arXiv:2511.03475v3 [cs.LG] 23 Feb 2026

CONTEXTPILOT: FAST LONG-CONTEXT INFERENCE VIA CONTEXT REUSE

Yinsicheng Jiang [*1] Yeqi Huang [*1] Liang Cheng [1] Cheng Deng [1] Xuan Sun [1] Luo Mai [1]

ABSTRACT

AI applications increasingly depend on long-context inference, where LLMs consume substantial context to support stronger reasoning. Common examples include retrieval-augmented generation, agent memory layers, and multi-agent orchestration. As input contexts get longer, prefill latency becomes the main bottleneck. Yet today's prefill acceleration techniques face a trade-off: they either preserve reasoning quality but deliver little KV-cache reuse, or improve reuse at the cost of degraded reasoning quality.

We present CONTEXTPILOT, a system that accelerates prefill by introducing *context reuse* as a new mechanism for faster long-context inference. CONTEXTPILOT introduces a context index to identify overlapping *context blocks* across LLM interactions (e.g., across users and turns). It further proposes context ordering and de-duplication techniques to maximize KV-cache reuse. To preserve reasoning quality under reuse, it introduces succinct context annotations that prevent quality degradation. Finally, CONTEXTPILOT is built around a modular architecture with a clean interface that integrates with existing inference engines. Extensive evaluation shows that CONTEXTPILOT reduces LLM prefill latency by up to 3× compared to state-of-the-art methods while preserving reasoning quality. At longer context lengths, it can even improve reasoning quality. CONTEXTPILOT is open-sourced at: https://github.com/EfficientContext/ContextPilot.

1 INTRODUCTION

Long-context inference is now central to many AI applications. Whether through retrieval-augmented generation (RAG) (Lewis et al., 2020), AI memory layers such as Mem0 (Chhikara et al., 2025), multi-agent orchestration, or simply appending in-context examples, modern workloads routinely feed LLMs tens to hundreds of thousands of tokens of external context. In a typical pipeline, a retriever (e.g., FAISS, Qdrant, ElasticSearch) or memory store fetches relevant documents, chunks, or memories for a user query, and an inference engine (e.g., SGLang, vLLM, TensorRT-LLM) consumes them as input context.

We call these discrete units of external context *context blocks (CBs)*. During the *prefill* phase, the engine computes key–value (KV) caches, which are then reused during *decode* to generate output tokens sequentially. The key performance goal in prefill is to reduce time-to-first-token (TTFT). To that end, inference engines use a *prefix cache* that stores KV caches from prior prompts, avoiding recomputation for repeated inputs or prompts that share a prefix.

[*]Co-primary Authors [1]University of Edinburgh. Correspondence to: Yinsicheng Jiang <ysc.jiang@ed.ac.uk>, Yeqi Huang <yeqi.huang@ed.ac.uk>, Liang Cheng <L.cheng@ed.ac.uk>, Cheng Deng <cdeng@ed.ac.uk>, Xuan Sun <xuan.sun@ed.ac.uk>, Luo Mai <luo.mai@ed.ac.uk>.

To appear at 9th Annual Conference on Machine Learning and Systems (MLSys 2026).

AI applications are feeding LLMs ever larger amounts of external context to unlock stronger capabilities, making prefill latency a key bottleneck. This trend is driven by two forces. First, many studies show that more context—ranging from retrieved knowledge-base content to long-horizon memories in agentic systems—improves performance on complex reasoning tasks (e.g., lemmas in AI4Math) (Varambally et al., 2025), strengthens access to up-to-date information (e.g., AI4Search) (Alzubi et al., 2025; Zilliz, 2025; Gupta et al., 2026), and reduces hallucinations (Ayala & Bechard, 2024; Shuster et al., 2021; AboulEla et al., 2025). Second, while chunking is widely used to shrink per-request inputs, recent findings (Rajasekaran et al., 2025) suggest that overly aggressive chunking can harm reasoning quality; in contrast, processing larger, less fragmented context (e.g., full documents) often yields better results—further increasing prefill latency.

To speed up long-context inference, existing systems adopt two main techniques, yet each faces a trade-off between reuse efficiency and model accuracy. The first, *exact prefix matching*, used in systems such as RadixCache (Zheng et al., 2024), LMCache (Cheng et al., 2025), and RAGCache (Jin et al., 2024b), reuses cached KV states only when a new prompt exactly matches a previous prefix. This approach preserves accuracy but yields low cache-hit ratios in practice, as long-context workloads often retrieve large sets of documents or memories in varying orders, leaving most KV caches unused. The second category, *approximate KV-*

cache matching, exemplified by CacheBlend (Yao et al., 2025) and PromptCache (Gim et al., 2024), matches KV caches by floating-point similarity rather than exact prefixes. While this increases reuse and shortens TTFT, we observe in evaluation that it can significantly degrade model accuracy.

To reduce TTFT for long-context inputs without sacrificing accuracy, we propose a new approach based on the observation that real-world long-context workloads, often exhibit overlapping context blocks, commonly (i) across multiple turns within the same conversation and (ii) among parallel sessions (e.g., prompts or user queries) in domain-specific applications. Leveraging this observation, we identify three opportunities for *context reuse with negligible accuracy loss*: (1) *Ordering* context blocks to align prefixes with previously cached contexts, improving cache-hit ratios; (2) *De-duplicating* context blocks to avoid recomputation for already cached content; and (3) *Adding context annotations* to inform the model of original relevance ranking and deduplicated block locations, mitigating accuracy loss.

In this paper, we present CONTEXTPILOT, a system that accelerates prefill by introducing *context reuse* as a new mechanism for faster long-context inference. CONTEXTPILOT targets practical long-context settings with parallel sessions and multi-turn conversations, where substantial portions of the input context recur across requests. Our key contributions are summarized below.

(1) Context Indexing. We design an indexing mechanism that efficiently tracks cached context blocks across parallel sessions and multi-turn conversation histories. The index supports fast retrieval of previously stored contexts by (i) aligning prefix overlaps between the incoming context and cached blocks, and (ii) traversing blocks referenced in multi-turn histories to recover reusable segments beyond the immediate prefix match, while maintaining low construction and maintenance overhead.

(2) Context Ordering. We propose a context-ordering algorithm that queries the index to select and arrange context blocks both *within* a session (across turns) and *across* sessions, with the explicit goal of maximizing cache hit ratio under a fixed context budget. To mitigate potential accuracy loss introduced by reordering, we introduce concise *order annotations* that preserve the original relevance ranking and structural cues, allowing the LLM to interpret the reordered prompt consistently with the intended semantics.

(3) Context De-Duplication. We further improve reuse efficiency via context de-duplication. By querying the index, CONTEXTPILOT identifies context blocks that overlap with already cached contexts (including partial overlaps), and replaces duplicated spans with succinct *location annotations* that point to their original occurrences in the prompt (or prior turns), thereby avoiding redundant prefill while preserving accuracy.

Extensive evaluations show that CONTEXTPILOT delivers strong performance across diverse baselines and real-world datasets. Across long-context workloads—including RAG (multi-turn, multi-session, and hybrid), agentic memory systems (Mem0), and emerging multi-agent reasoning paradigms—it reuses contexts to accelerate prefill, outperforming state-of-the-art systems (CacheBlend, LMCache, RadixCache, and RAGCache) by 1.5–3$\times$ on MultihopRAG, NarrativeQA, QASPER, and MT-RAG with negligible accuracy loss. As context length grows, CONTEXTPILOT can even improve reasoning quality and answer accuracy, thanks to its novel context-annotation design.

Notably, CONTEXTPILOT can effectively reduce the prefill latency of very large MoE reasoning models (Jiang et al., 2025): on DeepSeek-R1 (671B), it increases cache hit ratio from 5–6% to 38–60%, improving prefill throughput by 1.52–1.81$\times$ on 16 and 32 GPUs, demonstrating strong multi-GPU scalability. We will also report results on MiniMax2.5, GLM5, DeepSeek-V3.2, GPT-OSS-120B, and Kimi-K2.5 as they become available.

Building on these results, we are working towards broader academic and industry deployment with multiple adopters, and have open-sourced CONTEXTPILOT on GitHub. We expect it to be deployed in more challenging multi-user serverless scenarios (Fu et al., 2024) and can serve as an extensible software foundation for context engineering (Hua et al., 2025; Zhan et al., 2025), replay (Liu et al., 2025b), management (Chang & Geng, 2025; Yu et al., 2025), and optimization (Kang et al., 2025; Li et al., 2025).

2 BACKGROUND AND MOTIVATION

2.1 Long-context inference systems

Long-context inference systems augment LLMs with external context blocks—retrieved documents, chunks, or memories—to enhance factual grounding and reasoning. We use the term *context block* throughout this paper to refer to any discrete unit of external context injected into the model, whether a retrieved document, a document chunk, or a memory entry. Two dominant paradigms drive this trend: (1) *Retrieval-augmented generation (RAG)* (Lewis et al., 2020; Gao et al., 2024) retrieves the top-K most relevant documents per query from an external corpus, serving both online latency-sensitive services (e.g., semantic search, dialogue, deep research (Zilliz, 2025; Guo et al., 2024b)) and offline throughput-oriented pipelines (e.g., large-scale annotation, synthetic data generation (Shen et al., 2025; Zhou et al., 2024; NVIDIA, 2024; Zhang et al., 2025c)). (2) *AI memory layer* systems (e.g., Mem0 (Chhikara et al., 2025)) dynamically extract, consolidate, and retrieve user-specific memories across sessions (Hu et al., 2025), injecting rele-

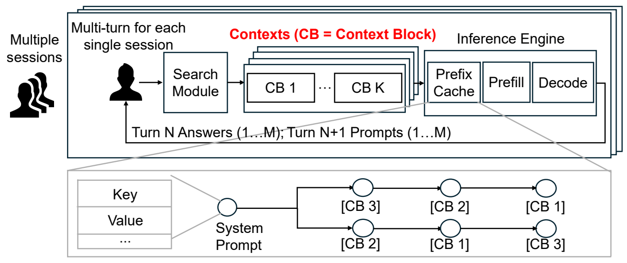


Figure 1. Overview of a long-context inference system with prefix caching.

vant context blocks into each query to enable personalized interactions. In both cases, the inference engine performs *prefill* to encode these context blocks and *decode* to generate responses.

A typical system (Figure 1) alternates between retrieval (or memory lookup) and generation across sessions and dialogue turns. At each turn, M concurrent prompts receive K relevant context blocks each. The inference engine encodes the context and generates responses, which feed into the next step with updated dialogue history, enabling efficient multi-turn reasoning.

These long-context inference systems often use prefix caching to improve prefill efficiency (Lumer et al., 2026). A trie-based implementation (Zheng et al., 2024) organizes tokens hierarchically, with each node storing a token sequence and its KV cache, enabling longest-prefix matching through a single traversal. An alternative hash-table design (Kwon et al., 2023) directly maps complete prefixes to KV-block identifiers.

2.2 Emerging challenge: growing context lengths

Long-context inference systems face a critical prefill latency bottleneck as modern LLMs demand expanding context windows. This is driven by two reasons: (1) *increasing the number of retrieved context blocks* to broaden information coverage (Li et al., 2024; Jin et al., 2024a; Yue et al., 2025; Laban et al., 2024; Chung et al., 2025), and (2) enriching contextual information by *retrieving complete documents or full memory histories* and applying context engineering methods (Rajasekaran et al., 2025).

Analysis of our workload data reveals that both approaches deliver significant accuracy gains. Scaling the retrieval parameter (k) from lower to higher values enhances accuracy by as much as 20%, while retrieving full documents achieves similar performance improvements, confirmed by recent context engineering studies (Zhang et al., 2025b).

