EPIC: Efficient Position-Independent Caching for Serving Large Language Models

Junhao Hu[† 1] **Wenrui Huang**[2] **Weidong Wang**[2] **Haoyi Wang**[1] **Tiancheng Hu**[1] **Qin Zhang**[3] **Hao Feng**[3] **Xusheng Chen**[3] **Yizhou Shan**[3] **Tao Xie**[4]

Abstract

Large Language Models (LLMs) show great capabilities in a wide range of applications, but serving them efficiently becomes increasingly challenging as requests (prompts) become more complex. Context caching improves serving performance by reusing Key-Value (KV) vectors, the intermediate representations of tokens that are repeated across requests. However, existing context caching requires exact prefix matches across requests, limiting reuse cases in settings such as few-shot learning and retrieval-augmented generation, where immutable content (e.g., documents) remains unchanged across requests but is preceded by varying prefixes. Position-Independent Caching (PIC) addresses this issue by enabling modular reuse of the KV vectors regardless of prefixes. We formalize PIC and advance prior work by introducing EPIC, a serving system incorporating our new *LegoLink* algorithm, which mitigates the inappropriate "attention sink" effect at every document beginning, to maintain accuracy with minimal computation. Experiments show that EPIC achieves up to 8× improvements in Time-To-First-Token (TTFT) and 7× throughput gains over existing systems, with negligible or no accuracy loss.

1. Introduction

Large Language Models (LLMs) are now fundamental to various emerging applications such as question answering, chatbots, education, and medicine (Zhou et al., 2024). Users interact with LLMs by submitting requests, or prompts that consist of text-like tokens. As LLMs' capabilities continue to grow, their usage has shifted from simple dialogues to more complex tasks, such as multi-document question answering, few-shot learning, and tool use. These tasks typically involve long prompts comprising relatively immutable token chunks (compared to mutable user instructions) such as system messages, few-shot examples, and documents. Notably, such immutable chunks are frequently repeated across requests.

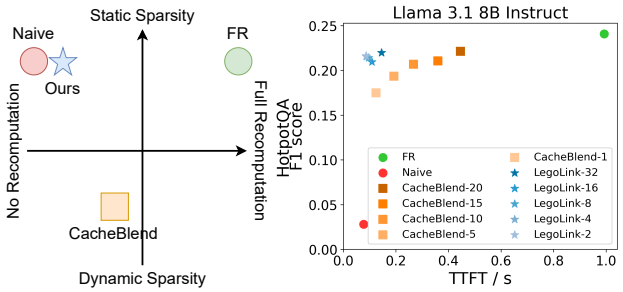

Figure 1. Left: Design space of position-independent context caching. Right: The x-axis shows the computation overhead or TTFT, while the y-axis shows accuracy. Different shades of the same color indicate variants of the same algorithm.

Context Caching[1] (CC) is an emerging approach that reuses Key-Value (KV) vectors, the intermediate representations of repeated tokens in previous requests to reduce computation, and is generally categorized into two types. First, **prefix-based CC** matches the current request against previous ones to reuse the KV vectors of the longest common prefix. Although prefix-based CC remains the dominant approach in existing systems (kim; gem, b; Zheng et al., 2024; Kwon et al., 2023), it requires exact prefix matches across requests, limiting reuse cases in settings such as few-shot learning and Retrieval-Augmented Generation (RAG), where immutable chunks (e.g., documents) remain unchanged across requests but are preceded by varying prefixes. Second, **Position-Independent Caching (PIC)** (Figure 2 (b)) extends prefix-

[†]This work was completed during his internship at Huawei Cloud. [1]SCS, Peking University; Key Lab of HCST (PKU), MOE, China [2]School of Computer Science, Nanjing University, Nanjing, China [3]Huawei Cloud, Shanghai, China [4]Key Lab of HCST (PKU), MOE; SCS, Peking University, China. Correspondence to: Tao Xie <taoxie@pku.edu.cn>, Yizhou Shan <shanyizhou@huawei.com>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.

[1]Also referred to as prompt caching.

based CC, enabling modular reuse of the KV vectors of immutable tokens, regardless of their prefixes (Yao et al., 2025). Although PIC significantly increases reuse opportunities (Figure 6), it deviates from standard attention mechanisms, resulting in potential accuracy degradation; thus, ensuring accurate recovery becomes its main challenge.

To tackle the challenge of PIC, we formalize its usage within a two-step framework analogous to compilation and linking (Figure 2). First, the **compile** step involves submitting individual immutable chunks to the LLM to generate and store their respective KV vectors. Second, the **link** step retrieves and concatenates cached KV vectors and recomputes a subset of KV vectors to maintain accuracy.

