CRAFT: Chunk-Level KV Cache Reuse for Efficient RAG Serving

Anonymous Author(s)

Abstract

Large Language Models (LLMs) are increasingly deployed in retrieval-augmented generation (RAG) settings, where external documents are incorporated at inference time to improve response quality. While RAG enhances accuracy, it also increases input length and serving cost, making prefill latency and key-value (KV) cache computation major bottlenecks in LLM serving systems. These bottlenecks arise because the KV cache size grows with sequence length and batch size, placing significant pressure on GPU memory, memory bandwidth, and time-to-first-token (TTFT) in interactive workloads. In practice, many requests retrieve overlapping document chunks, yet existing prefix caching mechanisms can only reuse KV cache when inputs share identical token prefixes. As a result, substantial reuse opportunity remains unexploited due to variations in chunk ordering.

In this paper, we observe that retrieved RAG chunks are typically semantically self-contained, making their ordering a free optimization variable. By exploiting this property, we reorder chunks into a consistent order across requests to improve prefix alignment and KV reuse. Based on this insight, we present CRAFT, a framework that restructures retrieved chunks to better satisfy the reuse conditions of existing prefix caching. CRAFT reorders chunks using access-frequency statistics so that frequently shared chunks appear earlier and more consistently across requests, and eliminates redundant computation in multi-turn settings through conversation-scoped chunk deduplication. We implement CRAFT on top of vLLM and evaluate it across multiple RAG benchmarks and models. Results show that CRAFT reduces TTFT by 1.2–1.6× on evaluated workloads, improving latency and resource efficiency in LLM serving, with no measurable degradation in response quality.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities across a broad range of real-world applications [1, 8, 38, 40, 43], but are constrained by their static training data and finite parametric knowledge, making them prone to factual errors, outdated information, and hallucinations when handling queries that demand up-to-date or domain-specific information [18, 23]. Retrieval-Augmented Generation (RAG) mitigates this by retrieving external knowledge at inference time: documents are segmented into chunks (typically 128–512 tokens), which are retrieved and prepended to the input query [5, 29, 37], with each user query (a *request*) retrieving a set of these chunks.

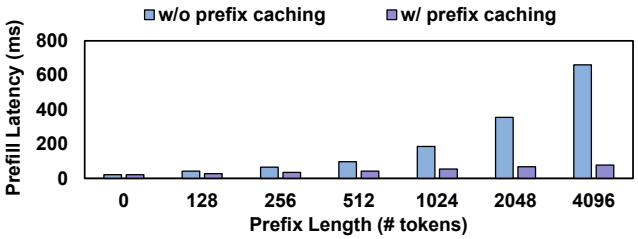

Figure 1. Prefill latency characterization.

As LLM deployments scale, efficient serving becomes critical. RAG further amplifies this challenge by increasing input length and prefill cost due to retrieved chunks. A key bottleneck is key-value (KV) cache management, whose memory and bandwidth overhead grow linearly with sequence length and batch size. In addition, the prefill stage dominates time-to-first-token (TTFT), a key latency metric for interactive workloads. As a result, reducing redundant KV computation directly improves both memory efficiency and end-to-end serving latency, making it a critical optimization for high-throughput LLM systems. These challenges are particularly pronounced in RAG workloads, where augmented inputs further increase sequence length and exacerbate KV cache growth and prefill cost. Prior work [25, 47] suggests, and our analysis confirms, that in RAG serving, retrieved chunks often overlap across requests, yet their KV caches are recomputed repeatedly, leading to substantial redundant computation and memory traffic.

While modern systems mitigate this overhead using prefix caching, reuse is only possible when inputs share identical token prefixes. This is widely adopted in systems such as vLLM [28] and SGLang [54]. Figure 1 illustrates the effectiveness of prefix caching. The "w/o prefix caching" bars show prefill latency without KV reuse, while "w/ prefix caching" bars show prefill latency when cached KV is reused. When requests share identical prefixes, KV reuse significantly reduces prefill latency compared to full recomputation.

In RAG, however, overlapping chunks can appear in different orders across requests, breaking prefix alignment and preventing reuse. This mismatch between high content overlap and low prefix alignment represents a fundamental inefficiency in KV cache utilization. Importantly, this inefficiency arises not from limitations in the prefix caching mechanism itself, but from the input structure presented to it. This redundancy is further amplified in multi-turn RAG settings, where a *conversation* consists of multiple *turns* (query–response

pairs), and each turn issues a new request that may retrieve overlapping chunks with earlier turns. As a result, a large fraction of reusable computation remains unexploited.

To quantify this inefficiency, we analyze chunk overlap patterns across requests in realistic workloads. We find that, on average, 43% of chunks retrieved by a request are also retrieved by prior requests, yet only 11% of chunks are prefix-aligned and reusable under standard prefix caching. This mismatch reveals a fundamental inefficiency: the bottleneck is not the lack of reusable content, but the inability of existing mechanisms to exploit it due to positional misalignment. Prior chunk-level KV reuse approaches (e.g., RAGCache [25], CacheBlend [47]) assume fixed retriever-provided chunk ordering and therefore cannot fully exploit this opportunity.

Crucially, this limitation stems from an overlooked degree of freedom: while the ordering of retrieved chunks does not affect generation quality, it determines whether prefix caching can be applied. Retrieved chunks are typically semantically self-contained, allowing their order to be treated as an optimization variable without changing task semantics. By reordering chunks into a consistent order across requests, shared chunks can be aligned into common prefix positions, enabling prefix caching to capture substantially more reuse. Furthermore, in multi-turn RAG, we find that retrieved chunks often recur across turns within the same conversation, providing an additional opportunity to eliminate redundant computation through simple deduplication.

Based on these insights, we present CRAFT, a framework for RAG LLM serving systems that improves KV cache reuse by reordering chunks into a consistent order across requests. Rather than modifying the KV cache or attention mechanism, CRAFT restructures inputs to better satisfy the reuse conditions of existing prefix caching. It reorders retrieved chunks using access-frequency statistics so that frequently shared chunks appear earlier and more consistently across requests, increasing the likelihood of prefix alignment. CRAFT then indexes reusable chunk-prefix KV using a Chunk-Prefix Tree (CP-Tree) and, in multi-turn settings, eliminates redundant computation via conversation-scoped chunk deduplication. Unlike prior chunk-level KV reuse systems (e.g., RAGCache [25]), which operate on the retriever's original chunk ordering, CRAFT treats chunk order as an optimization variable and restructures inputs to improve prefix alignment across requests and *make prefix caching and prior chunk-level KV reuse systems applicable more often.*

We evaluate CRAFT in vLLM across multiple workloads and show that chunk reordering across requests is an effective and practical approach to improving KV cache utilization. Overall, this paper makes the following contributions:

- We identify a key limitation of token-level prefix caching in RAG workloads: although retrieved chunks overlap substantially across requests, variations in their ordering prevent effective KV cache reuse.

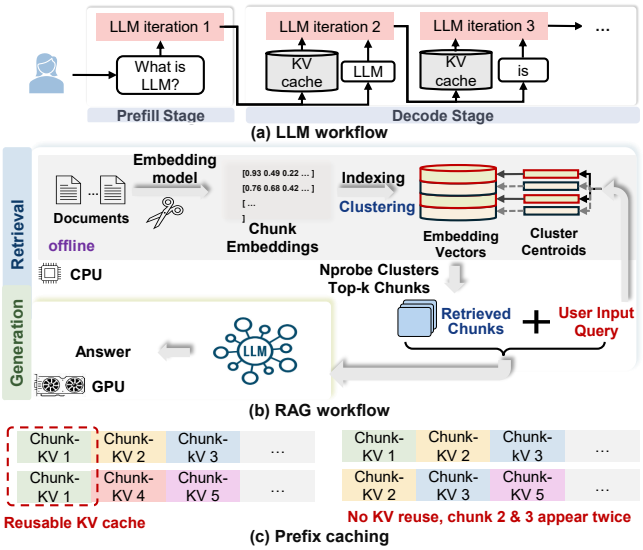

Figure 2. Overview of LLM inference, RAG workflow, and prefix caching.