However, expanded context windows (i.e., longer context block inputs) introduce substantial prefill overhead and can even degrade reasoning quality beyond a certain length (Du et al., 2025; Raju et al., 2026). Our trace data shows that LLM inference engines often process 20k–130k prefill tokens, leading to 3–10 second latency when executing 32B dense models on a single H100 GPU. For larger models such as Mixture-of-Experts (MoEs), the prefill latency can be even higher. As a result, the prefill becomes the dominant bottleneck, downgrading user experience and preventing long-context applications from being widely deployed.

2.3 Issues of existing KV cache reuse methods

To address the growing cost of longer retrieved contexts, existing KV-cache reuse methods exhibit several issues:

Exact-prefix matching yields low KV-cache reuse. Existing prefix-caching mechanisms rely heavily on exact token-level matching, e.g., RadixCache (Zheng et al., 2024), or document-level matching, e.g., LMCache (Cheng et al., 2025) and RAGCache (Jin et al., 2024b): even minor variations, such as whitespace differences or slightly reordered tokens and documents, prevent reuse. Our evaluation (Section 7.1) shows that despite substantial overlap in retrieved documents across related queries, cache hit ratios remain persistently low. For example, for the dataset multihopRAG with Qwen3-32B, the KV-cache hit ratio is only 4.6%, indicating low KV cache reuse. For NarrativeQA with Llama3.3-70B, the hit ratio is also only 5.5%, leaving most cache unused.

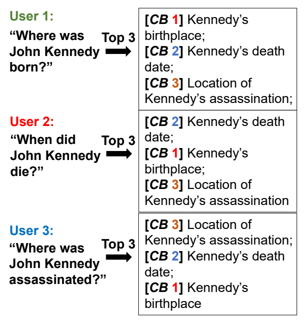
Approximate KV-cache matching degrades quality. To improve low cache-hit ratios, recent techniques such as CacheBlend (Yao et al., 2025) adopt approximate KV-cache matching. Instead of exact-prefix matching, they measure similarity in KV values (floating-point vectors) and reuse cached states when the proximity exceeds an empirically decided threshold. However, KV-value similarity is *not* a reliable indicator of whether cached states can be reused across different contexts and requests. Approximate matching degrades accuracy, with errors compounding over multi-turn interaction. Our evaluations (Section 7.1) show that across multiple models (e.g., Qwen3-32B, Qwen3-4B, Llama3.3-70B) and datasets (e.g., MultihopRAG, NarrativeQA, QASPER), approximate matching can degrade accuracy by 9–11% (dropping from around 60% to approximately 50%), preventing its deployment in many services where high fidelity is necessary.

3 Design Overview

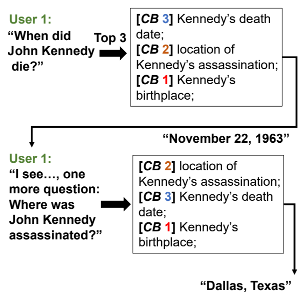
3.1 Observation: significant overlap in long context

Our design is motivated by a key observation: real-world long-context workloads exhibit substantial overlap in context blocks across both sessions and conversation turns:

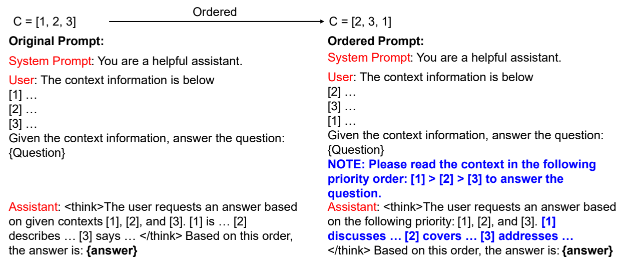
(1) Overlap across sessions. Figure 2a illustrates overlapping retrievals among multiple users querying different as-


User 1:
"Where was John Kennedy born?"
Top 3
[CB 1] Kennedy's birthplace;
[CB 2] Kennedy's death date;
[CB 3] Location of Kennedy's assassination;
User 2:
"When did John Kennedy die?"
Top 3
[CB 2] Kennedy's death date;
[CB 1] Kennedy's birthplace;
[CB 3] Location of Kennedy's assassination
User 3:
"Where was John Kennedy assassinated?"
Top 3
[CB 3] Location of Kennedy's assassination;
[CB 2] Kennedy's death date;
[CB 1] Kennedy's birthplace

(a) Multi-session overlap.


User 1:
"When did John Kennedy die?"
Top 3
[CB 3] Kennedy's death date;
[CB 2] location of Kennedy's assassination;
[CB 1] Kennedy's birthplace;
"November 22, 1963"
User 1:
"I see..., one more question: Where was John Kennedy assassinated?"
[CB 2] location of Kennedy's assassination;
[CB 3] Kennedy's death date;
[CB 1] Kennedy's birthplace;
"Dallas, Texas"

(b) Multi-turn overlap.



C = [1, 2, 3]

Original Prompt:

System Prompt: You are a helpful assistant.

User: The context information is below
[1] ...
[2] ...
[3] ...
Given the context information, answer the question:
{Question}

Assistant: <think>The user requests an answer based on given contexts [1], [2], and [3]. [1] is ... [2] describes ... [3] says ... </think> Based on this order, the answer is: **{answer}**

C = [2, 3, 1]

Ordered Prompt:

System Prompt: You are a helpful assistant.

User: The context information is below
[2] ...
[3] ...
[1] ...
Given the context information, answer the question:
{Question}
NOTE: Please read the context in the following priority order: [1] > [2] > [3] to answer the question.

Assistant: <think>The user requests an answer based on the following priority: [1], [2], and [3]. **[1] discusses ... [2] covers ... [3] addresses ...** </think> Based on this order, the answer is: **{answer}**

(c) Context annotations for recovering relevance ranking semantics.

Figure 2. Context overlap and reuse opportunities in long-context inference.

pects of the same person. Although the retrieved documents appear in different orders reflecting per-query relevance, their content largely coincides. Trace studies on MultihopRAG (Tang & Yang, 2024), NarrativeQA (Kočiský et al., 2017), and QASPER (Dasigi et al., 2021) confirm this trend: as shown in Figure 3, 79.2%, 57.4%, and 49.6% of questions respectively draw from the top 20% most frequently accessed documents, indicating extensive context sharing across sessions.

(2) Overlap within multi-turn conversations. Figure 2b shows that in multi-turn interactions, users often revisit related topics, causing the retriever to return the same documents with slightly different rankings. As previous turns become part of the input, later retrievals frequently duplicate content already present in the cached history. Our MT-RAG (Katsis et al., 2025) trace study quantifies this effect: on average, 40% of retrieved documents in any turn overlap with earlier ones in the same session.

3.2 Design opportunities for context reuse

The significant overlap among context blocks reveals clear opportunities for *context reuse*, boosting KV-cache hit ratio. Specifically, we identify three opportunities that commonly arise in real-world long-context applications:

(1) Ordering context blocks across sessions boosts KV-cache reuse. As shown in Figure 2a, if the context blocks

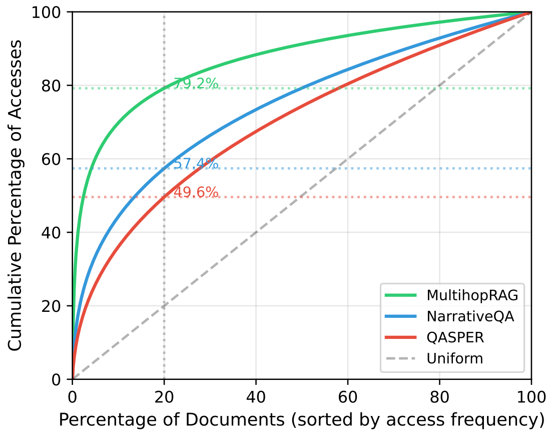


Figure 3. Document access distribution (CDF) across three datasets. The vertical dashed line marks 20% of documents; horizontal lines indicate the cumulative percentage of accesses from these top documents.

for the second and third users are reordered to match the first user's sequence, all three contexts would share an identical prefix, achieving 100% KV-cache reuse.

Trace-based reordering experiments on MultihopRAG, NarrativeQA, and QASPER confirm this potential. Aligning context block order with prefix-cache structure raises KV-cache hit ratio to 38.9%, 20.2%, and 16.5%, respectively, representing 3–8× higher utilization than the baseline (Section 7.3). Thus, strategic context block reordering can dramatically cut redundant prefill computation across users.

Crucially, such reordering incurs minor accuracy loss: only 0.1–3.3% on the same datasets (Section 7.3). As shown in Table 1, our reproduction of the DEmO ordering study (Guo et al., 2024a) with newer models confirms that modern LLMs are substantially less sensitive to input ordering than earlier generations, with near-zero variance on datasets (SST2 (Socher et al., 2013), SNLI (Bowman et al., 2015), SUBJ (Pang & Lee, 2004), CR (Hu & Liu, 2004)) that showed large gaps in the original study. The small residual degradation arises because prefix-optimized orderings can occasionally move important context blocks toward the middle of the list, exposing them to the lost-in-the-middle effect (Liu et al., 2023).

We later discuss strategies to largely recover this minor loss. Note that context block ordering poses no additional privacy or security risks, sharing the same guarantees as prior KV-cache reuse methods (e.g., RadixCache).

(2) De-duplicating multi-turn overlaps reduces prefill cost. Figure 2b shows that multi-turn retrievals often return overlapping context blocks across conversation turns. By deduplicating these blocks and processing only new content

Table 1. Reproducing DEmO ordering study with newer models. Modern LLMs show negligible ordering gaps even on datasets that showed large gaps in the original study.

Dataset	**GPT-3.5**		**GPT-5.1**	
	Random	DEmO	Random	DEmO
SST2 (Socher et al., 2013)	93.8	93.8	92.0	93.8
SNLI (Bowman et al., 2015)	72.6	72.6	83.2	83.2
SUBJ (Pang & Lee, 2004)	71.3	71.6	77.5	77.0
CR (Hu & Liu, 2004)	93.8	93.8	94.7	92.9
Avg	82.9	83.0	86.9	86.7

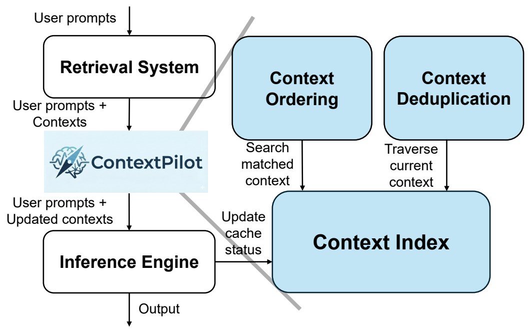


Figure 4. System Overview of ContextPilot.

together with dialogue history, the amount of contextual data during prefill can be greatly reduced, lowering computation cost.

Our MT-RAG trace study quantifies this benefit and shows that de-duplication causes only 1–3% accuracy degradation, which can be recovered with techniques discussed later. This minor loss occurs because the LLM can still access the deduplicated content through prior conversation history, preserving quality while avoiding duplicated computation.

(3) Context annotation compensates for accuracy perturbation from ordering and de-duplication. Carefully designed context annotation can largely compensate for the negligible accuracy perturbation introduced by context block ordering and de-duplication. As shown in Figure 2c, these annotations help the model reconstruct the original ordering relationships within its internal tokens, mitigating accuracy loss.

Evaluations show that context annotations effectively recover accuracy—and in multi-hop reasoning tasks, even improve it beyond the unordered baseline. On NarrativeQA and MultihopRAG, for instance, accuracy increases by 0.3–3.9% relative to approximate matching methods (Section 7.3), confirming the effectiveness of incorporating context annotations for enhanced reasoning.

Note that the context annotation does not affect the model's instruction-following ability, as it only conveys minimal retrieval metadata without altering the user prompt.

3.3 ContextPilot system overview

Prior work treats accuracy (e.g., context graphs, agentic memory) and system performance (exact prefix caching) in isolation. CONTEXTPILOT uniquely bridges this gap through three contributions: (1) a context index with a novel distance function that actively reorders documents to maximize prefix reuse—converting cache misses into hits that prior systems cannot achieve; (2) succinct context annotations that allow LLMs to recover semantic priority despite reordering; and (3) multi-turn context traversal that identifies and deduplicates previously memorized documents, reducing prefill overhead especially under model context length constraints. This co-design enables simultaneous gains in efficiency and quality that neither approach achieves alone.

