RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation

CHAO JIN, School of Computer Science, Peking University, Beijing, China
ZILI ZHANG, School of Computer Science, Peking University, Beijing, China
XUANLIN JIANG, School of Electronics Engineering and Computer Science, Peking University, Beijing, China
FANGYUE LIU, School of Computer Science, Peking University, Beijing, China
SHUFAN LIU, ByteDance Seed, Beijing, China
XUANZHE LIU, School of Computer Science, Peking University, Beijing, China
XIN JIN, School of Computer Science, Peking University, Beijing, China

Retrieval-Augmented Generation (RAG) has demonstrated substantial advancements in various natural language processing tasks by integrating the strengths of large language models (LLMs) and external knowledge databases. However, the retrieval step introduces long sequence generation and extra data dependency, resulting in long end-to-end latency.

Our analysis benchmarks current RAG systems and reveals that, while the retrieval step poses performance challenges, it also offers optimization opportunities through its retrieval pattern and streaming search behavior. We propose RAGCache, a latency-optimized serving system tailored for RAG. RAGCache leverages the retrieval pattern to organize and cache the intermediate states of retrieved knowledge in a *knowledge tree* across the GPU and host memory hierarchy, reducing LLM generation time. RAGCache employs *dynamic speculative pipelining* to exploit the streaming search behavior, overlapping retrieval with LLM generation to minimize end-to-end latency. We implement RAGCache based on vLLM and Faiss, and evaluate it on both open-source and production datasets. Experimental results demonstrate that RAGCache reduces the time to first token (TTFT) by up to 4× and improves the throughput by up to 2.1× compared to vLLM integrated with Faiss.

CCS Concepts: • **Software and its engineering** → **Scheduling**; **Cloud computing**; • **Computer systems organization** → **Cloud computing**.

Additional Key Words and Phrases: Retrieval-augmented generation, LLM inference, caching

This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4500700, the Scientific Research Innovation Capability Support Project for Young Faculty under Grant ZYGXQNJSKYCXNLZCXM-I1, the Fundamental Research Funds for the Central Universities, Peking University, and the National Natural Science Foundation of China under Grant 62172008 and 62325201. Chao Jin, Zili Zhang, Fangyue Liu, Xuanzhe Liu, and Xin Jin are also affiliated with Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.
Authors' Contact Information: Chao Jin, School of Computer Science, Peking University, Beijing, Beijing, China; e-mail: chaojin@pku.edu.cn; Zili Zhang, School of Computer Science, Peking University, Beijing, Beijing, China; e-mail: zzlcs@pku.edu.cn; Xuanlin Jiang, School of Electronics Engineering and Computer Science, Peking University, Beijing, Beijing, China; e-mail: xuanlinjiang@outlook.com; Fangyue Liu, School of Computer Science, Peking University, Beijing, Beijing, China; e-mail: lfy-wanqiu@stu.pku.edu.cn; Shufan Liu, ByteDance Seed, Beijing, Beijing, China; e-mail: liushufan.amos@bytedance.com; Xuanzhe Liu, School of Computer Science, Peking University, Beijing, Beijing, China; e-mail: liuxuanzhe@pku.edu.cn; Xin Jin, School of Computer Science, Peking University, Beijing, Beijing, China; e-mail: xinjinpku@pku.edu.cn.

ACM 1557-7333/2025/9-ART
https://doi.org/10.1145/3768628

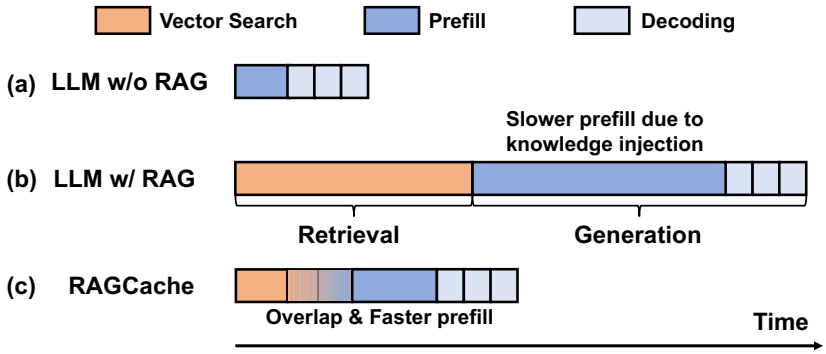


Fig. 1. RAG latency breakdown.

1 Introduction

Retrieval-augmented generation (RAG) [1, 2] is a hybrid approach in natural language processing (NLP) that combines retrieval-based and generation-based models to improve the quality and relevance of generated text. Traditional large language models (LLMs) like GPT-3 [3] generate text based solely on the training data, which struggle to incorporate the latest information or complex knowledge patterns. With informative external knowledge, RAG has achieved comparable or even better performance than LLMs fine-tuned for specific downstream tasks [4].

For an RAG request, RAG includes two main steps: retrieval and generation. Initially, relevant documents are retrieved from a knowledge database, where the documents are typically encoded as feature vectors using embedding models and retrieved through vector similarity search. Then, RAG injects the retrieved documents (i.e., external knowledge) into the original request and feeds the augmented request to the LLM for generation. With the help of the retrieved documents, RAG expands LLMs' knowledge base and contextual understanding, thereby improving the generation quality [4].

However, the two-step process of RAG poses significant challenges in terms of end-to-end latency as shown in Figure 1. First, with knowledge injection, RAG introduces long sequence generation, resulting in higher computation costs and extended generation time. For instance, an initial request of 100 tokens may be augmented with retrieved documents totaling 1000 tokens, leading to over 10× the computation latency compared to the original request. Second, RAG serving suffers from the data dependency between the retrieval and generation steps, where the generation must wait for the retrieval to complete. This results in idle GPU resources during retrieval and further exacerbates end-to-end latency.

Recent work [5, 6], focusing on system optimizations of LLM generation, has made progress in mitigating the long sequence issue by sharing intermediate states during inference. vLLM [6] manages the intermediate states in non-contiguous memory blocks to allow fine-grained state sharing across multiple generation iterations for a single request. SGLang [5] identifies the reusable intermediate states across different requests for LLM applications like multi-turn conversations and tree-of-thought [7]. However, these efforts only optimize for LLM inference without considering the characteristics of RAG. They cache the intermediate states in GPU memory, which has limited capacity considering the long sequences in augmented requests, leading to suboptimal performance.

To identify optimization opportunities, we conduct a system characterization of RAG, measuring the performance of current RAG systems under various datasets and retrieval settings with representative LLMs (§3). Our analysis reveals two key *opportunities* to optimize end-to-end latency for RAG systems. First, identical documents often recur across multiple RAG requests. Additionally, a small fraction of documents accounts for most retrievals, so we are able to cache intermediate states for frequently accessed documents to reduce redundant computation. Second, the retrieval step follows a streaming search behavior, where the vector search algorithms continuously update the

top-k results during the search process. We can leverage this observation to break the data dependency between retrieval and generation and minimize the end-to-end latency.

Based on these insights, we propose RAGCache, a latency-optimized serving system tailored for RAG. RAGCache is the first system that exploits the data characteristics and system behavior of the retrieval step to minimize end-to-end latency. For long sequence generation, RAGCache organizes the intermediate states of retrieved documents into a *knowledge tree*, efficiently caching them within the GPU and host memory hierarchy. Frequently accessed documents are stored in fast GPU memory, while less accessed ones are placed in slower host memory. For the data dependency, RAGCache employs a *dynamic speculative pipelining* strategy, using temporary search results for speculative LLM generation to overlap retrieval with LLM inference.

There are two main challenges in realizing RAGCache. First, RAG systems are sensitive to the order of retrieved documents. For example, consider two documents D_1 and D_2, and two requests Q_1 and Q_2. Let the relevant documents for Q_1 be $[D_1, D_2]$ and for Q_2 be $[D_2, D_1]$, where $[D_1, D_2]$ means D_1 is more relevant than D_2. The intermediate states (i.e., key-value tensors) of $[D_1, D_2]$ differ from those of $[D_2, D_1]$ because, in LLMs, the key-value tensor for a new token depends on preceding tokens in the attention mechanism [8]. Unfortunately, we cannot simply swap the order of D_1 and D_2, as recent studies have shown that the generation quality of LLMs is affected by document order [9, 10]. We use a knowledge tree to organize the intermediate states of retrieved documents in the GPU and host memory hierarchy and design a prefix-aware Greedy-Dual-Size-Frequency (PGDSF) replacement policy that comprehensively considers the document order, size, frequency, and recency to minimize the miss rate. We also propose a cache-aware request scheduling approach to further improve the hit rate.

Second, breaking the data dependency between retrieval and generation requires algorithm-system co-design since different vector search algorithms have different system characteristics. This dependency is challenging to eliminate because incorrect retrieval results can degrade the quality of generation. To address this, we propose dynamic speculative pipelining, which divides the vector search process into multiple stages and overlaps the computation of both steps while maintaining generation quality. For the system implementation, we focus on two representative vector search algorithms, IVF [11] and HNSW [12], and design a pipelined vector search scheme that adapts to these algorithms.

We implement an RAGCache prototype and evaluate it on both open-source and internal datasets. Experimental results show that RAGCache outperforms the state-of-the-art solution, vLLM [6] integrated with Faiss [13], by up to 4× on time to first token (TTFT) and improves the throughput by up to 2.1×. Compared to SGLang [5], which reuse the intermediate states in GPU memory, RAGCache reduces the TTFT by up to 3.5× and improves the throughput by up to 1.8×.

In summary, we make the following contributions.

- We conduct a detailed system characterization of RAG, which reveals the performance limitations and optimization opportunities.
- We propose RAGCache, to the best of our knowledge, the first serving system that exploits the characteristics of RAG to minimize the end-to-end latency.
- We design a multilevel dynamic cache with a prefix-aware GDSF replacement policy to minimize LLM generation time and a dynamic speculative pipelining approach to minimize the end-to-end latency.
- We implement a RAGCache prototype. The evaluation shows that RAGCache outperforms state-of-the-art solutions by up to 4× on TTFT and 2.1× on throughput.