- We show that retrieved chunks are typically semantically self-contained, and that reordering them does not affect generation quality, enabling systems to make chunk order consistent across requests.
- We present CRAFT, a lightweight framework that improves KV cache reuse by reordering chunks to increase prefix alignment across requests.
- We implement CRAFT on top of vLLM and evaluate it across multiple RAG benchmarks and models, demonstrating up to 38% reduction in KV computation and 1.2–1.6× improvement in TTFT with no measurable degradation in response quality.

2 Background

2.1 Basic LLM Concepts

Modern Large Language Models (LLMs) are typically based on the Transformer architecture [42], which uses a self-attention mechanism to process input sequences. As shown in Figure 2(a), the model first performs a prefill stage, computing and storing key (K) and value (V) vectors for all input tokens in a key-value (KV) cache. It then enters the decode stage, where tokens are generated iteratively using the KV cache, with new K and V vectors appended at each step. The KV cache grows linearly with sequence length and the number of active requests, making it a major bottleneck in high-throughput serving systems due to increased memory overhead and computation latency.

2.2 Retrieval Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) enhances LLMs by incorporating external knowledge, enabling more accurate and up-to-date responses [5, 29, 37]. In RAG, each query

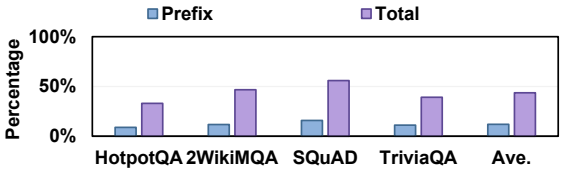

Figure 3. Chunk overlap across requests (prefix vs total).

retrieves a small set of relevant documents, segmented into chunks and prepended to the query as context (Figure 2(b)). RAG follows a two-stage pipeline with CPU-side retrieval and GPU-side generation. Offline, documents are segmented, embedded, and indexed for similarity search. Online, a query is encoded and used to retrieve top-k relevant chunks, typically via approximate nearest neighbor (ANN) methods [33]. We adopt a cluster-based ANN approach (e.g., IVF), where the query first identifies nearest clusters via centroid comparison, followed by refined search within those clusters. The retrieved chunks are then prepended to the query to form the augmented prompt for generation.

2.3 Prefix Caching

To reduce KV cache latency, LLM serving systems adopt prefix caching, which reuses KV states when input prompts share an identical token prefix [13, 22, 28, 47]. This technique is widely used in systems such as vLLM [28], SGLang [54], and RAGCache [25]. The KV cache for a prefix is computed once and reused by subsequent requests, avoiding redundant computation and storage (Figure 2(c)). This is valid because the KV cache of a prefix depends only on its tokens and is independent of the suffix. However, prefix caching requires strict token-level alignment: any variation in the input breaks reuse, even when inputs share substantial content. This limitation is particularly pronounced in RAG, although different queries may retrieve overlapping chunks, variations in their ordering and composition disrupt prefix alignment, preventing KV reuse under standard prefix caching.

Basic Terminology. To simplify the discussion, we use *prefix* to denote a contiguous sequence of tokens at the beginning of an input sequence. Accordingly, a *chunk-prefix* refers to an ordered sequence of chunks that appear at the beginning of an input. We use the suffix *-KV* to denote the corresponding KV cache. For example, *chunk-KV* denotes the KV cache associated with a chunk.

3 Motivation

3.1 Mismatch Between Chunk Redundancy and Prefix Caching

We observe a fundamental inefficiency in RAG workloads: although retrieved chunks overlap substantially across requests, prefix caching captures only a small fraction of potential chunk-KV reuse due to positional misalignment. We

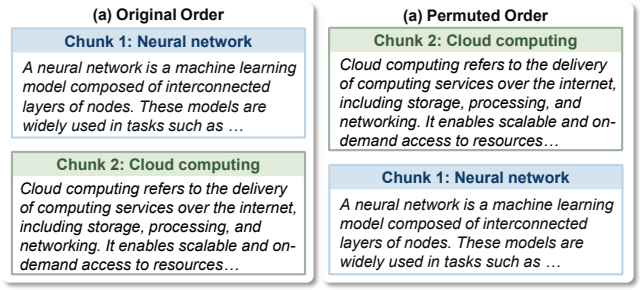

Figure 4. Chunk examples.

quantify this mismatch and show that it arises from a structural limitation of prefix-based reuse. We further demonstrate that it can be addressed by reordering chunks to improve prefix alignment, without sacrificing generation quality, and extend the analysis to multi-turn settings where additional redundancy arises.

We analyze chunk overlap patterns across N inference requests, denoted by $\{R_i\}_{i=1}^{N}$. Note that KV cache reuse requires not only shared chunks but also that they appear at the same prefix positions. We define two metrics: *prefix*, which captures reuse under prefix caching, and *total*, which captures overlap independent of position. Both metrics compare each request with prior requests, since reuse is only possible from earlier computations.

Formally, for each request R_i, we identify the best-matching prior request among $\{R_j \mid j < i\}$. The *prefix* metric computes the maximum longest common prefix (LCP) length with any prior request:

$$\text{Prefix} = \frac{1}{N-1}\sum_{i=2}^{N}\frac{\max_{j<i}\text{LCP}(R_i, R_j)}{k}.$$

The *total* metric counts the maximum number of shared chunks regardless of position:

$$\text{Total} = \frac{1}{N-1}\sum_{i=2}^{N}\frac{\max_{j<i}|R_i \cap R_j|}{k}.$$

Both metrics are averaged over all requests and normalized by the number of retrieved chunks k.

Consider two requests as a concrete example:

$$R_1 = [C_1, C_4, C_5, C_6, C_7], \quad R_2 = [C_1, C_2, C_3, C_4, C_5].$$

Their longest common prefix has length 1 (only C_1), so prefix-based reuse is limited to a single chunk. Yet the two requests share three chunks in total (C_1, C_4, C_5), representing a far larger reuse opportunity that prefix caching cannot directly exploit. As shown in Figure 3, this gap is substantial: on average, only **11%** of chunks are prefix-aligned, while **43%** are shared across requests. This indicates that most reuse opportunities are not captured by existing prefix caching due to positional misalignment.

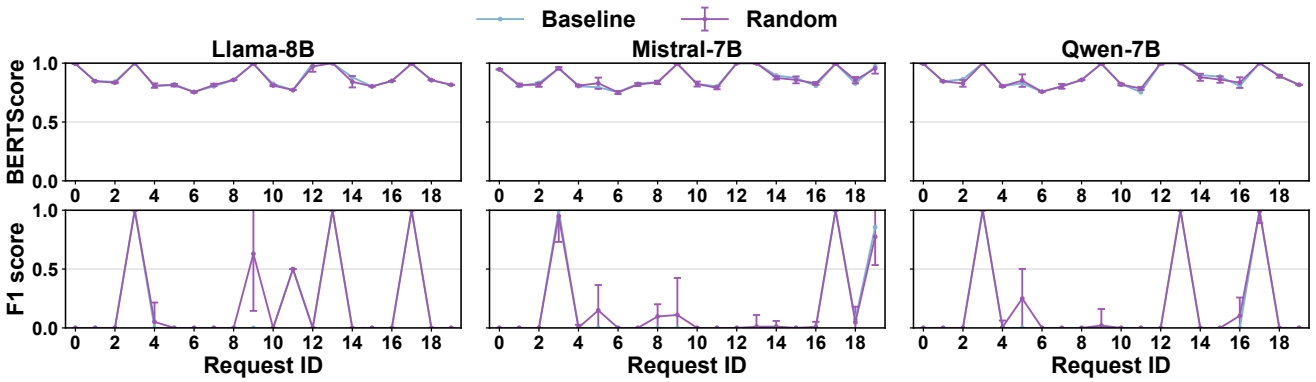

Figure 5. Comparing response quality under baseline chunk order and 100 random permutations (mean ± standard deviation), showing negligible impact of chunk reordering.