CONTEXTPILOT realizes the three design opportunities above, achieving *context reuse with negligible accuracy loss*. It features a clean, minimal interface compatible with common retrieval modules (e.g., FAISS and ElasticSearch), AI memory layer stores (e.g., Mem0), and inference engines (e.g., SGLang and vLLM), requiring only request ID tracking in the prefix cache of each engine without affecting existing functionality, enabling rapid deployment. Specifically, CONTEXTPILOT takes user prompts and their context blocks (retrieved documents, chunks, or memories), updates the context to enable effective reuse, and then passes the updated context to the inference engine for processing.

Figure 4 illustrates the key components in CONTEXTPILOT: a *context index* that tracks prefix-cache state, a *context ordering mechanism* that reorders and schedules context blocks for maximum cache hits, and a *context de-duplication mechanism* that removes redundant blocks across multi-turn conversations. The following sections describe each component in detail.

4 CONTEXT INDEX

The context index is designed to: (1) efficiently track the inference engine's prefix-cache status to enable KV-cache reuse; (2) support fast lookup of previously stored KV caches via prefix matching, enabling cross-session context reuse when overlaps exist; and (3) traverse KV caches in multi-turn conversations to detect duplicated context.

4.1 Key designs for context index

Figure 5 illustrates the structure of the context index with an example. The left panel shows the index tree, and the right panel shows the corresponding prefix-cache status. The index is organized as a tree whose root represents an empty context. Each node corresponds to a prefix stored in the prefix cache and contains child nodes that extend this prefix. Every node maintains four attributes: (1) the context

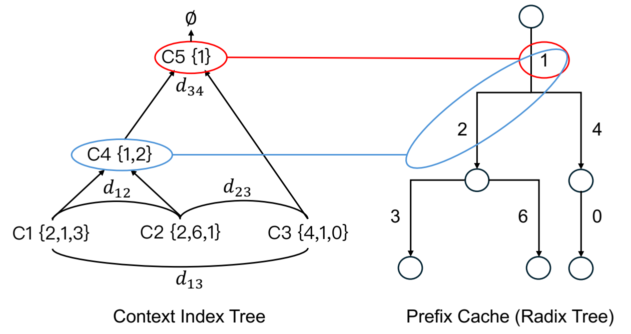


Figure 5. Context index construction with prefix-cache semantics.

containing context block IDs, (2) the search path from the root to this node, (3) an access frequency counter for cache eviction, and (4) the clustering distance at which the node was created.

Index creation. The index is built via hierarchical clustering based on prefix matching. First, we compute pairwise distances between all contexts using their overlap rate. Next, we iteratively merge the closest pair, creating a virtual node whose context is the sorted intersection representing their shared prefix. Finally, each leaf node records its search path from the root, enabling efficient traversal for both cross-session prefix matching and multi-turn duplicate detection.

As shown in Figure 5, the process begins with C1 $\{2, 1, 3\}$, C2 $\{2, 6, 1\}$, and C3 $\{4, 1, 0\}$ as leaf nodes. Since C1 and C2 have the smallest distance (sharing $\{1, 2\}$), they merge first into a virtual node C4 with context $\{1, 2\}$. C3 then merges with C4 to form the root C5 with context $\{1\}$. The resulting tree has C1–C3 as leaves storing their search paths from C5, while C4 and C5 serve as virtual nodes representing shared prefixes for cache reuse.

This construction runs in $O(N^2)$ time, where N is the number of contexts, and is fully parallelizable on CPUs and GPUs. In practice, building the index for 2,000 contexts takes 8 s on CPUs and 0.82 s on GPUs. The space complexity is $O(N \cdot K)$, where K is the average number of context blocks per query. Because the index stores only block IDs and metadata rather than full texts, its space overhead is minimal. The complete hierarchical clustering pseudocode is provided in Algorithm 4 (Appendix D).

Quantifying the overlapping between contexts. A key challenge in index construction is quantifying the overlap between contexts. We propose a *context distance function* that satisfies two requirements: (1) it captures the number of shared documents between contexts, and (2) it accounts for their positional alignment, since retrieval systems rank documents by query relevance.

To illustrate the need for this design, consider four contexts: A $\{3, 5, 1, 7\}$, B $\{2, 6, 3, 5\}$, C $\{3, 5, 8, 9\}$, and D $\{2, 6, 4, 0\}$. A naive overlap-only metric assigns identical distances (0.5) to pairs A–B, B–C, and B–D because each shares two documents. However, B and D share $\{2, 6\}$ at positions 1–2, while A and B share $\{3, 5\}$ at different positions. Our distance function (Equation 1) assigns a smaller distance to B–D, as their overlaps occur in similar positions, reflecting both overlap magnitude and positional alignment. Such patterns cannot be captured by conventional distance measures like cosine, L1, or L2 similarity, which ignore positional structure. More formally, our distance function is defined as:

$$d_{ij} = 1 - \frac{|S_{ij}|}{\max(|C_i|, |C_j|)} + \alpha \cdot \frac{\sum_{k \in S_{ij}} |p_i(k) - p_j(k)|}{|S_{ij}|} \quad (1)$$

where S_{ij} denotes the set of shared documents, $p_i(k)$ is the position of document k in context i, and $\alpha \in [0.001, 0.01]$ ensures overlap count remains the dominant factor while incorporating positional alignment.

Index update. The context index stays synchronized with the inference engine's prefix cache through lightweight request ID tracking. Each leaf node is associated with a request ID maintained by the engine. When the engine evicts cached entries, it sends the corresponding request IDs to ContextPilot, which looks up the affected nodes via a request-to-node mapping and removes them. Empty parent nodes are recursively pruned to keep the tree compact. The overall update cost is $O(h)$, where h is the tree height, requiring only a single traversal per eviction.

4.2 Key operations with context index

The context index provides two key operations:

Context search. CONTEXTPILOT frequently searches for previously stored contexts based on the current one to enable reuse. The index search algorithm (Algorithm 1) efficiently locates matching contexts by greedily descending from the root, selecting at each level the child with the minimum distance while recording positions to form a search path. The search stops upon reaching a leaf or when all children are equidistant, indicating the longest shared prefix. Updates are localized and efficient: matching an internal node appends the new context as a child ($O(1)$), while matching a leaf creates a new internal node with their intersection ($O(|C|)$). Unlike K-Means re-clustering or HNSW graph rebuilding, these updates require no tree restructuring, enabling dynamic index maintenance with minimal overhead.

For example, given context C6 $\{2, 1, 4\}$, we search the index in Figure 5. C6 first compares with the root's child C5 and finds a shared prefix $\{1\}$, descending to C5 and recording its position [0]. At C5, C6 shares $\{1, 2\}$ with C4 but only $\{1\}$ with C3, so it selects C4 and appends another [0], yielding [0,0]. At C4's children C1 $\{1, 2, 3\}$ and C2 $\{1, 2, 6\}$, all have equal distance, so the search stops and identifies C4

Algorithm 1 Context Index Tree Search

```
Require: Context $C = \{b_1, \ldots, b_k\}$, root node $R$
Ensure: Search path, best matching node
 1: $cur \leftarrow R$, $path \leftarrow []$
 2: while $cur$ has children do
 3:     $best \leftarrow \arg\min_{c \in cur.\text{children},\, C \cap c.\text{blocks} \neq \emptyset} \textsc{Dist}(C, c)$
 4:     if no overlapping child found then
 5:         break
 6:     end if
 7:     $path.\textsc{append}(\text{index of } best)$
 8:     if $best$ is leaf then
 9:         return $path$, $best$
10:     end if
11:     $cur \leftarrow best$
12: end while
13: return $path$, $cur$
```

as the best match with path [0,0]. C6 is then inserted into C4's children list at position 2, forming the final search path [0,0,2].

Search complexity scales with tree height. For contexts with common prefixes, $h = O(\log n)$ yields $O(|C| \cdot \log n)$ complexity, where n denotes the number of stored contexts. Empirically, search takes approximately 0.068 ms per request (Appendix C.3), negligible compared to prefill latency.

Context traversal. In multi-turn conversations, ContextPilot updates node context lengths by traversing the index using the stored search path. Starting from the root, it sequentially follows indices along the path until reaching the target node, then performs the update. Traversal costs $O(h)$ and is subsumed by the search overhead above.

5 Context Ordering

The context ordering mechanism aims to: (1) reorder retrieved contexts according to the current prefix-cache status to maximize KV-cache reuse; (2) schedule the ordered contexts to the inference engine with awareness of cache generation and eviction policies to enhance hit ratio; and (3) insert context annotations that recover pre-ordering semantics and maintain accuracy.

5.1 Context ordering algorithm

Formally, the context ordering algorithm (Algorithm 2) takes a batch of requests with their retrieved context blocks as input, reorders them based on prefix matches from the context index, and returns ordered contexts with maximized shared prefixes.

As illustrated in Figure 6, we begin with initialization contexts C1 $\{2, 1, 3\}$, C2 $\{2, 6, 1\}$, and C3 $\{4, 1, 0\}$, followed by new contexts C6 $\{2, 1, 4\}$, C7 $\{5, 7, 8\}$, and C8 $\{1, 2, 9\}$. Initialization contexts inherit prefixes from their parent nodes (C1, C2 from C4 with $\{1, 2\}$; C3 from C5 with $\{1\}$), while new contexts search the index (C6 and C8 match C4 and inherit $\{1, 2\}$). Each context then concatenates its matched prefix with remaining documents in their original order, producing C1 $\rightarrow \{1, 2, 3\}$, C2 $\rightarrow \{1, 2, 6\}$, C6 $\rightarrow \{1, 2, 4\}$, and C8 $\rightarrow \{1, 2, 9\}$. Unmatched contexts (e.g., C7) remain unchanged and form standalone branches. This strategy ensures overlapping contexts share common prefixes while preserving the ranking of non-shared documents.

Algorithm 2 Context Ordering for Prefix Sharing

```
Require: Context $C$ with context blocks, Context Tree root node $R$
Ensure: Reordered context $C'$
 1: $bestMatch \leftarrow \textsc{FindBestMatchNode}(C, R)$
 2: if $bestMatch$ = null then
 3:     $C' \leftarrow C$
 4:     return $C'$
 5: end if
 6: $prefix \leftarrow bestMatch.context$
 7: $remaining \leftarrow C \setminus prefix$
 8: $C' \leftarrow prefix \cup remaining$
 9: return $C'$

10: function $\textsc{FindBestMatchNode}(C, R)$
11: if $C$ is initialization context then
12:     return $C.parent$
13: else
14:     return $\textsc{SearchTree}(C, R)$
15: end if
16: end function
```

The algorithm is invoked whenever ContextPilot processes a new request. It runs in $O(|C| \cdot \log n)$ time, where n is the number of stored contexts, taking approximately 0.047 ms per request (Appendix C.3)—negligible compared to prefill.

5.2 Scheduling requests with ordered contexts

After ordering contexts, ContextPilot must schedule their execution to align with the inference engine's KV-cache generation and eviction policies; otherwise, cache reuse becomes ineffective. We therefore design a scheduling algorithm that: (1) reuses the search paths obtained during context ordering to avoid redundant tree lookups; (2) groups contexts by the first element of their search path, naturally separating cache regions; and (3) sorts contexts within each group by path length in descending order, ensuring longer prefix matches execute before shorter ones.