2 Background

Retrieval-Augmented Generation (RAG) marks a notable advancement in natural language processing (NLP) and machine learning by integrating large language models (LLMs) with the extensive information available in external knowledge databases. RAG enhances generative models by dynamically retrieving relevant information from a

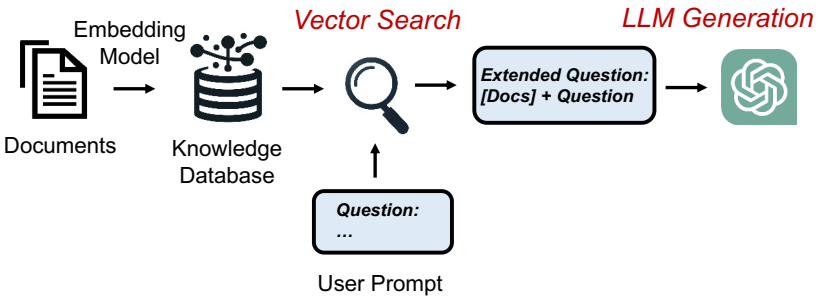


Fig. 2. RAG workflow.

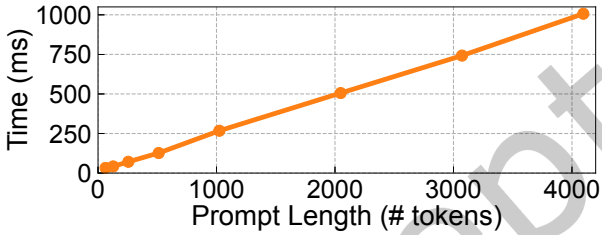


Fig. 3. Generation time with different input lengths.

corpus during the generation process, leading to more accurate, relevant, and contextually enriched responses. This hybrid approach leverages the deep contextual understanding of LLMs alongside the precision of knowledge retrieval. Recent studies [1, 2, 14–17] have demonstrated that RAG significantly improves the generation quality across various benchmarks when compared to purely generative models. The RAG framework has since been widely applied to diverse tasks such as question answering [18, 19], content creation [20], and code generation [21, 22], showcasing its versatility and promise.

As shown in Figure 2, RAG follows a two-step workflow: *retrieval* and *generation*, combining offline preparation with real-time processing for enhanced performance. In the offline phase, RAG converts external knowledge sources, like documents, into high-dimensional vectors using advanced embedding models and indexes them in a vector database for efficient retrieval. Upon receiving a user request, RAG first accesses this vector database to conduct a vector similarity search, retrieving semantically relevant documents. It then combines the retrieved documents with the user request to create an augmented request, which is processed by an LLM to generate a more informed and contextually rich response.

In an RAG workflow, retrieval is typically handled by CPUs, while LLM generation runs on GPUs. From a system perspective, the end-to-end performance of RAG depends on both steps. The retrieval time is influenced by the vector database's scale and the users' accuracy requirements—searching more vectors boosts accuracy but increases time. The generation time is determined by the model size and sequence length.

3 RAG System Characterization

In this section, we present a comprehensive system characterization. We first analyze RAG's end-to-end latency. Then, we explore retrieval patterns for caching optimizations and discuss the streaming search behavior.

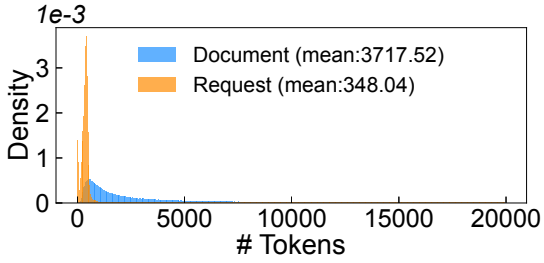


Fig. 4. The distribution of the number of tokens.

3.1 RAG Latency Breakdown

Retrieval time. The retrieval step typically uses Approximate Nearest Neighbor (ANN) [11] search to find the vectors in the database most similar to the input. Its execution time varies based on the database size and accuracy requirements, typically ranging from tens of milliseconds to a few seconds [23, 24]. This is because ANN search involves a fundamental tradeoff between accuracy and latency. For example, an ANN algorithm might achieve 90% recall, meaning 90% of the returned results are among the true top-k nearest vectors, by exploring only a small fraction of the index. To approach exact results, however, a much more exhaustive search is necessary. Therefore, when applications demand high-fidelity retrieval, the search latency can increase substantially, potentially becoming a bottleneck that takes as long as, or even longer than, the generation step.

Genration time. LLM inference can be divided into two distinct phases: prefill and decoding. The prefill phase involves computing the key-value tensors of the input tokens, while the decoding phase generates the output token in an auto-regressive manner based on the previously generated key-value tensors. The prefill phase is particularly time-consuming, as it requires to compute the entire input sequence's key-value tensors.

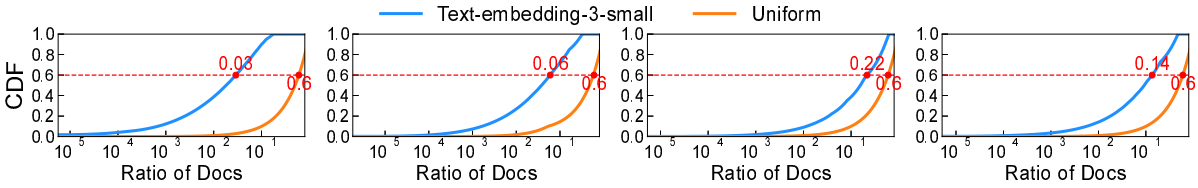
We evaluate the generation time with fixed output length and different input lengths on Llama2-7B, the smallest model in the Llama2 series [25]. Larger models will have longer generation time. The backend system is vLLM [6] equipped with one NVIDIA A10G GPU. Figure 3 shows that the generation time, mainly dominated by the prefill phase, increases rapidly with sequence length and reaches one second when the sequence length is larger than 4000 tokens.

The sequence length in the LLM generation step is the token number of the original request plus the retrieved document. We generate a document dataset based on the Wikipedia corpus [26] with ~0.3 million documents from most popular Wikipedia pages. Figure 4 demonstrates the distribution of the document length and the request length. The document length is significantly longer than the request length of the MMLU dataset [27]. With an average document length of 3718 tokens, the corresponding generation time is markedly higher than that of the original request.

3.2 Opportunity 1: Knowledge Caching

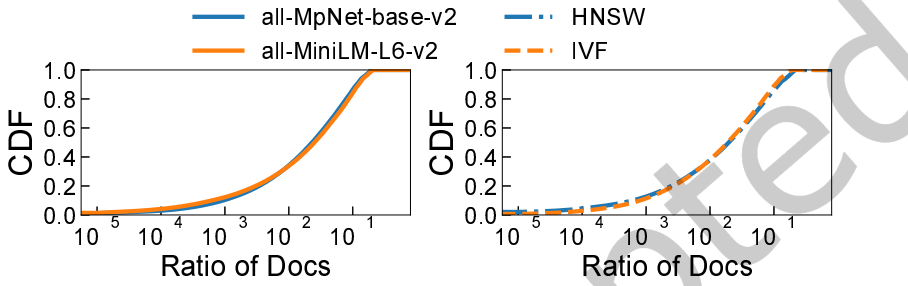
The generation step's performance bottleneck primarily arises from processing the long sequence's key-value tensors in attention blocks. A simple yet effective optimization for RAG involves caching these key-value tensors of previously retrieved documents. For example, let requests, Q_1 and Q_2, both refer to the same document, D_1. If Q_1 arrives first, the key-value tensors of D_1 are computed, and we can cache the key-value tensors. When Q_2 arrives, we can reuse the cached key-value tensors to reduce the prefill latency of Q_2. The average prefill latency with caching is calculated as follows:

$$\textit{Prefill Latency} = \textit{Miss Rate} \times \textit{Full Prefill Latency} + (1 - \textit{Miss Rate}) \times \textit{Cache Hit Latency} \quad (1)$$



(a) MMLU. (b) Google Natural Questions. (c) HotpotQA. (c) TrivialQA.

Fig. 5. Retrieval pattern on different datasets.



(a) Different embedding models. (b) Different ANN indexes.

Fig. 6. Retrieval pattern under different settings.

Miss rate. Cache performance is primarily driven by the miss rate, which is directly affected by the retrieval pattern. For example, a 100% miss rate occurs when each request retrieves a unique document, making caching intermediate states of retrieved documents meaningless. We analyze the retrieval patterns in four representative question-answering datasets for RAG: MMLU [27], Google Natural Questions [28], HotpotQA [29], and TriviaQA [30]. Using the `text-embedding-3-small` model [31] from OpenAI [32], we convert Wikipedia documents to vectors for retrieval, with top-1 document retrieval based on FlatL2 ANN index, i.e., exact search on the entire dataset with Euclidean distance. Figure 5 shows the CDF of accessed documents, revealing a skewed retrieval pattern where a small fraction of documents accounts for most retrieval requests. In MMLU, for example, 60% of requests refer to just 3% of documents, which is 20× less than the uniform distribution. This observation suggests a low miss rate to cache the frequently accessed documents.

Further analysis on additional embedding models and ANN indexes (vector search algorithms) is shown in Figure 6. All results exhibit a similar retrieval pattern regardless of which embedding model or ANN index is used. The results are consistent with FlatL2 index, which indicates the potential for caching optimization under different settings.

We also validate our findings using a production RAG dataset from Company-X, a leading LLM service provider, which we refer to as Dataset-X. Dataset-X consists of hundreds of documents in a knowledge base designed to assist internal employees in answering questions about service details. Over the course of several days, we collected hundreds of user requests and analyzed the retrieval patterns. Like the public datasets, Dataset-X exhibits a skewed retrieval pattern, with 2% of documents account for 60% of requests.

Full prefill computation. To quantify full computation, we compare the LLM prefill latency with and without caching partial intermediate states (i.e., key-value cache of prefixes). We set the original request length to 32 tokens and vary the prefix length from 128 to 4096 tokens. Figure 7 illustrates that caching significantly reduce the prefill latency. When cached, only the request tokens' key-value tensors are computed, while full prefill computation

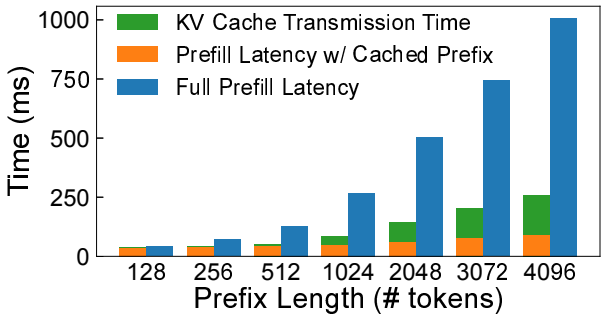


Fig. 7. Prefill latency characterization.

requires to calculate the key-value tensors for the entire sequence. The full prefill latency is up to 11.5× longer than that with cached prefix, highlighting the substantial performance improvement achieved by caching intermediate states of accessed documents.

