LETHE: FORGETTING IRRELEVANT CONTEXT FOR EFFICIENT PREFILL IN RAG AND TOOL-CALLING

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic, retrieval-augmented generation (RAG), and tool-calling applications repeatedly process largely unchanged context, yet standard KV caching only applies when the reused context forms an identical prefix. We introduce **LETHE**, a training-free method for non-prefix KV reuse. Rather than caching chunks independently and later reconstructing missing cross-attention, LETHE builds a shared cache in which chunks already interact, then selectively corrects the influence of chunks excluded from the current request. An offline divergence analysis is performed once per chunk to identify tokens that should be recomputed at runtime, while the remaining tokens are reused from cache. The same mechanism applies to both retrieved text and tool definitions without modifying or fine-tuning the base model. Across Qwen3-8B, Qwen3-14B, and Llama-3.1-8B-Instruct, LETHE recomputes as little as 4.9% of context tokens and reduces full-prompt FLOPs by 50.7–87.5%. On HotpotQA, the best evaluated LETHE setting for each model is within 2.8 EM points of full recomputation and LETHE-K outperforms EPIC-16 by 7.1 EM points on average; on BFCL, the best LETHE setting is within 1.5 percentage points of full-recomputation function-call accuracy across all three models. With a fixed 10% recomputation budget, LETHE achieves 5.3–6.3$\times$ wall-clock prefill speedup at 10k context tokens on a single RTX A6000 GPU. Overall, across three models and both text retrieval and tool calling, LETHE preserves near-baseline task quality while substantially reducing computation, with long-context prefill speedups reaching about 6$\times$.

1 INTRODUCTION

Agentic, retrieval-augmented generation (RAG), and tool-calling applications repeatedly process large contexts that change only slightly between requests (Lewis et al., 2020; Schick et al., 2023; Patil et al., 2025). Standard KV caching is effective when requests share an identical prefix (Kwon et al., 2023; Zheng et al., 2024), but it does not directly handle the non-prefix case in which a query selects a different subset of previously seen passages or tool definitions. Once the selected chunks change, the cached states no longer match the cross-attention produced by processing the new combination from scratch.

We introduce **LETHE**, a training-free method for efficient non-prefix KV reuse. Most prior methods begin with chunks cached independently and then try to reconstruct cross-chunk interaction that was never computed. LETHE takes the opposite approach: it builds a shared cache in which chunks have already interacted causally, then selectively corrects the residual influence of chunks excluded from the current request. The key asymmetry is therefore *missing relevant interaction* versus *extra irrelevant interaction*. LETHE starts with the former already present and attempts to remove only the most harmful part of the latter.

For each chunk, LETHE compares its representation inside the shared cache with a solo version of the same chunk and stores a per-token divergence ranking. This analysis is performed once when the chunk enters the cache. At query time, requested chunks are extracted, RoPE-repositioned, and only the highest-divergence tokens are recomputed. LETHE-K and LETHE-M derive adaptive per-chunk budgets from the divergence distribution; fixed P10/P20/P30 variants expose explicit compute budgets. The same mechanism applies to retrieved text and structured tool definitions without modifying or fine-tuning the model.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Table 1: Comparison with closely related non-prefix KV reuse methods. **Once per chunk**: the token-selection signal is computed a single time for each chunk, rather than freshly for every incoming request. **Adaptive budget**: the recompute ratio is derived per chunk from that chunk's own statistics, rather than one fixed ratio applied everywhere. **No training**: the method requires no fine-tuning or additional pretraining of the base model. **Direct recompute**: Selected tokens are recomputed using the base model's ordinary forward operations rather than a learned or blended KV correction. **Tool-calling support**: the method is demonstrated or described as applying directly to tool/function-definition chunks, not only free-text retrieval chunks. ✓ = yes, × = no.

	Once per chunk	Adaptive budget	No training	Direct recompute	Tool-calling support
EPIC Hu et al. (2025)	✓	×	✓	✓	×
CacheBlend Yao et al. (2025)	×	×	✓	✓	×
CacheSlide Liu et al. (2026)	×	×	×	×	×
CacheTune Li et al. (2026)	✓	×	✓	✓	×
AgentKVShift Pandey et al. (2026)	✓	×	✓	×	×
LETHE (ours)	✓	✓	✓	✓	✓

CacheSlide evaluates one agent (SWE-Agent-Bench) involving general tool use (file edits, shell commands via an agent-computer interface), but not structured function-definition chunks in the sense evaluated here (e.g. BFCL-style JSON tool schemas); we mark it × on that basis.