Figure 7 illustrates this process. In the baseline order C6, C3,

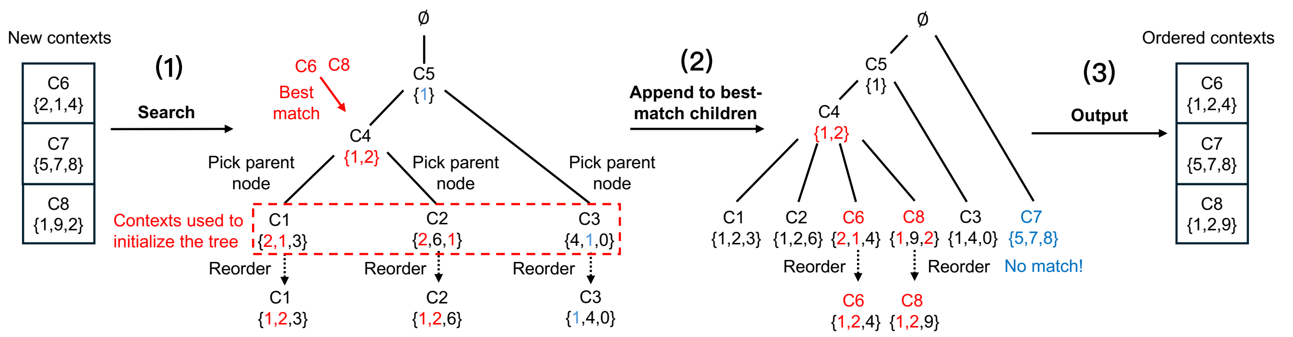


Figure 6. Example for ordering context: (1) find best-matching nodes, (2) reorder by shared prefix, and (3) append it as a child.

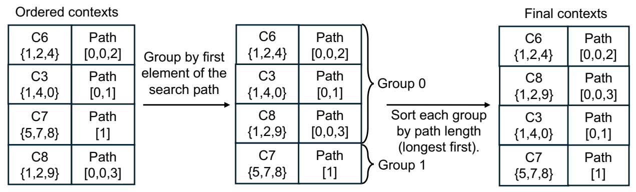


Figure 7. Example of scheduling requests with ordered contexts.

C7, C8, limited cache capacity allows only one context: C6 caches $\{1, 2, 4\}$, but C3 reuses only $\{1\}$ and evicts $\{2, 4\}$. C7 causes a full miss, caching $\{5, 7, 8\}$ and evicting all previous entries, which then forces another miss for C8 despite its shared prefix $\{1, 2\}$ with C6. This inefficiency arises because contexts with shared prefixes are not executed consecutively.

Our scheduler reorders execution to C6, C8, C3, C7, grouping prefix-sharing contexts together. C6 first caches $\{1, 2, 4\}$, then C8 immediately reuses $\{1, 2\}$ before eviction. C3 and C7 run afterward without disrupting this reuse, maximizing cache hit ratio.

Our scheduler performs $O(N)$ grouping by root-prefix path and $O(N \log N)$ in-group sorting over N contexts, with negligible real-time overhead. In contrast, existing indexing methods such as RAGCache and SGLang's LPM use a **global prefix selection that rescans a radix tree with M nodes at each decision point**, yielding $O(N \log M) + O(N \log N)$ overall as cache state evolves. By draining groups sequentially, our method **avoids repeated tree searches**, better preserves reuse under tight KV budgets, and keeps complexity independent of M. The full scheduling pseudocode is given in Algorithm 5 (Appendix D).

5.3 Context annotation for context ordering

Why reordering is safe. As shown in Section 3.2, our reproduction of the DEmO study (Guo et al., 2024a) confirms that modern LLMs are substantially less sensitive to input ordering (Table 1), explaining why reordering for cache efficiency introduces only minor accuracy perturbation and making lightweight correction mechanisms sufficient.

Why annotations still help. Despite the reduced sensitivity, reordering can still cause minor accuracy perturbation on some datasets (e.g., -1.1% on QASPER). Annotations mitigate this by reducing the model's reliance on positional signals to infer relevance. On multi-hop tasks where chaining evidence across context blocks benefits from explicit guidance, annotations not only recover lost accuracy but actively *improve* it beyond the no-reordering baseline (e.g., +4.0% F1 on MultihopRAG with Qwen3-32B; see Appendix C.2). Gains are consistent across model scales: Qwen3-4B gains +1.4% on MultihopRAG and +1.3% on NarrativeQA, while Qwen3-32B gains +4.0% and +1.2% respectively. Attention map analysis (Appendix B) confirms that annotations reshape internal attention, aligning it with semantic rather than positional priority.

Annotation mechanism. We provide the LLM with succinct annotations indicating the original relevance ranking of context blocks. Reordering contexts alters this ranking, which encodes document relevance critical for answer quality. Consider context C6, where the retriever returns documents in order $\{2, 1, 4\}$. The baseline prompt is:

[system prompt] → [Doc_2] → [Doc_1] → [Doc_4] → [question]

After reordering to $\{1, 2, 4\}$ for cache efficiency, we append an *order annotation* before the question:

[system prompt] → [Doc_1] → [Doc_2] → [Doc_4] →
[order annotation] → [question]

The annotation explicitly specifies the original relevance priority:

"Please read the context in the following priority order: [Doc_2] > [Doc_1] > [Doc_4] and answer the question."

This short instruction adds negligible token overhead during prefill yet effectively preserves the model's ability to attend to documents by their original relevance ranking. As a result, ContextPilot achieves aggressive cache optimization with minor accuracy perturbation ($\leq$1% on most datasets), and often improved accuracy on multi-hop reasoning tasks.

Algorithm 3 Context De-duplication

```
Require: New context $C_{\text{new}}$, context index $\mathcal{I}$, conversation ID $id$
Ensure: Deduplicated context $C'$, location annotations $H$
 1: $S \leftarrow \mathcal{I}$.seen_blocks$[id]$  ▷ blocks indexed in prior turns
 2: $C' \leftarrow []$, $H \leftarrow []$
 3: for each block $b$ in $C_{\text{new}}$ do
 4:     if $b \in S$ then
 5:         $h \leftarrow$ “refer to $[b]$ in previous conversation”
 6:         $C'$.APPEND$(h)$; $H$.APPEND$(h)$
 7:     else
 8:         $C'$.APPEND$(b)$
 9:     end if
10: end for
11: $S \leftarrow S \cup C_{\text{new}}$  ▷ register for future turns
12: return $C'$, $H$
```

6 Context Deduplication

The context de-duplication mechanism has two goals: (1) eliminate context blocks that share an exact prefix with previously processed ones whose KV caches are already stored, thereby minimizing redundant prefill computation; and (2) provide annotations informing the LLM which content has been de-duplicated and where the corresponding information resides in the earlier context.

Context de-duplication algorithm. Algorithm 3 formalizes the de-duplication procedure. The context index maintains a per-conversation record of all context blocks processed in prior turns, populated when the first turn's context is ordered and inserted into the index (Section 5). Given a new context, the algorithm looks up these previously indexed blocks, identifies overlaps, generates a location annotation for each duplicate, and registers the current turn's blocks for future comparisons.

We illustrate with an example. Consider a session where user C6 initially retrieves context $\{1, 2, 4\}$ in the first turn. During context ordering (Section 5), these blocks are inserted into the context index and recorded in C6's conversation history. In the second turn, a new query yields $\{1, 5, 2\}$. The algorithm queries the index's conversation record, identifies $\{1, 2\}$ as already cached from the first turn, and replaces them with location annotations, leaving only the novel block $\{5\}$ to be fully processed. The new blocks are then added to the conversation record for future turns. The algorithm runs in $O(|C|)$ time, taking approximately 0.144 ms per request (Appendix C.3)—negligible compared to prefill.

Context annotation for de-duplicated context blocks. Simply removing duplicates can degrade answer quality. To maintain quality, we insert *location annotations* that direct the LLM to corresponding context blocks in the conversation history.

For C6's second turn, the baseline prompt is:

[first turn context] → [first turn Q&A] → [Doc_1] → [Doc_5] → [Doc_2] → [second turn question]

With de-duplication and location annotations, the prompt becomes:

[first turn context] → [first turn Q&A] → [annotation_1] → [Doc_5] → [annotation_2] → [second turn question]

Here, *annotation_1* and *annotation_2* are references such as *“Please refer to [Doc_1] in the previous conversation”* and *“Please refer to [Doc_2] in the previous conversation.”* These annotations guide the LLM to prior context without repeating prefill.

7 Evaluation

Our evaluation of ContextPilot shows: (1) ContextPilot improves prefill throughput by up to 3.1× and accuracy by up to 4.0% over numerous state-of-the-art systems and methods across multi-turn, multi-session, and hybrid RAG workloads, (2) It outperforms strong baselines by 1.5-3× in throughput and 1.9-3.7% in accuracy in emerging agentic AI applications, (3) Each component (context ordering, de-duplication, and annotation design) yields clear gains and robustness with negligible overhead, and (4) Benefits scale with longer contexts and larger retrieval sizes.

Evaluation setup. Our ContextPilot implementation supports SGLang 0.4.6 and vLLM 0.10.0, requiring only request ID tracking in each engine's prefix cache without affecting existing functionality, making the changes easy to upstream and merge.

We compare ContextPilot against the following baselines: (i) LMCache (version 0.3.8), representing the state of the art in prompt caching; (ii) CacheBlend, the state of the art in KV-cache matching, integrated with LMCache; and (iii) RadixCache, based on SGLang's implementation with a Longest-Prefix-Match scheduling policy.

We omit additional baselines that achieve comparable performance to those above. (iv) HiCache (Xie et al., 2025) extends RadixCache by expanding prefix caches to lower-tier memory; since it directly builds on RadixCache, we compare against RadixCache instead. (v) RAGCache adopts a similar radix-tree structure at document granularity and shows comparable performance to RadixCache. It is not open-sourced and thus excluded from our evaluation.

Our evaluation is conducted on two GPU clusters: (1) a 16× H100 GPU cluster and (2) a 12× A6000 GPU cluster.

Table 2. Multi-session RAG results: F1 (%) and prefill throughput for four methods across three models on three datasets.

		LMCache		CacheBlend		Radix Cache		**ContextPilot (Ours)**	
Dataset	**Model**	F1	Prefill Throughput	F1	Prefill Throughput	F1	Prefill Throughput	F1	Prefill Throughput
MultihopRAG	Qwen3-4B-Instruct-2507	35.2	34710.9	34.8	85405.3	35.2	58990.1	**36.6**	**106799.5**
	Qwen3-32B	60.4	14708.6	50.1	36128.6	60.4	17682.6	**64.4**	**36296.1**
	Llama3.3-70B-Instruct	**62.9**	11596.4	54.9	14134.2	**62.9**	14777.1	**62.9**	**30046.7**
NarrativeQA	Qwen3-4B-Instruct-2507	16.0	39276.4	11.3	42819.5	16.0	39492.4	**17.3**	**57681.3**
	Qwen3-32B	28.4	15514.0	19.8	16913.2	28.4	15598.8	**29.6**	**22780.4**
	Llama3.3-70B-Instruct	37.8	12575.7	31.3	13710.5	37.8	12644.9	**38.4**	**18468.8**
QASPER	Qwen3-4B-Instruct-2507	**27.9**	29349.2	21.9	36273.5	**27.9**	33034.6	26.8	**46619.9**
	Qwen3-32B	**36.0**	15568.4	29.3	20238.9	**36.0**	17523.0	34.9	**24733.4**
	Llama3.3-70B-Instruct	**33.8**	12000.1	27.9	14829.7	**33.8**	13507.3	**33.8**	**19061.0**

For all baselines, we tune system parameters for optimal performance and accuracy, aligning our configurations with the best results reported in their respective papers. We set $\alpha = 0.001$ for the distance metric in Equation 1 across all experiments.

7.1 Performance in Retrieval-Augmented Generation

We evaluate on four RAG datasets: QASPER (Dasigi et al., 2021), MultihopRAG (Tang & Yang, 2024), NarrativeQA (Kočiský et al., 2017), and MT-RAG (Katsis et al., 2025). For QASPER, MultihopRAG, and NarrativeQA, we use a chunk size of 1024 following (Bhat et al., 2025), while MT-RAG performs document-level retrieval without chunking. We use `gte-Qwen2-7B-Instruct` as the embedding model, FAISS for similarity search on MultihopRAG and NarrativeQA, and BM25 for QASPER and MT-RAG. This setup demonstrates CONTEXTPILOT's effectiveness across diverse retrieval paradigms.