Cache hit. The cache hit comprises two parts: prefill computation of request tokens and loading the key-value cache of retrieved documents. The former is negligible compared to miss penalty. As for the latter one, the limited GPU memory contrasts sharply with the large size of the key-value cache from retrieved documents. This discrepancy necessitates leveraging the host memory to extend the caching system, accommodating a greater volume of documents. However, this introduces a potential overhead: the transmission of key-value cache between the GPU and host memory. Figure 7 shows the key-value cache transmission time for various prefix lengths. The cache hit latency is the sum of the prefill time with cached prefix and the transmission time. Even with the transmission overhead, the cache hit latency is up to 3.9× lower than the full prefill latency, demonstrating the advantages of caching intermediate states of retrieved documents.

3.3 Opportunity 2: Retrieval-Generation Overlapping

For the data dependency between the retrieval and generation steps, the optimization opportunity lies in the streaming behavior of vector search. Specifically, vector search algorithms like IVF [11] and HNSW [12] typically maintain a queue of top-k candidate documents, which are ranked by their similarity to the request. During the retrieval process, the top-k documents in the queue are continuously updated i.e., some documents with greater similarity are inserted into the queue. Interestingly, the final top-k documents emerge early in the retrieval step [23, 33]. We compare the temporary search results with the final search results using MMLU as the request dataset and Wikipedia as the document base. A request is actually finished when it reaches the final top-k documents. Figure 8 illustrates that 95% of requests are actually finished after completing 13% of the search process. Since the vector search is processed on CPUs, this observation suggests that we can start LLM generation with the temporary search results to utilize the idle GPU resources and overlap the retrieval and generation steps.

4 RAGCache Overview

We present RAGCache, an RAG serving system designed to optimize the end-to-end latency. RAGCache caches the key-value tensors of retrieved documents across multiple requests to minimize redundant computation, and dynamically overlaps retrieval and LLM generation to minimize the latency.

Architecture overview. We provide a brief overview of RAGCache in Figure 9. When a request arrives, the RAG controller first retrieves relevant documents from the external knowledge database and forwards them to the cache retriever to find matching key-value tensors. If not found, RAGCache directs the LLM inference engine to produce

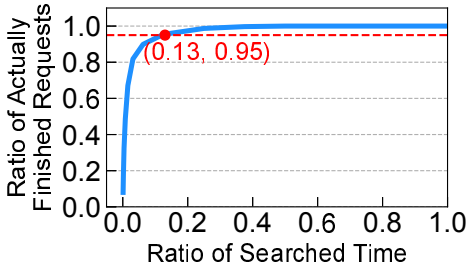


Fig. 8. Search pattern in vector search.

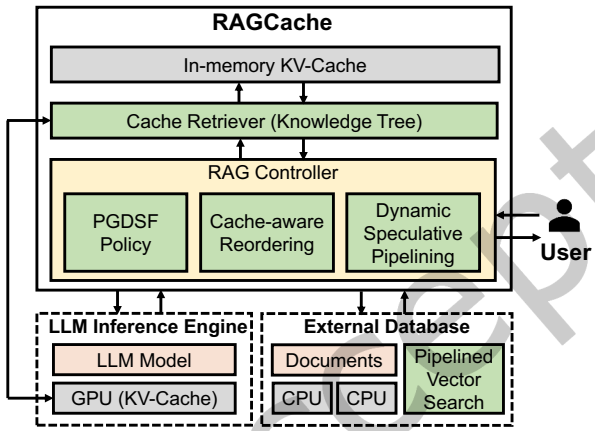


Fig. 9. RAGCache overview.

new tokens. Otherwise, the request and tensors are sent to the inference engine, which uses a prefix caching kernel for token generation. After generating the first token, the key-value tensors are relayed back to the RAG controller, which caches the tensors from the accessed documents and refreshes the cache's status. Finally, the generated answer is delivered to the user as the response.

Cache retriever. The cache retriever efficiently locates the key-value tensors for documents stored in the in-memory cache, utilizing a knowledge tree to organize these tensors. This tree, structured as a prefix tree based on document IDs, aligns with the LLM's position sensitivity to the document order. Each path within this tree represents one specific sequence of documents referenced by a request, with each node holding the key-value tensor of a referred document. Different paths may share the same nodes, which indicates the shared documents across different requests. This structure enables the retriever to swiftly access the key-value tensors of documents in their specified order.

RAG controller. The RAG controller orchestrates the interactions with some system optimizations tailored for RAG. Prefix-aware Greedy Dual-Size Frequency (PGDSF) policy is employed to minimize the cache miss rate. PGDSF calculates a priority based on the frequency, size of key-value tensors, last access time, and prefix-aware recomputation cost. The cache eviction is determined by the priority, which ensures the most valuable tensors are retained. Cache-aware reordering schedules the requests to improve cache hit rate and prevent thrashing, while also ensuring request fairness to mitigate starvation issues. Dynamic speculative pipelining and pipelined vector

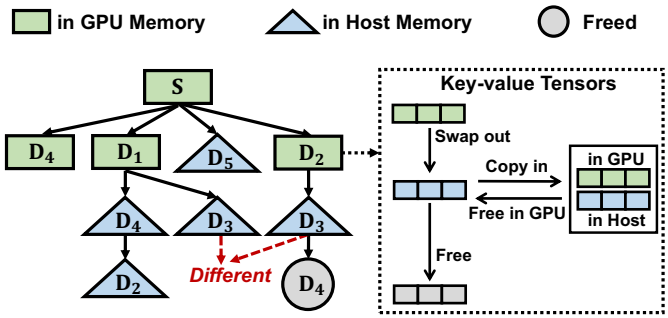


Fig. 10. Knowledge tree.

search is designed to overlap the knowledge retrieval and LLM inference to minimize the latency. This optimization leverages the mid-process generation of retrieval results to initiate LLM inference early.

5 RAGCache Design

In this section, we present the design of RAGCache. We first introduce the cache structure and the prefix-aware replacement policy (§5.1). Then, we describe the cache-aware reordering strategy to improve the cache hit rate (§5.2). Finally, we present the dynamic speculative pipelining approach to overlap knowledge retrieval and LLM inference (§5.3).

5.1 Cache Structure and Replacement Policy

Different from traditional cache systems that cache individual objects, RAGCache caches the key-value tensors of the retrieved documents that are sensitive to the referred order. For example, consider two document sequences: $[D_1, D_3]$ with key-value tensors KV and $[D_2, D_3]$ with KV'. Although $KV[1]$ and $KV'[1]$ both pertain to D_3, they are different in values. This discrepancy arises because the key-value tensor for a given token is generated based on the preceding tokens, underscoring the order-dependence of key-value tensors.

To facilitate fast retrieval while maintaining the document order, RAGCache structures the documents' key-value tensors with a knowledge tree, as depicted in Figure 10. This tree assigns each document to a node, which refers to the memory addresses of the document's key-value tensors. Following vLLM [6], RAGCache stores the key-value tensors in non-continuous memory blocks for KV cache reuse. The root node S denotes the shared system prompt. A path from the root to a particular node represents a sequence of documents.

The knowledge tree indirection leverages vLLM's page allocation mechanism. Instead of replicating the actual KV cache, each node in the knowledge tree—representing a single document—stores only the metadata of the corresponding KV cache pages, such as their indices. Consequently, insertion and eviction operations on the tree only involve modifying this metadata, thereby avoiding costly data transfers between GPU and host memory.

This design inherently allows RAGCache to serve multiple requests simultaneously through overlapping paths in the tree. RAGCache retrieves tensors by prefix matching along these paths. During the prefix matching process, if a subsequent document is not located among the child nodes, the traversal is promptly terminated, and the identified document sequence is returned. This method ensures efficiency with a time complexity of $O(h)$, where h represents the tree's height.

Prefix-aware Greedy-Dual-Size-Frequency (PGDSF) replacement policy. With the knowledge tree, RAGCache has to decide each node's placement within a hierarchical cache. Nodes that are accessed more frequently are ideally stored in GPU memory for faster access speeds, while those accessed less often are allocated to the slower

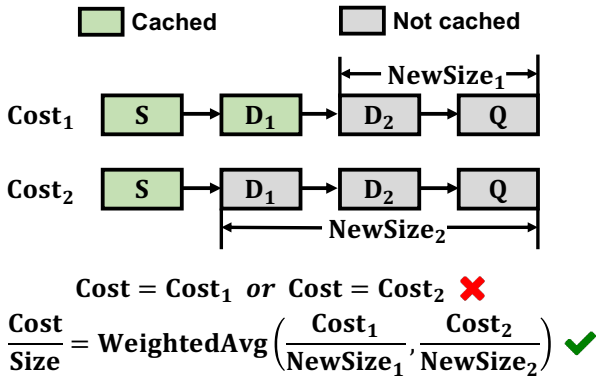


Fig. 11. Cost estimation in PGDSF.

host memory or simply freed. To optimize node placement, RAGCache employs a prefix-aware Greedy-Dual-Size-Frequency (PGDSF) replacement policy, which is based on the classic GDSF policy [34]. Unlike traditional caching strategies such as LRU, which neglect the variable sizes of documents, PGDSF evaluates each node based on its access frequency, size, and access cost. This method utilizes limited storage capacity by maintaining the most beneficial nodes, whose *priority* is defined as follows:

$$Priority = Clock + \frac{Frequency \times Cost}{Size} \tag{2}$$

Nodes with lower priority are evicted first. *Clock* tracks node access recency. We maintain two separate logical clocks in the RAG controller for GPU and host memory, respectively, to adapt to the cache hierarchy. Each clock starts at zero and updates with every eviction. When a document is retrieved, its node's clock is set and its priority is adjusted. Nodes with older clock, indicating less recent use, receive lower priorities. Let E be the set of evicted nodes in one eviction operation. The clock is updated accordingly:

$$Clock = \max_{n \in E} Priority(n) \tag{3}$$

Frequency represents the total retrieval count for a document within a time window. This count is reset to zero upon system start or cache clearance. The priority is proportional to the frequency, and thus more frequently accessed documents have higher priorities. *Size* reflects the number of tokens in a document post-tokenization, directly influencing the memory required for its key-value tensors. *Cost*, defined as the time taken to compute a document's key-value tensors, varies with GPU computational capacity, document size, and the sequence of preceding documents.