> ***Observation 1:*** Chunk redundancy across requests is high, but prefix-aligned chunk-KV reuse is limited due to positional misalignment.

Observation 1 raises a natural question: *can we reorder chunks to improve prefix alignment without affecting generation quality?* We find that this is possible due to *semantic permutation invariance*: reordering semantically self-contained chunks has a negligible impact on generation quality. The intuition is straightforward. As illustrated in Figure 4, each chunk corresponds to a semantically self-contained passage, typically spanning 128 to 512 tokens or more [25, 47]. At this granularity, each chunk encapsulates a complete and coherent piece of information. When multiple such chunks are concatenated to form the augmented prompt, the LLM processes each chunk as an independent unit of evidence, and the combined meaning remains largely unchanged regardless of their ordering. This is in contrast to arbitrary token sequences, where word order is essential for grammatical structure and semantic coherence.

To validate this, we measure the impact of chunk permutation on generation quality across multiple models. Figure 5 compares outputs generated using the original chunk order (Baseline) against 100 independent random chunk order permutations for 20 requests from HotpotQA. The x-axis represents the request index (i.e., 20 independent query instances), while each subplot corresponds to a different LLM (LLaMA-8B, Mistral-7B, and Qwen-7B). For each request, the random curve shows the mean over 100 runs, with error bars indicating the standard deviation. All generated outputs, including those from the baseline (i.e., original chunk order) and the random permutations, are evaluated against ground-truth answers from the dataset. We refer to Section 5 for full experimental details. We use BERTScore [50], which measures semantic similarity via contextualized embeddings, and F1 score [45], which measures token-level overlap. For both metrics, higher values indicate better quality, and a score of 1 corresponds to an exact match with the ground truth. In practice, BERTScore values above 0.8 indicate a strong match where the essential content and context are preserved. Since we aim to validate that chunk permutation does not affect generation quality, similar mean scores and low variance across permutations (as indicated by the error bars) support this claim. As shown in Figure 5, the BERTScore of outputs generated under random chunk permutations closely matches that of the baseline, indicating that chunk reordering does not affect the semantic content of the responses. The F1 score exhibits slightly larger variation for some requests, which is expected, as it is sensitive to exact wording differences rather than meaning. In some cases, random permutations achieve slightly higher scores than the baseline; this reflects natural variability in generation and differences in phrasing, rather than improvement over the original order. Overall, both metrics remain consistent across all orderings, confirming that chunk permutation has a negligible impact on generation quality.

> ***Observation 2:*** Chunk order has negligible impact on generation quality when chunks are semantically self-contained, as is typically the case in standard RAG settings (e.g., 128–512 tokens per chunk).

Observations 1 and 2 together reveal that the limitation of prefix caching is not inherent to the workload, but to the input structure. By reordering chunks to align shared content into common prefixes, it is possible to significantly increase KV reuse without affecting generation quality. Returning to our earlier example:

$$\text{Request}_1 = [C_1, C_4, C_5, C_6, C_7], \text{Request}_2 = [C_1, C_2, C_3, C_4, C_5].$$

Without chunk permutation, only C_1 is prefix-aligned, so KV reuse is limited to one chunk. Since chunk permutation does not affect generation quality, we can permute the chunk

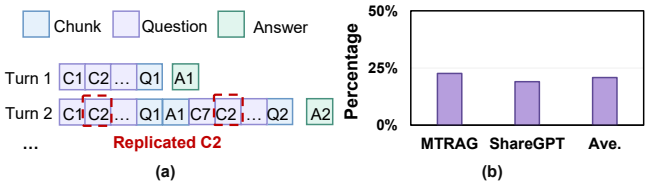

Figure 6. Overview of multi-turn RAG.

order of Request_2 as:

$$\text{Request}_2' = [C_1, C_4, C_5, C_2, C_3],$$

aligning chunks C_1, C_4, and C_5 at the shared prefix. This triples the reusable chunk-KV, directly reducing redundant computation without degrading output quality.

3.2 Chunk Redundancy in Multi-Turn Conversations

We extend our analysis to multi-turn conversation RAG, where a *conversation* consists of multiple *turns* (each turn is a query–response pair), and each turn independently performs retrieval while accumulating conversation history. As illustrated in Figure 6(a), the prompt at each turn concatenates the newly retrieved chunks with the full conversation history. A key characteristic of multi-turn conversations is that user queries within a conversation are typically topically related, leading to repeated retrieval of similar chunks across turns. Figure 6(b) reports this effect on two representative multi-turn RAG datasets (MTRAG and ShareGPT), which are the multi-turn datasets used in our evaluation (see Section 5 for details). We observe that the same chunks are frequently retrieved across turns within a conversation: on average, **21%** of chunks retrieved in a given turn have already appeared in earlier turns of the same conversation. These chunks are computed during prefill despite their KV cache already being available in previous turns, resulting in unnecessary memory traffic and computation.

We evaluate whether removing these duplicated chunks within a conversation degrades generation quality. Consider a two-turn example:

$$\text{Request} = [C_1, C_2, C_3, C_4, C_5, \text{Turn}_1\_\text{Query}, \text{Turn}_1\_\text{Response}, C_1, C_3, C_6, C_7, C_8, \text{Turn}_2\_\text{Query}].$$

Removing duplicated chunks (C_1, C_3) from the second turn:

$$\text{Request}_{\text{dedup}} = [C_1, C_2, C_3, C_4, C_5, \text{Turn}_1\_\text{Query}, \text{Turn}_1\_\text{Response}, C_6, C_7, C_8, \text{Turn}_2\_\text{Query}].$$

As shown in Figure 7, deduplication introduces negligible changes in both BERTScore and F1 across all evaluated requests compared to the baseline. The x-axis represents the request index across the evaluated conversations. This is because the relevant information from duplicated chunks has already been incorporated into the KV cache from earlier turns, which remains accessible in the conversation context. To further quantify this effect, we conduct a paired

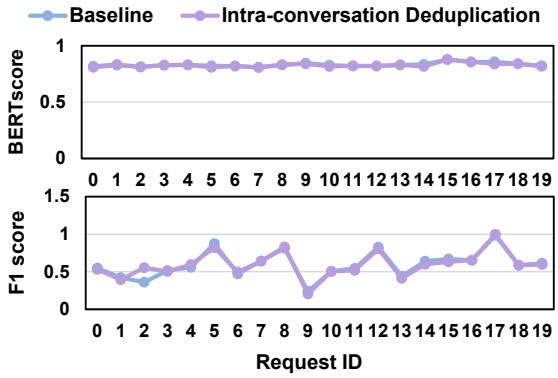

Figure 7. Generation quality comparison.

t-test between the baseline and the deduplicated results for BERTScore across all requests. The test yields $p = 0.47$, indicating no statistically significant difference ($p > 0.05$) [6].

> ***Observation 3:*** In multi-turn RAG, a significant fraction of retrieved chunks are repeated across turns, leading to redundant KV computation that can be eliminated without affecting generation quality.

Takeaway. These observations reveal two complementary sources of inefficiency in KV cache management for RAG systems. First, prefix caching captures only a small fraction of reuse due to positional misalignment, despite high cross-request chunk redundancy (11% vs. 43%). Second, multi-turn interactions introduce additional redundancy through repeated retrieval of the same chunks, where 21% of chunks in a given turn have already appeared in earlier turns of the same conversation. Together, these findings suggest that KV reuse can be significantly improved by restructuring inputs at the chunk level, motivating a design that aligns shared chunks across requests and eliminates redundant computation within conversations.

4 CRAFT Design

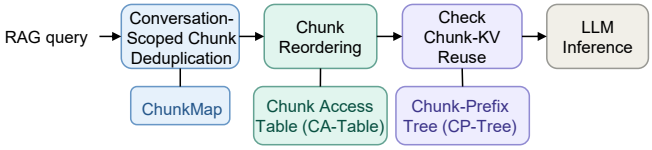

Figure 8. CRAFT overview.