We further test CONTEXTPILOT under three RAG workloads: (1) multi-session, where independent users query concurrently; (2) multi-turn, for extended single-user dialogues; and (3) combined multi-session and multi-turn, representing production-scale conversational systems. For multi-session experiments, CONTEXTPILOT operates in *offline* mode: all contexts are pre-fetched and the context index is built before large-batch inference begins. For multi-turn and Mem0 experiments, CONTEXTPILOT operates in *online* mode with cold-start: the context index is incrementally built and updated as each turn arrives.

Multi-session RAG. We evaluate multi-session RAG on QASPER, MultihopRAG, and NarrativeQA using three models—Qwen3-4B-Instruct-2507, Qwen3-32B, and Llama3.3-70B-Instruct—on H100 GPUs with top-k=15.

Table 2 reports F1 scores and prefill throughput. CONTEXTPILOT delivers up to 3.08×, 2.05×, and 2.13× speedups over LMCache, RadixCache, and CacheBlend on MultihopRAG, and 1.3–1.6× gains on NarrativeQA and QASPER. These gains arise from context reordering that maximizes prefix overlap, improving cache hits and reducing redundant computation. In contrast, LMCache and RadixCache depend on exact prefix matching, causing recomputation even for overlapping content, while LMCache also incurs high CPU offloading costs for long contexts.

CONTEXTPILOT maintains accuracy via order annotations that preserve original retrieval rankings. CacheBlend, however, degrades sharply—F1 drops to 11.3 on NarrativeQA with Qwen3-4B versus 17.3 for CONTEXTPILOT—due to approximate KV matching and selective recomputation disrupting coherence. Notably, CONTEXTPILOT can even improve accuracy (e.g., 60.4→64.4 on MultihopRAG with Qwen3-32B) as order annotations help models prioritize key context blocks during prefill, yielding richer contextual reasoning.

Effectiveness with ever larger MoE models. We further evaluate *DeepSeek-R1 (671B)* on a GPU cluster with 32 H20 GPUs, provided by a potential industry adopter. On MultihopRAG, CONTEXTPILOT increases the cache hit ratio from 5% to 60%; on NarrativeQA, it raises the hit ratio from 6% to 38%. These improvements translate into 1.81x and 1.52x higher prefill throughput on 16xH20, respectively. Scaling to 32xH20 yields similar speedups, confirming that our approach generalizes to larger reasoning models and multi-node deployments. We provide a more detailed analysis of the DeepSeek-R1 results in Appendix A. We will also report results on MiniMax2.5, GLM5, DeepSeek-V3.2, GPT-OSS-120B, and Kimi-K2.5 as they become available.

Multi-turn RAG. We evaluate multi-turn RAG on the MT-RAG dataset using Qwen3-4B-Instruct-2507, Llama3.1-8B-Instruct, and Qwen3-30B-A3B-Thinking-2507 on a single H100 GPU. Models with longer context windows are used to handle the growing conversation history. Answer accuracy is measured via the LLM-as-a-judge method from RADBench (Kuo et al., 2025) with GPT-5, as recommended by MT-RAG.

Table 3a reports accuracy and time-to-first-token (TTFT). CONTEXTPILOT cuts TTFT by removing redundant document processing across turns through context de-duplication. It achieves 3.45×, 3.35×, and 3.09× speedups over LM-

Table 3. Performance metrics across different tasks: (a) MT-RAG results, (b) Hybrid RAG sessions, (c) Context index construction latency.

Method	**Metric**	Qwen3-4B	Llama3.1-8B	Qwen3-30B
LMCache	Acc.	62.56	**68.46**	75.12
	TTFT	0.76	1.04	1.08
CacheBlend	Acc.	50.33	56.52	X
	TTFT	0.30	0.48	X
RadixCache	Acc.	62.56	**68.46**	75.12
	TTFT	0.44	0.49	0.61
ContextPilot	Acc.	**64.27**	68.12	**75.81**
	TTFT	**0.22**	**0.31**	**0.35**

(a) Accuracy (%) and time-to-first-token (TTFT)(s) on MT-RAG for four methods across three models. X = not supported.

	# Sessions				
Method	**2**	**4**	**8**	**16**	**32**
LMCache	0.81	0.87	0.94	1.19	1.72
CacheBlend	0.40	0.42	0.45	0.54	0.78
RadixCache	0.46	0.49	0.53	0.67	0.97
ContextPilot	**0.24**	**0.29**	**0.34**	**0.41**	**0.65**

(b) Time-to-first-token (seconds) performance in seconds for Qwen3-4B-Instruct-2507 model evaluated under hybrid RAG workloads with varying concurrent session counts ranging from 2 to 32 sessions.

	# Contexts (N_{ctx})					
k	**128**	**512**	**4k**	**8k**	**12k**	**100k**
3	0.64	0.65	1.51	3.54	7.48	687.64
5	0.66	0.66	1.55	3.58	7.55	688.21
10	0.67	0.68	1.59	3.63	7.63	689.07
15	0.69	0.69	1.62	3.67	7.69	687.95
20	0.71	0.72	1.66	3.72	7.78	690.12

(c) Context index construction latency (seconds) as a function of the number of contexts N_{ctx} and top-k. Columns labeled **128–100k** denote the total contexts inserted at build time (N_{ctx}; *1k* = 1,000).

Cache on Qwen3-4B, Llama3.1-8B, and Qwen3-30B, respectively, and up to 2.00× over RadixCache and 1.55× over CacheBlend. These gains arise from detecting and skipping duplicated contexts that baselines repeatedly recompute.

ContextPilot also preserves accuracy via location annotations that direct models to previously seen documents. CacheBlend, by contrast, drops to 50.33% on Qwen3-4B versus 64.27% for ContextPilot, as its approximate KV matching and selective recomputation disrupt multi-turn coherence. ContextPilot even improves accuracy in some cases (e.g., from 62.56% to 64.27% on Qwen3-4B) by maintaining contextual continuity while avoiding redundant computation.

Multi-session, multi-turn RAG. We evaluate the combined multi-session and multi-turn scenario under real-world deployment using Qwen3-4B-Instruct-2507 on H100 GPUs, varying concurrency from 2 to 32 sessions.

Table 3b shows that ContextPilot achieves the lowest TTFT at all concurrency levels by ordering contexts to maximize prefix overlap. At 2 sessions, it delivers 3.38×, 1.92×, and 1.67× speedups over LMCache, RadixCache, and CacheBlend, respectively; at 32 sessions, the gains remain substantial at 2.65×, 1.49×, and 1.20×. These consistent improvements demonstrate that ContextPilot's context ordering effectively maintains prefix overlap as concurrent, diverse queries scale, while baselines suffer from cache thrashing and redundant recomputation.

7.2 Performance in Agentic Applications

We study two representative agentic workloads that are increasingly common in production—multi-agent reasoning and AI memory—both of which repeatedly surface long contexts across turns and sessions, making them natural candidates for context reuse.

Multi-agent reasoning. We evaluate ContextPilot within an emerging multi-agent reasoning paradigm *Chain-of-Agent (CoA)* (Zhang et al., 2024), which mitigates long-context limitations by coordinating multiple specialized agents through natural-language communication. Each worker agent handles a document segment for localized reasoning, while a manager agent aggregates their intermediate results into a final answer.

ContextPilot enhances CoA through agent-aware routing. In multi-session settings, when documents appear in multiple queries, they are routed to the agent that processed them before, enabling KV-cache reuse. In multi-turn conversations, repeated documents are deduplicated, and location annotations direct agents to prior content, eliminating redundant prefilling while preserving CoA's distributed reasoning.

We deploy three CoA configurations on MultihopRAG, each with 15 agents using Llama3.1-8B, Llama3.2-3B, or Qwen3-4B-Instruct, where each agent processes one retrieved document (k=15). With Qwen3-4B, accuracy increases from **48.3%** to **50.2%** and throughput by **1.8×**; with Llama3.1-8B, accuracy rises from **50.7%** to **54.4%** with a **2.1×** speedup. These results show that ContextPilot reduces redundant processing and boosts both accuracy and throughput via effective KV-cache reuse across agents.

Agentic memory systems. Agentic memory systems are an important long-context workload because they repeatedly retrieve and attach user-specific memories across turns and sessions, creating substantial cross-request context overlap and making them a natural fit for context reuse. Hence, we evaluate ContextPilot with Mem0 (Chhikara et al., 2025), a popular AI memory system that stores and fetches user-specific memories as context blocks for personalized inference. We configure Mem0 with Qwen3-4B on the LoCoMo (Maharana et al., 2024) benchmark and evaluate at memory search depths k=20 and k=100, using GPT-4.1 as an LLM judge for accuracy.

ContextPilot treats Mem0's fetched memory entries as context blocks, indexing and reordering them via the context index in online mode. At k=100, ContextPilot reduces TTFT from 0.101 s to 0.055 s (**1.83**× speedup) with a minor accuracy trade-off (0.437→0.420). At k=20, although LoCoMo conversations are relatively short (∼26K tokens across all turns on average), ContextPilot still improves both TTFT (0.038 s→0.031 s, 1.23×) and accuracy (0.440→0.460), as context reordering places more relevant memories earlier. These results confirm the effectivness of ContextPilot in memory-augmented AI workloads.

7.3 Performance breakdown, overhead and robustness

Contribution of ordering and scheduling. We analyze how each of ContextPilot's two components—*ordering* and *scheduling*—contributes to performance on MultihopRAG (k=15) across two inference engines (SGLang, vLLM) and two models (Qwen3-32B, Llama3.3-70B-Instruct) on H100 GPUs. As shown in Figure 8, each component adds incremental cache hit ratio gains. For SGLang with Qwen3-32B, the baseline hit ratio of 8.49% rises to 20.56% with ordering alone and to 33.97% with scheduling added—a 4× overall improvement. vLLM with Llama3.3-70B follows a similar trend: 10.7% → 30.8% → 43.2%. These gains directly translate to reduced prefill computation and lower TTFT.

Performance with long-running workloads. It is necessary to verify that ContextPilot's gains persist under realistic, long-running workloads (details in Appendix C.1). The results confirm that the improvements are sustained throughout the workload: ContextPilot maintains ∼34% cache hit ratio versus the baseline's ∼7%, a consistent 5× advantage. Cumulative radix-tree reuse further quantifies the benefit over time: ContextPilot achieves 10.33M cached tokens versus the baseline's 2.42M (4.27×) on Llama3.3-70B-Instruct, and 10.50M versus 2.75M (3.82×) on Qwen3-32B. The "w/o Scheduling" variant (6.85M, 2.83×) shows that both ordering and scheduling contribute to the sustained prefill acceleration.

Model accuracy under context reuse. Because ContextPilot modifies the original order of context via context reordering and de-duplication, we verify that these optimizations do not trade correctness for speed. A full per-component breakdown (Appendix C.2) shows that reordering alone causes ≤1% F1 variation, while adding annotations not only recovers this gap but even improves accuracy by +1.4–4.4%. We leave a deeper investigation of this surprising behavior for future work.

Overhead of context index construction. We evaluate index construction overhead as context count scales from 128 to 100K under varying retrieval depths (k=5–20) on NVIDIA A6000 GPUs, spanning online deployments (128–512 contexts), offline workloads (up to 12K), and an extreme stress test (100K). As shown in Table 3c, construction scales smoothly with context count and remains stable across retrieval depths, showing minimal sensitivity to k. At

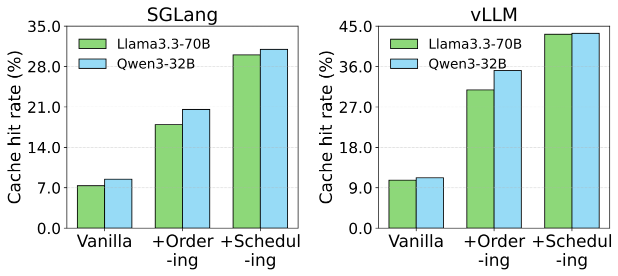


Figure 8. Performance breakdown of key components.