PGDSF achieves prefix awareness for RAG systems in two aspects: *Cost* estimation and node placement. Unlike GDSF, where costs are straightforward (e.g., object size in web caching), RAG costs involve complex LLM generation dynamics. For example, Figure 11 shows varying costs incurred by the same request denoted as $[S, D_1, D_2, Q]$. To estimate the cost for D_2, directly using the cost where $[S, D_1]$ is cached or only S is cached is imprecise. Besides, the latter case's cost also includes the time to compute the key-value tensors for D_1 and Q. PGDSF addresses this problem by replacing $Cost/Size$ in Formula 2 as follows:

$$\frac{Cost}{Size} = \frac{1}{m} \sum_{i=1}^{m} \frac{Cost_i}{NewSize_i} \tag{4}$$

where m is the number of requests that access the document but do not have the document cached. $Cost_i/NewSize_i$ represents the compute time per non-cached token for the i-th request. Such estimation inherently considers the

Algorithm 1 Knowledge Tree Operations

```
1: function UPDATE_NODE_IN_GPU(node, is_cached, α, β)
2:     // α and β are cached and non-cached sizes of the request
3:     node.Frequency ← node.Frequency + 1
4:     if is_cached is false then
5:         // Bilinear interpolation to estimate the cost
6:         Find α_l < α < α_h and β_l < β < β_h from the profiler
7:         T(α, β) ← BILINEAR(T(α_l, β_l), T(α_l, β_h), T(α_h, β_l), T(α_h, β_h))
8:         UPDATE_AVG_COST(node, T(α, β))
9:     node.Priority ← Clock + node.AvgCost × node.Frequency
10:
11: function EVICT_IN_GPU(required_size)
12:     E ← ∅ // Evicted nodes in GPU
13:     S ← {n ∈ GPU ∧ n.Children ∉ GPU} // Leaf nodes in GPU
14:     while Σ_{n∈E} n.Size < required_size do
15:         n ← arg min_{n∈S} n.Priority
16:         E ← E ∪ {n}
17:         Clock ← max{Clock, n.Priority}
18:         if n.Parent.Children ∉ GPU then
19:             S ← S ∪ {n.Parent}
```

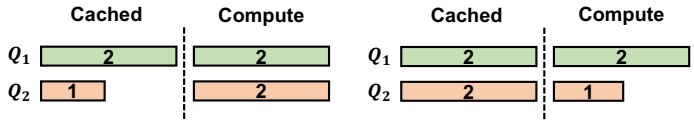
document size by amortizing the cost to all non-cached tokens. As for $Cost_i$, RAGCache profiles the LLM prefill time with varying cached and non-cached token lengths offline and uses bilinear interpolation to estimate the cost for a given request. Document retrieval triggers an update in node frequency, cost estimation and clock within the knowledge tree, or initiates a new node for documents not previously cached.

PGDSF orchestrates node placement in the knowledge tree, which is divided into GPU, host, and free segments, as illustrated in Figure 10. Nodes in GPU memory serve as parent nodes to those in host memory, establishing a hierarchical structure. RAGCache dynamically manages node eviction across these segments for efficiency. Specifically, when the GPU memory is full, RAGCache swaps the least priority node in leaf nodes to the host memory. RAGCache applies a similar process for host memory oversubscription. This eviction strategy upholds the tree's hierarchical partitioning, which is pivotal for aligning with the memory hierarchy and prefix sensitivity in LLM generation. A node relies on its parent node for key-value tensor calculation, emphasizing the need for prioritizing parent node placement for rapid retrieval.

Algorithm 1 outlines the operations for updating and evicting nodes in the GPU memory of the knowledge tree. $T(\alpha, \beta)$ represents the estimated compute time for a request with α cached tokens and β non-cached tokens. Upon a document retrieval by a request, RAGCache updates the cost using bilinear interpolation (line 6–9) if the document is not cached, EVICT_IN_GPU evicts nodes from the GPU memory to accommodate new requests and updates the clock according to Formula 3. If a parent node becomes a leaf following eviction, it is added to the candidate set S.

Swap out only once. The GPU connects to the host memory via the PCIe bus, which has significantly lower bandwidth than GPU HBM. To reduce data transfers, RAGCache adopts a swap-out-only-once strategy as shown in Figure 10. The key-value tensors of a node are swapped out to the host memory only on the first eviction and remain there until fully removed from the cache. For subsequent evictions in GPU memory, RAGCache directly frees the node without data copy. Since host memory is typically one to two orders of magnitude larger than GPU memory, keeping one copy of the key-value tensors in host memory is efficient and acceptable.

Multi-level eviction. When an incoming request necessitates the eviction of data from GPU memory, two scenarios are considered. First, if the victim node has already been backed up to host memory (per our swap-out-only-once



(a) Differ in cached length. (b) Differ in compute length.

Fig. 12. Cache-aware reordering.

strategy), the eviction simply requires freeing this node in the knowledge tree. Alternatively, if the host memory is also at capacity, a node must be evicted from the host as well, following a procedure similar to the EVICT_IN_GPU function detailed in Algorithm 1.

Critically, these eviction operations are highly efficient. As they only involve manipulating metadata within the knowledge tree, the physical memory pages holding the evicted KV cache are immediately made available for the new request without any data copying. Our evaluation in §7.5 confirms that the scheduling overhead for this process is under one millisecond, an amount that is negligible compared to the overall request processing time.

5.2 Cache-aware Reordering

Cache hit rate is vital for RAGCache's cache efficiency, yet the unpredictable arrival pattern of user requests results in substantial cache trashing. The requests referring to the same documents may not be issued together, affecting cache efficiency. For illustration, let requests $\{Q_i, i\%2 == 0\}$ and $\{Q_i, i\%2 == 1\}$ target documents D_1 and D_2, respectively. The cache capacity is one document. The sequence $\{Q_1, Q_2, Q_3...\}$ causes frequent swapping of the key-value cache of D_1 and D_2, rendering a zero cache hit rate. Conversely, rearranging requests to $\{Q_1, Q_3, Q_5, Q_2, Q_4, Q_6, Q_7, \ldots\}$ optimizes cache utilization, which improves the hit rate to 66%. This exemplifies how strategic request ordering can mitigate cache volatility and enhance cache efficiency.

Before introducing the cache-aware reordering algorithm, we first consider two scenarios to illustrate the key insights. We assume that the recomputation cost is proportional to the recomputation length in this example. The first scenario (Figure 12(a)) considers requests with identical recomputation demands but varying cached context lengths, under a cache limit of four. With an initial order of $\{Q_1, Q_2\}$, the system must clear Q_2's cache space to accommodate Q_1's computation, then reallocate memory for Q_1's processing. It effectively utilizes Q_1's cache while discarding Q_2's. This results in a total computation cost of $2 + 1 + 2 = 5$. Conversely, ordering as Q_2, Q_1 utilizes Q_2's cache but discards Q_1's, which increases computation to $2 + 2 + 2 = 6$. Thus, cache-aware reordering advocates prioritizing requests with larger cached contexts to enhance cache efficiency, as they bring larger benefits.

In the second scenario (Figure 12(b)), we examine requests with identical cached context lengths but varying recomputation demands, given a cache capacity of five. For a sequence $\{Q_1, Q_2\}$, the system must clear Q_2's cache to allocate space for Q_1's computation, given only one available memory slot. This necessitates recomputing Q_2 entirely, resulting in a computation cost of $2 + 2 + 1 = 5$. In contrast, the sequence $\{Q_2, Q_1\}$ allows for direct computation of Q_2 due to adequate cache availability. It reduces the total computation to $2 + 1 = 3$. Hence, cache-aware reordering is beneficial when it prioritizes requests with shorter recomputation segments, as this approach minimizes the adverse effects on cache efficiency.

Drawing from these insights, we introduce a cache-aware reordering algorithm aimed at improving cache efficiency. RAGCache employs a priority queue for managing incoming requests, prioritizing them based on their impact on cache performance. Specifically, requests are selected for processing based on a priority metric, defined as:

$$OrderPriority = \frac{Cached\ Length}{Computation\ Length}$$

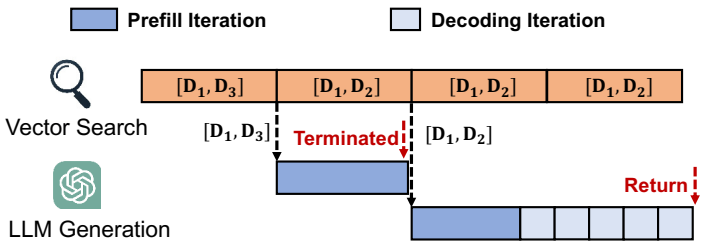


Fig. 13. Speculative pipelining.

This formula prioritizes requests that are likely to enhance cache efficiency——those with a larger cached portion relative to their computation needs. By adopting this cache-aware reordering, RAGCache increases the cache hit rate and decreases the total computation time, optimizing resource use and system performance. To avoid starvation, RAGCache sets a window for each request to ensure that all requests are processed no later than the window size.

5.3 Dynamic Speculative Pipelining

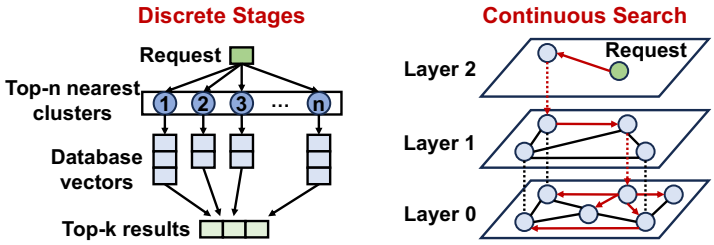
As we discuss in §3.3, the vector search time is determined by the knowledge base size and user configurations. If the vector database grows to a larger scale or the retrieving requires a higher accuracy, the retrieval step may incur a substantial latency. To mitigate the impact of retrieval latency, RAGCache employs dynamic speculative pipelining to overlap knowledge retrieval and LLM inference. The key insight behind this technique is that the vector search may produce the final results early in the retrieval step, which can be leveraged by LLM for speculative generation ahead of time.