We present CRAFT, a framework for RAG LLM serving systems that improves KV cache reuse by aligning shared content into a consistent order across requests. As shown in Section 3, RAG exhibits high cross-request chunk redundancy, but prefix caching fails to exploit this redundancy due to chunk positional misalignment. CRAFT addresses this limitation by introducing a chunk-level abstraction that

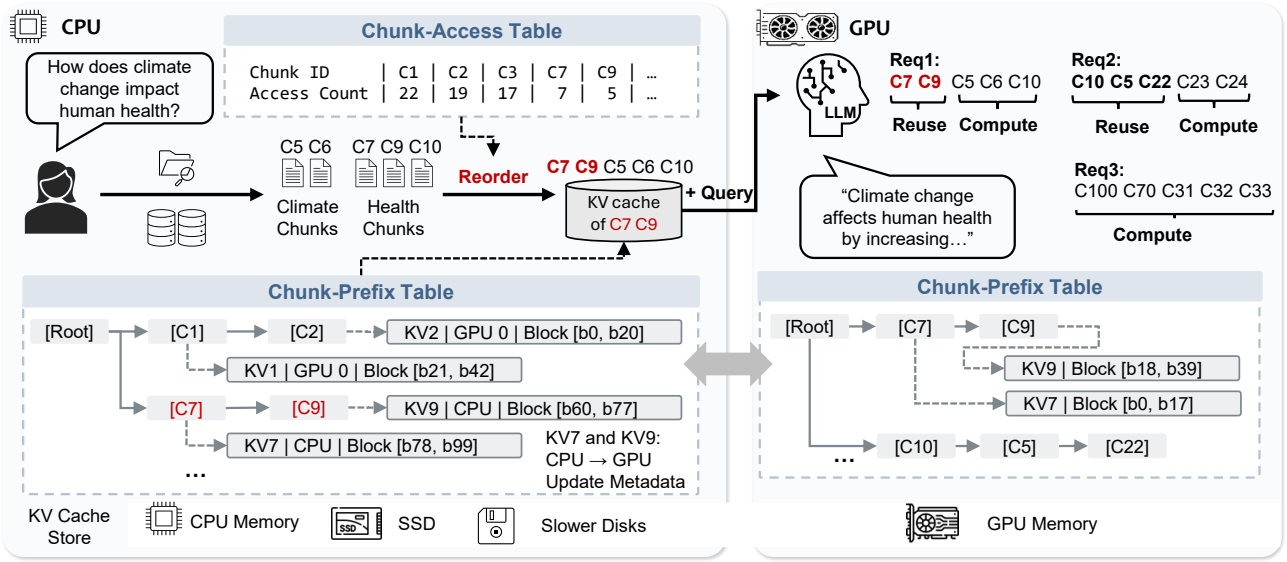

Figure 9. System overview of CRAFT for chunk-level KV cache reuse in RAG serving.

decouples KV reuse from strict token ordering. It reorganizes retrieved chunks to increase prefix alignment and indexes reusable chunk-prefix KV through a dedicated data structure. The system consists of two key components: (1) a *Chunk-Access Table (CA-Table)* that captures cross-request access patterns, and (2) a *Chunk-Prefix Tree (CP-Tree)* that indexes reusable chunk-prefix KV. Together, they enable a unified request processing pipeline consisting of chunk reordering, prefix lookup, KV reuse, and state update. CRAFT further extends this design to multi-turn settings by eliminating redundant chunks within conversations.

4.1 Chunk-Level KV Abstraction

CRAFT introduces a chunk-level abstraction for KV cache management. Each retrieved passage is treated as an atomic chunk, identified by a globally unique key (e.g., document ID and offset). KV reuse is performed at the granularity of chunk sequences, while token-level KV management is retained for the remaining input. This abstraction enables efficient tracking, matching, and reuse of chunk-KV across requests.

Chunk-Access Table. We maintain a Chunk-Access Table (CA-Table) over recent requests, where each chunk is associated with an access count, as illustrated in Figure 9. After the CPU completes retrieval for a request, the table is updated by incrementing the access count of each retrieved chunk. To accurately capture recent access patterns, we employ a sliding-window mechanism. This ensures that the table reflects the characteristics of recent requests rather than being dominated by outdated access history. In particular, chunks that were frequently accessed in the past but are no longer active will gradually lose influence as they fall outside the window. Without this mechanism, stale access counts could bias the reordering policy and lead to suboptimal reuse decisions. We evaluate the sensitivity of the sliding-window size in Section 6.2.4.

Chunk-Prefix Tree. We maintain a Chunk-Prefix Tree (CP-Tree) to track reusable chunk-prefixes, as illustrated in Figure 9. The CP-Tree is a radix-tree-like structure, where each node represents a chunk, and each path from the root to a node corresponds to a chunk-prefix. Each node stores metadata associated with the prefix, including references to KV cache entries, their storage location (e.g., GPU or CPU), and the corresponding token span. Note that CRAFT follows the memory management scheme of vLLM [28], where KV cache is represented as contiguous blocks in the logical space but stored in non-contiguous regions in physical memory. Specifically, each chunk-KV occupies a contiguous range of logical KV blocks. For example, as shown in the figure, chunk-KV-9 spans a contiguous sequence of logical blocks (e.g., b_{90} to b_{102}). Then, with the vLLM engine, these logical blocks are mapped to non-contiguous physical memory locations, allowing efficient KV reuse.

In our hierarchical design, the CPU maintains a global CP-Tree that stores complete metadata for all reusable chunk-prefixes, including KV locations and token spans, and is responsible for all update operations such as prefix promotion, extension, and eviction handling. Each GPU maintains a compact local CP-Tree that maintains only the subset of chunk-prefixes whose KV cache resides on that GPU. This is optimized for fast lookup, enabling efficient prefix matching without CPU involvement. When a chunk-KV is evicted or migrated, the CPU updates the global CP-Tree and asynchronously propagates updates to the affected GPU-local

CP-Tree. This design maintains consistency while avoiding expensive synchronization overhead.

4.2 Chunk-Prefix Management

We now describe how requests are processed using the Chunk-Access Table (CA-Table) and Chunk-Prefix Tree (CP-Tree).

Chunk Permutation for Prefix Alignment. In RAG workloads, the same retrieved chunks often appear in different positions across requests, which prevents them from forming identical token prefixes and limits KV cache reuse under standard prefix caching. Importantly, our goal is not to modify or bypass prefix caching, but to increase the likelihood that its reuse condition, identical prefixes, is satisfied. Based on Observation 2, permuting chunk-level content does not affect LLM generation quality when chunks are semantically self-contained. This allows us to restructure the input sequence so that shared chunks appear in consistent prefix positions across requests. By doing so, we preserve the validity of prefix caching while expanding its applicability. The key challenge is to determine an effective and consistent ordering. An inappropriate ordering may fail to improve prefix alignment and can even reduce reuse. Moreover, the permutation strategy should be consistent across requests, so that similar chunk sets are arranged in a similar prefix order, which also simplifies system design. Therefore, the permutation strategy should (i) capture cross-request chunk access patterns, (ii) promote frequently accessed chunks to appear early in the sequence, and (iii) remain lightweight for online computation. We leverage the information maintained in the CA-Table to permute retrieved chunks accordingly:

$$\mathbf{Chunks}_{\text{reordered}} = \text{sort}(\mathbf{C}, -\text{freq}(c)).$$

This places frequently accessed chunks earlier in the sequence, increasing the likelihood that shared chunks across different requests align in the prefix. Consider two requests with retrieved chunks $\{C_1, C_2, C_3\}$ and $\{C_2, C_3, C_4\}$, without reordering, they do not share a common prefix. Suppose the CA-Table assigns higher access frequency to C_2 and C_3. After reordering, both sequences become $[C_2, C_3, etc.]$, resulting in a shared prefix of length 2, enabling the corresponding chunk-KV to be directly reused via prefix caching. This strategy is effective because chunk access frequency reflects cross-request reuse patterns. Frequently accessed chunks are more likely to co-occur across requests, and consistently placing them at the front increases prefix alignment. As a result, we do not alter the prefix caching mechanism itself; rather, we reshape inputs so that more requests satisfy its reuse condition, thereby enabling more effective chunk-KV reuse.