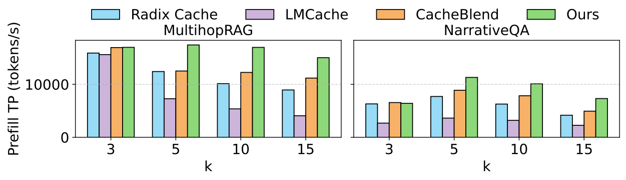


Figure 9. Prefill throughput under different top k values.

12K contexts, construction completes in 7.48 s—negligible relative to total prefill latency. Even at 100K, ContextPilot finishes within 12 minutes on GPU, far faster than multi-hour offline prefilling, and can be further accelerated via multi-GPU indexing. Overall, index construction is a lightweight, one-time cost amortized across many queries, with per-request overhead of just ∼0.2 ms (Appendix C.3)—negligible compared to seconds of prefill latency—making ContextPilot practical for large-scale deployments.

System overhead with zero context overlap. Zero context overlap represents the worst case for ContextPilot, as it isolates pure system overhead with no reuse benefit. Using a synthetic RAG workload with no retrieval overlap, ContextPilot adds only 0.72 s of prefill latency for 1K contexts (one-hour job), demonstrating that querying the context index during operation incurs negligible overhead.

Performance with growing context lengths. We evaluate ContextPilot under varying retrieval depths (k=3,5,10,15) on NarrativeQA and MultihopRAG using A6000 GPUs. As shown in Figure 9, ContextPilot consistently achieves the highest prefill throughput across all k values. On MultihopRAG, it sustains 1.5–2.0× speedups over baselines as k grows from 3 to 15; on NarrativeQA, it maintains 1.3–1.6× gains. Notably, while baseline throughput declines at larger k due to reduced cache reuse, ContextPilot remains stable—its context ordering preserves prefix overlap even as retrieved contexts grow longer.

Effectiveness of context annotations. We tested placing annotations before and after questions, observing negligible impact on accuracy (variations below 0.5%). Attention heatmap analysis further confirms that annotations effectively guide the model toward relevant documents, improving downstream accuracy. We include this analysis in the supplementary materials (Appendix B).

Impact of prefix cache size. The prefix KV-cache stores precomputed attention states for reuse across requests; a larger cache retains more contexts, increasing the chance of a cache hit and reducing redundant prefill computation. We evaluate this effect on MultihopRAG across A6000 (48 GB) and H100 (80 GB) GPUs. Because ContextPilot orders contexts to maximize prefix overlap, it benefits disproportionately from larger caches: SGLang hit ratios improve from 29.64% to 33.97%, and vLLM from 35.90% to 43.4%, translating directly into higher prefill throughput. In contrast, baselines see smaller gains since their context ordering does not systematically exploit the additional capacity.

8 Related Works

RAG system optimization. System-level approaches such as METIS (Ray et al., 2025) and Chameleon (Jiang et al., 2024) optimize workflow and hardware efficiency, jointly tuning retrieval settings or using heterogeneous accelerators (He et al., 2025) to reduce latency and boost throughput. ContextPilot complements them by improving KV-cache reuse.

Reranking in retrieval systems. Rerankers refine retrieval results via learned ranking models (Adeyemi et al., 2024; Li et al., 2023; Zhang et al., 2025d); HyperRAG (An et al., 2025) further enables KV reuse at the reranking stage. ContextPilot instead operates downstream, ordering reranked document IDs to maximize prefix overlap while preserving relevance through order annotations.

Fine-tuning with positional re-encoding. Methods like BlockAttention, KVLink, and TurboRAG (Ma et al., 2025; Yang et al., 2025a; Lu & Tang, 2024) fine-tune models to reuse KV caches via position re-encoding and pre-stored states. They improve efficiency but require heavy training and large cache storage. ContextPilot is training-free and shows strong promise in numerous RAG applications.

Faster KV-cache compute. CacheBlend (Yao et al., 2025) and related works (Liu et al., 2024; Agarwal et al., 2025; Yang et al., 2025b; Deng et al., 2025; Liu et al., 2025a; Du et al., 2026; Pan et al., 2025; Tang et al., 2024; Wu et al., 2025; Chen et al., 2025; Zhang et al., 2025a) optimize KV-cache computation through compression, decoding, parallel encoding, cache sharing, or FLOP-aware admission and eviction policies. ContextPilot complements these by operating at the context level, ordering and de-duplicating inputs to maximize reuse, and can be combined with them for further gains.

9 Conclusion

We presented ContextPilot, a context-reuse system that accelerates long-context prefill with negligible accuracy loss. ContextPilot uniformly represents external inputs as *context blocks* and applies a common set of mechanisms—context indexing, ordering, de-duplication, and succinct annotations to boost prefix KV cache reuse across diverse workloads. Its modular design enables new system and algorithmic research on context engineering, management, and optimization for long-context AI. Looking ahead, we envision ContextPilot as a co-optimization framework that jointly decides *what* context to feed and *how* to serve it, maximizing both inference efficiency and output quality.

Acknowledgements

We sincerely thank our shepherd and the MLSys reviewers for insightful feedback that significantly improved this manuscript. We also thank Minjie Wang (formerly Amazon AI Labs, now The University of Hong Kong (Shanghai X-Lab)) for valuable early-stage discussions, and Ryan Tsui (The University of Edinburgh) for contributions to the implementation. We further thank the Tencent WeChat (Weixin) Group for access to H20 GPU clusters for large-scale MoE evaluation and for insightful feedback. Finally, we thank the School of Informatics, University of Edinburgh and the Edinburgh International Data Facility (EIDF) for providing GPU resources and funding that supported this research.

References

AboulEla, S., Zabihitari, P., Ibrahim, N., Afshar, M., and Kashef, R. Exploring rag solutions to reduce hallucinations in llms. In *2025 IEEE International systems Conference (SysCon)*, pp. 1–8, 2025. doi: 10.1109/SysCon64521.2025.11014810.

Adeyemi, M., Oladipo, A., Pradeep, R., and Lin, J. Zero-shot cross-lingual reranking with large language models for low-resource languages. In Ku, L.-W., Martins, A., and Srikumar, V. (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 650–656, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-short.59. URL `https://aclanthology.org/2024.acl-short.59/`.

Agarwal, S., Sundaresan, S., Mitra, S., Mahapatra, D., Gupta, A., Sharma, R., Kapu, N. J., Yu, T., and Saini, S. Cache-craft: Managing chunk-caches for efficient retrieval-augmented generation. *Proc. ACM Manag. Data*, 3(3), June 2025. doi: 10.1145/3725273. URL `https://doi.org/10.1145/3725273`.

Alzubi, S., Brooks, C., Chiniya, P., Contente, E., von Gerlach, C., Irwin, L., Jiang, Y., Kaz, A., Nguyen, W., Oh, S., Tyagi, H., and Viswanath, P. Open deep search: Democratizing search with open-source reasoning agents, 2025. URL https://arxiv.org/abs/2503.20201.

An, Y., Cheng, Y., Park, S. J., and Jiang, J. Hyperrag: Enhancing quality-efficiency tradeoffs in retrieval-augmented generation with reranker kv-cache reuse, 2025. URL https://arxiv.org/abs/2504.02921.

Ayala, O. and Bechard, P. Reducing hallucination in structured outputs via retrieval-augmented generation. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, pp. 228–238. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.naacl-industry.19. URL http://dx.doi.org/10.18653/v1/2024.naacl-industry.19.

Bhat, S. R., Rudat, M., Spiekermann, J., and Flores-Herr, N. Rethinking chunk size for long-document retrieval: A multi-dataset analysis, 2025. URL https://arxiv.org/abs/2505.21700.

Bowman, S. R., Angeli, G., Potts, C., and Manning, C. D. A large annotated corpus for learning natural language inference. In *EMNLP*, 2015.

Chang, E. Y. and Geng, L. Sagallm: context management, validation, and transaction guarantees for multi-agent llm planning. *arXiv preprint arXiv:2503.11951*, 2025.

Chen, G., Feng, Q., Ni, J., Li, X., and Shieh, M. Q. RAPID: Long-context inference with retrieval-augmented speculative decoding. In *International Conference on Machine Learning (ICML)*, pp. 8093–8107. PMLR, 2025.

Cheng, Y., Liu, Y., Yao, J., An, Y., Chen, X., Feng, S., Huang, Y., Shen, S., Du, K., and Jiang, J. Lmcache: An efficient kv cache layer for enterprise-scale llm inference, 2025. URL https://arxiv.org/abs/2510.09665.

Chhikara, P., Khant, D., Aryan, S., Singh, T., and Yadav, D. Mem0: Building production-ready ai agents with scalable long-term memory, 2025. URL https://arxiv.org/abs/2504.19413.

Chung, Y., Kakkar, G. T., Gan, Y., Milne, B., and Ozcan, F. Is long context all you need? leveraging LLM's extended context for NL2SQL. In *Proceedings of the VLDB Endowment*, 2025. URL https://arxiv.org/abs/2501.12372.

Dasigi, P., Lo, K., Beltagy, I., Cohan, A., Smith, N. A., and Gardner, M. A dataset of information-seeking questions and answers anchored in research papers, 2021. URL https://arxiv.org/abs/2105.03011.

Deng, Y., You, Z., Xiang, L., Li, Q., Yuan, P., Hong, Z., Zheng, Y., Li, W., Li, R., Liu, H., Mouratidis, K., Yiu, M. L., Li, H., Shen, Q., Mao, R., and Tang, B. Alayadb: The data foundation for efficient and effective long-context llm inference. In *Companion of the 2025 International Conference on Management of Data*, SIGMOD/PODS '25, pp. 364–377, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715648. doi: 10.1145/3722212.3724428. URL https://doi.org/10.1145/3722212.3724428.

Du, D., Cao, S., Cheng, J., Mai, L., Cao, T., and Yang, M. Bitdecoding: Unlocking tensor cores for long-context llms with low-bit kv cache. In *2026 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2026.

Du, Y., Tian, M., Ronanki, S., Rongali, S., Bodapati, S., Galstyan, A., Wells, A., Schwartz, R., Huerta, E. A., and Peng, H. Context length alone hurts LLM performance despite perfect retrieval. In *Findings of the Association for Computational Linguistics: EMNLP*, 2025. URL https://arxiv.org/abs/2510.05381.

Fu, Y., Xue, L., Huang, Y., Brabete, A., Ustiugov, D., Patel, Y., and Mai, L. Serverlessllm: Low-latency serverless inference for large language models. In Gavrilovska, A. and Terry, D. B. (eds.), *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pp. 135–153. USENIX Association, 2024. URL https://www.usenix.org/conference/osdi24/presentation/fu.

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, M., and Wang, H. Retrieval-augmented generation for large language models: A survey, 2024. URL https://arxiv.org/abs/2312.10997.

Gim, I., Chen, G., seob Lee, S., Sarda, N., Khandelwal, A., and Zhong, L. Prompt cache: Modular attention reuse for low-latency inference, 2024. URL https://arxiv.org/abs/2311.04934.

Guo, Q., Wang, L., Wang, Y., Ye, W., and Zhang, S. What makes a good order of examples in in-context learning. In Ku, L.-W., Martins, A., and Srikumar, V. (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 14892–14904, Bangkok, Thailand, August 2024a. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.

884. URL https://aclanthology.org/2024.findings-acl.884/.

Guo, S., Deng, C., Wen, Y., Chen, H., Chang, Y., and Wang, J. Ds-agent: Automated data science by empowering large language models with case-based reasoning. *arXiv preprint arXiv:2402.17453*, 2024b.

