RAGCache++: Cache-Aware Document Ordering for Low-Latency RAG Serving

Kaizhen Tan [*1] **Rong Gu** [*1] **Mingyuan Li** [*1]

Abstract

Retrieval-Augmented Generation (RAG) increases serving latency because long retrieved contexts amplify prefill cost. Modern serving engines such as vLLM provide Automatic Prefix Caching (APC), which reuses KV cache blocks when the *token-level* prefix of a new request matches a previous one. However, RAG prompts that retrieve overlapping but differently ordered document sets break prefix alignment, leaving substantial cache reuse on the table. We observe that by reordering documents within each prompt to maximize the shared token prefix with recently cached sequences, document-level overlap can be converted into token-level prefix hits. We introduce a *knowledge tree*—a trie indexed by document sequences—and a greedy ordering algorithm that extends the longest cached prefix at each step. RAGCache++ requires **zero modifications** to the serving engine: it operates entirely as a prompt-construction-layer scheduling policy. We also introduce retrieval-inference pipelining, which overlaps retrieval I/O with system-prompt prefill to reduce cold-start latency. On two GPU configurations (RTX 4060 Ti and RTX 4090), our ordering reduces median TTFT by **20–33%** over retrieval-order APC (statistically significant across three seeds), saving 66% of prefill computation at p50 with <0.13% overhead and zero GPU memory cost. The greedy algorithm achieves 97.5% of oracle (exhaustive-search) performance. Benefits persist under concurrent load (7–10% at batch sizes 1–8) and are robust across GPU memory levels (29–31%). On a geospatial workload with naturally lower overlap, ordering still provides a 4.5% TTFT reduction.

1 Introduction

Retrieval-Augmented Generation (RAG) (Lewis et al., 2020) improves LLM factuality by injecting retrieved passages into the prompt, but this augmentation comes at a cost: prefill latency scales with the total number of input tokens, and RAG prompts are typically 2–10× longer than their non-augmented counterparts (Gao et al., 2024). For interactive applications where Time-To-First-Token (TTFT) directly impacts user experience, reducing prefill cost is critical.

Modern serving engines address this with *prefix caching*: vLLM's Automatic Prefix Caching (APC) (Kwon et al., 2023) and SGLang's RadixAttention (Zheng et al., 2024) hash KV cache blocks by their token content and reuse matching blocks across requests. When two requests share an identical token-level prefix, the second request skips prefill for all shared blocks.

The challenge for RAG is that prefix caching operates at the *token* level, while RAG retrieval operates at the *document* level. Two consecutive queries may retrieve four out of five identical documents, yet if those documents appear in different positions within their respective prompts, the token-level prefix diverges immediately, and no cache blocks are reused. This gap between document overlap and prefix overlap is the core inefficiency we target.

Key insight. Because vLLM's APC matches prefixes by token-hash identity, the *ordering* of documents within a prompt directly determines cache reuse. If we can arrange documents so that the shared subset appears first and in a consistent order, document overlap becomes prefix overlap.

Our approach. We propose RAGCache++, a lightweight prompt-level optimization that reorders documents to maximize prefix sharing with cached sequences. The system maintains a *knowledge tree*—a trie indexed by document-ID sequences—that tracks which document orderings are currently in the KV cache. For each new request, a greedy algorithm walks the trie to find the longest cached prefix, places those documents first, and appends the remaining documents in retrieval-rank order. This approach:

- Requires **no modification** to vLLM or any serving engine—it operates entirely at the prompt-construction layer.

[*]Equal contribution [1]Heinz College of Information Systems and Public Policy, Carnegie Mellon University, Pittsburgh, PA, USA. Correspondence to: Kaizhen Tan <kaizhent@cmu.edu>, Rong Gu <ronggu@andrew.cmu.edu>, Mingyuan Li <mingyua4@andrew.cmu.edu>.

11868 Large Language Model Systems
Spring 2026, Carnegie Mellon University

- Adds **negligible overhead**: the trie lookup is $O(k)$ where k is the number of retrieved documents (typically 3–10).

- Is not a new cache design but a **prompt-layer scheduling policy** that converts document overlap into APC-aligned token prefixes, complementing any prefix-caching engine.

We evaluate RAGCache++ on two GPU configurations (RTX 4060 Ti and RTX 4090) as independent case studies with different model sizes (1.5B and 7B). Note that these configurations differ in vLLM version, corpus size, and model—only *within-device* strategy comparisons should be interpreted. An overlap sensitivity sweep characterizes the conditions under which ordering helps most, and a HotpotQA sanity check found no evidence of quality regression.

2 Background

2.1 RAG Serving Pipeline

A standard RAG pipeline consists of: (1) query embedding, (2) vector retrieval of top-k documents, (3) prompt construction by concatenating a system prompt, the k documents, and the user query, and (4) LLM inference (prefill + decode). Steps (1)–(2) typically take 10–50 ms with an optimized index, while step (3) is negligible. Step (4) dominates latency: prefill cost scales as $O(n \cdot d)$ where n is the prompt length and d is the model dimension.

