial divergence is caused by a single token flip.

We demonstrate this phenomenon empirically using Llama-3-8B and Llama-3-70B models on the ShareGPT dataset. We first execute 256 requests with batch size one—i.e., without dynamic batching—to obtain reference ("ground-truth") output tokens. We then re-run the same requests under dynamic batching at a load of 6 queries per second and compare each request's output against its reference. In both runs, we fix the per-request output length to 512 tokens.

We quantify divergence using two metrics: the *first consistent span* of a request denotes the number of initial output tokens that are identical across the two runs, while the *second consistent span* measures the number of matching tokens between the first and second inconsistent token positions. Figure 6 shows the CDF of the length of consistent spans. In the common case, the first consistent span covers 10s-100s of initial tokens, with some requests having all 512 tokens in the first consistent span. However, once a token diverges, the sequences rarely re-align: the second consistent span is at most one for 80% requests, indicating that after the first mismatch, subsequent tokens are highly likely to differ from the reference execution.

Observation-2 (O2). *GPU kernels used by LLMs are generally position-invariant: a token's output is independent of its position within the batch as long as it is computed with a fixed batch size.*

Position-invariance arises from two natural properties of GPU kernels: (1) all elements within a batch share a common reduction schedule, and (2) this schedule depends only on the batch size and is identical across all inputs of that size.

GEMM kernels provide a simple example. For an input matrix $A \in \mathbb{R}^{M \times K}$, a GEMM kernel computes all M output elements using the same reduction order R. Further, this

Figure 7. Position-invariant kernels produce the same output for a given input element irrespective of its position in the batch, as long as the total batch size is fixed. In this example, $T_1' == T_1''$ if the kernel is position-invariant.

reduction order R is fixed for all matrices of size $M \times K$. As a result, under a fixed batch size, the output for each element depends only on its inputs and not on its position, satisfying position-invariance.[2] Figure 7 illustrates this property for a position-invariant kernel, while Table 2 summarizes the invariance properties of common LLM operators.

Ring-based AllReduce is neither batch-invariant nor position-invariant, as it reduces different chunks of a tensor in different orders around the ring (not all tokens of a batch share a common reduction schedule).

Additionally, the unquantized fused MoE kernel (Triton) [9] with variable tokens per expert is not guaranteed to be *batch-invariant*: it selects the kernel configurations based on the total number of tokens M in the batch, including `BLOCK_K`, so a different M can have different configurations and potentially different reduction order. For example, it sums over K in tiles of `BLOCK_K`, so a different M can regroup the sum. Using one fixed kernel tile configuration for all batch sizes removes this source of batch dependence, as vLLM does in its batch-invariant mode [53]. The unquantized fused MoE kernel is nonetheless *position-invariant*: the tile configuration, including `BLOCK_K`, is constant per launch for a fixed total number of tokens M, with no split-K and no atomics, so every token sums over K identically, and sorting tokens by expert only determines which tile a token lands in. Per-expert token counts vary with the batch, but the tile sizes are determined by the total token count, so position-invariance holds and LLM-42 reuses the kernel unmodified.

Observation-3 (O3). *Position-consistent reductions are sufficient for determinism: a given token position must go through the same reduction schedule across all runs of a given request; reduction schedules for different token positions within or across sequences can be different.*

Numerical differences in the output of a token arise from differences in how its own floating-point reductions are performed, not from the numerical values of other co-batched tokens. While batching affects how computations are scheduled, the computation for a given token position is functionally independent (we discuss some exceptions in §4.4).

[2]Note that such guarantees do not hold for kernels that implement reductions via atomic operations. Fortunately, kernels used in the LLM forward pass do not use atomic reductions [25].

Category	Operator	Invariant	
		Batch	Position
Matmul	CuBLAS GEMM	✗	✓
	Fused MoE (Triton)	✗	✓
Attention	FlashAttention-3‡	✓	✓
Communication	Multimem-based AllReduce*	✓	✓
	Ring-based AllReduce	✗	✗
	Tree-based AllReduce★	✓	✓
Normalization	RMSNorm†	✗	✓
	Fused RMSNorm + Residual†	✗	✓

Table 2. Invariance properties of common inference operators (‡num_splits=1, *CUDA 13.0+, ★specific NCCL settings, †vLLM/SGLang defaults).

Interactions across tokens occur only indirectly through execution choices—such as which partial sums are reduced together—not through direct data dependencies. Consequently, as long as a token position is always reduced using the same strategy, its output remains consistent.

The key difference between batch-invariant and position-invariant computation lies in the scope at which consistency is enforced. Batch-invariance effectively imposes a single global reduction strategy, preventing the system from adapting kernel configurations to different batch sizes. In contrast, position-invariance enforces consistency at the level of token positions rather than across all batch sizes. This allows different batch sizes to be optimized independently. This approach is also easier to deploy since many kernels already support position-consistent reductions.

Observation-4 (O4). *Current systems take an all-or-nothing approach: they either enforce determinism for every request or disable it entirely. Such a design is not a good fit for LLM inference.*

This is because many LLM workloads neither require bitwise reproducibility nor benefit from it. In fact, controlled stochasticity is often desirable as it enhances output diversity and creativity of LLMs [29, 31, 59]. In contrast, requests such as evaluation runs, safety audits, or regression testing require bit-level reproducibility. Enforcing determinism on all requests is an overkill.

It is worth highlighting that this all-or-nothing behavior stems from the batch-invariant approach itself that ties determinism to kernel design. Since all co-batched tokens go through the same set of kernels, determinism becomes a global property of the system rather than being selective. While one could run different requests through separate deterministic and nondeterministic kernels, doing so would fragment batches, complicate scheduling and hurt efficiency.

4 LLM-42

Since nondeterminism in LLM inference comes from dynamic batching, disabling it would make inference deterministic. However, dynamic batching is arguably the most powerful performance optimization for LLM inference [12, 32, 55, 63]. Therefore, our goal is to enable determinism in the presence of dynamic batching. In this section, we introduce LLM-42, a speculative decoding-inspired, scheduler-driven approach to achieve this goal.

4.1 Overall Design

LLM-42 exploits the observations presented in §3 as follows:
Leveraging O1 for scheduling-based determinism. Tokens decoded from a consistent state are mostly consistent. Based on this observation, we re-purpose speculative-decoding style mechanism to enable determinism via a decode–verify–rollback (DVR) protocol. DVR optimistically decodes tokens using the fast path and delegates deterministic enforcement to a separate verifier. Only tokens approved by the verifier are returned to the user, while the few that fail verification are discarded and recomputed. The key is to ensure that the verifier's output itself is consistent. We leverage O2 to achieve this.
Leveraging O2 for position-consistent reductions. To compute position-consistent reductions, we always verify a fixed T number of tokens. If fewer than T tokens are available (e.g., at the end of a sequence), we pad with dummy tokens.
Leveraging O3 for performance. Position-consistent reductions are sufficient for determinism. This observation lets us compute prefill and decode phases of a request using different reduction schedules. Because prefill is highly parallel even within a single request, it can be made deterministic simply by avoiding arbitrary batching of prefill tokens and eliminating the need for a verifier. The verifier is required only for the tokens generated by the decode phase.
Leveraging O4 for selective determinism: LLM-42 decouples determinism from token generation. This makes selective determinism straightforward: only requests that require deterministic inference incur verification. We expose this control via a new API flag `is_deterministic` $\in$ {`True`, `False`} that allows users to explicitly request determinism on a per-request basis; default is `False`.