Chunk-Prefix Lookup in CP-Tree. Given the reordered chunk sequence, we traverse the CP-Tree to find the longest matching chunk-prefix. The matching process starts from the first chunk and proceeds sequentially along the tree. Each path in the CP-Tree corresponds to a previously observed chunk-prefix with associated KV metadata. The traversal returns the maximum prefix length k such that the first k chunks match an existing path. For example, consider a reordered chunk sequence $[C_1, C_2, C_3]$. Starting from the root of the CP-Tree, we first check whether C_1 exists as a child node at the first level. If a match is found, we continue matching the next chunk C_2 along the same path. If both C_1 and C_2 are present but C_3 does not exist as a child of C_2, the longest matched prefix is $[C_1, C_2]$, and the traversal stops. This chunk-prefix is then used for chunk-KV reuse. If C_1 is not found at the first level, no prefix can be reused. If C_3 later appears frequently after $[C_1, C_2]$, it will be inserted into the CP-Tree as a child of C_2, forming a longer chunk-prefix $[C_1, C_2, C_3]$. This extended prefix can then be reused by future requests. Note that, based on the tree structure, CRAFT also supports *prefix slicing*: if a longer chunk-prefix is cached, shorter prefixes can be derived without additional storage by retrieving the corresponding nodes' KV metadata. For example, if the chunk-prefix $[C_1, C_2, C_3]$ is stored in the CP-Tree, then its sub-prefixes $[C_1]$ and $[C_1, C_2]$ can also be reused. Consider a request with reordered chunks $[C_1, C_{10}, C_{11}]$. Even if C_{10} and C_{11} are not frequently accessed and do not appear in the CP-Tree, the system can still reuse the KV cache for C_1 by retrieving it from the CP-Tree. The details of CP-Tree insertion and updates are described in a later paragraph (i.e., Table Update and Management).

KV Reuse and Suffix Computation. If a chunk-prefix of length k is found, we reuse the corresponding KV cache and only compute the remaining suffix. Each CP-Tree node stores metadata describing the location of the KV cache associated with the chunk. Thus, once a reusable chunk-prefix is identified, the system can directly locate the corresponding KV cache. If the chunk-prefix-KV resides in CPU memory, it is transferred to the target GPU together with the request. The GPU then reuses this KV cache and performs LLM inference only on the remaining suffix. If the chunk-prefix-KV is already resident on the target GPU, the request is sent to that GPU, and the cached chunk-prefix-KV is directly reused. One may notice that, in multi-GPU settings, this introduces a scheduling challenge: ideally, requests should be routed to GPUs that already hold the required chunk-prefix-KV to maximize reuse and minimize data transfer overhead. However, this may lead to severe load imbalance. For example, if many requests share a popular chunk-prefix whose KV cache resides on a single GPU, routing all such requests to that GPU can create a hotspot, while other GPUs remain underutilized. This imbalance can increase queueing delay and reduce overall system throughput. Designing an efficient KV-aware scheduling policy is non-trivial and beyond the scope of this work. To simplify the evaluation, for multi-GPU scenarios, we assume a simple baseline policy: requests are assigned in a round-robin manner, and chunk-prefix-KV

reuse is applied only if the assigned GPU already contains the corresponding KV cache.

Table Update and Maintenance. After processing each request, we update both the CA-Table and the CP-Tree. For the CA-Table, we increment the access count of each retrieved chunk by one. To reflect recent access patterns, we maintain the table using a sliding window and remove chunks that have not appeared in recent requests. We use a promotion threshold (its sensitivity is evaluated in Section 6.2.5) to determine when and which chunk-prefixes should be inserted into the CP-Tree. Specifically, if the chunks in a prefix all reach the threshold, we promote this to a reusable chunk-prefix and insert it into the CP-Tree. If a chunk-prefix already exists in the CP-Tree and is reused by a request, we further check whether the next chunk following this chunk-prefix reaches the threshold. If so, we extend the existing chunk-prefix by inserting the new chunk as a child node. The corresponding KV metadata (e.g., location and token span) is updated accordingly. When a KV cache is evicted or moved, the corresponding metadata entry in the CP-Tree is also updated or removed to maintain consistency.

Example. Assume the CA-Table contains two chunks C_1 and C_2 with access counts of 1, and the CP-Tree is empty. Let the promotion threshold be 2. Consider a request $[C_1, C_2, C_5]$. Since no prefix is found in the CP-Tree, the request is directly sent to GPU_1 for execution, and the CA-Table is updated to $C_1 : 2$, $C_2 : 2$, and $C_5 : 1$. As C_1 and C_2 both reach the threshold and appear consecutively in this request, we insert the prefix $[C_1, C_2]$ into the CP-Tree, where C_2 is a child of C_1, and record that their KV cache resides on GPU_1. Next, consider a request $[C_1, C_2, C_6]$. The chunk-prefix $[C_1, C_2]$ is found and reused, so only the KV cache for C_6 will be computed. The CA-Table is updated to $C_1 : 3$, $C_2 : 3$, and $C_6 : 1$, and the CP-Tree remains unchanged since C_6 does not reach the threshold. If a request with the same chunks appears again, $[C_1, C_2, C_6]$ is reused once more, and now C_6 reaches the threshold. We then extend the CP-Tree by inserting C_6 as a child of C_2, forming the chunk-prefix $[C_1, C_2, C_6]$ with updated KV metadata. Finally, suppose the KV cache for $[C_1, C_2]$ is evicted from GPU_1 to CPU memory. The corresponding CP-Tree entries are updated to reflect the new location. If the KV cache is removed entirely, the associated nodes or metadata entries are also removed.

4.3 Multi-Turn Conversation Optimization

In multi-turn RAG, chunks are often retrieved repeatedly across turns within the same conversation (Observation 3). This redundancy leads to repeated KV computation for the same chunks, even though their content has already been processed in earlier turns. Moreover, Observation 3 shows that removing these duplicated chunks within the same conversation does not affect generation quality. Therefore, to eliminate this redundancy, we propose a conversation-scoped chunk deduplication as shown in Figure 10. We maintain a per-conversation chunk set:

$$\text{ChunkMap} : \text{conv\_id} \rightarrow \{\text{chunk\_id}\}.$$

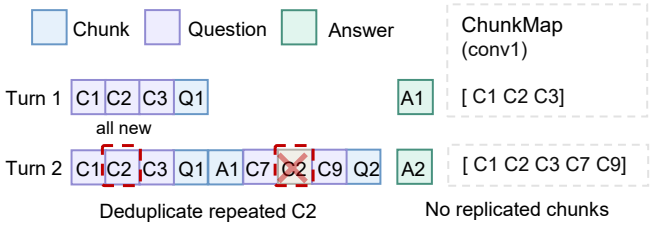

Figure 10. Conversation-scoped chunk deduplication.

At each turn, after retrieval, we check whether a chunk has already appeared in previous turns of the same conversation. If so, we remove this chunk from the current request to avoid redundant KV computation and data transfer. Only newly retrieved chunks are processed. For example, consider a conversation where the first turn retrieves $[C_1, C_2, C_3]$, and the second turn retrieves $[C_2, C_3, C_4]$. Without this optimization, C_2 and C_3 would be recomputed in the second turn. With the per-conversation chunk set, we detect that C_2 and C_3 have already been processed, and only compute C_4, reducing both latency and resource usage.