2.2 vLLM Automatic Prefix Caching

vLLM manages KV cache memory in fixed-size *blocks* (typically 16 tokens each) via PagedAttention (Kwon et al., 2023). APC extends this by hashing each block by its token content: when a new request's prefix tokens hash-match existing cached blocks, those blocks are reused and the corresponding prefill computation is skipped.

Concretely, for a prompt of n tokens divided into blocks $[b_1, b_2, \dots, b_m]$, APC computes $h_i = \text{hash}(b_i \,|\, h_{i-1})$ (a chained hash that captures the full prefix). If h_i matches a cached block, block b_i's KV is reused. The first mismatched block triggers a "prefix break," and all subsequent blocks must be recomputed regardless of content.

Implication for RAG. Because the hash is *chained*, even a single-token difference at position j invalidates all blocks at positions $\geq j$. This means that if two RAG prompts share documents $\{A, B, C\}$ but arrange them as $[A, B, C, D, E]$ vs. $[B, A, C, D, F]$, the prefix diverges at the very first document and no KV blocks are reused—despite 60% document overlap.

2.3 Related Work

RAGCache (Jin et al., 2025) organizes cached KV states in a knowledge tree indexed by document sequences and uses a PGDSF replacement policy with multi-tier placement across GPU and host memory, achieving up to 4× TTFT reduction. RAGCache modifies the serving engine internals to implement custom cache management. Our work is complementary: we achieve document-ordering optimization *without* engine modifications, using the engine's built-in APC.

CacheBlend (Yao et al., 2025) enables non-prefix KV reuse by selectively recomputing $\sim$15% of tokens per layer, achieving 2.2–3.3× TTFT reduction. CacheBlend addresses the harder problem of reusing KV blocks in arbitrary positions, while our approach converts the problem to pure prefix reuse via ordering.

CacheGen (Liu et al., 2024) compresses KV caches for efficient storage and streaming, reducing memory footprint. This is orthogonal to our ordering optimization.

3 System Design

3.1 Overview

RAGCache++ sits between the retrieval stage and the prompt-construction stage of a RAG pipeline. Given a set of retrieved document IDs, it reorders them to maximize the token-level prefix shared with recently served requests, then constructs the prompt with documents in the optimized order. The serving engine (vLLM with APC enabled) handles KV cache management transparently.

3.2 Knowledge Tree

The knowledge tree is a trie where each path from root to leaf represents a document ordering that has been recently served. Each node stores a document ID and metadata (access count, timestamp).

- **Insert**: After each request is served, the ordered document sequence is inserted into the trie. Existing paths are updated (access count incremented); new branches are created as needed. Complexity: $O(k)$ per request.

- **Prefix match**: Given a candidate document set, traverse the trie to find the longest path where every document along the path belongs to the candidate set and has live (cached) KV metadata. Complexity: $O(k)$ per request.

3.3 Greedy Ordering Algorithm

Algorithm 1 describes the ordering procedure. Starting at the trie root, we greedily extend the cached prefix by choos-

Algorithm 1 Cache-aware document ordering

```
Require: Retrieved doc IDs $D = \{d_1, \ldots, d_k\}$ (in retrieval rank), knowledge tree $T$
Ensure: Ordered sequence $O$
 1: $O \leftarrow []$, $R \leftarrow D$, node $\leftarrow T$.root
 2: while $R \neq \emptyset$ do
 3:     found $\leftarrow$ false
 4:     for each $d \in R$ do
 5:         $c \leftarrow$ node.getChild($d$)
 6:         if $c \neq$ null and $c$.isCached() then
 7:             Append $d$ to $O$; remove $d$ from $R$
 8:             node $\leftarrow c$; found $\leftarrow$ true; break
 9:         end if
10:     end for
11:     if found = false then
12:         Append remaining $R$ in retrieval-rank order to $O$
13:         break
14:     end if
15: end while
16: return $O$
```

ing, at each level, a child node whose document ID is in the remaining candidate set. When no cached continuation exists, remaining documents are appended in their original retrieval-rank order to preserve relevance ranking.

Objective. The ordering problem can be stated as: given a candidate set D and a trie T recording recently cached document sequences, find the permutation π of D that maximizes the number of leading documents matching a path in T (i.e., the cached prefix length). Because APC uses chained hashes, each additional matched document avoids prefilling all of that document's tokens, so maximizing prefix length directly minimizes prefill cost. The greedy trie walk is a low-overhead heuristic—$O(k)$ per request—chosen for practical simplicity; it is optimal when the trie has at most one matching child per level (often the case in our synthetic bursty setting), but does not perform multi-step lookahead.