4.2 Decode-verify-rollback (DVR)

DVR adopts a deferred approach to enforce determinism: it decodes tokens optimistically using a high-throughput, nondeterministic execution path, and then identifies and corrects any inconsistencies through verification and targeted recomputation. It ensures consistency in both logits and output tokens.

Figure 8 illustrates how DVR operates, assuming a verification window size of four tokens. The blue request requires deterministic output and its first output token T_0 is produced

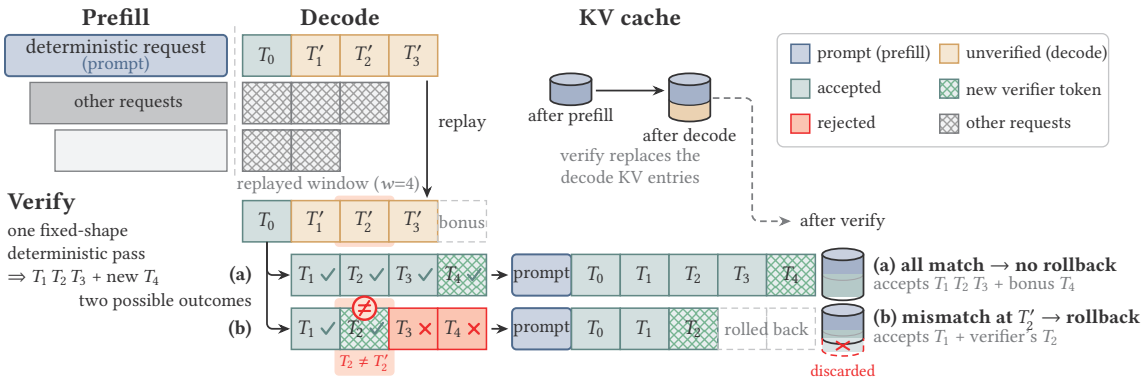


Figure 8. An example of decode-verify-rollback. After generating a fixed number of tokens through regular decoding, LLM-42 verifies them in parallel via a separate forward pass. (a) all the tokens generated in the decode phase pass verification; LLM-42 accepts all these tokens along with the new verifier-generated token (T_4). (b) some tokens do not match between decode and verification; LLM-42 accepts only the initial matching tokens (T'_1) and the following verifier-generated token (T_2); all other tokens are recomputed. In both cases, the verifier replaces the KV cache entries generated by the decode phase with its own.

by prefill that avoids arbitrary inter-request batching and is therefore deterministic. LLM-42 then generates candidate tokens T'_1, T'_2, and T'_3 using fast path with dynamic batching, where gray requests may be batched arbitrarily. The first input to the verifier should be consistent (in this case, T_0 is consistent since it comes from the prefill phase). The verifier replays these four tokens T_0 and T'_1–T'_3 as input and produces output tokens T_1–T_4. Verification has two possible outcomes: (1) all tokens pass verification, or (2) one or more tokens fail verification. We explain these two cases below.

Case-1: Verification succeeds. In the common case, verification reproduces the same tokens that the preceding decode iterations generated. For example, in Figure 8 (a), $T_1 = T'_1$, $T_2 = T'_2$ and $T_3 = T'_3$. In this case, LLM-42 accepts all these tokens; in addition, it also accepts the token T_4 since T_4 was generated by the verifier itself from a consistent state.

Case-2: Verification fails. Occasionally, a verification pass disagrees with one or more decoded tokens, e.g., only T'_1 matches with the verifier's output in Figure 8 (b). In this case, LLM-42 commits the output only up to the last matching token (T'_1); all subsequent tokens (T'_2 and T'_3) are rejected. The verifier-generated token that appears immediately after the last matching position (T_2) is also accepted and the KV cache is truncated at this position. The next decode iteration resumes from this repaired, consistent state, at the cost of recomputation for the failed tokens.

Making KV cache consistent. During the decode phase, the KV cache is populated by fast path iterations that execute under dynamic batching. Consequently, even when token verification succeeds, the KV cache corresponding to the verified window may still be inconsistent, since it was produced by nondeterministic execution. This inconsistency could affect tokens generated in future iterations even if tokens in the current window pass through the verifier. To prevent this, we overwrite the KV cache entries produced during decoding with those generated by the verifier. This way, both the emitted tokens and the KV cache state are consistent for subsequent iterations.

Guaranteed forward progress. Note that each verification pass produces at least one new consistent output token as shown in Figure 8. As a result, DVR guarantees forward progress even in contrived worst-case scenarios that might trigger rollbacks for every optimistically generated token. We have not observed any such scenario in our experiments.

Overall, DVR mirrors the structure of self-speculative decoding, but differs in several important ways as discussed in Table 1. Under DVR, decoded tokens result from a faithful execution of the model's forward pass, with deviations arising only from the floating-point rounding errors. In contrast, self-speculative execution techniques generate candidate tokens using modified forward computations e.g., by pruning attention/context [16, 23]. Because these approximations change the computation itself, errors compound quickly and hence speculation depth is typically limited to a few (2-8) tokens [16, 23, 34, 37]. In contrast, leveraging O1, DVR can safely speculate over longer (32-64) token sequences.

4.3 Grouped Verification

The efficiency of DVR depends on the size of verification window and involves a trade-off: smaller verification windows incur higher verification cost due to low compute inefficiency. At the same time, smaller windows require lower recomputation. In contrast, larger windows reduce verification cost at the expense of increased recomputation. We empirically characterize this trade-off by profiling these costs for Llama-3-8B and Llama-3-70B. We measure per-token verification cost based on the latency of verification forward pass. To quantify the recomputation cost, we execute 2048 requests from the ShareGPT dataset in an online setting at 12 queries

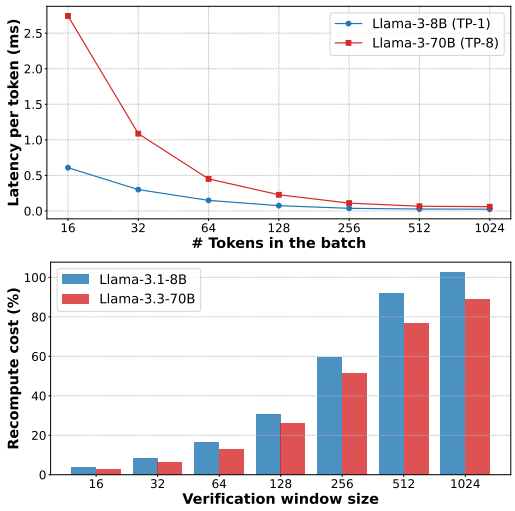


Figure 9. Verification (top) and recompute cost (bottom).

per second. For each verification window size, we measure the total number of tokens decoded versus the number of consistent tokens; the difference between the two denotes recomputation overhead.