To the best of our knowledge, CacheBlend (Yao et al., 2025) is the first[2] work that fits into our PIC framework (Figure 1), and has two major limitations. First, the time and resource complexity of the recomputation in the link step are the same as the original attention mechanism—$O(N^2)$, where N is the number of tokens in the given prompt. Figure 1 shows that, although CacheBlend-15 dynamically selects 15% of tokens for recomputation, for very long prompts, common in many applications today, this $O(15\%N^2)$ complexity remains slow and prone to out-of-memory (OOM) errors (Figure 9). Second, CacheBlend relies on dynamic attention sparsity, which incurs heavy runtime overhead in addition to the $O(N^2)$ recomputation. Figure 10 shows that the runtime overhead of CacheBlend takes around 16.3% to 63.56% of Time-To-First-Token (TTFT).

To overcome the limitations of CacheBlend, we develop EPIC (Efficient Position-Independent Caching), a serving system that incorporates our simple but effective algorithm named *LegoLink* with two characteristics. First, *LegoLink* reduces recomputation complexity to $O(kN) \sim O(N)$, where $k \ll N$ and increases with the number of immutable chunks instead of N. As described in Section 5, k could potentially become zero. Second, *LegoLink* relies on static attention sparsity, which selects the tokens to recompute beforehand, further improving performance. The static token selection is based on our key insight: the initial tokens of each immutable chunk disproportionately absorb attention, impeding subsequent tokens from attending to relevant context—a phenomenon known as "attention sink" (Xiao et al., 2024). *LegoLink* recomputes k ($k \leq 32$) initial tokens of each chunk (except the first chunk), allowing these tokens to recognize their non-initial positions and crippling their attention-sink ability.

We implement the EPIC serving system with the *LegoLink* algorithm based on one of the most widely used inference frameworks, vLLM (Kwon et al., 2023). We evaluate EPIC against the state-of-the-art CacheBlend (Yao et al., 2025) system, across six tasks with distinct characteristics and three model architectures with diverse training recipes. Compared to CacheBlend, EPIC achieves up to a $3\times$ improvement in TTFT with accuracy losses limited to within 7% (Figure 1) when serving single requests. Furthermore, EPIC provides up to an $8\times$ reduction in TTFT and a $7\times$ increase in throughput when serving multiple requests under varying rates. The code is available at: https://github.com/DerekHJH/epic.

In summary, this paper makes three major contributions:

- We formalize the PIC usage into a two-step framework, within which we consolidate existing literature and highlight potential directions for future research.
- We provide a detailed analysis of existing algorithms, based on which we propose a new *LegoLink* algorithm, which reduces up to $3\times$ *TTFT* while keeping accuracy losses limited to within 7%, compared to the state-of-the-art CacheBlend system.
- We implement the EPIC serving system by incorporating OpenAI-compatible context caching APIs, a KV store, and *LegoLink*. EPIC reduces up to $8\times$ *TTFT* and increases up to $7\times$ throughput when serving multiple requests under varying rates.

2. Background and Motivation

This section provides a primer on transformers, context caching, and its variant, Position-Independent Caching (PIC), along with a review of an existing PIC algorithm.

2.1. Autoregressive Generation and KV Cache

The generation process of Large Language Models (LLMs) consists of two distinct stages: the prefill stage and the decode stage. In the **prefill** stage, the model processes a sequence of prompt tokens all at once. It computes the Key (K) and Value (V) vectors for all prompt tokens, stores these vectors in the KV cache, and generates the first output token to initiate the decode stage. The time required to generate the first token is referred to as the Time-To-First-Token (TTFT). The prefill stage is primarily compute-bound, as it involves processing multiple tokens in parallel. In the **decode** stage, the model iteratively processes each newly generated token. It computes the KV vectors for the new token, appends these vectors to the KV cache, and generates the next token. This process repeats until a specified stopping criterion is met. Unlike the prefill stage, the decode stage is memory-bound as it computes little compared to the amount of memory access.

[2]Another related approach called PromptCache (Gim et al., 2024) is not of the PIC type.