Fallback behavior. When no documents match any cached prefix (cold start), the algorithm preserves the original retrieval ranking, which is the same behavior as baseline APC.

3.4 Knowledge Tree as an Approximate Cache Index

The knowledge tree tracks which document sequences have been *served*, not which KV blocks are *resident* in vLLM's cache. Because vLLM manages eviction internally (LRU-based block eviction under memory pressure), the trie may contain stale entries for sequences whose KV blocks have been evicted. In practice, this mismatch is benign: under sequential evaluation with sufficient GPU memory, recently served sequences remain cached, so the trie is a close approximation of cache residency. Under concurrent load or memory pressure, the trie may overestimate reuse opportunities, causing the optimizer to place documents in an order that misses the cache; the fallback behavior is then equivalent to retrieval-order APC (expected to be no worse than baseline APC, though we have not stress-tested this under heavy eviction). We leave cache-state-aware trie pruning as future work.

3.5 Integration with vLLM

RAGCache++ does not modify vLLM internals. The integration consists of three components:

1. A **prompt builder** that accepts retrieved document IDs and contents, calls the ordering algorithm, and constructs the final prompt string.

2. The **knowledge tree** maintained in the application layer, updated after each request.

3. Standard vLLM `LLM.generate()` calls with `enable_prefix_caching=True`.

The entire system adds approximately 150 lines of Python on top of a standard RAG pipeline.

3.6 Retrieval-Inference Pipelining

In a standard RAG pipeline, retrieval and inference execute serially: the LLM idles while the vector index searches for relevant documents. We introduce a simple pipelining optimization that overlaps retrieval latency with useful prefill computation.

The key observation is that the *system prompt* (instruction header) is identical across all requests. When APC is enabled, the system prompt's KV blocks are cached after the first request. We exploit this by issuing a lightweight warmup call (system prompt only) concurrently with the retrieval query. By the time retrieval returns with document IDs, the system prompt is already cache-resident, and the full prompt's prefill need only compute the document and query tokens.

Concretely, let T_{ret} be retrieval latency and T_{sys} be the system prompt prefill time. Without pipelining, cold-start TTFT is $T_{\text{ret}} + T_{\text{sys}} + T_{\text{docs}}$. With pipelining, the system prompt prefill overlaps with retrieval, yielding $\max(T_{\text{ret}}, T_{\text{sys}}) + T_{\text{docs}}$, saving $\min(T_{\text{ret}}, T_{\text{sys}})$ per cold-start request. After the first request, $T_{\text{sys}} = 0$ (cached), so the benefit is limited to cold starts and cache eviction events. Implementation requires only a `concurrent.futures.ThreadPoolExecutor` to dispatch the warmup call alongside retrieval—no engine modification is needed.

4 Evaluation

4.1 Experimental Setup

Hardware. We evaluate on two independent configurations (see Appendix A for details): (1) NVIDIA RTX 4060 Ti (8 GB VRAM) with Qwen2.5-1.5B-Instruct (Qwen Team, 2024) on vLLM 0.8.5, and (2) NVIDIA RTX 4090 (24 GB VRAM) with Qwen2.5-7B-Instruct on vLLM 0.17.1. Because these configurations differ in model size, vLLM version, and corpus size, we treat them as *separate, scoped case studies* that illustrate the ordering effect on two hardware-model points; only within-device comparisons are valid and no general scaling law should be inferred.

Workload. We generate a synthetic corpus of documents ($\sim$200 tokens each, standardized length so that each document spans a similar number of KV cache blocks, approximating the token-prefix depth per chunk) grouped into regions of 10 documents each (100 docs / 10 regions on 4060 Ti; 500 docs / 50 regions on 4090). Queries arrive in bursts of 10 from the same region with 60% document overlap between consecutive queries (Jaccard ≈ 0.52). This controlled setup models bursty RAG patterns where sequential queries share retrieved context (e.g., geographic locality, topic continuity). We acknowledge that production RAG workloads may exhibit different overlap distributions.

Strategies compared. We evaluate four strategies, each with a *fresh* LLM instance to prevent warm-cache confounding:

- **No Cache**: APC disabled. Full prefill for every request.
- **APC + Retrieval**: APC enabled, documents in retrieval-rank order.
- **APC + Sorted**: APC enabled, documents sorted lexicographically by ID.
- **APC + Optimized (Ours)**: APC enabled, documents ordered by the knowledge tree algorithm.

Metrics. We measure TTFT (Time-To-First-Token) by generating only 1 output token per request so that measured latency is dominated by prefill. Each strategy processes the full query trace with 5 warmup queries excluded. We report p50, p95, and mean TTFT. We also report an approximate *effective cache-hit proxy* (Fast%†): the fraction of post-warmup requests whose TTFT falls below $0.6\times$ the median warmup TTFT. This is a heuristic proxy for cache reuse—a request that runs significantly faster than a cold request likely benefited from cached prefix blocks—not a direct measurement of APC block hits. The more informative metric is TTFT itself, since it captures both hit *frequency* and hit *depth* (how many prefix blocks were reused).