Figure 9 (top) shows per-token verification cost as a function of verification window size. For smaller windows, the verification pass is memory-bound: each token performs very little computation, resulting in low arithmetic intensity and poor hardware utilization [11]. Consequently, the verification cost per-token is high—up to 0.6ms for Llama-3-8B and 2.7ms for Llama-3-70B. The cost is higher for Llama-3-70B because it is running on 8 GPUs which results in smaller computation per-GPU. As the verification window increases, the kernel transitions to being compute-bound, and the per-token cost drops sharply, reaching 0.025ms for Llama-3-8B and 0.06ms for Llama-3-70B at a window size of 1024, signifying a reduction of 23× for Llama-3-8B and 29× for Llama-3-70B.

Figure 9 (bottom) reports the total recomputation overhead. As expected, larger verification windows lead to higher recomputation: when a mismatch is detected, all tokens following the first mismatched position must be recomputed. This effect is clearly visible in the figure. For Llama-3-8B, the recomputation overhead is only 3.8% at a window size of 16, but rises sharply to 102% at a window size of 1024—effectively doubling the number of decoded tokens. A similar trend holds for Llama-3-70B, where the overhead increases from 3% at window size 16 to 88% at window size 1024.

To address this inherent trade-off between verification efficiency and recomputation, we propose *grouped verification*. Rather than verifying a large window for a single request (e.g., 512 tokens), we verify small, fixed-size windows across multiple requests in a single pass (e.g., 16 requests with 32 tokens each). This approach preserves the rollback efficiency of small windows for each request while retaining the throughput benefits of batched verification.

4.4 Operator-specific Discussion

This section highlights a few subtle sources of inconsistency in commonly used operators. We do not introduce new techniques here; instead, we adopt established approaches to achieve determinism and summarize them for completeness.

Attention. Attention involves reductions along multiple steps: reductions along the sequence dimension (e.g., in FlashDecoding-style sequence parallelism [21]) and reductions inside the softmax normalization step. To make it consistent, we use the FlashAttention-3 kernel with a fixed KV split tile size (4096, similar to SGLang). This is slightly suboptimal in some cases, but its effect is limited to verification passes - a small fraction of total forward passes (see Table 4). Hence, the end-to-end impact is also negligible.

Communication collectives. Classical ring-based AllReduce reduces tokens in different orders depending on their position in the batch. We use multimem-based AllReduce, introduced in CUDA 13.0, that is already batch-invariant [52].

Sampling. We adopt the sampler from SGLang. When temperature is zero, it selects the token with the highest logit (argmax) and if there are multiple maximal values then it returns the index of the first maximal value. For non-greedy sampling, we use SGLang's `multinomial_with_seed` function with a fixed seed, ensuring that the same input–seed pair always produces the same sample [45].

Overall, achieving determinism in LLM-42 requires reimplementing an operator only if the existing implementation is not position-invariant under fixed input shapes. LLM-42 also requires deterministic implementations of sampling components (e.g., sampler, top-k), but these are reused across models and incur only a one-time cost. In contrast, the dominant performance and engineering effort in LLM systems lies in optimizing GEMM, communication, and attention kernels—components that LLM-42 reuses without modification along with RMSNorm and FusedMoE kernels.

5 Evaluation

Our evaluation answers the following questions:

- How does LLM-42 compare against SGLang's deterministic and nondeterministic modes?
- How does LLM-42 perform for different mix of deterministic and nondeterministic traffic?
- How do configuration parameters affect the performance of grouped verification in LLM-42?

Models and environment: We evaluate two models: Llama-3-8B [3] and Llama-3-70B [4] in BF16. Llama-3-8B has 32 query heads, 8 KV heads, and 32 layers. Llama-3-70B has 64 query heads, 8 KV heads and 80 layers. The system has eight NVIDIA H100 GPUs, all connected via NVLink. Each

Dataset	Type	Mean	Median	Std. Dev.
ShareGPT	Input Length	304	136	491
	Output Length	192	118	212
ArXiv	Input Length	7017	6435	3479
Test Split	Output Length	198	191	74

Table 3. Datasets and their input/output context lengths.

Workload	2%	5%	10%	20%	50%	100%
ShareGPT	2.8%	6.5%	10.3%	16.0%	22.8%	25.9%
ArXiv	2.4%	3.3%	4.9%	14.8%	30.8%	35.0%
(1K,1K)	3.1%	4.0%	5.5%	8.4%	13.0%	19.2%
(1K,256)	3.1%	4.2%	5.8%	8.8%	13.6%	19.9%
(4K,1K)	2.9%	3.2%	4.2%	7.6%	14.9%	23.7%
(4K,4K)	3.0%	3.0%	3.1%	5.7%	11.2%	17.6%

Table 4. Verification passes as a percentage of total forward passes for Llama-3-70B (TP-8). Columns show fraction of traffic that requires determinism.

GPU has 80GB high-bandwidth memory and 132 SMs. The host CPU has 112 physical cores and 2TB of DRAM. We run Llama-3-8B on a single GPU (TP-1) and Llama-3-70B with tensor-parallelism across all GPUs (TP-8).

Workloads: For real workloads, we use popular datasets of ShareGPT [1] and ArXiv [19] whose characteristics are summarized in Table 3. In addition, we use synthetic workloads with varying input and output context lengths e.g., (4K, 1K) denotes a synthetic prompt of 4K input and 1K output tokens. To evaluate LLM-42 under different workload conditions, we vary the fraction of requests that require determinism between 2%, 5%, 10%, 20%, 50%, and 100%. Higher deterministic ratios are included for stress testing; we expect that only a small fraction (typically <10%) of requests will require determinism in practical deployments. Finally, note that deterministic ratio applies only to LLM-42; approaches based on batch-invariant computation support only global determinism. All experiments run 2048 requests.

Baselines: We implement LLM-42 in SGLang (v0.5.6) and use it as our primary baseline. We compare against two deterministic variants of SGLang: SGLang-det-triton uses Triton-based batch-invariant kernels [35] and SGLang-det-deepgemm uses DeepGEMM kernels [8]. To understand the upper limit on performance, we also compare with the nondeterministic version of SGLang referred to as SGLang-nondet. All experiments in §5.1 and §5.2 set LLM-42's per-request verification window size to 32 and number of requests per verification pass to 16, informed by our observations (§4.3) and ablation study; §5.3 evaluates the effect of varying these parameters separately. We validated SGLang's and LLM-42's determinism empirically by evaluating 90K requests at various load conditions and ensuring that each prompt produces the same output across all runs.

To show that the overhead of determinism is not unique to SGLang, we also present some results for vLLM (v0.18.0) in both deterministic and nondeterministic modes referred to as vLLM-det and vLLM-nondet. We find that for distributed inference in deterministic mode, vLLM uses inefficient tree-based NCCL AllReduce implementation with static configurations (1 thread, 1 channel). This implementation achieves $10-20\times$ lower bandwidth than default NCCL AllReduce for Llama-3-70B model (TP-8), causing unacceptable performance loss. For fairer benchmarking, we modify vLLM-det to use multi-mem based AllReduce and refer to it as vLLM-det-multimem.

Metrics: In offline inference, we evaluate throughput as the number of tokens processed per second. For online inference, we evaluate end-to-end request execution latency and time-to-first-token (TTFT).