Based on this observation, RAGCache introduces a speculative pipelining strategy that splits a request's retrieval process into several stages. In each stage, RAGCache ticks the vector database to send the candidate documents to the LLM engine for speculative generation. Then, the LLM engine starts a new speculative generation and terminates the previous generation if the received documents are different from the previous ones. If there is no difference, the LLM engine remains processing the previous generation. When the final top-k documents are produced, RAGCache sends the final results to the LLM engine. At this moment, the LLM engine returns the results of the latest speculative generation to users if it matches the final top-k documents. Otherwise, the LLM engine performs re-generation to ensure the correctness of the results.

As shown in Figure 13, we split the retrieval process into four stages. The top-2 documents in candidate queue are $[D_1, D_3]$, $[D_1, D_2]$, $[D_1, D_2]$, and $[D_1, D_2]$ in the four stages. After stage one is finished, RAGCache sends $[D_1, D_3]$ to the LLM engine for speculative generation. When stage two is finished, RAGCache sends $[D_1, D_2]$ to the LLM engine. The LLM engine finds that $[D_1, D_3]$ are different from $[D_1, D_2]$, and thus terminates the previous speculative generation and starts a new one. As for stage three, the LLM engine receives the same documents as stage two, and thus remains processing the previous generation. After the final stage, RAGCache sends the final top-2 documents to the LLM engine which is the same as the latest speculative generation. The LLM engine directly returns the speculative generation results to users.

Notably, RAGCache leverages the knowledge tree to retain the longest common document prefix when reconciling different speculative generations, thereby minimizing recomputation overhead. For example, as illustrated in Figure 13, when $[D_1, D_2]$ is received after $[D_1, D_3]$, the node representing D_3 is pruned from the tree. However, the KV cache associated with the common prefix, D_1, is retained. Consequently, the LLM engine can resume generation efficiently from this shared point without recomputing D_1's KV cache.

Pipelined vector search. The prerequisite for speculative pipelining is enabling pipelined vector search in the retrieval step, meaning the vector search must be divided into multiple stages and produce intermediate results at



(a) IVF. (b) HNSW.

Fig. 14. Different vector search algorithms.

the end of each stage. Figure 14 shows two widely-used vector search algorithms: IVF [11] and HNSW [12]. IVF partitions the vector space into multiple clusters and stores vectors within them. During search, IVF first locates the top-n closest clusters to the request vector and subsequently searches within these clusters. HNSW constructs multi-layer graphs to map the vector space, connecting vectors with edges based on similarity. HNSW searches the vectors by traversing the graph, maintaining a candidate list of the top-k nearest vectors. These algorithms exhibit different search characteristics: IVF operates in discrete stages (i.e., searching clusters one by one) while HNSW has a continuous search process. To support pipelined vector search, we adapt their search methods. For stage-based algorithms like IVF, we split the search into multiple stages, each processing a subset of clusters and returning the current top-k vectors. For continuous algorithms like HNSW, we measure the average search time for a given configuration and divide the entire time into smaller time slices. After each time slice, the current top-k vectors are returned. These modifications facilitate dynamic speculative pipelining while preserving the integrity of the final search results.

Dynamic speculative generation. The speculative pipelining allows RAGCache to overlap the retrieval and generation steps, which reduces the end-to-end latency of RAG systems. However, it may introduce extra LLM computation as some speculative generations are incorrect, potentially leading to performance degradation under high system loads. To address this problem, RAGCache dynamically enables speculative pipelining based on the system load.

We begin with a simplified analysis to show how to minimize end-to-end latency while managing system load. For simplicity, we assume both the vector search and LLM handle one request at a time. The vector search produces candidate results at the end of each stage at fixed intervals, much shorter than LLM generation time. Since the batch size is one, any incorrect speculative generation can be terminated immediately. In this case, the optimal strategy is simple: if a stage ends with different results, stop the current speculative generation (if any) and start a new one, as there is no termination cost.

The general RAG system is much more complex with parallel vector search and larger LLM batch sizes. We cannot immediately terminate speculative generations since they may be correct for other requests and the termination cost is non-negligible with batching. To handle this, we design a dynamic speculative pipelining strategy in Algorithm 2. The main idea is to start a speculative generation only if the retrieved documents change and the number of pending LLM requests falls below a predetermined maximum batch size for the prefill iteration ($max\_prefill\_bs$). The maximum prefill batch size is determined by the smaller number of tokens that can either fit within the GPU memory or fully utilize the GPU compute capabilities (i.e., streaming multiprocessors, SMs). The strategy terminates the incorrect speculative generation after the current LLM iteration, which does not affect other requests in the batch. This strategy overlaps the retrieval and generation steps as much as possible while gauging the system load.

Algorithm 2 Dynamic Speculative Pipelining Strategy

```
1: function DYNAMIC_SPECULATIVE_PIPELINING(request)
2:     D ← [ ]
3:     while the vector search of request is not finished do
4:         // Produce the candidate documents at the next stage
5:         D_temp ← VECTOR_SEARCH(request, D)
6:         if D_temp ≠ D then
7:             if {request, D} in queue then
8:                 Terminate {request, D} after the current iteration
9:             if queue.size < max_prefill_bs then
10:                queue.insert({request, D_temp})
11:        D ← D_temp
```

Dynamic speculative pipelining is orthogonal to, and fully integrated with, existing batching optimizations such as continuous batching [35] and chunked prefill [36]. The integration operates as follows. First, we assign a common speculative generation ID to all speculative generations originating from the same RAG request. At the end of each LLM generation iteration, the scheduler examines these IDs. If a speculative generation is found to be incorrect, it is terminated and removed from the batch using the same mechanism that handles sequences reaching their end-of-sequence (<eos>) token in continuous batching. For compatibility with chunked prefill, this process is adapted to handle the termination of partially-completed prefill requests. RAGCache retains the KV cache corresponding to the longest common document prefix that has already been processed, thereby maximizing computational reuse and ensuring no work is wasted.

5.4 Discussion and Limitations

Time between tokens (TBT). In addition to time to first token (TTFT), TBT is a crucial metric in LLM serving to assess the token streaming speed [37, 38]. RAG augments the request with external documents, significantly increasing the input length and thus the prefill latency, i.e., TTFT. Consequently, the primary concern for RAG systems is the prolonged TTFT [39]. RAGCache reduces TTFT by caching the KV cache of frequently retrieved documents. With RAGCache, TBT can be reduced or at least not worsen, as decoding iterations are less likely to be blocked by shorter prefill iterations. We evaluate RAGCache's TBT performance in §7.2.

Large top-k. As the top-k value increases, the number of document permutations grows factorially, reducing the likelihood of reuse. The key insight to mitigate this issue is that the top-k documents are exactly the first k documents in the top-k' set, where $k < k'$ (e.g., the top-2 documents are the same as the first two documents in top-5 documents), and caching only the first few documents can also lead to a considerable reduction in latency. Therefore, RAGCache caches documents with a lower top-k value (e.g., top-2) to balance the hit rate and cache efficiency. We perform an experiment to evaluate the impact of the top-k value on system performance in §7.3.

Large MoE models. As for more scaled-out MoE models like DeepSeek-V3 [40], RAGCache remains applicable because it is designed to avoid the re-computation of shared prefixes. This core principle is valid for all Transformer-based auto-regressive models, including MoE architectures. However, the relative performance gain on MoE models may be less pronounced than on traditional dense models, due to two primary considerations.

(i) Diminished computational savings: MoE models leverage sparse expert activation, meaning their prefix computation overhead is inherently lower than that of a dense model with a comparable parameter count. Consequently, the absolute time saved by caching constitutes a smaller fraction of the end-to-end latency.

(ii) Increased memory cost: MoE models are notoriously memory-intensive. Allocating additional memory for a KV cache exacerbates this pressure, creating a trade-off between using memory to reduce latency (via caching) versus using it to increase throughput (by supporting larger batch sizes).

Internet-scale knowledge bases. The fundamental principles of our approach——namely, prefix-aware caching and overlapping retrieval with prefill——are general and remain applicable to internet-scale data. However, effectively supporting an internet-scale knowledge base would introduce new system-level challenges. First, it would necessitate a more expansive KV cache storage solution, likely a multi-level hierarchy spanning GPU memory, host memory, and SSDs (e.g., Mooncake [41]). Furthermore, at that scale, the latency of billion-scale vector search would become the dominant bottleneck, far surpassing the prefill computation time, which would itself require significant optimization. Recent work, Hermes [42], stores the data across multiple nodes and performs relaxed parallel retrieval to improve efficiency, which can be integrated with RAGCache for internet-scale RAG applications.

Deployment at scale. As the escalation of LLMs and the knowledge bases, RAG applications increasingly require distributed deployments. Such at-scale deployments create a pressing need for efficient communication, not only for the data transfer between the retrieval and generation stages but also for the collective communications [43, 44] required within the distributed generation procedure itself.

As discussed in §2, RAG workflows are typically composed of offline processing and online inference phases. While many applications focus on real-time serving, offline RAG inference tasks also exist, such as Deep Research [45] proposed by OpenAI. In production environments, developing strategies for the efficient sharing [46, 47] of these online and offline workloads could significantly enhance resource utilization. We leave these studies as future work.

6 Implementation

We implement a system prototype of RAGCache with ~5000 lines of code in C++ and Python. Our implementation is based on vLLM [6] and Faiss [13]. We extend its prefill kernel in Pytorch [48] and Triton [49] to support prefix caching for different attention mechanisms, e.g., multi-head attention [8] and grouped-query attention [50].

Fault tolerance. We implement two fault-tolerant mechanisms in RAGCache to handle GPU failures and request processing failures. The GPU memory serves as RAGCache's first-level cache, storing the KV cache of the upper-level nodes in the knowledge tree hierarchy. Given the prefix sensitivity of LLM inference, a GPU failure would invalidate the lower-level nodes and therefore the entire tree. We replicate a portion of the most frequently accessed upper-level nodes (e.g., the system prompt) in the host memory for fast recovery. We also employ a timeout mechanism to retry the failed requests. If a request fails before completing its first iteration, it will be recomputed. Otherwise, the request can continue computation by reusing the stored KV cache.