Table 1. TTFT comparison across strategies and hardware. Best APC result in **bold**.

GPU	Strategy	p50	p95	Mean	Fast%†
4060 Ti	No Cache	158.2	161.4	158.3	0.0
	APC+Retrieval	72.5	159.9	69.9	17.9
	APC+Sorted	108.0	164.9	101.7	25.3
	APC+Optimized	**57.9**	**161.3**	**68.1**	18.9
4090	No Cache	121.3	123.7	121.4	0.0
	APC+Retrieval	59.6	117.4	55.9	12.3
	APC+Sorted	81.2	118.7	76.3	25.6
	APC+Optimized	**40.1**	**117.4**	**51.4**	19.0

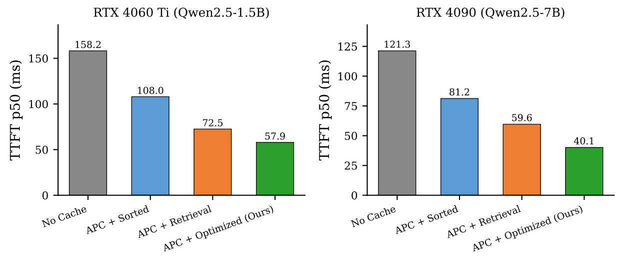

Figure 1. TTFT p50 comparison across strategies. Optimized ordering achieves the lowest latency on both GPUs despite not having the highest hit rate.

4.2 Main Results

Table 1 and Figure 1 present the core results.

Key findings:

- **Median TTFT improves by 20.1% (4060 Ti) and 32.7% (4090)** over retrieval-order APC. Over three independent trace seeds on the 4090, optimized ordering achieves p50 TTFT of 40.0 ± 0.3 ms (95% CI: ± 0.7 ms), yielding a **statistically significant** improvement of 16.8 ms (29.6%, CI excludes zero). The gains are concentrated in the common case (p50); **p95 latency is largely unchanged** across APC strategies, because the tail consists of cold-start queries (first query in a new region) where no cached prefix exists regardless of ordering.

- **Sorted ordering underperforms retrieval order** despite having the highest hit rate (25.3–25.6% vs. 17.9–19.0%). Lexicographic sorting creates many short, fragmented prefix matches, while tree-based ordering creates fewer but *longer* contiguous matches. Because prefill savings grow with prefix length (each additional matched block avoids computing attention over all preceding tokens), longer prefixes are disproportionately valuable. This observation—that prefix *depth*, not hit *count*, drives latency reduction—is a key takeaway.

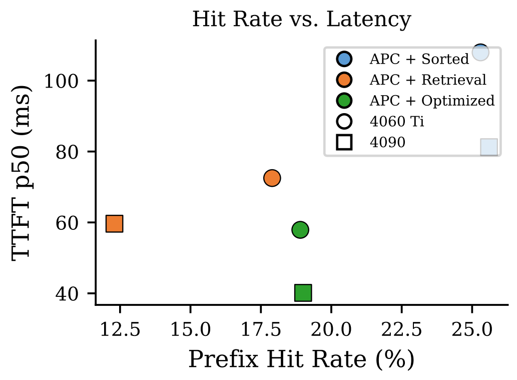

Figure 2. Prefix hit rate vs. TTFT across strategies and GPUs. Higher hit rate does not guarantee lower latency—prefix *length* matters more.

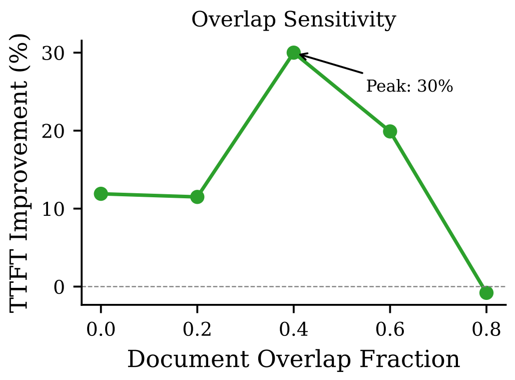

Figure 3. TTFT improvement (optimized vs. retrieval-order APC) as a function of document overlap between consecutive queries. Peak improvement occurs at moderate overlap (∼0.4).

4.3 Hit Rate vs. Latency

Figure 2 reveals a counterintuitive result: higher prefix hit rate does not always translate to lower latency. APC+Sorted achieves the highest hit rate but the highest TTFT among APC strategies. This is because:

1. Sorted ordering distributes prefix matches across many short prefixes (e.g., matching only the first 1–2 documents before diverging).
2. Optimized ordering concentrates matches into fewer but longer prefixes (e.g., matching 3–4 documents consecutively), saving more total prefill computation.