5.1 Offline Inference

We evaluate six workload configurations: two real traces (ArXiv and ShareGPT) and four synthetic workloads with varying input and output lengths. To capture the efficacy of LLM-42 on longer decoding horizons, we evaluate workload with up to 4K output tokens per-request. Figure 10 reports throughput; we summarize the key observations below.

SGLang. First, kernel choice matters for smaller models: on Llama-3-8B, SGLang-det-deepgemm is consistently faster than SGLang-det-triton, though both remain slower than nondeterministic execution (up to 18% and 51%); this advantage largely disappears for Llama-3-70B, where increased parallelism makes operations leaner and both deterministic variants perform significantly below the nondeterministic baseline (up to 45% and 52%). Second, the relative cost of determinism decreases with longer contexts, as attention dominates runtime and bottleneck shifts to memory bandwidth rather than compute. Finally, SGLang does not support selective determinism; deterministic execution is enforced globally—so even a small fraction of requests requiring determinism causes global slowdown.

vLLM. vLLM shows a similar determinism penalty as SGLang, consistently falling behind its nondeterministic baseline. The gap is especially pronounced for Llama-3-70B, where its default AllReduce kernel introduces significant communication overheads under deterministic execution. It degrades the end-to-end serving throughput by more than 80% in most cases. Enabling multi-mem AllReduce removes much of this communication bottleneck, but a noticeable gap of up to 28% still remains compared to vLLM-nondet.

LLM-42. LLM-42 closely tracks nondeterministic performance when deterministic traffic is low: at 0–10% deterministic traffic, its throughput is typically within 5–10% of

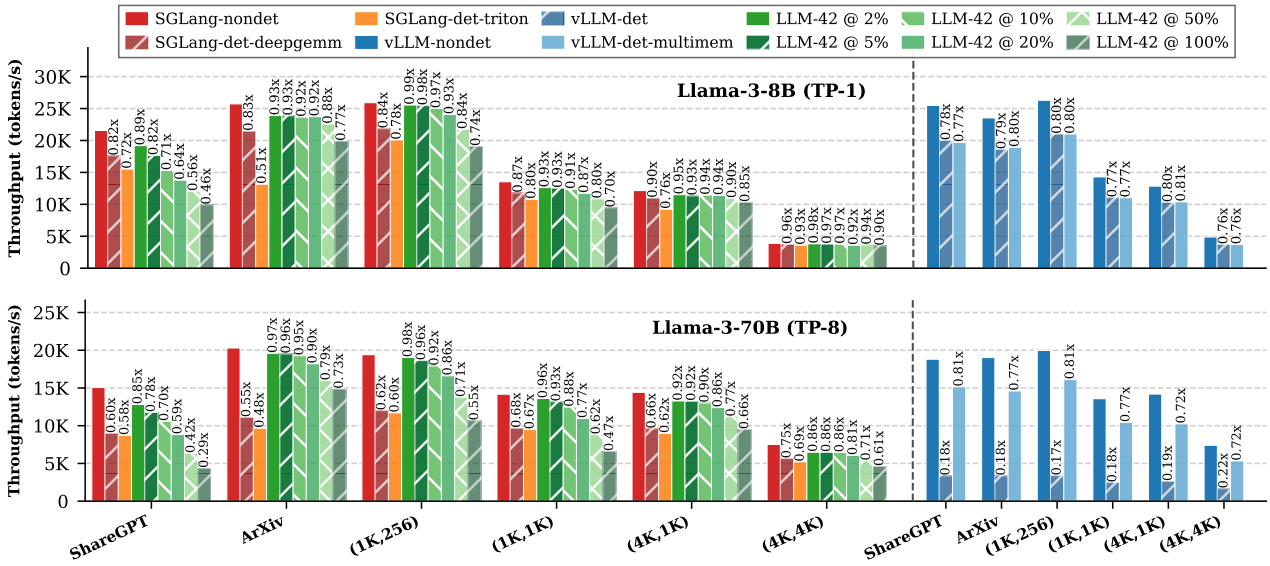


Figure 10. Throughput in offline inference. SGLang has only two modes: all requests run in either deterministic or nondeterministic mode whereas LLM-42 supports selective determinism (% values in the legend reflect the fraction of traffic that requires deterministic output).

SGLang-nondet across both models, and remains close even at 20%. The overhead is mainly due to verification passes: at low deterministic traffic, LLM-42 requires only 3–9% more forward passes than SGLang, but this increases with higher deterministic traffic to up to 35% (see Table 4). LLM-42's throughput degrades gradually as the deterministic fraction increases, with a more visible drop at 50% and the largest gap at 100%, consistent with its selective determinism. Even at 50% deterministic traffic, LLM-42 remains comparable to or better than SGLang-det-deepgemm in most workloads. One exception is ShareGPT, where LLM-42 is slower, especially at higher deterministic ratios. This workload contains many short prefills; since LLM-42 does not batch prefills of deterministic requests, it frequently executes them at low efficiency. This effect diminishes as context lengths increase. Further, LLM-42's performance trend is stable even for workloads with very long outputs such as (4K, 4K) workload.

Takeaway. LLM-42 bridges the gap between deterministic and nondeterministic execution. At up to 20% deterministic traffic, it delivers roughly 1.2–1.5× higher throughput than SGLang-det-deepgemm across most workloads, with larger gains for bigger models. The gap narrows as the deterministic fraction approaches 100%, but LLM-42 remains competitive in most cases. LLM-42 is also robust in long output scenarios.

5.2 Online Inference

Figure 11 shows the CDF of end-to-end latency under increasing load for both Llama-3-8B (top) and Llama-3-70B (bottom). The workload contains a mix of short context (ShareGPT), long context (ArXiv) and reasoning requests (generating up to 2K tokens each). We compare against SGLang-nondet and the faster deterministic baseline, SGLang-det-deepgemm. Furthermore, we run these experiments at load near the serving capacity of the baseline: below this range GPUs are heavily underutilized, while above it TTFT becomes unacceptably high. Therefore, evaluating lower or higher QPS is not particularly meaningful.

SGLang-det-deepgemm exhibits a clear rightward shift with a heavy tail, indicating substantially higher median and tail latencies than SGLang-nondet. This gap appears even for the 8B model: at QPS=4, it reaches 10.5s (P50) and 107s (P99), compared to 4.60s and 52.6s for SGLang-nondet, and widens further at higher load. The effect is more pronounced for the 70B model, where at QPS=4, SGLang-det-deepgemm reaches 189.5s (P50) and 685.0s (P99), while SGLang-nondet remains much lower. Overall, SGLang-nondet provides a practical lower bound, while deterministic execution incurs a large and growing penalty.

LLM-42 bridges this gap: it closely tracks SGLang-nondet when deterministic traffic is small, and degrades smoothly as the fraction increases. For the 8B model at QPS=5, LLM-42 achieves 11.9s (P50) and 109.7s (P99) at 2% deterministic traffic—close to SGLang-nondet and well below SGLang-det-deepgemm at the tail. Even at 20%, LLM-42 maintains a P99 of 128.7s versus 146.4s of SGLang-det-deepgemm. At higher fractions (e.g., 50%), LLM-42 begins to exceed the deterministic baseline on 8B, indicating earlier saturation.