One may be concerned that, as the conversation grows and exceeds the model's context length, earlier turns with chunks will be truncated, which may affect the generation quality of later responses. However, this does not impact generation quality in our setting. The information from earlier chunks has already been incorporated into the generated responses of previous turns. Therefore, removing earlier turns, even if their chunks are reused in later turns, does not lead to information loss for subsequent responses. For example, consider a conversation where turn 1 retrieves chunks $[C_1, C_2, C_3]$, and turn 5 retrieves $[C_2, C_3, C_5]$. Under our design, duplicated chunks $[C_2, C_3]$ are removed, and turn 5 processes only $[C_5]$. At a later turn (e.g., turn 9), the input includes a sequence: $C_1, C_2, C_3, \text{Turn}_1\_Query, \text{Turn}_1\_Response, \ldots, C_5, \text{Turn}_5\_Query, \text{Turn}_5\_Response, \ldots, C_7, C_8, C_9, \text{Turn}_9\_Query$. When the input exceeds the model's context length, the earliest portion (e.g., $C_1, C_2, C_3, Turn_1\_Query, Turn_1\_Response$) may be truncated. However, when generating $\text{Turn}_5\_Response$, the model has already consumed the chunks $[C_2, C_3, C_5]$, so the relevant information is incorporated into $\text{Turn}_5\_Response$. As a result, even if earlier chunks are later truncated, their information is preserved in intermediate responses such as $\text{Turn}_5\_Response$, which remain in the context. Therefore, the generation of $\text{Turn}_9\_Response$ is not affected.

Note that this multi-turn optimization is orthogonal to cross-request chunk-prefix reuse. While the CP-Tree enables reuse across different requests, the per-conversation chunk set eliminates redundancy within the same conversation, further improving overall efficiency.

5 Methodology

Hardware and Models. We evaluate on a CPU–GPU hybrid server, where vector search is executed on an Intel Xeon w9-3595X CPU, and LLM generation is performed on an NVIDIA A6000 GPU with 48 GB of memory. We consider three open-source LLMs of varying scales: LLaMA-3-8B [14], Mistral-7B [24], and Qwen2.5-7B [4, 19]. Our system is implemented on top of vLLM [28].

Corpus and Index. We use a Wikipedia passage corpus [7, 21] as the primary retrieval source, containing ~38M passages with knowledge up to 2022. Each passage corresponds to a paragraph-level segment and is paired with its originating article title. Following prior dense retrieval pipelines, we construct the input by concatenating the title and passage text. We encode each passage using Contriever [9] to obtain dense embeddings. The embeddings are indexed using FAISS [10] with an IVF4096 index and n_{nprobe} set to 128. At query time, we retrieve the top-k = 5 passages.

Datasets. We evaluate on four open-domain Question-Answer (QA) benchmarks, and for multi-turn conversation, we use ShareGPT and MTRAG:

- **HotpotQA** [46]: A QA dataset requiring reasoning over multiple supporting documents.
- **2WikiMultiHopQA** [16]: A QA dataset designed to test cross-document reasoning over Wikipedia.
- **SQuAD** [36]: A QA reading comprehension benchmark based on Wikipedia passages.
- **TriviaQA** [26]: A large-scale factoid QA dataset with evidence from Wikipedia and the web.
- **ShareGPT** [39]: A dataset of real-world multi-turn conversations between users and AI assistants, designed to evaluate context retention and coherence across dialogue turns. The dataset contains over 90,000 requests, we sample 10,000 requests for evaluation.
- **MTRAG** [20, 27]: A multi-turn retrieval-augmented generation benchmark that assesses a system's ability to retrieve and integrate relevant information across successive conversational turns.

Baselines. We compare against two baselines: (1) w/o prefix caching concatenates the retrieved chunks and feeds them directly to the LLM, recomputing all KV cache from scratch during prefill; (2) w/ prefix caching adopts prefix-based KV reuse for shared prompt prefixes across requests and represents the widely deployed reuse mechanism in production LLM serving systems (e.g., vLLM [28]).

We use prefix caching as the primary baseline because CRAFT improves the effectiveness of prefix-based KV reuse by restructuring inputs to increase prefix alignment. This allows us to isolate the impact of input-level optimization on reuse. Prior systems such as RAGCache [25] incorporate multiple orthogonal optimizations, including hierarchical KV caching, request scheduling, and speculative pipelining. In contrast, our work targets the input structure itself and is complementary to such system-level techniques; CRAFT can be applied on top of RAGCache to further improve reuse by increasing prefix alignment across requests.

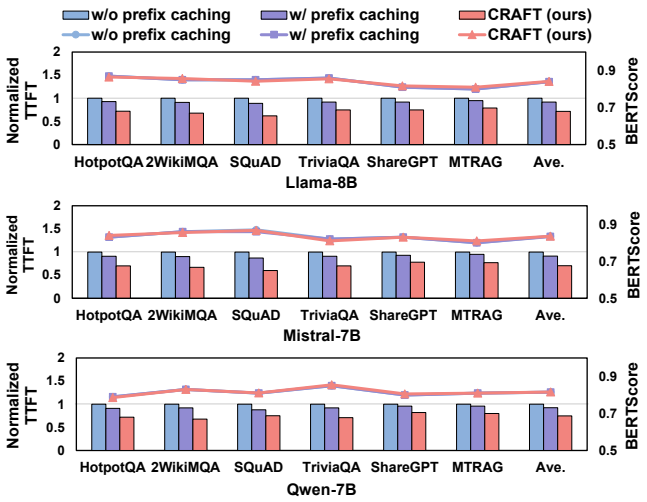

Figure 11. CRAFT reduces TTFT by 1.2–1.6× compared to full w/o prefix caching with no degradation in generation quality across six datasets and three models.

For fair comparison, all configurations use the same model parameters and batch size. CRAFT uses a sliding window size of 1,000 requests for the CA-Table and a default promotion threshold of 2. Sensitivity analyses for these two parameters are presented in Section 6.2.4 and Section 6.2.5.

6 Evaluation

6.1 Overall Performance

As shown in Figure 11, we evaluate average quality (BERTScore) and TTFT against baseline methods under a consistent setup across different datasets and models (Section 5), where each request retrieves the top-5 chunks. All results are normalized to w/o prefix caching. One can observe that compared to the w/o prefix caching and w/ prefix caching, CRAFT significantly reduces the TTFT by 1.2–1.6× with no degradation in generation quality. This improvement comes from reducing redundant chunk-KV computation through CRAFT. As shown in Figure 12, CRAFT eliminates 19%–38% of KV cache computation during the prefill stage on average. In this figure, the *general* bars represent the reduction in chunk-KV computation achieved by the main permutation mechanism, while the *conversation-scoped* bars correspond to the reduction from conversation-scoped chunk deduplication. Note that, for multi-turn conversation datasets, the contribution from the general mechanism is relatively small. This is due to the structure of multi-turn conversation inputs. In the first turn, retrieved chunks can be freely reordered to align with previously cached prefixes, enabling effective chunk-KV reuse. However, once the first turn is fixed, its chunks

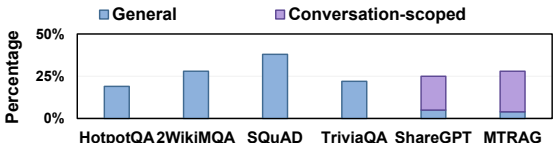

Figure 12. Reduction in KV cache computation.

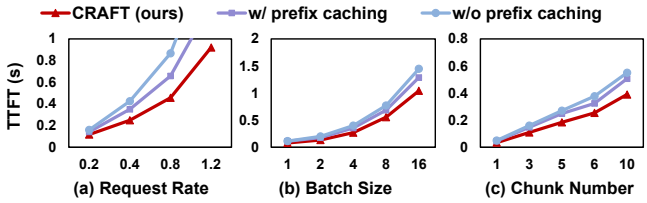

Figure 13. Performance under varying configurations.

and corresponding KV cache become part of the history for subsequent turns. In later turns, reordering newly retrieved chunks to the front in order to maximize reuse would require recomputing the KV cache of the existing conversation history, which is already available and should ideally be reused as-is. For example, after processing a first turn with prefix $C_1, C_2, C_3, Turn_1\_Q, Turn_1\_A$, the second turn may retrieve $C_4, C_5, C_6, Turn_2\_Q$. Even a chunk-prefix KV exists for C_4, C_5, C_6, moving them ahead of the existing prefix (i.e., forming $C_4, C_5, C_6, C_1, C_2, C_3, \dots$) would require recomputing the KV cache for C_1, C_2, C_3, which is unnecessary and counterproductive.