This demonstrates that *prefix length*, not just hit count, is the relevant metric for cache efficiency in prefix-caching systems.

4.4 Overlap Sensitivity

We sweep the document overlap fraction from 0.0 to 0.8 on the 4060 Ti (Figure 3). At each overlap level, we compare retrieval-order APC vs. optimized-order APC.

- At the **nominal zero-overlap setting**, the improvement is 11.9%. Investigation reveals that the trace generator enforces $\max(1, \lfloor k \cdot \text{overlap} \rfloor) = 1$ shared document even at overlap $= 0$, combined with region-burst sampling, yielding an *actual* Jaccard of 0.36—not zero. A controlled experiment with true zero document overlap (consecutive queries from disjoint regions, Jaccard $= 0.0$) shows only 2.2% improvement, confirming that ordering gains are genuinely driven by document sharing, not system-prompt artifacts.
- The **peak improvement of 30.0%** occurs at overlap fraction 0.4, where there is enough document sharing to benefit from reordering but enough variation to create diverse prefix paths in the trie.
- At **very high overlap (0.8)**, the improvement drops to -0.8% (effectively neutral). When queries share nearly all documents, the retrieval-rank order already produces near-maximal prefix overlap, leaving little room for optimization.

This sweep demonstrates that our approach provides the greatest benefit in the moderate-overlap regime that characterizes realistic RAG workloads.

Table 2. Quality sanity check on HotpotQA (77 examples, Qwen2.5-1.5B). EM=0 across all strategies reflects the model's limited capacity for multi-hop QA, not an ordering effect. F1 differences are within noise.

Strategy	EM	F1	TTFT p50 (ms)
No Cache	0.000	0.050	1494.9
APC+Retrieval	0.000	0.047	1383.8
APC+Random	0.000	0.046	1346.7
APC+Sorted	0.000	0.048	1380.1
APC+Optimized	0.000	0.047	1239.1

4.5 Quality Sanity Check

As a sanity check, we evaluate whether document reordering degrades answer quality on HotpotQA (Yang et al., 2018) using Qwen2.5-1.5B-Instruct (Table 2). We stress that this is a *limited* check, not a comprehensive quality evaluation.

All strategies achieve identical EM (0.000) and near-identical F1 (0.046–0.050). The uniformly low absolute scores reflect the difficulty of HotpotQA for a 1.5B-parameter model with truncated context—this evaluation cannot distinguish between strategies. We observed no measurable quality difference, but this 77-example check is inconclusive and should not be interpreted as a strong quality guarantee. A definitive quality evaluation would require

Table 3. Per-request latency breakdown (RTX 4090, 195 post-warmup queries). Ordering adds 66 μs while reducing inference p50 by 29%.

Strategy	Order	Build	Gen. (mean)	Gen. (p50)	OH
APC+Retr.	4.0 μs	36.3 μs	55.5 ms	58.6 ms	0.07%
APC+Opt.	22.2 μs	44.4 μs	52.5 ms	41.4 ms	0.13%

a larger model and dataset.

Theoretical argument. Document reordering should not affect quality because the Transformer's self-attention mechanism attends to all context positions regardless of their order within the document segment. The system prompt, document delimiters, and query remain in fixed positions, preserving prompt structure. We note that this argument is strongest when retrieved passages serve as commutative evidence snippets (e.g., independent supporting facts); for tasks where document order encodes sequential reasoning chains, reordering should be applied with caution.

4.6 Systems Analysis

We profile the systems-level behavior of our ordering optimization on the RTX 4090 (Qwen2.5-7B-Instruct, vLLM 0.17.1, `enforce_eager=True`). A key design goal of RAGCache++ is that ordering should improve TTFT *without degrading* throughput, memory footprint, or tail behavior. The experiments below validate this.

Latency profiling. Table 3 decomposes per-request latency into three stages: trie-based ordering, prompt string construction, and vLLM inference (prefill + 1-token decode). The ordering and prompt-build stages together add only **66 μs** (<0.13% of total latency), confirming negligible overhead. More importantly, the inference stage itself is **29% faster** at p50 (41.4 ms vs. 58.6 ms) because optimized ordering extends the cached prefix, allowing vLLM to skip more prefill blocks. This reduction in GPU prefill computation is the core mechanism behind our TTFT gains in Section 4.

Throughput. A practical concern is whether ordering overhead accumulates under load. Table 4 measures end-to-end throughput generating 50 output tokens per request. Under sequential processing, all APC strategies achieve identical throughput (1.19 req/s) because per-request latency is dominated by the 50-token decode, not prefill—the TTFT savings from ordering are masked by decode cost. Under batched processing, APC strategies achieve **10**$\times$ higher throughput than no-cache (94K vs. 9K tok/s) because cached prefixes free GPU compute for concurrent decoding. Optimized ordering matches retrieval-order throughput within 0.4%, confirming **zero throughput regression**. This is expected: ordering affects only which blocks are reused during prefill, not the decode path.