For the 70B model, however, LLM-42 consistently outperforms SGLang-det-deepgemm across all traffic mixes. Even at 100% deterministic traffic, LLM-42 achieves 134.1s (P50) and 554.4s (P99), compared to 189.5s and 685.0s for SGLang-det-deepgemm, showing better performance at larger scales.

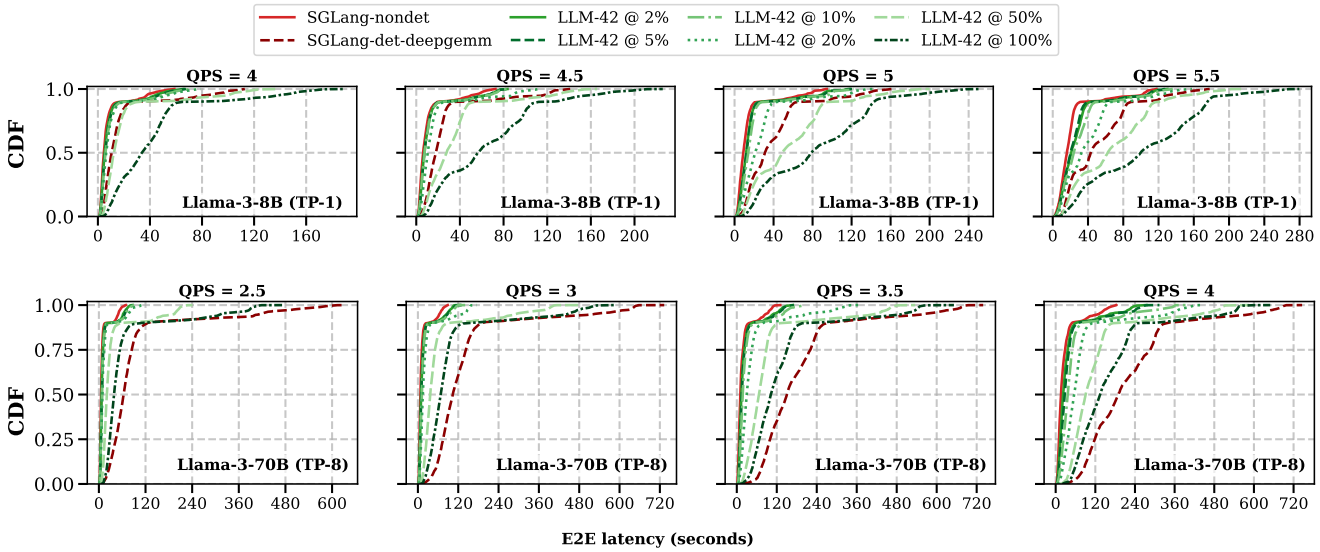


Figure 11. CDF of request end-to-end latency in online inference with varying load.

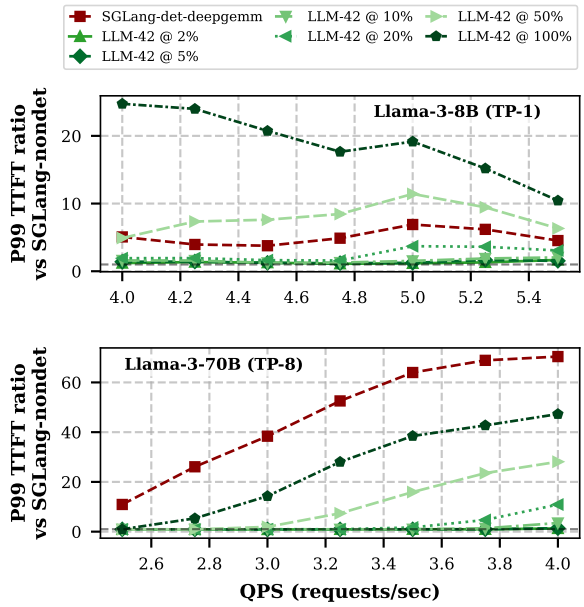


Figure 12. Impact of determinism on TTFT.

We observed similar trends in TTFT latency. At QPS=5.25 for Llama-3-8B, the median TTFT of SGLang-nondet and SGLang-det is 2.6 seconds and 22.7 seconds whereas LLM-42's median TTFT varies from 3.4 seconds at 2% to 64.9 seconds at 100% deterministic ratios. At QPS=4 for Llama-3-70B, the median TTFT of SGLang-nondet and SGLang-det is 0.84 seconds and 112 seconds whereas LLM-42's median TTFT varies from 0.94 seconds at 2% to 64.4 seconds at 100% deterministic ratios.

Figure 12 reports P99 TTFT normalized to SGLang-nondet and reflects a similar trend. For the 8B model, LLM-42 incurs modest increase in tail TTFT at low deterministic fractions (1.67× at 2%, 2.04× at 10%), remaining far below SGLang-det-deepgemm (4.53×) as long as deterministic traffic in less than 20%. At higher fractions, TTFT increases sharply (e.g., 10.4× at 100%), reflecting that the load exceeded LLM-42's serving capacity in this case but LLM-42 still outperforms SGLang-det-deepgemm at low deterministic traffic. Higher TTFT in deterministic settings is mostly due to higher scheduling delays. For Llama-3-70B, LLM-42 achieves much lower TTFT that SGLang-det-deepgemm in all cases whereas SGLang-det-deepgemm increases TTFT by up to 70×.

Takeaway. Determinism in SGLang comes with a substantial latency penalty, especially at high load. LLM-42 removes most of this gap: it stays close to SGLang-nondet at low deterministic fractions and significantly outperforms SGLang-det-deepgemm across a wide range of settings, while degrading gracefully as the fraction of deterministic traffic increases.

5.3 Ablation Study

Grouped verification reduces both verification and recomputation cost in LLM-42. It is governed by (1) the per-request verification window size and (2) the number of requests verified together. We ablate both on ShareGPT by varying the window size from 16 to 1024 tokens and the group size from 1 to 64 requests, restricting both to powers of two for simplicity (though any positive integer is valid). All requests are deterministic. Figure 13 shows recompute cost of the left-side heatmaps and total throughput normalized to the best LLM-42 configuration on the right-side heatmaps.

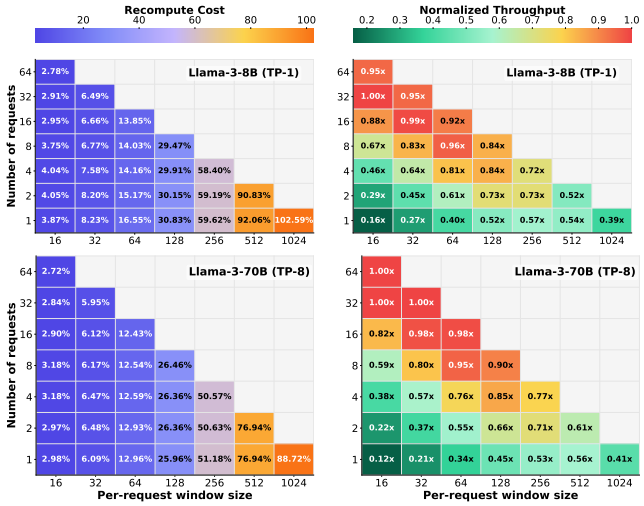


Figure 13. The effect of different verification strategies on recompute cost (left) and throughput (right).