7 Evaluation

In this section, we evaluate RAGCache from the following aspects: (i) overall performance against state-of-the-art approaches on open-source and production datasets; (ii) performance under general settings; (iii) ablation studies; and (iv) scheduling time of RAGCache.

Testbed. Most of our experiments are conducted on AWS EC2 g5.16xlarge instances, each with 64 vCPUs (AMD EPYC 7R32), 256 GiB host memory, and 25 Gbps NIC. Each instance is configured with one NVIDIA A10G GPU with 24 GiB memory and the GPU is connected to the host via PCIe 4.0×16. We run experiments with 7B models on a single g5.16xlarge instance and use 192 GiB host memory for caching unless otherwise stated. For large models, we use two NVIDIA H800 GPUs, each with 80 GiB memory and interconnected by NVLink. The two GPUs are connected to the host via PCIe 5.0×16. We use 384 GiB host memory for caching in this case.

Model	Layers	Q/KV Heads	MoE	Model Size	KV Size
Mistral-7B	32	32/8	no	14 GiB	0.125 MiB/token
Llama2-7B	32	32/32	no	14 GiB	0.5 MiB/token
Mixtral-8×7B	32	32/8	yes	96.8 GiB	0.125 MiB/token
Llama2-70B	80	64/8	no	140 GiB	0.3125 MiB/token

Table 1. Models used in the evaluation.

Models. We evaluate RAGCache with the Llama2 chat models [25] and the Mistral AI models [51, 52]. The model details are listed in Table 1. Most of the experiments are conducted with Mistral-7B and Llama2-7B. The two models have the same size but employ different attention mechanisms, i.e., grouped-query attention and multi-head attention. We also evaluate RAGCache with large models, Mixtral-8×7B and Llama2-70B, to demonstrate the scalability of RAGCache. Mixtral-8×7B is a mixture-of-experts (MoE) model with eight experts, and two experts are activated for each token. We deploy the large models on two H800 80GB GPUs for tensor parallelism and expert parallelism.

Open-source datasets. We use the Wikipedia dataset discussed in §3.2 as the knowledge base. For vector search, we use the IVF index with 1024 clusters and set the default top-k to 2. We deploy the vector database with four separate vCPUs and 30 GiB host memory on the same instance as the GPU and expose a RESTful API for document retrieval. Our evaluation uses two representative QA datasets, MMLU [27] and Natural Questions [28], to generate requests. MMLU is a multi-choice knowledge benchmark where the LLM outputs a single token (A/B/C/D) per question. Natural Questions comprises anonymized questions from Google Search and provides the reference answers for each question. For Natural Questions, we sample the output length for each question from the token length distribution of the reference answers. The average output length is 6 tokens, and 99% of the answers contain no more than 32 tokens. We sample a subset of the questions from the dataset, respecting the document retrieval distribution in §3.2, and randomly shuffle the questions to generate 1-hour workloads. In line with prior work [6, 53], we assign the arrival time for each request using a Poisson process parameterized by the arrival rate.

Production dataset. We also evaluate RAGCache with the production RAG dataset, Dataset-X, discussed in §3.2. Dataset-X has relatively long output lengths. It assists internal users at Company-X with their daily tasks. We use an internal knowledge base as the retrieval database and collect hundreds of user requests over the span of several days to form Dataset-X.

Metrics. We report the average time to first token (TTFT) as the main metric and measure the average time between tokens (TBT) for datasets with long outputs [37, 38]. System throughput is also evaluated, defined as the request rate the system can handle while keeping TTFT below a threshold, such as 5× the TTFT at the lowest request rate [54]. Additionally, we assess cache hit rate in the ablation study.

Baselines. We compare RAGCache with two baselines.

- **vLLM** [6], a state-of-the-art LLM serving system that supports iteration-level scheduling [35] and uses PagedAttention [6] to reduce memory fragmentation.
- **SGLang** [5], a high-performance LLM serving system that allows KV cache reuse across different requests in GPU memory and employs LRU as the replacement policy.

For fair comparison, the baselines are configured with the same model parallelism, maximum batch size, and vector database settings as RAGCache.

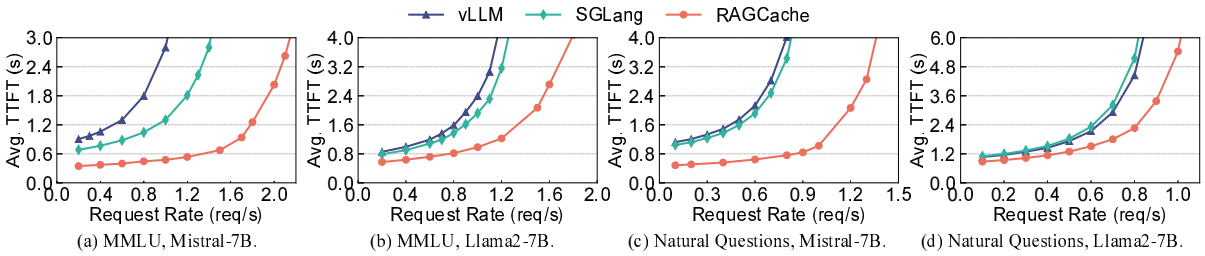


Fig. 15. Overall performance on MMLU and Natural Questions, using Mistral-7B and Llama2-7B as the models.

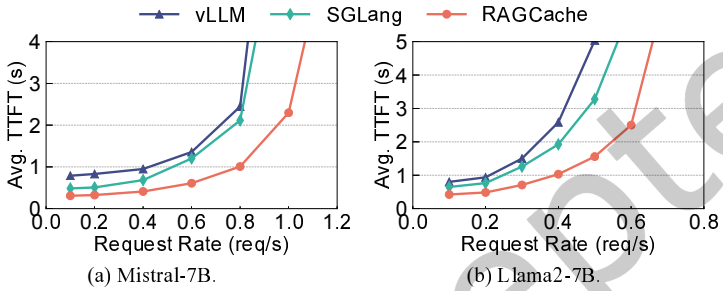


Fig. 16. Average TTFT on Dataset-X.

7.1 Overall Performance

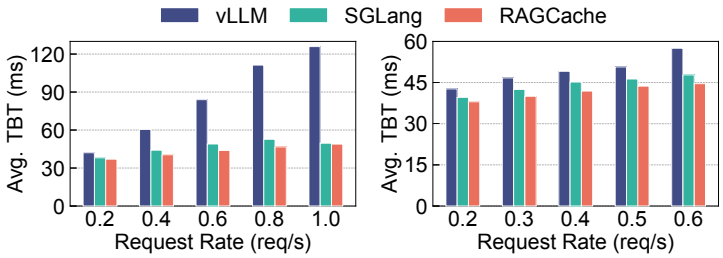
We first compare the overall performance of RAGCache against the baselines. We use MMLU and Natural Questions as the workloads and Mistral-7B and Llama2-7B as the models. The maximum batch size is set to 4. We vary the request rate and measure the average TTFT. Figure 15 shows the results on MMLU and Natural Questions, which we summarize as follows.

- RAGCache reduces the average TTFT by 1.2–4× compared to vLLM and 1.1–3.5× compared to SGLang under the same request rate. This is because RAGCache utilizes the GPU memory and host memory to cache the KV cache of hot documents and avoids frequent recomputation.
- Due to faster request processing, RAGCache achieves 1.3–2.1× higher throughput than vLLM and 1.2–1.8× higher throughput than SGLang.
- RAGCache outperforms the baselines across models with varying attention mechanisms on different datasets.

The results also reflect the differences between the models and datasets. The performance gap between RAGCache and vLLM is greater for Mistral-7B than for Llama2-7B. This is because Llama2-7B has a KV cache size 4× that of Mistral-7B for the same token count, resulting in a lower cache hit rate for Llama2-7B with the same cache size. According to our characterization in §3.2, MMLU benefits more from document caching than Natural Questions, with a wider performance improvement for MMLU than for Natural Questions. SGLang performs closely to vLLM for Natural Questions because the limited GPU memory restricts document locality. RAGCache, however, with its multilevel caching and the adapted knowledge tree, outperforms in both datasets.

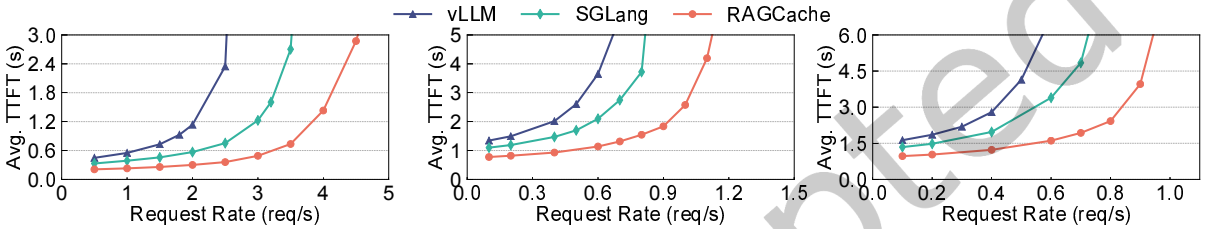
7.2 Benefits for Real-World Production Datasets

We then evaluate the performance of RAGCache on the production dataset, Dataset-X, which is collected from the internal RAG application at Company-X. Dataset-X features relatively longer outputs compared to the other



(a) Mistral-7B. (b) Llama2-7B.

Fig. 17. Average TBT on Dataset-X.



(a) Top-1. (b) Top-3. (b) Top-5.

Fig. 18. Performance with different top-k values.

two datasets, with an average output length nearing 100 tokens. Figure 16 illustrates that RAGCache achieves a 1.9–2.6× reduction in TTFT compared to vLLM and a 1.2–2.1× reduction against SGLang, while improving throughput by 1.2× over the baselines. As mentioned in §5.4, due to long sequence injections, TTFT is the main concern for RAG systems [39]. In addition to TTFT, we also report the average TBT to assess the effectiveness of RAGCache in token streaming speed as shown in Figure 17. RAGCache consistently achieves lower average TBT than both vLLM and SGLang, regardless of load conditions. This is because RAGCache processes requests faster with knowledge caching, reducing the likelihood of decoding iterations being blocked by prefill iterations.

7.3 Case Study

We conduct two case studies to demonstrate the benefits of RAGCache over the baselines under general settings. We use MMLU and Mistral-7B in the case studies.