Across Qwen3-8B, Qwen3-14B, and Llama-3.1-8B-Instruct on HotpotQA and BFCL, LETHE recomputes as little as 4.9% of context tokens and reduces full-prompt FLOPs by 50.7–87.5% while remaining close to full recomputation accuracy. In a controlled prefill-scaling benchmark with a 10% recomputation budget, LETHE achieves 5.3–6.3× wall-clock speedup at 10,000 context tokens across all three models on a single RTX A6000 GPU. On HotpotQA, LETHE outperforms EPIC by 7.1 EM points on average at comparable or lower recomputation cost. These results suggest that selectively *forgetting* irrelevant context can be an effective alternative to rebuilding missing interactions from independently cached chunks.

2 Related Work

Strict-prefix KV reuse is well established in systems such as PagedAttention (Kwon et al., 2023) and RadixAttention in SGLang (Zheng et al., 2024). The harder case is *non-prefix reuse*, where RAG or agentic requests assemble different subsets of previously seen chunks. Existing methods mainly differ in how they repair the resulting cross-attention mismatch and when they decide which tokens need correction. Table 1 summarizes the closest approaches.

Independent-cache reuse and correction. PromptCache (Gim et al., 2024) concatenates independently cached prompt modules without correction. CacheBlend (Yao et al., 2025) uses the current request to identify high-deviation tokens and recomputes a fixed fraction, while CacheSlide (Liu et al., 2026) periodically re-evaluates token importance and uses a learned positional/correction mechanism. These approaches either accept missing cross-chunk interaction or pay query-time cost to estimate how it should be repaired.

Once-per-chunk selection. CacheTune (Li et al., 2026) is closer to LETHE in timing: it computes a token-importance signal once per chunk and reuses it across requests. Its signal is intrinsic to the chunk's own KV states. LETHE instead measures a *relational* effect by comparing a chunk in isolation with the same chunk inside the shared cache. CacheTune uses a globally chosen recompute ratio for a hardware/storage setting, whereas LETHE can derive a separate budget from each chunk's divergence distribution.

Other approaches. AgentKVShift (Pandey et al., 2026) applies approximate corrections to reused agent memory. TurboRAG (Lu et al., 2025) and KVLink (Yang et al., 2025b) train models to tolerate non-prefix reuse, trading online correction for additional training. Quest (Tang et al., 2024) and ContextPilot (Jiang et al., 2026) address complementary decoding or serving-layer problems rather than LETHE's single-request cross-attention correction problem.

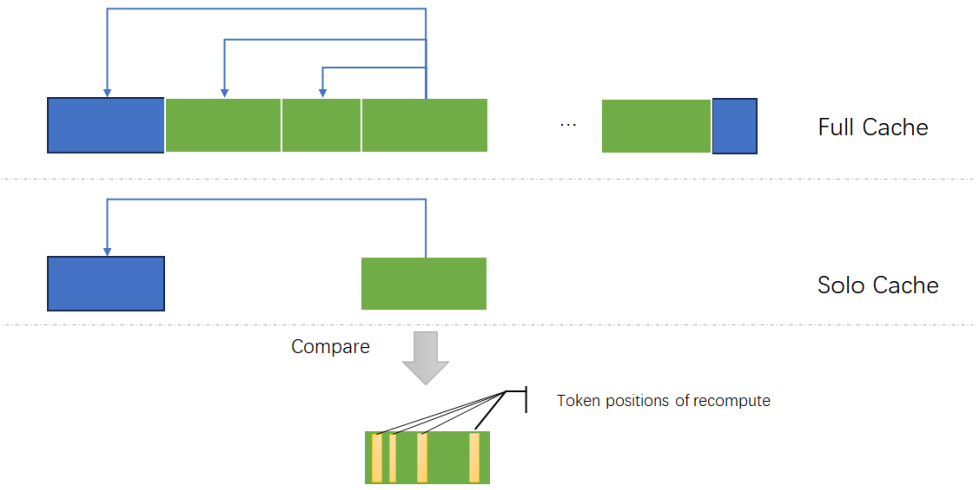


Figure 1: Build-time divergence construction in LETHE. Each chunk is represented both inside the shared cache and in isolation. Their difference produces a stored per-token divergence ranking used for later selective recomputation.

Complementary KV-cache efficiency. A separate line of work reduces the memory or decoding cost of an already constructed KV cache. H_2O (Zhang et al., 2023) evicts low-importance KV entries while retaining heavy hitters, and SnapKV (Li et al., 2024) compresses the cache using attention patterns observed near the end of the prompt. These methods are complementary to LETHE: they reduce the size or cost of a cache, whereas LETHE addresses reuse when a request assembles a new non-prefix subset of previously cached chunks.

Unlike independent-cache libraries, LETHE trades an unbounded chunk library for a bounded, order-preserving shared pool that fits within the model context window.