Without grouping (batch size 1; first rows in the heatmaps), throughput is non-monotonic in window size. The smallest window of 16 tokens requires minimum rollbacks (2-4%), yet achieves only 0.16× and 0.12× of peak LLM-42 throughput due to frequent, inefficient verification passes. Increasing the window size initially helps but overly large windows hurt: 8B model peaks at window 256 (0.57× of global peak) and 70B model at 512 (0.56×), dropping to 0.39× and 0.41× at window size of 1024. This reflects the verification–recomputation trade-off: small windows increase verification overhead, while large windows amplify recomputation (up to 8.23% for 8B model and 102% for 70B model).

Grouped verification substantially improves throughput without inflating recomputation (all rows above the first). Even batching four requests significantly increases throughput e.g., it reaches up to 0.84× and 0.85× of peak throughput for 8B and 70B models, respectively, with further gains as batching increases. Multiple configurations are near-optimal: for Llama-3-8B, (window size 16, num requests 32) and (window size 32, num requests 16) both achieve near-peak throughput, and for Llama-3-70B, five configurations lie within 2% of the peak throughput.

Takeaway. LLM-42's performance is not highly sensitive to the exact choice of parameters: as long as the per-request window is not excessively large (to avoid recomputation blow-up) and the total number of verified tokens per step is not too small (to avoid under-utilization), a broad range of configurations can yield similar performance.

5.4 Mixture-of-experts Models

When we evaluated SGLang deterministic baselines (both SGLang-det-triton and SGLang-det-deepgemm) on MoE models, we found that these implementations are not deterministic! Figure 14 quantifies SGLang's nondeterminism with a

QPS	6	10	14	18
6	–	–	–	–
10	662	–	–	–
14	300	393	–	–
18	665	11	392	–

Number of requests with non-unique samples at different QPS value pairs in SGLang-det.

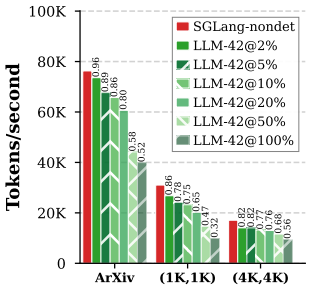


Figure 14. MoE model evaluation on Qwen3-30B-A3B.

benchmark wherein we ran 10K requests at different load on Qwen3-30B-A3B [5, 50] deployed with TP-4. It shows that SGLang-det-deepgemm produces non-unique samples for many requests across different runs (we obtained similar results for SGLang-det-triton). In contrast, LLM-42 is deterministic; each request produces a unique sample at all loads. LLM-42's performance trend is consistent with those seen in other models: it operates at 0.96× and 0.89× the throughput of SGLang-nondet at 2% and 5% deterministic traffic for ArXiv, with performance degrading slowly as the fraction of deterministic traffic increases. Importantly, these results suggest that by reducing kernel-level constraints, LLM-42 makes it easier to achieve determinism for different models.

6 Discussion

In this section, we briefly discuss some alternative approaches to LLM-42, the challenges inherent to these approaches, and a recently proposed method for improving the efficiency of verified speculation in LLM-42.

6.1 Deterministic Record and Replay

Since variations in batch size are the primary cause of nondeterminism, one could consider logging batch sizes and then replaying the exact batch sizes during different runs of the same request. While this method would achieve determinism, it is not clear how a logging-based deterministic replay scheme could be deployed efficiently. Such a system would require logging (1) the prompt, (2) the model's temperature setting, and (3) the batch sizes used by the model to handle the prompt. The key issue is with item (3) – we need to log the batch sizes not only to handle the input prompt but also each of the thousands of output-token iterations (each of which can have a unique batch size because of iterative scheduling!) and then replay that exact setting to recreate the deterministic response. However, replaying a fixed batch size at each iteration fundamentally conflicts with dynamic batching that adds/removes requests dynamically in each iteration. If the recorded batch size is higher than the in-flight batch size, it would require padding, wasting compute. On the contrary, if the recorded batch size is lower, then one

or more in-flight requests would need to be deferred, preempted, or migrated. Managing these scenarios will likely add significant complexity to a serving system.

6.2 Split Deterministic and Nondeterministic Traffic

LLM-42 enables a single replica to serve deterministic and nondeterministic traffic. An alternate approach is to disaggregate the two types of traffic and serve each from separate GPU pools. While splitting GPUs this way is feasible, it may cause inefficiencies. Consider a cloud provider serving tens or hundreds of models. There would be a few head models that serve a large fraction of overall traffic and many tail models that each serve a modest amount of traffic. For tail models that are served by few replicas, provisioning another replica just to handle a fraction of the model's requests will result in poor utilization and may not make economic sense. In contrast, LLM-42 can easily absorb the deterministic load for these models within the existing replicas.

Assuming the separate pools are optimally provisioned, let p denote the fraction of deterministic requests, and let T_n and T_d denote the throughput of SGLang-nondet and SGLang-det, respectively. The expected service time for a randomly selected request is

$$\frac{p}{T_d} + \frac{1-p}{T_n}. \tag{1}$$

Hence, the aggregate throughput of the partitioned deployment is

$$T_{\text{Sep}}(p) = \left(\frac{p}{T_d} + \frac{1-p}{T_n}\right)^{-1}. \tag{2}$$

which is the weighted harmonic mean of T_d and T_n. LLM-42 should be deployed whenever its throughput exceeds that of the separate-pool approach, i.e., $T_{\text{LLM-42}}(p) > T_{\text{Sep}}(p)$. Based on our experiments, this condition generally holds for $p < 0.5$, suggesting that LLM-42 is preferable unless the majority of traffic is deterministic.

6.3 Efficient Opportunistic Verification

LLM-42's throughput drops significantly when majority of the traffic is deterministic due to its always-on verifier. However, this cost is not fundamental. Recent work MarginGate [18] shows that this overhead can be reduced by 2.23× on Llama-3.1-8B and 1.99× on Qwen2.5-14B through opportunistic verification. The key observation is that batch-induced token flips occur almost exclusively at decode steps with a very small top-1/top-2 logit margin; however, to conservatively capture these rare flips, MarginGate marks a larger set of low-margin steps for verification. While only 0.3–1.3% of decode steps actually experience a flip, a conservative margin threshold identifies roughly 15–19% of steps as potentially vulnerable, allowing verification to be skipped on the remaining high-margin steps. With this optimization, LLM-42's verifier need not run on every step.