Different top-k values. Users may have varying requirements for the number of retrieved documents. We evaluate the performance of RAGCache and the baselines with commonly used top-k values: 1, 3, and 5. We set the maximum batch size to 4 and truncate the documents in the top-5 experiment to fit within GPU capacity limits. Figure 18 shows that RAGCache outperforms vLLM by 1.7–3.1× and SGLang by 1.2–2.5× in average TTFT across these top-k values. Despite the factorial growth in document permutations with increasing top-k values, RAGCache maintains its advantage by caching frequently-used documents. This is because the knowledge tree always evicts the node furthest from the root, ensuring that the most frequently used prefixes remain in the cache.

Large models. The second case study evaluates RAGCache with larger models using MMLU as the workload. We deploy Mixtral-8×7B and Llama2-70B on two H800 80GB GPUs. For each model, we set the maximum batch size to the lesser of what fits in GPU memory or fully utilizes the SMs (e.g., 8 for Mixtral-8×7B and 4 for Llama2-70B). Figure 19 shows that under low request rates, RAGCache reduces the average TTFT by 1.4–2.1× compared to

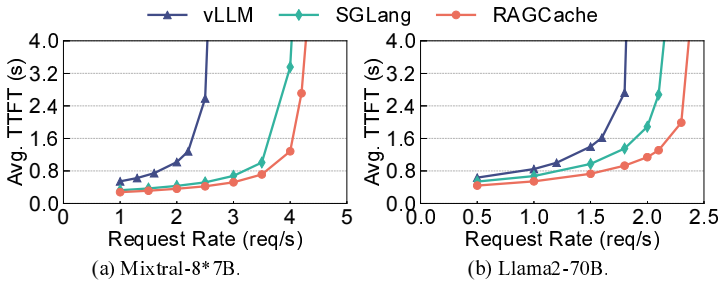


Fig. 19. Performance under large models.

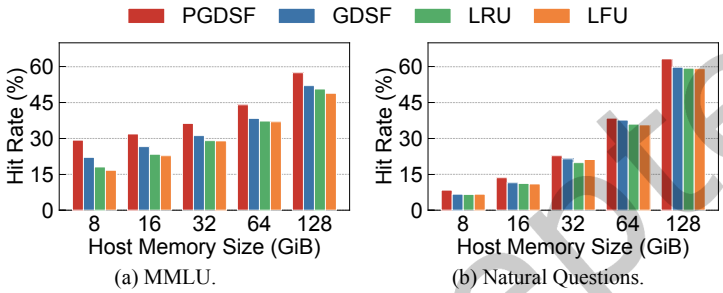


Fig. 20. Ablation study on cache replacement policy.

vLLM. SGLang performs better on H800 than on A10G GPUs due to increased GPU memory for caching, but RAGCache still surpasses SGlang by 1.2–2.6× in average TTFT.

7.4 Ablation Study

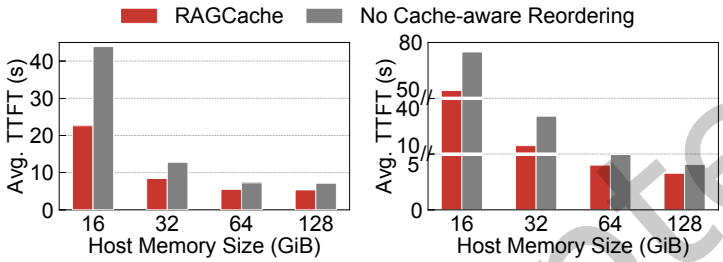
Prefix-aware GDSF policy. We compare RAGCache with versions of RAGCache that use native GDSF, LRU, and LFU as the replacement policy. For GDSF, we set the recomputation cost of a document proportional to its size, which aligns with our profiling results in Figure 3. We vary the host memory for caching from 8 GiB to 128 GiB, set the request rate to 0.8 req/s, and report the hit rate and average TTFT. The hit rate for top-2 retrieval is defined as the ratio of hit documents to retrieved ones. For example, if the cached document sequence is $[D_1, D_2]$ and the requested one is $[D_1, D_3]$, the hit rate is 50%. Figure 20 shows the hit rates for MMLU and Natural Questions, and Table 2 lists the average TTFT. PGDSF achieves the highest hit rates across different host memory sizes, improving 1.02–1.32× over GDSF, 1.06–1.62× over LRU, and 1.06–1.75× over LFU, as it captures the varying sizes, access patterns, and recomputation costs of different document prefixes. With higher hit rates, RAGCache achieves 1.05–1.29× lower average TTFT than the baselines.

Cache-aware reordering. Then we evaluate the impact of cache-aware reordering. Reordering works when the request queue is saturated. We set the request rate to 2.5 req/s for MMLU and 1.4 req/s for Natural Questions, which are slightly higher than the throughput of RAGCache. We set the reordering window size to 32 and vary the host memory size from 16 GiB to 128 GiB. Figure 21 shows that RAGCache reduces the average TTFT by 1.2–2.1× with cache-aware reordering, which demonstrates its effectiveness under high request rates.

Our reordering mechanism uses a scheduling window of 32 requests, which means the maximum queueing length for any individual request is bounded by this size. The actual extent of reordering is typically smaller than this limit. Similar to most caching systems, user requests exhibit temporal locality. Therefore, newly arriving requests often

Host Memory Size	MMLU				Natural Questions			
	PGDSF	**GDSF**	**LRU**	**LFU**	**PGDSF**	**GDSF**	**LRU**	**LFU**
8 GiB	1.38	1.68	1.78	1.81	2.85	3.35	3.41	3.36
16 GiB	1.32	1.55	1.61	1.63	2.50	2.89	2.92	2.98
32 GiB	1.23	1.45	1.50	1.49	2.00	2.09	2.25	2.20
64 GiB	1.06	1.27	1.28	1.29	1.32	1.47	1.56	1.55
128 GiB	0.83	0.98	1.01	1.03	0.78	0.92	0.95	0.95

Table 2. Average TTFT (seconds) of different replacement policies with varying host memory size.



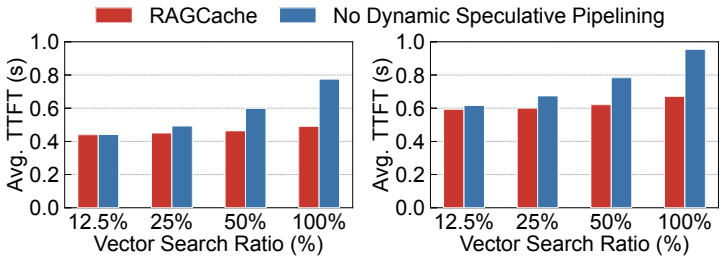
(a) MMLU. (b) Natural Questions.

Fig. 21. Ablation study on cache-aware reordering.

have a lower probability of a cache hit than requests that are already at the head of the queue. Consequently, the likelihood of a reordering event spanning the full window size is low.

Dynamic speculative pipelining. Finally, we evaluate the effectiveness of dynamic speculative pipelining against a baseline, No Dynamic Speculative Pipelining (No DSP), which waits for vector search completion before starting LLM generation. We adjust the proportion of vectors to be searched, relative to the total number of vectors in the database (referred to as the vector search ratio), ranging from 12.5% to 100%. Note that while the search accuracy increases with the vector search ratio, the search time also extends. We use MMLU and Natural Questions as the workloads and set the request rate to 0.1 req/s. Figure 22 demonstrates that RAGCache achieves up to 1.6× TTFT reduction with dynamic speculative pipelining.

Note that the semantics of "speculative" in our dynamic speculative pipelining differ from its use in speculative decoding [55]. The latter introduces randomness through modified rejection sampling [55], resulting in a probabilistic acceptance rate (between 0 and 1) for each step. Our dynamic speculative pipelining, in contrast, operates on deterministic vector search results where a correct outcome is guaranteed, not probabilistic. Therefore, a more intuitive metric for our method's effectiveness is the fraction of the total search process required to identify the correct final result. Table 3 presents this comparison by measuring the average non-overlapping vector search time, i.e., the duration that the vector search does not overlap with the LLM generation using the final retrieval result, for RAGCache (T_1) against the baseline No DSP (T_2). Consequently, the ratio T_1/T_2 represents the fraction of time required to secure the correct result with our method, while $1 - T_1/T_2$ quantifies the relative time savings achieved through dynamic speculative pipelining. For example, for the MMLU query set where the retrieval step searches the entire dataset, RAGCache saved 76.9% of the vector search time. Overall, dynamic speculative pipelining allows RAGCache to decrease non-overlapping vector search time by 1.5–4.3× and leads to a lower TTFT.



(a) MMLU. (b) Natural Questions.

Fig. 22. Ablation study on speculative pipelining.

Vector Search Ratio	MMLU		Natural Questions	
	RAGCache	**No DSP**	**RAGCache**	**No DSP**
12.5%	52.1 ms	78.5 ms	67.7 ms	105.8 ms
25%	59.2 ms	135.9 ms	72.9 ms	163.4 ms
50%	69.7 ms	243.7 ms	94.2 ms	282.5 ms
100%	97.4 ms	422.3 ms	145.0 ms	446.1 ms

Table 3. Average non-overlapping vector search time under different settings.

7.5 Scheduling Time

We measure RAGCache's scheduling time, including the time for knowledge tree lookup and update, request reordering, and speculative pipelining decisions. Using MMLU as the workload and Mistral-7B as the model, we range the request rate from 0.5 to 2 req/s. The scheduling time ranges from 0.872 to 0.906 millisecond, which is negligible compared to the second-level TTFT.

8 Related Work

RAG. RAG [1, 2, 14–17] enhances the generation quality of LLMs by incorporating relevant knowledge from external databases. Several works [14, 16, 17, 56] suggest iterative retrieval throughout generation to further improve the response quality. RAGCache supports iterative retrieval by treating the intermediate iterations as separate requests and caching the corresponding KV cache of the documents.

Vector search. RAG systems convert user prompts into vectors and uses approximate nearest neighbor (ANN) indexes like IVF [11, 23, 24, 57] and graph indexes [12, 58, 59] for efficient and accurate similarity search. RAGCache extracts the temporary search results for speculative LLM generation and thus pipelines the search process with LLM inference.