We use EPIC (Hu et al., 2025) as the primary training-free baseline. LETHE combines five properties: its selection signal is computed once per chunk; its budget adapts per chunk; selected tokens receive an ordinary model forward rather than a learned blend; it requires no fine-tuning; and it applies to both RAG passages and tool definitions. Most importantly, LETHE begins from a shared cache where cross-chunk interaction already exists and selectively removes unwanted influence, rather than reconstructing interaction missing from independently cached chunks.

3 METHOD

3.1 OVERVIEW

LETHE operates on a bounded reusable pool $\mathcal{C} = \{c_1, \ldots, c_n\}$ that fits within the model context window. Larger applications may maintain multiple pools; LETHE's algorithm is unchanged within each pool. Each chunk is processed once in shared context and once in isolation to obtain a stored divergence ranking. At query time, LETHE assembles the requested subset, corrects token positions, and selectively recomputes only the most divergent tokens.

The design is intentionally approximate. Independent caches begin with missing relevant cross-chunk interaction. LETHE instead begins with genuine causal interaction among chunks in the shared cache, but reused states may retain residual influence from chunks later excluded by a query. LETHE targets this residual influence rather than attempting to reconstruct all interactions from scratch. Figure 1 illustrates how the divergence signal is constructed, and Figure 2 shows how it is used at query time.

3.2 BUILD-TIME DIVERGENCE SIGNAL

Chunks enter the shared cache in a fixed build order π. When a new chunk c_i is appended, causal masking guarantees that previously cached chunks do not change, so only c_i must be processed. LETHE also builds a *solo* version of c_i with only the fixed prompt header as prefix. For each token t and layer l, we compare its key state in the two versions.

For each layer, cosine similarity is computed independently over the head dimension for each KV head and then averaged across heads, yielding one similarity value per token and layer. For RAG we use worst-layer divergence,

$$d_i(t) = 1 - \min_l \cos\left(k_i^{\text{solo},l}(t), k_i^{\text{build},l}(t)\right), \tag{1}$$

where $\cos(\cdot,\cdot)$ denotes the head-averaged cosine similarity. Tool-calling instead uses the corresponding mean across layers. We selected these reductions empirically: worst-layer divergence gave slightly better HotpotQA performance, while mean-layer divergence performed better on BFCL. We rank tokens by $d_i(t)$ from largest to smallest. Key divergence alone is used for selection because it closely tracks value divergence in our measurements.

Let μ_i and σ_i be the mean and standard deviation of the token divergences in chunk c_i. LETHE stores two adaptive cutoffs:

$$K_i = |\{t : d_i(t) > \mu_i + 2\sigma_i\}|, \qquad M_i = |\{t : d_i(t) > \mu_i + \sigma_i\}|. \tag{2}$$

K_i is the stricter budget and M_i the looser one. We also evaluate fixed top-p budgets P10/P20/P30 using the same stored ranking. All rankings and cutoffs are computed once per chunk; no divergence statistics are recomputed at query time. Solo caches are discarded afterward; runtime metadata is one token ranking and two scalar cutoffs per chunk in addition to the shared KV cache.

3.3 QUERY-TIME REUSE

Given a requested subset $S \subseteq \mathcal{C}$, LETHE performs three steps, summarized in Figure 2.

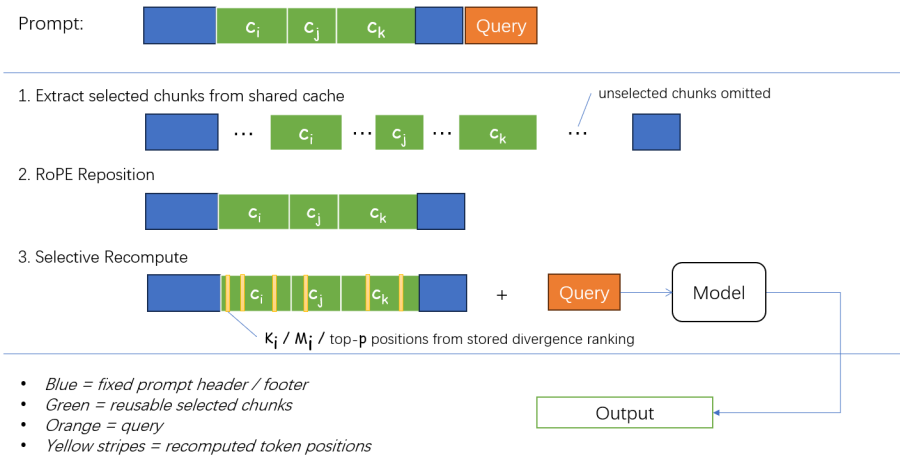


Figure 2: Query-time reuse in LETHE. LETHE extracts selected chunks from the shared cache, preserves build order, repositions cached keys to their new RoPE positions, and recomputes only positions selected by the stored divergence ranking (K_i, M_i, or top-p). Remaining states are reused and combined with the current query for inference.