7 Related Works

Recent advances in LLM inference systems have focused on optimizing throughput, latency, and resource utilization [11, 28, 30, 32, 41, 49, 55, 63]. Orca [55] pioneered continuous batching, enabling low-latency serving by dynamically scheduling requests at the granularity of individual iterations. vLLM [32] introduced PagedAttention, a memory management abstraction that allows higher batch sizes and reuse of KV cache across requests, dramatically improving system throughput. Sarathi-Serve [11] further improved the throughput–latency trade-off through chunked prefills to better exploit GPU parallelism. DistServe [63] and Splitwise [40] proposed disaggregated inference architectures that separate prefill and decode workloads across specialized servers, mitigating interference. Additionally, compute–communication overlap techniques such as Comet [58], FlashOverlap [26], TileLink [62], TokenWeave [24], Syncopate [42], etc., hide communication latency in distributed LLM inference by overlapping it with computation, improving both latency and throughput. Complementary lines of work such as speculative decoding [17, 33, 36] and FlexGen [47] pursue orthogonal optimizations by reducing the number of autoregressive steps or offloading memory to CPU DRAM and SSDs. While these systems improve performance, they leave the trade-offs between determinism and efficiency unexplored.

Enabling determinism in LLMs is drawing increasing attention [18, 64]. Song et al. [48] argued that nondeterminism can distort evaluation results, calling for reproducibility-aware benchmarking. Yuan et al. [56] linked reproducibility failures to mixed-precision arithmetic and fused kernel inconsistencies, showing their impact on multi-step reasoning accuracy. Rainbird AI [13] emphasized deterministic inference as a requirement for traceability and compliance in enterprise settings and safety-critical domains.

Recent work by Zhang et al. [60] proposes tensor-parallel invariant kernels that eliminate training–inference mismatch by ensuring bitwise deterministic results across different tensor parallel sizes for LLM inference. We find that Multimem-based communication kernels are already usable for deterministic inference in multi-GPU settings.

8 Conclusion

Enabling determinism in LLM inference is tedious today. Existing systems achieve determinism through batch-invariant computation—an approach that requires rewriting GPU kernels at a time when both hardware and model architectures are evolving rapidly. Moreover, batch-invariant kernels impose a global performance penalty, even on requests that do not require determinism. We present LLM-42, a simpler alternative that minimizes the need for new kernel implementations while supporting selective determinism.

References

[1] 2023. ShareGPT_Vicuna_unfiltered. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered/resolve/main/ShareGPT_V3_unfiltered_cleaned_split.json.

[2] 2024. CUTLASS Tutorial: Mastering the NVIDIA® Tensor Memory Accelerator (TMA). https://research.colfax-intl.com/tutorial-hopper-tma/.

[3] 2024. Meta-Llama-3.1-8B. https://huggingface.co/meta-llama/Meta-Llama-3-8B.

[4] 2024. Meta-Llama-3.3-70B. https://huggingface.co/meta-llama/Meta-Llama-3-70B.

[5] 2025. Qwen/Qwen3-30B-A3B. https://huggingface.co/Qwen/Qwen3-30B-A3B.

[6] 2025. SGLang batch-invariant ops. https://github.com/sgl-project/sglang/blob/main/python/sglang/srt/batch_invariant_ops/batch_invariant_ops.py.

[7] 2025. SGLang layernorm. https://github.com/sgl-project/sglang/blob/main/python/sglang/srt/layers/layernorm.py.

[8] 2025. Support DeepGEMM for deterministic inference. https://github.com/sgl-project/sglang/pull/12142.

[9] 2025. vLLM Fused MoE. https://github.com/vllm-project/vllm/blob/main/vllm/model_executor/layers/fused_moe/fused_moe.py.

[10] 2025. vLLM RMSNorm. https://github.com/vllm-project/vllm/blob/main/csrc/layernorm_kernels.cu.

[11] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[12] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. 2023. SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills. arXiv:2308.16369 [cs.LG] https://arxiv.org/abs/2308.16369

[13] Rainbird AI. 2025. *Deterministic Graph-Based Inference for Guardrailing Large Language Models*. Technical Report. Rainbird Technologies Ltd. https://rainbird.ai/wp-content/uploads/2025/03/Deterministic-Graph-Based-Inference-for-Guardrailing-Large-Language-Models.pdf Accessed: 2025-10-28.

[14] Shyamal Anadkat and Thinking Machines Lab. 2023. How to make your completions outputs consistent with the new seed parameter. https://cookbook.openai.com/examples/reproducible_outputs_with_the_seed_parameter#model-level-features-for-consistent-outputs

[15] Berk Atıl, Sarp Aykent, Alexa Chittams, Lisheng Fu, Rebecca J. Passonneau, Evan Radcliffe, Guru Rajan Rajagopal, Adam Sloan, Tomasz Tudrej, Ferhan Ture, Zhe Wu, Lixinyu Xu, and Breck Baldwin. 2025. Non-Determinism of "Deterministic" LLM System Settings in Hosted Environments. In *Proceedings of the 5th Workshop on Evaluation and Comparison of NLP Systems*, Mousumi Akter, Tahiya Chowdhury, Steffen Eger, Christoph Leiter, Juri Opitz, and Erion Çano (Eds.). Association for Computational Linguistics, Mumbai, India, 135–148. https://aclanthology.org/2025.eval4nlp-1.12/

[16] 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 *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 203, 27 pages.

[17] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating Large Language Model Decoding with Speculative Sampling. *arXiv preprint arXiv:2302.01318* (2023). https://arxiv.org/abs/2302.01318

[18] Kexin Chu, Yang Zhou, and Wei Zhang. 2026. MarginGate: Sparse Margin-Triggered Verification for Batch-Invariant LLM Inference. arXiv:2605.30218 [cs.LG] https://arxiv.org/abs/2605.30218

[19] Arman Cohan, Franck Dernoncourt, Doo Soon Kim, Trung Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. 2018. A Discourse-Aware Attention Model for Abstractive Summarization of Long Documents. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Association for Computational Linguistics, New Orleans, Louisiana, 615–621. doi:10.18653/v1/N18-2097

[20] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=mZn2Xyh9Ec

[21] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. 2023. Flash-Decoding for long-context inference. https://crfm.stanford.edu/2023/10/12/flashdecoding.html.

[22] Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, Ahmed Aly, Beidi Chen, and Carole-Jean Wu. 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)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 12622–12642. doi:10.18653/v1/2024.acl-long.681

[23] 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* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 561, 20 pages.

[24] Raja Gond, Nipun Kwatra, and Ramachandran Ramjee. 2025. TokenWeave: Efficient Compute-Communication Overlap for Distributed LLM Inference. *arXiv preprint arXiv:2505.11329* (2025). https://arxiv.org/abs/2505.11329

[25] Horace He and Thinking Machines Lab. 2025. Defeating Nondeterminism in LLM Inference. https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

[26] Ke Hong, Xiuhong Li, Minxu Liu, Qiuli Mao, Tianqi Wu, Zixiao Huang, Lufang Chen, Zhong Wang, Yichong Zhang, Zhenhua Zhu, Guohao Dai, and Yu Wang. 2025. Efficient and Adaptable Overlapping for Computation and Communication via Signaling and Reordering. *arXiv preprint arXiv:2504.19519* (2025).