KV cache management. KV cache is widely used to accelerate the decoding phase of LLM inference [6, 35, 37, 53, 60]. Recent efforts aim to reduce the KV cache's memory footprint by quantization [61], compression [62–64], and self-attention with a subset of tokens [60, 65]. These methods introduce approximation to the generation process, while RAGCache preserves the exact KV cache of documents without affecting generation quality. Inspired by virtual memory in operating systems, vLLM [6] manages the KV cache at page granularity and proposes PagedAttention to prevent external fragmentation. RAGCache integrates the page-level management for KV cache sharing and improves over vLLM by leveraging the characteristics of RAG to cache the KV cache of the knowledge documents.

KV cache reusing. Recent efforts [5, 39, 66–68] propose to reuse the KV cache across requests to reduce redundant computation. Prompt Cache [66] allows flexible reuse of the same tokens at different positions, while CacheGen [39]

compresses the KV cache for efficient reuse. CacheBlend [68] uses selective KV recomputation to fuse multiple pre-computed KV caches and intelligently recomputes a small, critical fraction of them to restore the missing cross-attention information. These approaches may generate inaccurate responses due to the approximation of the KV cache. SGLang [5] and ChunkAttention [67] identify the reusable KV cache in GPU memory. RAGCache leverages RAG's retrieval pattern and builds a multilevel caching system, leading to higher performance with unchanged generation results.

RAG performance optimization. Recent efforts have introduced targeted optimizations for the end-to-end RAG workflow. A concurrent work, TeleRAG [69], proposes lookahead prefetching to optimize the retrieval step across the pre-generation, retrieval, and post-generation stages in multi-turn RAG. It prefetches IVF clusters during pre-generation and employs a GPU-CPU hybrid vector search to accelerate retrieval. Hermes [42] targets retrieval with trillion-scale datasets, where retrieval becomes the primary bottleneck. It employs a distributed datastore for parallel retrieval and uses a sampling-based hierarchical search for efficient retrieval with acceptable algorithmic performance. Other works optimize RAG performance by leveraging novel hardware architectures. HeterRAG [70] employs DIMM-based Processing-in-Memory (PIM) for the high-memory-capacity retrieval stage and HBM-based PIM for the high-memory-bandwidth generation stage. RAGX [71] targets the retrieval bottleneck in RAG that uses persistent storage by proposing a programmable in-storage accelerator to co-locate computation (vector search) with data (vector embeddings). RAGCache can be integrated with these solutions for further optimization in more complex RAG scenarios.

9 Conclusion

We present RAGCache, a latency-optimized serving system tailored for RAG. Based on a detail RAG system characterization, RAGCache employs a knowledge tree with a prefix-aware replacement policy to minimize redundant computation in the LLM and a dynamic speculative pipelining mechanism to overlap the knowledge retrieval and LLM inference to minimize end-to-end latency. We evaluate RAGCache with a variety of models on both open-source and production workloads. Experimental results show that RAGCache outperforms the state-of-the-art solution, vLLM integrated with Faiss, by up to 4× on TTFT and 2.1× on throughput.

Acknowledgments

We thank Editor in Chief of TOCS, Sam H. Noh and Robbert van Renesse, and the anonymous Associate Editor and reviewers from TOCS, for their insightful feedback.

References

[1] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[2] 2024. LangChain. https://python.langchain.com/docs/get_started/introduction. (2024).

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[4] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. 2024. Benchmarking large language models in retrieval-augmented generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 17754–17762.

[5] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2023. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104* (2023).

[6] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[7] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems* 36 (2023), 11809–11822.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[9] Hung-Ting Chen, Fangyuan Xu, Shane A Arora, and Eunsol Choi. 2023. Understanding retrieval augmentation for long-form question answering. *arXiv preprint arXiv:2310.12150* (2023).

[10] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.

[11] Artem Babenko and Victor Lempitsky. 2014. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence* 37, 6 (2014), 1247–1260.

[12] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[13] 2024. Pinecone: Introduction to Facebook AI Similarity Search (Faiss). (2024). https://www.pinecone.io/learn/series/faiss/faiss-tutorial/.

[14] Wenqi Jiang, Shuai Zhang, Boran Han, Jie Wang, Bernie Wang, and Tim Kraska. 2024. Piperag: Fast retrieval-augmented generation via algorithm-system co-design. *arXiv preprint arXiv:2403.05676* (2024).

[15] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. 2022. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*. PMLR, 2206–2240.

[16] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. 2023. In-context retrieval-augmented language models. *Transactions of the Association for Computational Linguistics* 11 (2023), 1316–1331.

[17] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. *arXiv preprint arXiv:2212.10509* (2022).

[18] Shamane Siriwardhana, Rivindu Weerasekera, Elliott Wen, Tharindu Kaluarachchi, Rajib Rana, and Suranga Nanayakkara. 2023. Improving the domain adaptation of retrieval augmented generation (RAG) models for open domain question answering. *Transactions of the Association for Computational Linguistics* 11 (2023), 1–17.

[19] Daniel Adiwardana, Minh-Thang Luong, David R So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, et al. 2020. Towards a human-like open-domain chatbot. *arXiv preprint arXiv:2001.09977* (2020).

[20] Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. 2022. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive nlp. *arXiv preprint arXiv:2212.14024* (2022).

[21] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722* (2022).

[22] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[23] Zili Zhang, Chao Jin, Linpeng Tang, Xuanzhe Liu, and Xin Jin. 2023. Fast, approximate vector queries on very large unstructured datasets. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 995–1011.

[24] Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin. 2024. Fast Vector Query Processing for Large Datasets Beyond {GPU} Memory with Reordered Pipelining. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 23–40.

[25] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[26] 2024. Wikipedia (en) embedded with cohere.ai multilingual-22-12 encoder. https://huggingface.co/datasets/Cohere/wikipedia-22-12-en-embeddings/. (2024).

[27] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300* (2020).

[28] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. 2019. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics* 7 (2019), 453–466.

[29] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[30] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. 2017. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

[31] 2024. OpenAI text-embedding-3 model. https://openai.com/blog/new-embedding-models-and-api-updates/. (2024).

[32] 2024. OpenAI. https://openai.com/. (2024).

[33] Conglong Li, Minjia Zhang, David G Andersen, and Yuxiong He. 2020. Improving approximate nearest neighbor search through learned adaptive early termination. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2539–2554.

[34] Ludmila Cherkasova. 1998. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories Palo Alto, CA, USA.

[35] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[36] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 117–134.

[37] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 193–210.

[38] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.

[39] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. 2024. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 38–56.

[40] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2024. DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437* (2024).

[41] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 155–170.

[42] Michael Shen, Muhammad Umar, Kiwan Maeng, G. Edward Suh, and Udit Gupta. 2025. Hermes: Algorithm-System Co-design for Efficient Retrieval-Augmented Generation At-Scale. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 958–973.

[43] Xin Jin, Zhen Zhang, Yunshan Jia, Yun Ma, and Xuanzhe Liu. 2024. SDCC: Software-defined collective communication for distributed training. *Science China Information Sciences* 67, 9 (2024), 192104.

[44] Yongji Wu, Yechen Xu, Jingrong Chen, Zhaodong Wang, Ying Zhang, Matthew Lentz, and Danyang Zhuo. 2024. MCCS: A Service-based Approach to Collective Communication for Multi-Tenant Cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 679–690.

[45] 2025. Introducing Deep Research. https://openai.com/index/introducing-deep-research/. (2025).

[46] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent {GPU} sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 69–85.

[47] Xuanzhe Liu, Yihao Zhao, Shufan Liu, Xiang Li, Yibo Zhu, Xin Liu, and Xin Jin. 2024. MuxFlow: efficient GPU sharing in production-level clusters with more than 10000 GPUs. *Science China Information Sciences* 67, 12 (2024), 222101.

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* (2019).

[49] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.
[50] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245* (2023).
[51] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).
[52] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024).
[53] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920* (2023).
[54] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 663–679.
[55] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating Large Language Model Decoding with Speculative Sampling. *arXiv preprint arXiv:2302.01318* (2023).
[56] Zhihao Zhang, Alan Zhu, Lijie Yang, Yihua Xu, Lanting Li, Phitchaya Mangpo Phothilimthana, and Zhihao Jia. 2024. Accelerating retrieval-augmented language model serving with speculation. *arXiv preprint arXiv:2401.14021* (2024).
[57] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems* 34 (2021), 5199–5212.
[58] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. In *Proceedings of the VLDB Endowment*.
[59] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in neural information processing Systems* 32 (2019).
[60] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453* (2023).
[61] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in neural information processing systems* 35 (2022), 30318–30332.
[62] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. 2020. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on machine learning*. PMLR, 5958–5968.
[63] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2023. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems* 36 (2023), 52342–52364.
[64] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2023. Model tells you what to discard: Adaptive kv cache compression for llms. *arXiv preprint arXiv:2310.01801* (2023).
[65] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2023. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems* 36 (2023), 34661–34710.
[66] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems* 6 (2024), 325–338.
[67] Lu Ye, Ze Tao, Yong Huang, and Yang Li. 2024. ChunkAttention: Efficient Self-Attention with Prefix-Aware KV Cache and Two-Phase Partition. *arXiv preprint arXiv:2402.15220* (2024).
[68] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*. 94–109.
[69] Chien-Yu Lin, Keisuke Kamahori, Yiyu Liu, Xiaoxiang Shi, Madhav Kashyap, Yile Gu, Rulin Shao, Zihao Ye, Kan Zhu, Stephanie Wang, Arvind Krishnamurthy, Rohan Kadekodi, Luis Ceze, and Baris Kasikci. 2025. TeleRAG: Efficient Retrieval-Augmented Generation Inference with Lookahead Retrieval. *arXiv preprint arXiv:2502.20969* (2025).
[70] Chaoqiang Liu, Haifeng Liu, Dan Chen, Yu Huang, Yi Zhang, Wenjing Xiao, Xiaofei Liao, and Hai Jin. 2025. HeterRAG: Heterogeneous Processing-in-Memory Acceleration for Retrieval-augmented Generation. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 884–898.

[71] Rohan Mahapatra, Harsha Santhanam, Christopher Priebe, Hanyang Xu, and Hadi Esmaeilzadeh. 2025. In-Storage Acceleration of Retrieval Augmented Generation as a Service. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 450–466.

Received 21 April 2025; revised 22 August 2025; accepted 5 September 2025