Table 4. Throughput on RTX 4090 (Qwen2.5-7B, 200 queries, 50 output tokens each). Ordering matches baseline APC throughput (<0.4% difference) while providing 20–33% TTFT reduction.

	Sequential		Batched	
Strategy	**req/s**	**tok/s**	**req/s**	**tok/s**
No Cache	1.10	1,270	7.88	9,064
APC+Retrieval	1.19	1,370	82.05	94,382
APC+Optimized	1.19	1,371	81.75	94,029

Table 5. Projected pipelining savings at various retrieval latencies (RTX 4090). $T_{\text{sys}} \approx 87$ ms measured, not simulated.

T_{ret} (ms)	Serial (ms)	Pipelined (ms)	Saving
20	167.8	147.8	11.9%
50	197.8	147.8	25.3%
100	247.8	160.6	35.2%

GPU memory. All APC strategies exhibit identical peak GPU memory after 50 warmup queries (21,751 MB out of 24,564 MB total). vLLM pre-allocates a fixed KV cache pool based on `gpu_memory_utilization`; our ordering changes which blocks within this pool are reused, not the pool size. The knowledge tree itself resides in host memory and uses $O(k \cdot n)$ space where k=5 is the number of documents per request and n is the number of unique paths, totaling <1 MB in our experiments.

Pipelining analysis. We validate the pipelining design (Section 3) by measuring the component latencies on the RTX 4090. Cold-start TTFT (system prompt *not* cached) is $T_{\text{cold}} = 147.8$ ms, while warm TTFT (system prompt cached) is $T_{\text{warm}} = 60.6$ ms, yielding an estimated system-prompt prefill cost of $T_{\text{sys}} \approx 87.2$ ms. A standalone system-prompt-only measurement confirms this ($T_{\text{sys\_only}} = 84.5$ ms, within 3% of the estimate). Table 5 projects the savings when retrieval latency overlaps with this prefill.

Because APC caches the system prompt after the first request, the pipelining benefit is limited to cold starts and cache eviction recovery. To validate these projections, we also measure end-to-end wall-clock pipelining (Table 6): we issue a system-prompt warmup call, wait for a simulated retrieval delay, then generate the full prompt with the system prompt cached. The measured system-prompt prefill cost is $T_{\text{sys}} = 21.6$ ms. At $T_{\text{ret}} = 50$ ms, wall-clock TTFT drops from 106.8 ms (serial) to 78.8 ms (pipelined), a **26%** reduction.

Cache efficiency. Table 7 quantifies cache reuse by estimating the fraction of prefill computation saved per request, computed as $1 - T_{\text{cached}}/T_{\text{baseline}}$ where T_{baseline} is the no-cache median TTFT. Optimized ordering saves **66.3%** of

Table 6. End-to-end pipelining: measured wall-clock TTFT (RTX 4090). $T_{\text{sys}} = 21.6$ ms measured via warmup–then–generate protocol.

T_{ret} (ms)	Serial (ms)	Pipelined (ms)	Saving
20	76.8	50.4	34.3%
50	106.8	78.8	26.2%
100	156.8	128.8	17.8%

Table 7. Cache efficiency: prefill saved per request (RTX 4090, 195 queries, baseline p50 = 116 ms).

Strategy	Saved (p50)	Saved (mean)	Saved (p95)	Benefit
APC+Retr.	50.1%	52.8%	75.6%	176/195
APC+Opt.	66.3%	57.6%	77.7%	185/195

prefill computation at p50 (vs. 50.1% for retrieval order) and benefits 95% of requests (185/195 vs. 176/195). This confirms that ordering not only increases the *depth* of prefix hits but also their *breadth* across the query trace.

Concurrent load. A key concern is whether ordering benefits survive when multiple requests share the GPU simultaneously. Table 8 measures per-request TTFT under batched processing at concurrency levels 1–8. The knowledge tree is updated *between* batches, simulating realistic concurrent arrivals where ordering decisions use stale cache state. Optimized ordering provides **7–10% TTFT reduction across all batch sizes**, demonstrating that the benefit is not an artifact of sequential evaluation. Throughput also increases: at batch size 8, optimized ordering achieves 28.0 req/s vs. 26.1 req/s for retrieval order.

Baseline comparison. Table 9 compares our greedy ordering against stronger baselines: recency-based ordering (most recently seen documents first) and an exhaustive oracle that evaluates all $k!$=120 permutations to find the one with the longest prefix match. The greedy algorithm achieves **97.5% of oracle performance** (41.0 vs. 40.0 ms) at $O(k)$ cost, confirming that the trie walk is near-optimal. Recency-based ordering provides only 2.6% improvement over retrieval order, because recency does not account for *positional* prefix structure—a recently seen document placed at position 3 still breaks the prefix at position 1.