Assemble. Requested chunks are extracted from the shared cache and presented in build order π restricted to S. The fixed header is reused directly. For RAG, the very short footer is recomputed; tool-calling footer handling is described below.

Reposition. Removing unselected chunks changes the positions of retained tokens. LETHE applies an exact RoPE rotation (Su et al., 2024) to each retained cached key from its build-time position to its new position. Values require no positional rotation.

Selectively recompute. For each retained chunk, LETHE recomputes the first K_i or M_i tokens in its stored divergence ranking, or the requested top-p fraction. Selected positions receive the model's ordinary decoder computation against the current assembled cache, and their new KV states replace the cached ones. Unselected positions retain their repositioned shared-cache states.

In our implementation, selected positions from all retained chunks are gathered per decoder layer and processed together under the causal mask, so later positions attend to updated earlier positions without a separate forward per chunk. Q/K/V projections, query-side attention, and feed-forward computation therefore run only for selected tokens, while attention still has access to the assembled cached K/V context. The reported FLOP savings thus correspond to skipped transformer computation rather than masking after a full forward pass.

3.4 Tool-Calling and Chunk Order

For tool-calling, each chunk is one JSON function definition; cache construction, repositioning, and divergence-based selection are unchanged. The longer fixed tool-calling footer is cached and repositioned rather than recomputed. For Llama-3.1-8B-Instruct only, we additionally force recomputation of function-name and parameter-name tokens because divergence-only selection produced more identifier errors; Section 5.2 ablates this rule.

LETHE preserves build order when assembling selected chunks so reused causal relationships remain consistent with the shared cache. Our evaluation randomizes build order within each cache block, directly exercising this assumption. If a recurring workload requires a different order or combines chunks from incompatible pools, a separate shared cache can be built for that working set; cache-management policy is outside LETHE's core algorithm.

4 Experiments

Models and tasks. We evaluate Qwen3-8B and Qwen3-14B (Yang et al., 2025a), and Llama-3.1-8B-Instruct (Grattafiori et al., 2024), without fine-tuning. For RAG, we use 500 HotpotQA (Yang et al., 2018) questions with supporting and distractor passages as reusable text chunks. For tool-calling, we use all 200 questions in the BFCL v4 (Patil et al., 2025) *multiple* category, where each query is paired with 2–4 candidate function definitions; each function definition is treated as one chunk. BFCL correctness is computed with its deterministic AST-based evaluator.

Cache protocol. A LETHE pool is bounded by the model context window. For HotpotQA, the 500 questions are split into five disjoint blocks of 100 and one shared cache is built per block; BFCL is split into two 100-question blocks. A single randomized build order is fixed within each block, and all methods for a model use exactly the same caches and questions. This paired setup also tests LETHE's use of a fixed build order rather than a query-specific chunk order (§3.4).

Methods. We compare: **Baseline**, which fully recomputes all selected chunks; **NoRecompute**, which only extracts and repositions shared-cache chunks; **LETHE-K** and **LETHE-M**, using the adaptive $\mu + 2\sigma$ and $\mu + \sigma$ cutoffs from §3.2; and **LETHE-P10/P20/P30**, which use the same divergence ranking but recompute a fixed 10/20/30% of each chunk. On HotpotQA we also compare **EPIC-8/16** (Hu et al., 2025), the closest training-free non-prefix reuse baseline discussed in §2. We omit EPIC on BFCL because the original method does not specify or evaluate structured function-definition chunks, and adapting it would introduce additional design choices. For Llama tool-calling, we additionally ablate the forced function/parameter-name refresh from §3.4.

Metrics. HotpotQA uses exact match (EM) as the primary main-text metric; token-level F1, recall, and precision are reported in Appendix A. BFCL reports wrong function calls out of 200. For Llama-3.1-8B-Instruct only, we exclude `type_error:simple` cases from both Baseline and every LETHE configuration because these are serialization-type mismatches rather than function or

Table 2: HotpotQA, 500 questions. LETHE entries are EM / full-prompt FLOPs saved (%). Baseline and NoRecompute show EM only; EPIC FLOPs are omitted.

Method	Qwen3-8B	Llama-3.1-8B	Qwen3-14B
Baseline	.634	.646	.616
NoRecompute	.606	.574	.598
LETHE-K	.632 / 74.5	.594 / 68.1	.610 / 74.6
LETHE-M	.632 / 69.2	.604 / 63.2	.614 / 68.4
LETHE-P10	.632 / 70.7	.582 / 65.2	.608 / 70.9
LETHE-P20	.626 / 62.8	**.618** / 57.9	.602 / 63.1
LETHE-P30	.612 / 54.9	.616 / 50.7	.608 / 55.2
EPIC-8	.538	.526	.518
EPIC-16	.562	.528	.532

argument-value selection errors; the same filter is applied symmetrically. Thus the Llama BFCL counts are intended for paired Baseline-versus-LETHE comparison, not direct comparison with external leaderboard scores.