Overhead. CRAFT introduces modest overhead from CPU-side metadata management. The CA-Table maintains a sliding window over recent requests, and its size is bounded by the number of unique chunks within the window (e.g., on the order of a few thousand entries in our setting), where each entry stores only a chunk identifier and a small counter, resulting in a compact footprint of less than 1 MB in our evaluations. Chunk reordering operates on a small set of retrieved chunks (typically $k \leq 10$), resulting in negligible sorting cost. The CP-Tree stores only promoted chunk-prefixes, which effectively limits its growth; each node contains lightweight metadata (e.g., chunk id, KV location, and token span), making its memory footprint small compared to the KV cache (e.g., tens of MB in our evaluations). Both CA-Table updates and CP-Tree lookups involve simple hash or pointer-based operations over a short sequence, leading to low and stable access latency. Overall, these overheads remain small relative to the performance gains from chunk-KV reuse.

6.2 Sensitivity Analyses

6.2.1 Request rate. As shown in Figure 13(a), we compare CRAFT with w/o prefix caching and w/ prefix caching under different request rates. As the request rate increases, TTFT rises across all three methods due to higher queuing contention. We observe that CRAFT consistently achieves lower TTFT while maintaining higher throughput.

6.2.2 Batch size. Figure 13(b) shows the TTFT under varying batch sizes. We observe that CRAFT consistently outperforms both w/o prefix caching and w/ prefix caching across different batch sizes. Note that, as the batch size increases, the decoding phase benefits more from parallelism and improves faster, while the prefill phase becomes the dominant component of the latency. As a result, optimizations that reduce prefill computation become increasingly important at larger batch sizes. Since CRAFT reduces redundant prefill stage computation through chunk-KV reuse, its advantage becomes more pronounced as the batch size grows.

6.2.3 Chunk number. Figure 13(c) presents the impact of varying the number of retrieved chunks. As the number of chunks increases, the TTFT of all methods increases with input length. However, CRAFT maintains a consistently lower TTFT than both baselines across all batch sizes.

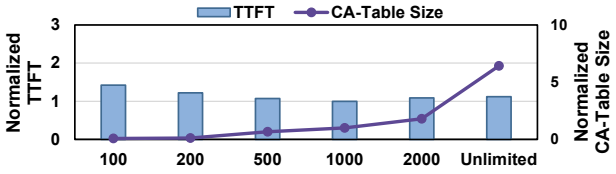

Figure 14. Sliding window size for CA-Table in CRAFT.

6.2.4 Sliding window size for CA-Table. Figure 14 shows the impact of different sliding window sizes for the CA-Table. As the window size increases, the CA-Table records chunk access counts over more past requests, which increases the table size. We observe that a moderate window size (i.e., 1,000) achieves a good balance between performance and table size. When the window size is too small, the access counts are computed from only a few recent requests, so they fluctuate frequently and do not reliably reflect which chunks are repeatedly accessed, leading to suboptimal chunk ordering. When the window size becomes very large (or unlimited), the table includes many chunks that were frequently accessed in the past but no longer appear in recent requests. These outdated counts bias the ordering decision and reduce prefix alignment, resulting in higher TTFT and increased metadata overhead. This shows that limiting the window size is necessary to keep the access counts both representative of current workloads and effective for chunk reordering, while also controlling the table size.

6.2.5 Promotion threshold for CP-Tree. Figure 15 shows the impact of different promotion thresholds for CP-Tree construction. We observe that a smaller threshold (e.g., 2) consistently achieves the best performance across datasets. With a small threshold, chunk prefixes are promoted earlier,

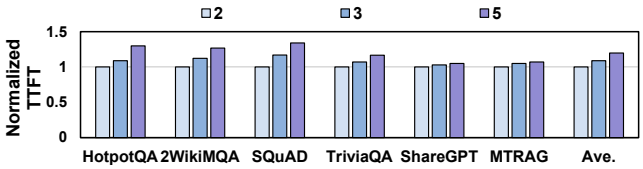

Figure 15. Promotion threshold for CP-Tree in CRAFT.

allowing the system to capture reuse opportunities quickly and increase the likelihood of prefix matching. As the threshold increases (e.g., 3 or 5), prefix promotion becomes more conservative, delaying insertion into the CP-Tree and reducing reuse opportunities, which leads to higher TTFT. These results indicate that aggressively promoting chunk-prefixes is beneficial for maximizing chunk-KV reuse.

6.3 Synergy with CacheBlend

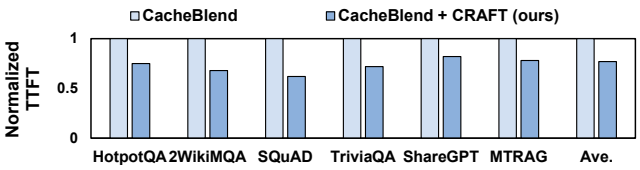

Figure 16. CacheBlend and CacheBlend with CRAFT.

Figure 16 compares the TTFT of CacheBlend [47] with the combined approach (CacheBlend + CRAFT). We observe that integrating CRAFT with CacheBlend further reduces TTFT by 1.29× beyond using CacheBlend alone. This improvement arises because the two methods are orthogonal and optimize different aspects of chunk-level KV reuse. CacheBlend reuses chunk KV by selectively recomputing a small fraction of the token KV cache within each chunk. In contrast, CRAFT increases the likelihood that shared chunks are aligned in the prefix through reordering, enabling their chunk-KV to be directly reused without such recomputation. As a result, their benefits complement each other naturally, combining them allows more KV cache to be reused directly, leading to further TTFT improvement.

7 Related Work

KV cache optimizations. The KV cache stores key and value tensors computed during the prefill phase for reuse in subsequent autoregressive decoding steps, avoiding redundant computation. However, its computation overhead and memory footprint grow linearly with sequence length and batch size, creating a major bottleneck for long-context and high-throughput inference. A broad body of prior work optimized the KV cache efficiency across multiple granularities. At the token level, methods such as H2O [52] and SnapKV [30] evict less-attended tokens, while quantization approaches like KIVI [32] and KVQuant [17] compress activations to fewer bits. Other techniques exploit redundancy via low-rank decomposition and merging. At the system level, prefix caching enables cross-request KV reuse when inputs share identical prefixes [28, 54]. Moreover, the following two approaches specifically optimize KV reuse for RAG workloads. CacheBlend [47] enables partial KV reuse by recomputing a small fraction of tokens within a chunk when prefixes are not fully aligned. RAGCache [25] organizes chunk-level KV cache in a radix-tree structure across GPU and host memory, and incorporates scheduling and replacement policies to overlap retrieval and inference. However, both approaches assume a fixed chunk order and can only reuse KV when identical chunk sequences appear in the same prefix positions, leaving substantial reuse opportunities untapped when shared chunks appear in different positions across requests.

General-purpose LLM inference systems. A large body of work improves LLM serving efficiency through better batching, scheduling, and memory management [11, 15, 31, 41, 44, 48, 51, 53]. Orca [49] introduces iteration-level continuous batching to improve GPU utilization, while vLLM [28] improves memory efficiency with PagedAttention for KV cache management. Sarathi [2] proposes chunked prefill to balance TTFT and decode latency. Recent systems [12, 34, 35, 55] further explore memory and scheduling efficiency, including disaggregated LLM serving architectures that separate prefill and decode stages across heterogeneous resources, and fine-grained scheduling policies that improve GPU utilization under high concurrency. Production systems such as TensorRT-LLM and DeepSpeed-Inference [3] provide highly optimized kernels and parallelism support. CRAFT is complementary to these systems by enabling more effective KV cache reuse.