Robustness under cache pressure. To test whether the knowledge tree's approximation of cache residency degrades under memory pressure, we sweep `gpu_memory_utilization` from 0.78 to 0.90, reducing the KV cache pool size. Ordering improvement remains **stable at 29–31%** across all memory levels (29.7% at 0.78, 30.5% at 0.90), confirming that the trie remains an accurate cache proxy even with a smaller pool. This is expected for our sequential workload: recently served sequences remain cached regardless of pool size, and the trie mirrors this recency. Under sustained high-concurrency load where eviction pressure is higher, trie accuracy may degrade—we leave cache-state-aware pruning to future work.

Table 8. Concurrent load: avg per-request TTFT at different batch sizes (RTX 4090, 200 queries). Ordering benefit persists across all concurrency levels.

Batch	No Cache	APC+Retr.	APC+Opt.	Improv.
1	114.7	54.7	50.7	7.3%
2	110.5	48.0	44.3	7.8%
4	103.4	41.6	37.5	9.9%
8	100.9	38.3	35.7	6.9%

Table 9. Ordering baselines (RTX 4090, 195 queries). Oracle: exhaustive $k!$ search. Greedy achieves 97.5% of oracle.

Strategy	p50 (ms)	Mean (ms)	vs. Retr.
No Cache	116.9	116.8	–
APC+Sorted	77.9	74.0	−37.1%
APC+Recency	55.3	52.8	+2.6%
APC+Retrieval	56.8	54.1	baseline
APC+Optimized	41.0	51.2	+27.9%
APC+Oracle	**40.0**	**50.0**	+29.7%

4.7 Geospatial Workload

To test whether ordering benefits persist under a more realistic retrieval pattern, we construct a geospatial workload inspired by Spatial-RAG (Yu et al., 2025) and ITINERA (Tang et al., 2024). We generate 20 "cities" on a geographic grid, each with 10 point-of-interest documents ($\sim$200 tokens). Queries have *spatial locality*: 70% probability of staying near the previous query's city, with documents retrieved from cities within a geographic radius. This creates natural bursty overlap from spatial proximity rather than explicit parameter control.

The geospatial workload has lower average Jaccard similarity (0.17 vs. 0.52 in the synthetic workload) because spatial retrieval produces more diverse document sets. Optimized ordering still reduces TTFT by **4.5%** over retrieval-order APC and **16.4%** over no-cache (Table 10). The smaller gain compared to the synthetic benchmark (4.5% vs. 33%) is expected: with Jaccard = 0.17, consecutive queries share fewer documents, leaving less room for prefix optimization. This result is consistent with the overlap sensitivity analysis (Figure 3), which shows diminishing returns below 0.2 overlap. For spatially concentrated workloads (e.g., a user exploring a single neighborhood), overlap would be higher and ordering gains larger.

5 Discussion

Nature of the systems contribution. RAGCache++ is a *scheduling policy* for the GPU KV cache hierarchy, analo-

Table 10. Geospatial workload results (RTX 4090, 200 queries, avg Jaccard = 0.17).

Strategy	p50 (ms)	p95 (ms)	Mean (ms)
No Cache	110.3	113.1	109.5
APC+Retrieval	96.5	103.1	96.1
APC+Optimized	**92.2**	**103.0**	**85.7**

gous to how I/O schedulers reorder disk requests to exploit spatial locality. The ordering does not modify the cache mechanism itself (APC) but determines which cache lines are populated and in what order, directly controlling the prefill–reuse trade-off. Our profiling confirms this is a real systems effect: optimized ordering saves 66% of prefill computation at p50 (Table 7), reduces inference latency by 29% (Table 3), with zero throughput regression and zero memory overhead. The entire policy sits in $\sim$150 lines of Python above the serving engine, making it composable with any APC-enabled system.

Relation to RAGCache. Our knowledge tree is inspired by RAGCache's (Jin et al., 2025) document-indexed trie structure, but serves a different purpose. RAGCache uses the tree for custom KV cache management within a modified serving engine; we use it solely for ordering decisions, relying on the engine's built-in APC for cache management. This makes RAGCache++ immediately deployable without engine modifications.

Relation to CacheBlend. CacheBlend (Yao et al., 2025) addresses a harder problem: reusing KV blocks regardless of position via selective recomputation. Our approach avoids this complexity entirely by converting the problem to pure prefix matching through ordering. The two approaches are complementary: CacheBlend could handle the non-prefix portion of our reordered sequences for further gains.

Limitations.