We measure computation with operation-level FLOP counting over the executed PyTorch model forwards, including a custom SDPA counter using the actual Q/K/V tensor shapes. We report both context FLOPs and full-prompt FLOPs (context plus the uncached query). We additionally report wall-clock prefill latency in the controlled scaling benchmark below.

Latency benchmark. We measure prefill latency on a single NVIDIA RTX A6000 48 GB GPU with CUDA 12.8, using bfloat16 precision, batch size 1, and the same Hugging Face SDPA attention backend for Baseline and LETHE. We evaluate reusable context lengths from 100 to 10,000 tokens with a 10-token cached prefix and a 10% recomputation budget. The benchmark measures prefill only; query-token processing and autoregressive decoding are excluded. LETHE's timed region includes mmap-backed cache retrieval, CPU-to-GPU KV transfer, RoPE repositioning, runtime mask construction, and selective transformer recomputation, while Baseline fully recomputes the same context span. To isolate runtime scaling, the 10% selected-token ranking is fixed by a deterministic seed and constructed outside the timed region, matching LETHE's offline ranking stage. We synchronize CUDA before and after each measurement, use 5 warmup iterations per context length followed by 30 measured iterations, and repeat the benchmark in three independent runs for each model and method.

5 RESULTS

The main result is an accuracy–compute trade-off rather than a single universally best configuration. LETHE-K is the most compute-efficient operating point, while LETHE-M often provides a balanced accuracy–compute operating point; fixed P10/P20/P30 expose explicit compute budgets. Tables 2 and 3 report exact task results. Figure 3 visualizes the HotpotQA accuracy–compute trade-off, while Figure 4 shows how LETHE's wall-clock prefill advantage scales with reusable context length.

5.1 HOTPOTQA

On Qwen3-8B, LETHE-K/M/P10 are within 0.2 EM points of full recomputation; on Qwen3-14B, LETHE-M is also within 0.2 points. Llama is harder: its best setting, P20, is 2.8 points below Baseline. LETHE nevertheless substantially exceeds EPIC on every model. In particular, LETHE-K improves over EPIC-16 by 7.0, 6.6, and 7.8 EM points on Qwen3-8B, Llama, and Qwen3-14B respectively (7.1 points on average).

Non-monotonicity. Accuracy is not strictly monotonic with recomputation budget (e.g., P10 can exceed P30). Because LETHE mixes recomputed and reused states, more recomputation does not guarantee better task output; we therefore treat P10/P20/P30 as operating points rather than a monotonic sweep.

Table 3: BFCL, 200 questions. LETHE entries are wrong calls (lower is better) / full-prompt FLOPs saved (%). Llama uses the symmetric type-error filter described in §4.

Method	Qwen3-8B	Llama-3.1-8B	Qwen3-14B
Baseline	9	7	12
LETHE-K	10 / 87.5	13 / 81.5	12 / 87.2
LETHE-M	11 / 79.6	11 / 67.4	11 / 78.3
LETHE-P10	10 / 84.1	10 / 71.3	**10** / 84.1
LETHE-P20	11 / 76.1	13 / 63.9	15 / 76.1
LETHE-P30	13 / 68.2	13 / 56.4	14 / 68.2

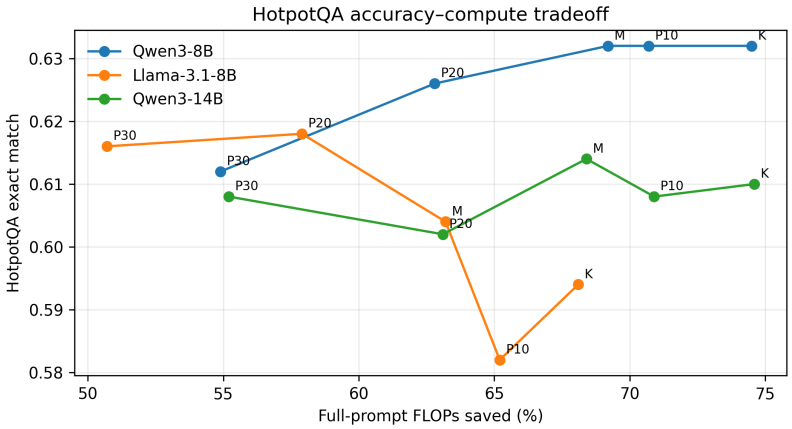


Figure 3: HotpotQA accuracy–compute trade-off for LETHE operating points. Full-prompt FLOPs saved are measured relative to full recomputation; labels identify adaptive K/M and fixed P10/P20/P30 settings. Exact values are in Table 2.