8 Conclusion

This paper proposes CRAFT, a framework for RAG LLM serving systems that improves KV cache reuse by reordering chunks into a consistent order across requests. We showed that existing prefix caching exposes only a small fraction of available reuse (11%) despite substantial cross-request chunk overlap (43%), due to positional misalignment. To address this limitation, CRAFT reorganizes inputs through frequency-guided chunk reordering, placing shared chunks in a consistent order across requests to improve prefix alignment and KV reuse. In addition, CRAFT eliminates redundant computation in multi-turn settings, where up to 21% of retrieved chunks repeat across turns, through conversation-scoped deduplication. Our results demonstrate that CRAFT improves TTFT by 1.2–1.6× while preserving response quality, showing that restructuring input layout is an effective and practical approach for improving KV cache utilization in high-throughput LLM serving systems.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] 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.

[3] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[4] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[5] 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.

[6] George Casella and Roger Berger. 2024. *Statistical inference*. Chapman and Hall/CRC.

[7] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. 2017. Reading Wikipedia to Answer Open-Domain Questions. *arXiv preprint arXiv:1704.00051* (2017).

[8] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of machine learning research* 24, 240 (2023), 1–113.

[9] Facebook AI Research. 2022. Contriever: Unsupervised Dense Retrieval. https://github.com/facebookresearch/contriever. GitHub repository.

[10] Facebook AI Research. 2023. FAISS: A Library for Efficient Similarity Search and Clustering of Dense Vectors. https://github.com/facebookresearch/faiss. GitHub repository.

[11] Ruibo Fan, Xiangrui Yu, Xinglin Pan, Zeyu Li, Weile Luo, Qiang Wang, Wei Wang, and Xiaowen Chu. 2026. ZipServ: Fast and memory-efficient LLM inference with hardware-aware lossless compression. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*. 2264–2280.

[12] Jingqi Feng, Yukai Huang, Rui Zhang, Sicheng Liang, Ming Yan, and Jie Wu. 2025. Windserve: Efficient phase-disaggregated llm serving with stream-based dynamic scheduling. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 1283–1295.

[13] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. {Cost-Efficient} large language model serving for multi-turn conversations with {CachedAttention}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 111–126.

[14] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[15] Mert Hidayetoglu, Aurick Qiao, Michael Wyatt, Jeff Rasley, Yuxiong He, and Samyam Rajbhandari. 2026. Shift parallelism: Low-latency, high-throughput llm inference for dynamic workloads. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*. 1749–1763.

[16] Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. 2020. Constructing a Multi-hop QA Dataset for Comprehensive Evaluation of Reasoning Steps. *arXiv preprint arXiv:2011.01060* (2020).

[17] C Hooper et al. 2024. Kvquant: Towards 10m-token context. *Advances in Neural Information Processing Systems (NeurIPS)* (2024).

[18] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems* 43, 2 (2025), 1–55.

[19] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

[20] IBM Research. 2025. MT-RAG Benchmark: A Multi-Turn Conversational Benchmark for Retrieval-Augmented Generation. https://github.com/IBM/mt-rag-benchmark. GitHub repository.

[21] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. 2023. Atlas: Few-shot learning with retrieval augmented language models. *Journal of Machine Learning Research* 24, 251 (2023), 1–43.

[22] Jinwoo Jeong and Jeongseob Ahn. 2025. Accelerating LLM Serving for Multi-turn Dialogues with Efficient Resource Management. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*. 1–15.

[23] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM computing surveys* 55, 12 (2023), 1–38.

[24] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

[25] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Shufan Liu, Xuanzhe Liu, and Xin Jin. 2025. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *ACM Transactions on Computer Systems* 44, 1 (2025), 1–27.

[26] 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 2017 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[27] Yannis Katsis, Sara Rosenthal, Kshitij Fadnis, Chulaka Gunasekara, Young-Suk Lee, Lucian Popa, Vraj Shah, Huaiyu Zhu, Danish Contractor, and Marina Danilevsky. 2025. MTRAG: A Multi-Turn Conversational Benchmark for Evaluating Retrieval-Augmented Generation Systems. *arXiv preprint arXiv:2501.03468* (2025).

[28] 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.

[29] 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.

[30] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. Snapkv: Llm knows what you are looking for before generation.

Advances in Neural Information Processing Systems 37 (2024), 22947–22970.

[31] Zejia Lin, Hongxin Xu, Guanyi Chen, Zhiguang Chen, Yutong Lu, and Xianwei Zhang. 2026. Bullet: Boosting gpu utilization for llm serving via dynamic spatial-temporal orchestration. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*. 290–306.

[32] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750* (2024).

[33] 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.

[34] 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.

[35] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Heyi Tang, Feng Ren, Teng Ma, Shangming Cai, Yineng Zhang, Mingxing Zhang, et al. 2024. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *ACM Transactions on Storage* (2024).

[36] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[37] 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.

[38] Shuo Ren, Can Xie, Pu Jian, Zhenjiang Ren, Chunlin Leng, and Jiajun Zhang. 2025. Towards scientific intelligence: A survey of llm-based scientific agents. *arXiv preprint arXiv:2503.24047* (2025).

[39] ShareGPT. 2023. ShareGPT: A Collection of Real-World Multi-Turn Conversations with ChatGPT. https://huggingface.co/datasets/RyokoAI/ShareGPT52K. Dataset.

[40] Sina Shool, Sara Adimi, Reza Saboori Amleshi, Ehsan Bitaraf, Reza Golpira, and Mahmood Tara. 2025. A systematic review of large language model (LLM) evaluations in clinical medicine. *BMC Medical Informatics and Decision Making* 25, 1 (2025), 117.

[41] Peng Tang, Jiacheng Liu, Xiaofeng Hou, Yifei Pu, Jing Wang, Pheng-Ann Heng, Chao Li, and Minyi Guo. 2026. MoE-APEX: An Efficient MoE Inference System with Adaptive Precision Expert Offloading. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*. 1185–1200.

[42] 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).

[43] Yiquan Wu, Siying Zhou, Yifei Liu, Weiming Lu, Xiaozhong Liu, Yating Zhang, Changlong Sun, Fei Wu, and Kun Kuang. 2023. Precedent-enhanced legal judgment prediction with llm and domain-model collaboration. In *Proceedings of the 2023 conference on empirical methods in natural language processing*. 12060–12075.

[44] Jiaming Xu, Jiayi Pan, Hanzhen Wang, Yongkang Zhou, Jiancai Ye, Yu Wang, and Guohao Dai. 2026. SpeContext: Enabling Efficient Long-context Reasoning with Speculative Context Sparsity in LLMs. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*. 1832–1847.

[45] Reda Yacouby and Dustin Axman. 2020. Probabilistic extension of precision, recall, and f1 score for more thorough evaluation of classification models. In *Proceedings of the first workshop on evaluation and comparison of NLP systems*. 79–91.

[46] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 conference on empirical methods in natural language processing*. 2369–2380.

[47] 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.

[48] Jinjun Yi, Zhixin Zhao, Yitao Hu, Ke Yan, Weiwei Sun, Hao Wang, Laiping Zhao, Yuhao Zhang, Wenxin Li, and Keqiu Li. 2026. PAT: Accelerating LLM Decoding via P refix-A ware A t tention with Resource Efficient Multi-Tile Kernel. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*. 1396–1412.

[49] 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.

[50] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675* (2019).

[51] Ziyi Zhang, Ziheng Jiang, Chengquan Jiang, Menghan Yu, Size Zheng, Haibin Lin, Xin Liu, and Henry Hoffmann. 2026. SwiftSpec: Disaggregated Speculative Decoding and Fused Kernels for Low-Latency LLM Inference. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*. 2197–2211.

[52] 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.

[53] Yilong Zhao, Shuo Yang, Kan Zhu, Lianmin Zheng, Baris Kasikci, Yifan Qiao, Yang Zhou, Jiarong Xing, and Ion Stoica. 2026. BlendServe: Optimizing Offline Inference with Resource-Aware Batching. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*. 255–273.

[54] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.

[55] 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.