Gupta, N., Chatterjee, R., Haas, L., Tao, C., Wang, A., Liu, C., Oiwa, H., Gribovskaya, E., Ackermann, J., Blitzer, J., Goldshtein, S., and Das, D. DeepSearchQA: Bridging the comprehensiveness gap for deep research agents, 2026. URL https://arxiv.org/abs/2601.20975.

He, C., Huang, Y., Mu, P., Miao, Z., Xue, J., Ma, L., Yang, F., and Mai, L. Waferllm: Large language model inference at wafer scale. In Zhou, L. and Zhou, Y. (eds.), *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, pp. 257–273. USENIX Association, 2025. URL https://www.usenix.org/conference/osdi25/presentation/he.

Hu, M. and Liu, B. Mining and summarizing customer reviews. In *KDD*, 2004.

Hu, Y., Liu, S., Yue, Y., Zhang, G., Liu, B., Zhu, F., Lin, J., Guo, H., Dou, S., Xi, Z., et al. Memory in the age of AI agents, 2025. URL https://arxiv.org/abs/2512.13564.

Hua, Q., Ye, L., Fu, D., Xiao, Y., Cai, X., Wu, Y., Lin, J., Wang, J., and Liu, P. Context engineering 2.0: The context of context engineering. *arXiv preprint arXiv:2510.26493*, 2025.

Jiang, W., Zeller, M., Waleffe, R., Hoefler, T., and Alonso, G. Chameleon: a heterogeneous and disaggregated accelerator system for retrieval-augmented language models. *Proc. VLDB Endow.*, 18(1):42–52, 2024.

Jiang, Y., Fu, Y., Huang, Y., Nie, P., Lu, Z., Xue, L., He, C., Sit, M.-K., Xue, J., Dong, L., Miao, Z., Du, D., Xu, T., Zou, K., Ponti, E., and Mai, L. Moe-cap: Benchmarking cost, accuracy and performance of sparse mixture-of-experts systems. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Jin, B., Yoon, J., Han, J., and Arik, S. O. Long-context llms meet rag: Overcoming challenges for long inputs in rag, 2024a. URL https://arxiv.org/abs/2410.05983.

Jin, C., Zhang, Z., Jiang, X., Liu, F., Liu, X., Liu, X., and Jin, X. Ragcache: Efficient knowledge caching for retrieval-augmented generation, 2024b. URL https://arxiv.org/abs/2404.12457.

Kang, M., Chen, W.-N., Han, D., Inan, H. A., Wutschitz, L., Chen, Y., Sim, R., and Rajmohan, S. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*, 2025.

Katsis, Y., Rosenthal, S., Fadnis, K., Gunasekara, C., Lee, Y.-S., Popa, L., Shah, V., Zhu, H., Contractor, D., and Danilevsky, M. Mtrag: A multi-turn conversational benchmark for evaluating retrieval-augmented generation systems, 2025. URL https://arxiv.org/abs/2501.03468.

Kirsch, L., Harrison, J., Sohl-Dickstein, J., and Metz, L. General-purpose in-context learning by meta-learning transformers. *arXiv preprint arXiv:2212.04458*, 2022.

Kočiský, T., Schwarz, J., Blunsom, P., Dyer, C., Hermann, K. M., Melis, G., and Grefenstette, E. The narrativeqa reading comprehension challenge, 2017. URL https://arxiv.org/abs/1712.07040.

Kuo, T.-L., Liao, F.-T., Hsieh, M.-W., Chang, F.-C., Hsu, P.-C., and Shiu, D.-S. Rad-bench: Evaluating large language models capabilities in retrieval augmented dialogues, 2025. URL https://arxiv.org/abs/2409.12558.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

Laban, P., Fabbri, A. R., Xiong, C., and Wu, C.-S. Summary of a haystack: A challenge to long-context LLMs and RAG systems. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 9885–9903, 2024.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. Retrieval-augmented generation for knowledge-intensive nlp. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Li, C., Liu, Z., Xiao, S., and Shao, Y. Making large language models a better foundation for dense retrieval, 2023.

Li, G., Zhou, X., Zhang, X., and Zhang, H. Database perspective on LLM inference systems. *Proceedings of the VLDB Endowment*, 18, 2025. URL https://www.vldb.org/pvldb/vol18/p5504-li.pdf.

Li, Z., Li, C., Zhang, M., Mei, Q., and Bendersky, M. Retrieval augmented generation or long-context llms? a comprehensive study and hybrid approach. *arXiv preprint arXiv:2407.16833*, 2024.

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., and Liang, P. Lost in the middle: How language models use long contexts, 2023. URL https://arxiv.org/abs/2307.03172.

Liu, Y., Li, H., Cheng, Y., Ray, S., Huang, Y., Zhang, Q., Du, K., Yao, J., Lu, S., Ananthanarayanan, G., Maire, M., Hoffmann, H., Holtzman, A., and Jiang, J. Cachegen: Kv cache compression and streaming for fast large language model serving, 2024. URL https://arxiv.org/abs/2310.07240.

Liu, Y., Huang, Y., Yao, J., Feng, S., Gu, Z., Du, K., Li, H., Cheng, Y., Jiang, J., Lu, S., Musuvathi, M., and Choukse, E. Droidspeak: Kv cache sharing for cross-llm communication and multi-llm serving, 2025a. URL https://arxiv.org/abs/2411.02820.

Liu, Y., Si, C., Narasimhan, K. R., and Yao, S. Contextual experience replay for self-improvement of language agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14179–14198, 2025b.

Lu, S. and Tang, Y. Turborag: Accelerating retrieval-augmented generation with precomputed kv caches for chunked text. *arXiv preprint arXiv:2410.07590*, 2024.

Lumer, E., Nizar, F., Jangiti, A., Frank, K., Gulati, A., Phadate, M., and Subbiah, V. K. Don't break the cache: An evaluation of prompt caching for long-horizon agentic tasks, 2026. URL https://arxiv.org/abs/2601.06007.

Ma, D., Wang, Y., and Tian, L. Block-attention for efficient prefilling, 2025. URL https://arxiv.org/abs/2409.15355.

Maharana, A., Lee, D.-H., Tulyakov, S., Bansal, M., Barbieri, F., and Fang, Y. Evaluating very long-term conversational memory of LLM agents. In *ACL*, 2024.

NVIDIA. How nvidia uses rag to generate synthetic data for llm improvement. https://developer.nvidia.com/blog, 2024.

Pan, R., Wang, Z., Jia, Z., Karakus, C., Zancato, L., Dao, T., Wang, Y., and Netravali, R. Marconi: Prefix caching for the era of hybrid LLMs. In *Proceedings of Machine Learning and Systems (MLSys)*, 2025. URL https://arxiv.org/abs/2411.19379.

Pang, B. and Lee, L. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In *ACL*, 2004.

Rajasekaran, P., Dixon, E., Ryan, C., and Hadfield, J. Effective context engineering for AI agents, 2025. URL https://www.anthropic.com. Anthropic Engineering blog.

Raju, R., Ji, M., Upasani, S., Li, B., and Thakker, U. The limits of long-context reasoning in automated bug fixing, 2026. URL https://arxiv.org/abs/2602.16069.

Ray, S., Pan, R., Gu, Z., Du, K., Feng, S., Ananthanarayanan, G., Netravali, R., and Jiang, J. Metis: Fast quality-aware rag systems with configuration adaptation. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, pp. 606–622, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400718700. doi: 10.1145/3731569.3764855. URL https://doi.org/10.1145/3731569.3764855.

Shen, H., Yan, H., Xing, Z., Liu, M., Li, Y., Chen, Z., Wang, Y., Wang, J., and Ma, Y. Ragsynth: Synthetic data for robust and faithful rag component optimization, 2025. URL https://arxiv.org/abs/2505.10989.

Shuster, K., Poff, S., Chen, M., Kiela, D., and Weston, J. Retrieval augmentation reduces hallucination in conversation, 2021. URL https://arxiv.org/abs/2104.07567.

Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., and Potts, C. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013.

Tang, J., Zhao, Y., Zhu, K., Xiao, G., Kasikci, B., and Han, S. 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.

Tang, Y. and Yang, Y. Multihop-rag: Benchmarking retrieval-augmented generation for multi-hop queries, 2024. URL https://arxiv.org/abs/2401.15391.

Varambally, S., Voice, T., Sun, Y., Chen, Z., Yu, R., and Ye, K. Hilbert: Recursively building formal proofs with informal reasoning, 2025. URL https://arxiv.org/abs/2509.22819.

Wu, W., Pan, Z., Fu, K., Wang, C., Chen, L., Bai, Y., Wang, T., Wang, Z., and Xiong, H. TokenSelect: Efficient long-context inference and length extrapolation for LLMs via dynamic token-level KV cache selection. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 21275–21292, 2025.

Xie, Z., Xu, Z., Zhao, M., An, Y., Mailthody, V. S., Mahlke, S., Garland, M., and Kozyrakis, C. Strata: Hierarchical context caching for long context language model serving, 2025. URL https://arxiv.org/abs/2508.18572.

Yang, J., Hou, B., Wei, W., Bao, Y., and Chang, S. Kvlink: Accelerating large language models via efficient kv cache reuse, 2025a. URL https://arxiv.org/abs/2502.16002.

Yang, X., Chen, T., and Chen, B. Ape: Faster and longer context-augmented generation via adaptive parallel encoding, 2025b. URL https://arxiv.org/abs/2502.05431.

Yao, J., Li, H., Liu, Y., Ray, S., Cheng, Y., Zhang, Q., Du, K., Lu, S., and Jiang, J. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 94–109, 2025. doi: 10.1145/3689031.3696098. URL https://doi.org/10.1145/3689031.3696098.

Yu, H., Chen, T., Feng, J., Chen, J., Dai, W., Yu, Q., Zhang, Y.-Q., Ma, W.-Y., Liu, J., Wang, M., and Zhou, H. MemAgent: Reshaping long-context LLM with multi-conv RL-based memory agent, 2025. URL https://arxiv.org/abs/2507.02259.

Yue, Z., Zhuang, H., Bai, A., Hui, K., Jagerman, R., Zeng, H., Qin, Z., Wang, D., Wang, X., and Bendersky, M. Inference scaling for long-context retrieval augmented generation. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025.

Zhan, H., Guo, Y., Wen, B., Li, Z., Ling, C., and Zhao, L. A survey of context engineering for large language models, 2025. URL https://arxiv.org/abs/2507.13334.

Zhang, H., Ji, X., Chen, Y., Fu, F., Miao, X., Nie, X., Chen, W., and Cui, B. PQCache: Product quantization-based KVCache for long context LLM inference. *Proceedings of the ACM on Management of Data*, 3(3):1–30, 2025a.

Zhang, Q., Hu, C., Upasani, S., Ma, B., Hong, F., Kamanuru, V., Rainton, J., Wu, C., Ji, M., Li, H., Thakker, U., Zou, J., and Olukotun, K. Agentic context engineering: Evolving contexts for self-improving language models, 2025b. URL https://arxiv.org/abs/2510.04618.

Zhang, X., Zhang, P., Luo, S., Tang, J., Wan, Y., Yang, B., and Huang, F. Culturesynth: A hierarchical taxonomy-guided and retrieval-augmented framework for cultural question-answer synthesis, 2025c. URL https://arxiv.org/abs/2509.10886.

Zhang, Y., Sun, R., Chen, Y., Pfister, T., Zhang, R., and Arik, S. Chain of agents: Large language models collaborating on long-context tasks. *Advances in Neural Information Processing Systems*, 37:132208–132237, 2024.

Zhang, Y., Li, M., Long, D., Zhang, X., Lin, H., Yang, B., Xie, P., Yang, A., Liu, D., Lin, J., Huang, F., and Zhou, J. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025d.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

Zhou, J., Liu, Z., Liu, Z., Xiao, S., Wang, Y., Zhao, B., Zhang, C. J., Lian, D., and Xiong, Y. Megapairs: Massive data synthesis for universal multimodal retrieval, 2024. URL https://arxiv.org/abs/2412.14475.