Why selective recomputation matters. NoRecompute already retains much of Baseline accuracy, supporting LETHE's shared-cache starting point: its Baseline gaps are 2.8, 7.2, and 1.8 EM points for Qwen3-8B, Llama, and Qwen3-14B. Selective recomputation then closes most of those gaps—92.9%, 61.1%, and 88.9%, respectively, using each model's best LETHE setting. Thus the shared cache provides a strong starting approximation, while recomputation remains important when stale cross-context influence is more damaging.

Adaptive budgets are genuinely per-chunk. To verify that K/M do not merely behave like hidden fixed percentages, we analyze one 100-question cache block per model (600 chunks, 62,479 tokens total). The token-weighted K budget is 5.0–6.0% across the three models and M is 11.8–13.1%, but individual chunks vary: across all 600 chunks, the central 80% spans 3.8–7.6% for K and 9.7–17.0% for M. The similar global operating points therefore emerge from different per-chunk allocations rather than an imposed fixed ratio.

Human check of lexical metrics. On the 200-question Llama subset where LETHE-M has the largest automated gap, Baseline vs. LETHE-M differs by 5.0 EM points and 3.85 F1 points, but by only 2.5 points under manual correct/incorrect judgment. We use this only as a check that lexical metrics can overstate the observed accuracy gap, not as a replacement for the full automated evaluation.

Compute. LETHE-K reduces context FLOPs by 93.6–94.1% and full-prompt FLOPs by 68.1–74.6% across the three models. Even P30 reduces full-prompt FLOPs by 50.7–55.2%. The smaller full-prompt savings reflect the uncached query, whose computation is identical across methods.

5.2 BFCL Tool-Calling

With only 200 questions, differences of a few calls should not be over-interpreted. LETHE-P10 is within three wrong calls of Baseline on all three models and is two calls better than Baseline on Qwen3-14B, while saving 71.3–84.1% of full-prompt FLOPs. LETHE-M gives a similarly stable operating point: relative to Baseline it changes the wrong-count by $+2, +4, -1$ while saving 67.4–79.6% of full-prompt FLOPs.

Llama identifier ablation. The model-specific forced refresh in §3.4 is important. Without forcing function- and parameter-name tokens, Llama produces 22 and 21 wrong calls under LETHE-K/M; with forcing, these fall to 13 and 11. We therefore retain the rule as an empirical Llama-specific adjustment rather than claiming a general mechanism.

Compute. LETHE-K reduces full-prompt FLOPs by 81.5–87.5% on BFCL (87.2–87.5% for the two Qwen models). Llama saves less because the identifier rule increases its effective recomputation budget. Overall, the two tasks show the same pattern: large reductions in executed transformer computation with a modest, model-dependent accuracy cost.

5.3 Prefill Latency Scaling

Figure 4 reports the controlled LETHE 10% prefill benchmark described in §4. Detailed measurements for all context lengths are reported in Appendix B. LETHE is slower at 100 tokens because fixed cache-management and selective-recomputation overhead dominates such short contexts, but it is already faster by 300 tokens on all three models. The advantage then grows steadily with context length. At 1,000 tokens, LETHE is 2.58×, 2.68×, and 3.17× faster than Baseline on Qwen3-8B, Llama-3.1-8B, and Qwen3-14B, respectively; at 5,000 tokens the speedups grow to 4.37×, 4.35×, and 5.44×. At 10,000 tokens, LETHE reduces mean prefill latency from 3.227 s to 0.604 s on Qwen3-8B, from 2.935 s to 0.558 s on Llama-3.1-8B, and from 5.066 s to 0.803 s on Qwen3-14B, corresponding to **5.35×, 5.26×, and 6.31× speedups**. These gains include mmap-backed cache retrieval, CPU-to-GPU KV transfer, RoPE repositioning, runtime mask construction, and selective transformer recomputation, rather than timing only the reduced transformer forward.

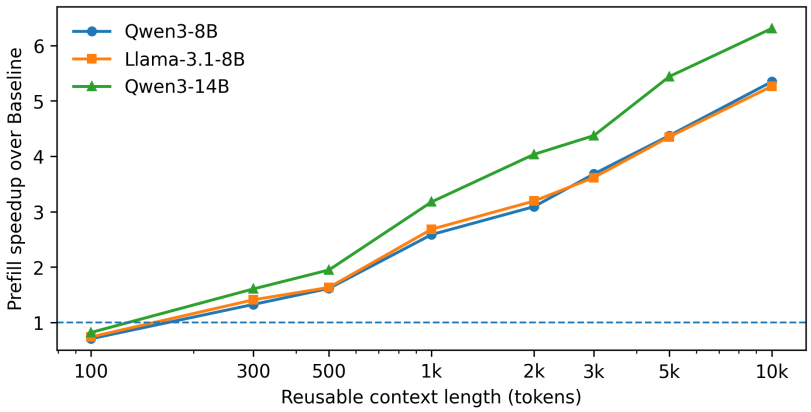


Figure 4: Wall-clock prefill speedup of LETHE with a 10% recomputation budget over full recomputation as reusable context length increases. Values use the mean latency across three independent runs per model and method; each run uses 5 warmup iterations and 30 measured iterations per context length on a single RTX A6000 48 GB GPU. The dashed line marks break-even (1×).