[27] Adnan Hoque, Less Wright, Chih-Chieh Yang, Mudhakar Srivatsa, and Raghu Ganti. 2024. Accelerating a Triton Fused Kernel for W4A16 Quantized Inference with SplitK work decomposition. arXiv:2402.00025 [cs.DC] https://arxiv.org/abs/2402.00025

[28] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2024. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. arXiv:2401.11181 [cs.DC] https://arxiv.org/abs/2401.11181

[29] Tanmay Joshi, Shourya Aggarwal, Anusa Saha, Aadi Pandey, Shreyash Dhoot, Vighnesh Rai, Raxit Goswami, Aman Chadha, Vinija Jain, and Amitava Das. 2026. Stochastic CHAOS: Why Deterministic Inference Kills, and Distributional Variability Is the Heartbeat of Artificial Cognition. arXiv:2601.07239 [cs.AI] https://arxiv.org/abs/2601.07239

[30] Aditya K. Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. 2025. POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Association for Computing Machinery, New York, NY, USA, 897–912. https://doi.org/10.1145/3676641.3715996

[31] Sungwoo Kim, Divya Patel, and Tian Xu. 2025. A Comprehensive Analysis of Large Language Model Outputs: Similarity, Diversity, and

Bias. *arXiv preprint arXiv:2505.09056* (2025). https://arxiv.org/abs/2505.09056

[32] 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* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[33] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2022. Fast Inference from Transformers via Speculative Decoding. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*. https://arxiv.org/abs/2211.17192

[34] 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* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 795, 13 pages.

[35] Thinking Machines. 2025. Batch Invariant Ops. https://github.com/thinking-machines-lab/batch_invariant_ops/tree/main. (2025).

[36] Jonathan Mamou, Oren Pereg, Daniel Korat, Moshe Berchansky, Nadav Timor, Moshe Wasserblat, and Roy Schwartz. 2024. Dynamic Speculation Lookahead Accelerates Speculative Decoding of Large Language Models. In *Proceedings of The 4th NeurIPS Efficient Natural Language and Speech Processing Workshop (PMLR 262)*. 456–467. https://proceedings.mlr.press/v262/mamou24a.html

[37] 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 932–949. doi:10.1145/3620666.3651335

[38] NVIDIA. 2019. CUTLASS: Fast Linear Algebra in CUDA. https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/. NVIDIA Developer Blog.

[39] Charlie Parker. 2025. How can I ensure deterministic text generation with vLLM, and does it support a global set_seed? https://stackoverflow.com/questions/79467847/how-can-i-ensure-deterministic-text-generation-with-vllm-and-does-it-support-a.

[40] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Inigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 118–132. doi:10.1109/ISCA59077.2024.00019

[41] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1133–1150. doi:10.1145/3669940.3707256

[42] Xinwei Qiang, Yue Guan, Zhengding Hu, Yufei Ding, and Adnan Aziz. 2026. Syncopate: Efficient Multi-GPU AI Kernels via Automatic Chunk-Centric Compute-Communication Overlap. arXiv:2601.20595 [cs.DC] https://arxiv.org/abs/2601.20595

[43] Ranajoy Sadhukhan, Jian Chen, Zhuoming Chen, Vashisth Tiwari, Ruihang Lai, Jinyuan Shi, Ian En-Hsu Yen, Avner May, Tianqi Chen, and Beidi Chen. 2024. Magicdec: Breaking the latency-throughput tradeoff for long context generation with speculative decoding. *arXiv preprint arXiv:2408.11049* (2024).

[44] sglang. 2025. Support deterministic inference with Batch Invariant Ops. https://github.com/sgl-project/sglang/issues/10278

[45] Team SGLang. 2025. Towards Deterministic Inference in SGLang and Reproducible RL Training. https://lmsys.org/blog/2025-09-22-sglang-deterministic/.

[46] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. arXiv:2407.08608 [cs.LG] https://arxiv.org/abs/2407.08608

[47] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. arXiv:2303.06865 [cs.LG]

[48] Yifan Song, Guoyin Wang, Sujian Li, and Bill Yuchen Lin. 2025. The Good, The Bad, and The Greedy: Evaluation of LLMs Should Not Ignore Non-Determinism. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Luis Chiruzzo, Alan Ritter, and Lu Wang (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, 4195–4206. doi:10.18653/v1/2025.naacl-long.211

[49] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1348–1362. doi:10.1109/HPCA61900.2025.00102

[50] Qwen Team. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[51] vllm. 2025. Batch-invariant Inference. https://github.com/orgs/vllm-project/projects/29

[52] vLLM. 2025. Same data all reduce on H20, but results are different. https://github.com/NVIDIA/nccl/issues/1497#issuecomment-3210819243

[53] Team vLLM. 2025. Batch Invariance. https://docs.vllm.ai/en/latest/features/batch_invariance/.

[54] Bram Wasti. 2025. Deepseek-v3 Batch Invariant on 8xH100. https://github.com/vllm-project/vllm/pull/26609.

[55] 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)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[56] Jiayi Yuan, Hao Li, Xinheng Ding, Wenya Xie, Yu-Jhe Li, Wentian Zhao, Kun Wan, Jing Shi, Xia Hu, and Zirui Liu. 2025. Give Me FP32 or Give Me Death? Challenges and Solutions for Reproducible Reasoning. *arXiv preprint arXiv:2506.09501* (2025). https://arxiv.org/abs/2506.09501

[57] 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)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 11263–11282. doi:10.18653/v1/2024.acl-long.607

[58] Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, Quan Chen, and Xin Liu. 2025. Comet: Fine-grained Computation-communication Overlapping for Mixture-of-Experts. arXiv:2502.19811 [cs.DC] https://arxiv.org/abs/2502.19811

[59] Wei Zhang, Rui Li, and Hao Chen. 2024. Integrating Randomness in Large Language Models. *arXiv preprint arXiv:2407.03582* (2024). https://arxiv.org/abs/2407.03582

[60] Ziyang Zhang, Xinheng Ding, Jiayi Yuan, Rixin Liu, Huizi Mao, Jiarong Xing, and Zirui Liu. 2025. Deterministic Inference across Tensor Parallel Sizes That Eliminates Training-Inference Mismatch.

arXiv:2511.17826 [cs.LG] https://arxiv.org/abs/2511.17826

[61] Yilong Zhao, Jiaming Tang, Kan Zhu, Zihao Ye, Chi-Chih Chang, Chaofan Lin, Jongseok Park, Guangxuan Xiao, Mohamed S Abdelfattah, Mingyu Gao, et al. 2025. Accelerating Large-Scale Reasoning Model Inference with Sparse Self-Speculative Decoding. *arXiv preprint arXiv:2512.01278* (2025).

[62] Size Zheng, Jin Fang, Xuegui Zheng, Qi Hou, Wenlei Bao, Ningxin Zheng, Ziheng Jiang, Dongyang Wang, Jianxi Ye, Haibin Lin, Li-Wen Chang, and Xin Liu. 2025. TileLink: Generating Efficient Compute-Communication Overlapping Kernels using Tile-Centric Primitives. arXiv:2503.20313 [cs.DC] https://arxiv.org/abs/2503.20313

[63] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

[64] Zhenting Zhu, Lucas Thai, Shan Yu, Yicheng Liu, Yifan Qiao, Chenxi Wang, Harry Xu, and Junyi Shu. 2026. Demystifying Numerical Instability in LLM Inference: Achieving Reproducible Inference for Mission-Critical Tasks with HEAL. *arXiv preprint arXiv:2606.21023* (2026).