Zilliz. Deepsearcher: Open-source deep research on private data. GitHub repository, 2025. URL https://github.com/zilliztech/deep-searcher.

A DeepSeek-R1 Results

Table 4 presents end-to-end results for DeepSeek-R1 on MultihopRAG and NarrativeQA datasets, evaluated on 16×H20 and 32×H20 GPUs. ContextPilot achieves 1.81× and 1.52× throughput improvements on MultihopRAG and NarrativeQA respectively, with these speedups remaining consistent across both 16 and 32 GPU deployments using context-aware routing.

Table 4. DeepSeek-R1 results formatted for vertical readability. By stacking hardware configurations, the table width is minimized while preserving all data points.

Method	Hardware	Prefill TP (tok/s)	Cache Hit	F1 (%)
Dataset: MultihopRAG				
Vanilla	16×H20	9636.69	5.12%	64.15
	32×H20	18406.08	4.17%	64.15
ContextPilot w/o Annotations	16×H20	17498.75	60.37%	64.09
	32×H20	33072.64	58.41%	64.09
ContextPilot (Ours)	16×H20	**17498.75**	**60.37%**	**64.68**
	32×H20	**33072.64**	**58.41%**	**64.68**
Dataset: NarrativeQA				
Vanilla	16×H20	8687.98	6.08%	40.20
	32×H20	16247.32	5.14%	40.20
ContextPilot w/o Annotations	16×H20	13201.52	38.24%	40.38
	32×H20	24686.84	35.71%	40.38
ContextPilot (Ours)	16×H20	**13201.52**	**38.24%**	**41.08**
	32×H20	**24686.84**	**35.71%**	**41.08**

B Attention Map Analysis

Since context engineering (Rajasekaran et al., 2025) and in-context learning (Kirsch et al., 2022) strongly influence model inference, we analyze attention patterns when explicit annotations are introduced to recall the original relevance ranking.

Figure 10 and Figure 11 compare the final-layer attention maps of Qwen3 and LLaMA3.3 under this setup. When given explicit document-priority cues, both models exhibit consistent attention behaviors despite architectural differences. They correctly focus on document tokens ([Doc_1], [Doc_2], [Doc_3]), reflecting awareness of the mismatch between the reordered and original sequences, as indicated by intersections between queries in the annotation region and keys in the context region. As the context re-aligns with the original sequence, both models emphasize ([Doc_2]) while parsing ([Doc_1]) and ([Doc_3]), showing that the cue ([Doc_2] > [Doc_1] > [Doc_3]) effectively directs cross-document attention. Hence, explicit annotations reshape internal attention, aligning it with semantic rather than positional priority.

This finding supports a central hypothesis of ContextPilot: explicit annotations can make a comeback from the accuracy lost to reordering and filtering by re-establishing alignment with the original retrieval semantics.

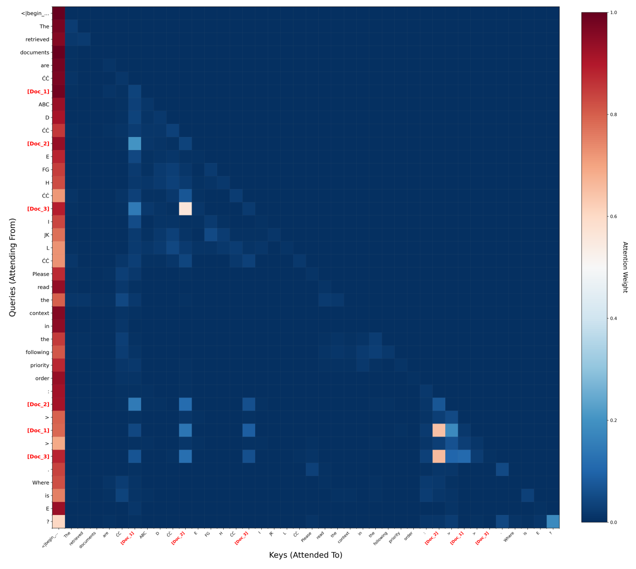


Figure 10. Attention map of the last layer attention of LLaMA3.3 (Head 14) for the prompt "The retrieved documents are [Doc_1] ABCD [Doc_2] EFGH [Doc_3] IJKL. Please read the context in the following priority order: [Doc_2] > [Doc_1] > [Doc_3]. Where is E?".

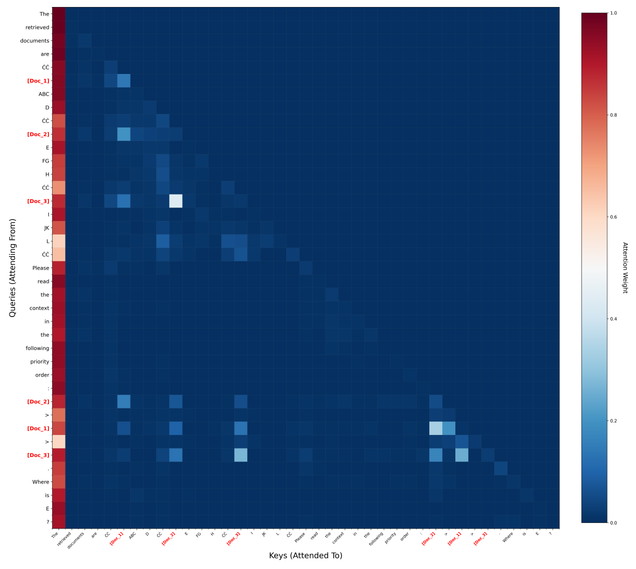


Figure 11. Attention map of the last layer attention of Qwen3 (Head 9) for the prompt "The retrieved documents are [Doc_1] ABCD [Doc_2] EFGH [Doc_3] IJKL. Please read the context in the following priority order: [Doc_2] > [Doc_1] > [Doc_3]. Where is E?".

C Additional Evaluation Details

C.1 Time-Series Metrics

Figure 12 shows how cache hit ratio evolves as the workload progresses. ContextPilot maintains approximately 34% cache hit ratio compared to baseline's 7% throughout the entire workload, demonstrating a sustained 5× improvement that is not a transient warm-up effect.

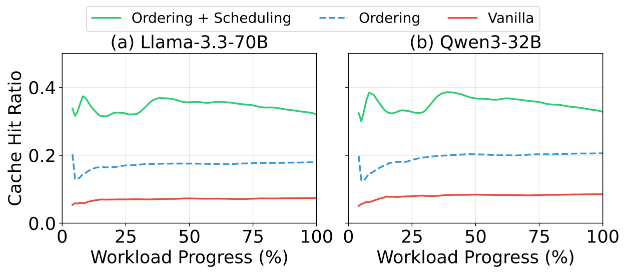


Figure 12. Cache hit ratio over workload progress for Llama-3.3-70B and Qwen3-32B. ContextPilot maintains ~5× higher hit ratio throughout execution.

Figure 13 presents cumulative cached tokens as a metric for radix tree prefix reuse. ContextPilot achieves 10.33M cached tokens versus baseline's 2.42M at completion on Llama3.3-70B-Instruct (4.27×), and 10.50M versus 2.75M on Qwen3-32B (3.82×). The "w/o Scheduling" variant (6.85M, 2.83×) confirms that both ordering and scheduling contribute to the improvement.

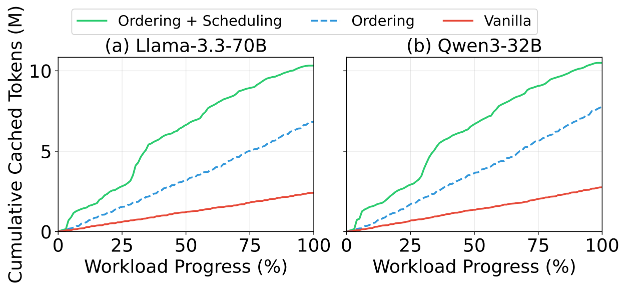


Figure 13. Cumulative cached tokens (radix tree reuse) over workload progress for Llama-3.3-70B and Qwen3-32B. ContextPilot achieves 4× better prefix reuse.

C.2 Accuracy Breakdown

Table 5 provides a detailed breakdown of accuracy contributions from each ContextPilot component.

Table 5. Accuracy breakdown by component. Reordering alone: ≤1% F1 variation; adding annotations: +1.4–4.4% gains over reordered baseline.

Model	Configuration	MultihopRAG	NarrativeQA
Qwen3-32B	Baseline	60.4%	28.4%
	+ Reordering	60.0%	28.2%
	+ Annotation	64.4%	29.6%
	+ Scheduling	**64.4%**	**29.6%**
Qwen3-4B	Baseline	35.2%	16.0%
	+ Reordering	34.5%	15.2%
	+ Annotation	36.6%	17.3%
	+ Scheduling	**36.6%**	**17.3%**

C.3 Per-Request Overhead

Table 6 reports the per-request overhead of ContextPilot components, measured on 2K requests with $k = 15$ on NVIDIA A6000.

Table 6. Per-request overhead breakdown. Total overhead (~0.2 ms) is negligible compared to seconds of prefill latency.

Component	Latency (ms)
Search	0.068
Reordering	0.047
De-duplication	0.144
Total	**~0.2**

D Detailed Algorithm Pseudocode

Algorithm 4 provides the full pseudocode for context index construction via hierarchical clustering, and Algorithm 5 details the tree-based request grouping and scheduling procedure.

Algorithm 4 Context Index Construction via Hierarchical Clustering

```
Input: Batch of $n$ contexts $\mathcal{S} = \{s_1, s_2, \dots, s_n\}$, distance function $d(\cdot, \cdot)$
 1: // Phase 1: Distance computation and clustering
 2: $D \leftarrow$ pairwise distances using $d(s_i, s_j)$ ▷ GPU or CPU
 3: $Z \leftarrow$ LINKAGE($D$)   ▷ hierarchical clustering
 4: // Phase 2: Build tree with deduplication
 5: for $i = 1$ to $n$ do
 6:     Create leaf node $v_i$ with $v_i$.content $\leftarrow s_i$
 7:     if $\exists\, v_j$ with $v_j$.content $= s_i$ then
 8:         Redirect $v_i \rightarrow v_j$; $v_j$.freq += 1   ▷ dedup
 9:     end if
10: end for
11: for each merge $(c_1, c_2, \delta)$ in $Z$ do
12:     $v \leftarrow$ new internal node
13:     $v$.content $\leftarrow c_1$.content $\cap$ $c_2$.content
14:     $v$.children $\leftarrow [c_1, c_2]$;   $c_1$.parent, $c_2$.parent $\leftarrow v$
15: end for
16: Remove empty internal nodes; relink children to grandparents
17: // Phase 3: Top-down prefix-aligned reordering
18: Compute search paths from root to all nodes
19: for each node $v$ in BFS order from root do
20:     if $v$ is not root and $v$.parent.docs $\neq \emptyset$ then
21:         $v$.docs $\leftarrow v$.parent.docs$\oplus$($v$.docs$\setminus v$.parent.docs)
22:     end if
23: end for
24: return reordered contexts from leaf nodes, search paths
```

Algorithm 5 Search-Path-Based Request Grouping and Scheduling

```
 1: Input: $N$ contexts with search paths $\{P_1, \dots, P_N\}$ from ordering
 2: Output: Scheduled execution order
 3: // Phase 1: Group by root prefix   ▷ $O(N)$
 4: $\mathcal{G} \leftarrow$ empty map
 5: for $i = 1$ to $N$ do
 6:     $key \leftarrow P_i[0]$   ▷ first element of search path
 7:     $\mathcal{G}[key]$.APPEND($i$)
 8: end for
 9: // Phase 2: Sort within each group   ▷ $O(N \log N)$
10: for each group $G \in \mathcal{G}$ do
11:     Sort $G$ by $|P_i|$ descending   ▷ longer paths first
12: end for
13: // Phase 3: Order groups and flatten
14: Sort groups by $|G|$ descending   ▷ largest groups first
15: return concatenation of all groups
```