6 Limitations

LETHE operates on reusable chunk pools that fit within the model context window; larger applications therefore require multiple pools and an external cache-management policy. Its divergence

ranking is computed once per chunk and is intentionally query-independent, so it cannot anticipate which excluded context will matter for a particular future request. For example, it may spend recomputation budget on a token whose influential predecessor chunks are still retained in that request. Our evaluation covers three open-weight models and two tasks. The latency evaluation is a controlled single-GPU, batch-size-1 prefill benchmark rather than a complete production serving system; batched serving may exhibit different efficiency, and end-to-end latency will additionally depend on cache placement, storage hierarchy, query processing, decoding, and serving-engine integration. Our implementation uses Python-level hooks and standard attention kernels, leaving further systems and kernel optimization as future work.

7 Conclusion

We introduced LETHE, a training-free approach to non-prefix KV reuse that starts from a shared cache with genuine cross-chunk interaction and selectively corrects the influence of excluded context. Across HotpotQA and BFCL, LETHE preserves accuracy while substantially reducing transformer computation, with adaptive per-chunk cutoffs providing automatic accuracy–compute operating points. In a controlled prefill-scaling benchmark with a 10% recomputation budget, LETHE achieves 5.3–6.3$\times$ wall-clock speedup at 10,000 context tokens across all three models. These results suggest that selectively forgetting irrelevant context is a promising alternative to reconstructing interactions from independently cached chunks.

AI Use Statement

Generative AI tools were used to assist with manuscript organization and editing, drafting and rewriting portions of the manuscript, related-work discovery, reviewer-style critique, experimental-design feedback, code-level debugging suggestions, and interpretation and presentation of experimental results. All experiments, measurements, code changes, references, and scientific claims were reviewed and verified by the authors. The authors take full responsibility for the final content of this work.

References

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 6, pp. 325–338, 2024.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. The Llama 3 herd of models, 2024. arXiv:2407.21783.

Junhao Hu, Wenrui Huang, Weidong Wang, Haoyi Wang, Tiancheng Hu, Qin Zhang, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. Epic: Efficient position-independent caching for serving large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 24391–24402. PMLR, 2025.

Yinsicheng Jiang, Yeqi Huang, Liang Cheng, Cheng Deng, Xuan Sun, and Luo Mai. Contextpilot: Fast long-context inference via context reuse. In *Proceedings of Machine Learning and Systems (MLSys)*, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

Fei Li, Song Liu, Yan Liu, Jinhua Cui, Shiqiang Nie, Jinyu Wang, and Weiguo Wu. Adaptive KV cache reuse for fast long-context LLM serving (CacheTune), 2026. arXiv:2605.24022.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. SnapKV: LLM knows what you are looking for before generation. In *Advances in Neural Information Processing Systems*, volume 37, pp. 22947–22970, 2024.

Yang Liu, Yunfei Gu, Liqiang Zhang, Chentao Wu, Guangtao Xue, Jie Li, Minyi Guo, Junhao Hu, and Jie Meng. CacheSlide: Unlocking cross position-aware KV cache reuse for accelerating LLM serving. In *24th USENIX Conference on File and Storage Technologies (FAST '26)*, pp. 83–99, 2026.

Songshuo Lu, Hua Wang, Yutian Rong, Zhi Chen, and Yaohua Tang. TurboRAG: Accelerating retrieval-augmented generation with precomputed KV caches for chunked text. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6588–6601, 2025.

Nilesh Prasad Pandey, Jason Kong, Lanxiang Hu, Quanling Zhao, Yujie Zhao, Onat Gungor, Hao Zhang, and Tajana Rosing. AgentKVShift: Efficient KV cache reuse for agentic memory systems, 2026. arXiv:2607.21604.

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The Berkeley Function Calling Leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, pp. 48371–48392, 2025.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024. doi: 10.1016/j.neucom.2023.127063.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context LLM inference. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, pp. 47901–47911, 2024.

An Yang, Anfeng Li, Baosong Yang, et al. Qwen3 technical report, 2025a. arXiv:2505.09388.

Jingbo Yang, Bairu Hou, Wei Wei, Yujia Bao, and Shiyu Chang. KVLink: Accelerating large language models via efficient KV cache reuse. In *Advances in Neural Information Processing Systems*, 2025b.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380. Association for Computational Linguistics, 2018. doi: 10.18653/v1/D18-1259.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25)*, pp. 94–109, 2025.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H2O: Heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583, 2024.