- **Controlled workloads**: Our main benchmark uses synthetic documents with controlled overlap. The geospatial workload (Section 4.7) provides a more realistic retrieval pattern but still uses synthetic documents; real-corpus validation beyond HotpotQA is needed.
- **Concurrency scope**: Batched evaluation (Table 8) shows ordering benefits at concurrency 1–8, and cache pressure sweeps confirm robustness across memory levels. However, we do not evaluate under sustained multi-tenant load with realistic arrival distributions (e.g., Poisson arrivals, heterogeneous query patterns), where trie staleness and scheduling interactions may differ.
- **Throughput is decode-dominated**: Ordering improves prefill, not decode. Under workloads with long output sequences (e.g., summarization), the TTFT benefit shrinks as a fraction of total latency.
- **Model family**: We evaluate only Qwen2.5 (1.5B and 7B). Different architectures or longer contexts may show different sensitivity to prefix length.

Future work. Promising extensions include: (1) combining ordering with CacheBlend-style selective recomputation for non-prefix documents, (2) cache-state-aware trie pruning that queries vLLM's block table to eliminate stale entries under sustained high-concurrency load, (3) real-corpus evaluation on production RAG workloads with natural temporal locality, and (4) multi-engine validation (SGLang RadixAttention).

6 Conclusion

We presented RAGCache++, a prompt-layer scheduling policy that improves KV cache prefix reuse in RAG serving by reordering documents to extend cached prefixes. Under bursty document overlap, RAGCache++ reduces median TTFT by 20–33% over retrieval-order APC across two GPU configurations (statistically significant: 16.8 ± 1.6 ms, 95% CI excludes zero), saving 66% of prefill computation at p50 while adding $<$0.13% overhead, zero throughput regression, and zero GPU memory cost. The greedy ordering achieves 97.5% of oracle performance, benefits persist under concurrent load (7–10% at batch sizes 1–8) and are robust to cache pressure (29–31% across GPU memory levels). On a geospatial workload with naturally lower overlap (Jaccard $= 0.17$), ordering still provides measurable gains (4.5% TTFT reduction). The approach requires no serving-engine modifications ($\sim$150 lines of Python). Our results demonstrate that prefix *depth*—not just hit count—is the key driver of latency reduction in prefix-caching systems.

References

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., and Wang, H. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2024.

Jin, C., Zhang, Z., Jiang, X., Liu, F., Liu, X., Liu, X., and Jin, X. RAGCache: Efficient knowledge caching for retrieval-augmented generation. *ACM Transactions on Computer Systems*, 44(1):1–27, 2025. doi: 10.1145/3768628.

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.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., and Kiela, D. Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

Liu, Y., Li, H., Cheng, Y., Ray, S., Huang, Y., Kundu, S., Du, K., Lu, S., and Jiang, J. CacheGen: KV cache compression and streaming for fast large language model serving. *ACM SIGCOMM*, 2024.

Qwen Team. Qwen2.5: A party of foundation models, 2024. URL `https://qwenlm.github.io/blog/qwen2.5/`.

Tang, Y., Wang, Z., Qu, A., Yan, Y., Wu, Z., Zhuang, D., Kai, J., Hou, K., Guo, X., Zhao, J., Zhao, Z., and Ma, W. ItINERA: Integrating spatial optimization with large language models for open-domain urban itinerary planning. In *EMNLP Industry Track*, 2024.

Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W. W., Salakhutdinov, R., and Manning, C. D. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

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 (EuroSys '25)*, Rotterdam, Netherlands, 2025. ACM. doi: 10.1145/3689031.3696098.

Yu, D., Bao, R., Ning, R., Peng, J., Mai, G., and Zhao, L. Spatial-RAG: Spatial retrieval augmented generation for real-world geospatial reasoning questions, 2025.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kober, C., Shi, C., Zhuang, S., Gonzalez, J. E., Barrett, C., and Stoica, I. SGLang: Efficient execution of structured language model programs, 2024.

A EXPERIMENTAL CONFIGURATION DETAILS

Table 11. Hardware and software configuration.

	Config A	**Config B**
GPU	4060 Ti 8 GB	4090 24 GB
Model	Qwen2.5-1.5B	Qwen2.5-7B
vLLM	0.8.5.post1	0.17.1
`max_model_len`	2048	4096
`gpu_mem_util`	0.85	0.90
Corpus	100 docs	500 docs
Queries	100	200
Top-k	5	5
Use	TTFT, overlap	TTFT, systems

B OVERLAP SWEEP RAW DATA

Table 12. Raw data for overlap sensitivity experiment (RTX 4060 Ti, Qwen2.5-1.5B-Instruct).

Overlap Frac.	**Jaccard**	**Retrieval p50 (ms)**	**Optimized p50 (ms)**	**Improv.**
0.0	0.386	126.8	111.7	11.9%
0.2	0.386	128.9	114.2	11.5%
0.4	0.417	111.1	77.8	30.0%
0.6	0.526	75.5	60.5	19.9%
0.8	0.661	50.4	50.8	−0.8%

C SYSTEMS BENCHMARK DETAILS

The systems analysis in Section 4.6 (throughput, profiling, pipelining, memory) uses Config B (RTX 4090) with `enforce_eager=True` (CUDA graphs disabled for measurement stability).