A Additional HotpotQA Metrics

Table 4 reports token-level F1, recall, and precision in addition to the exact-match (EM) results shown in the main paper. The overall trend is consistent with EM: LETHE substantially outperforms EPIC across all three models, although the best LETHE operating point varies by model and metric. On Qwen3-14B, LETHE-M nearly matches full recomputation across all four metrics.

Model	Method	EM	F1	Recall	Precision
Qwen3-8B	Baseline	.634	.797	.823	.816
	NoRecompute	.606	.752	.771	.772
	LETHE-K	.632	.776	.795	.794
	LETHE-M	.632	.776	.797	.796
	LETHE-P10	.632	.776	.796	.793
	LETHE-P20	.626	.778	.803	.795
	LETHE-P30	.612	.763	.785	.783
	EPIC-8	.538	.685	.719	.704
	EPIC-16	.562	.709	.740	.727
Llama-3.1-8B-Instruct	Baseline	.646	.798	.812	.821
	NoRecompute	.574	.712	.726	.733
	LETHE-K	.594	.734	.750	.754
	LETHE-M	.604	.738	.753	.759
	LETHE-P10	.582	.724	.742	.745
	LETHE-P20	.618	.748	.762	.769
	LETHE-P30	.616	.750	.765	.772
	EPIC-8	.526	.650	.650	.676
	EPIC-16	.528	.662	.668	.691
Qwen3-14B	Baseline	.616	.790	.835	.798
	NoRecompute	.598	.748	.774	.766
	LETHE-K	.610	.754	.780	.772
	LETHE-M	.614	.788	.835	.796
	LETHE-P10	.608	.759	.788	.776
	LETHE-P20	.602	.765	.807	.776
	LETHE-P30	.608	.770	.811	.785
	EPIC-8	.518	.680	.713	.693
	EPIC-16	.532	.702	.747	.712

Table 4: Additional HotpotQA results over 500 questions. EM is the primary metric used in the main paper; F1, recall, and precision are reported here for completeness.

B Detailed Prefill Latency Results

Table 5 reports the complete prefill-latency measurements used to produce Figure 4. Latencies are aggregated across three independent runs for each model and method. Each independent run uses 5 warmup iterations followed by 30 measured iterations per context length. The table reports the mean and sample standard deviation across the three independent run means. LETHE uses a fixed 10% recomputation budget in this controlled scaling benchmark.

Model / Context	Baseline (ms)	LETHE 10% (ms)	Speedup
Qwen3-8B / 100	41.4 ± 0.0	58.7 ± 0.9	0.71×
300	82.8 ± 1.8	62.5 ± 0.4	1.32×
500	112.2 ± 0.1	69.5 ± 1.2	1.61×
1k	211.3 ± 0.3	81.8 ± 0.7	2.58×
2k	427.4 ± 9.6	138.3 ± 2.2	3.09×
3k	678.0 ± 0.2	184.2 ± 1.8	3.68×
5k	1261.3 ± 1.0	288.4 ± 3.1	4.37×
10k	3227.1 ± 26.2	603.6 ± 7.9	5.35×
Llama-3.1-8B-Instruct / 100	36.9 ± 0.1	50.1 ± 0.2	0.74×
300	78.3 ± 1.6	55.6 ± 0.1	1.41×
500	104.3 ± 0.1	63.9 ± 0.4	1.63×
1k	197.2 ± 0.2	73.5 ± 0.3	2.68×
2k	388.0 ± 8.5	121.6 ± 0.9	3.19×
3k	620.5 ± 0.2	171.6 ± 1.0	3.62×
5k	1157.6 ± 0.8	266.1 ± 1.1	4.35×
10k	2935.2 ± 26.6	557.5 ± 4.5	5.26×
Qwen3-14B / 100	62.7 ± 0.1	76.6 ± 0.2	0.82×
300	134.9 ± 0.1	84.0 ± 0.9	1.60×
500	178.7 ± 0.3	91.8 ± 0.3	1.95×
1k	339.4 ± 0.2	106.9 ± 0.3	3.17×
2k	691.4 ± 0.3	171.4 ± 0.3	4.03×
3k	1105.1 ± 1.0	252.7 ± 1.2	4.37×
5k	2051.2 ± 1.4	377.1 ± 1.8	5.44×
10k	5066.0 ± 2.6	803.1 ± 4.5	6.31×

Table 5: Detailed wall-clock prefill latency across reusable context lengths. Values are mean $\pm$ sample standard deviation across three independent run means. Each run uses 5 warmup iterations and 30 measured iterations per context length on a single NVIDIA RTX A6000 48 GB GPU. The speedup column is Baseline latency divided by LETHE latency.