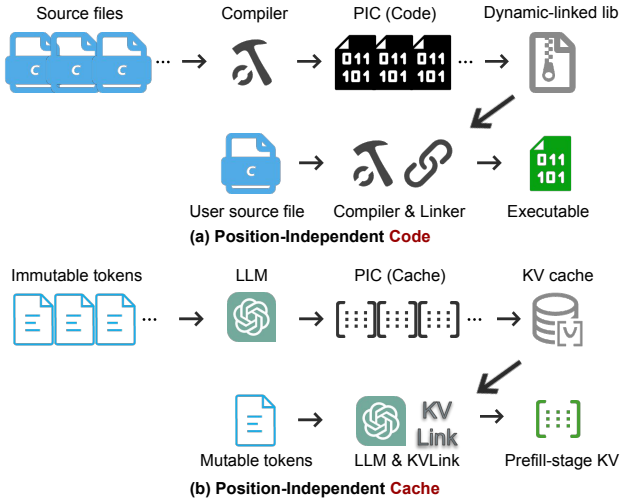

Figure 2. An analogy between position-independent code and position-independent cache.

2.2. Context Caching

LLMs' usage has shifted from simple dialogues to more complex tasks, such as multi-document question answering, few-shot learning, and tool use. These tasks typically involve long prompts comprising relatively immutable token chunks (compared to mutable user instructions/queries) such as system messages, few-shot examples, and documents. Notably, such immutable chunks are frequently repeated across requests. Context Caching (CC), also referred to as prompt caching, is an emerging approach that reuses the KV vectors of repeated tokens in previous requests (Hu et al., 2024a; Zheng et al., 2024; Liu et al., 2024; Kwon et al., 2023), speeding up the prefill stage and reducing TTFT. Context caching can be categorized into two types: prefix-based caching and Positional-Independent Caching (PIC).

Prefix-based caching, implemented in nearly all existing context caching systems (Kwon et al., 2023; Zheng et al., 2024; Gim et al., 2024), matches the current request against previous ones to reuse the KV vectors of the longest common prefix. This approach, however, requires an exact prefix match, as each token's KV vector depends on all preceding tokens and their absolute position IDs in the prompt. Consequently, even minor differences in the prefix invalidate the KV vectors of otherwise immutable chunks, requiring full recomputation. This constraint significantly limits reuse opportunities (lmc; Yao et al., 2025), especially in scenarios such as multi-document question answering or Retrieval-Augmented Generation (RAG), where immutable chunks (e.g., documents) remain unchanged across requests but are preceded by varying prefixes.

Position-Independent Caching (PIC), inspired by the classical position-independent code[3] that can be executed at any memory address (Hu et al., 2023), enables modular reuse of the KV vectors of immutable tokens (Yao et al., 2025), regardless of their prefix (Figure 2). PIC significantly increases reuse opportunities (Figure 6), but it deviates from standard attention mechanisms, resulting in potential accuracy degradation; thus, ensuring accurate recovery becomes its main challenge.

We formalize PIC usage within a two-step framework. First, the **compile** step involves submitting individual immutable chunks to the LLM to generate and store their respective KV vectors. In this step, each chunk is encoded with position IDs starting from zero, and the LLM performs only the prefill stage—generating KV vectors without any prefix or further token generation. This process is analogous to compiling C source files into position-independent relocatable code. The resulting KV vectors are stored in a cache, conceptually similar to packaging object code into a dynamically linked library. Second, the **link** step retrieves and concatenates cached KV vectors and recomputes a subset of KV vectors to mitigate accuracy degradation due to deviations from the standard attention mechanism. This recomputation involves both cached tokens and uncached tokens, such as user instruction/query tokens, which are computed for the first time[4]. This process is analogous to linking dynamically linked libraries with source code to produce an executable.

2.3. Existing Algorithms for PIC

CacheBlend (Yao et al., 2025) is the first PIC algorithm focusing on the link step, but two simpler algorithms also deserve attention, although they are too rudimentary to be classified as proper algorithms. First, *Naive* reuses cached KV vectors directly in the link step, without any recomputation (Figure 4, first row below the dashed line). This algorithm incurs zero linking overhead but leads to substantial accuracy degradation (Figure 6) due to violations of the attention mechanism—the PIC challenge. Second, Fully Recompute (*FR*) recomputes all KV vectors in the link step (Figure 4, second row below the dashed line). This algorithm preserves the standard attention mechanism and achieves the highest accuracy, but it eliminates the efficiency benefits of caching, resulting in the highest linking overhead (Figure 6).

To strike a balance between accuracy and linking overhead, CacheBlend (Yao et al., 2025) works as follows. First, it retrieves the needed KV vectors and concatenates them to obtain KV_old. Second, it recomputes all KV vectors in the first layer of the LLM, generating KV_new_1. Third,

[3] https://en.wikipedia.org/wiki/Position-independent_code

[4] Strictly speaking, uncached tokens are not recomputed, but for simplicity, we use "recomputation" to refer to both cases.

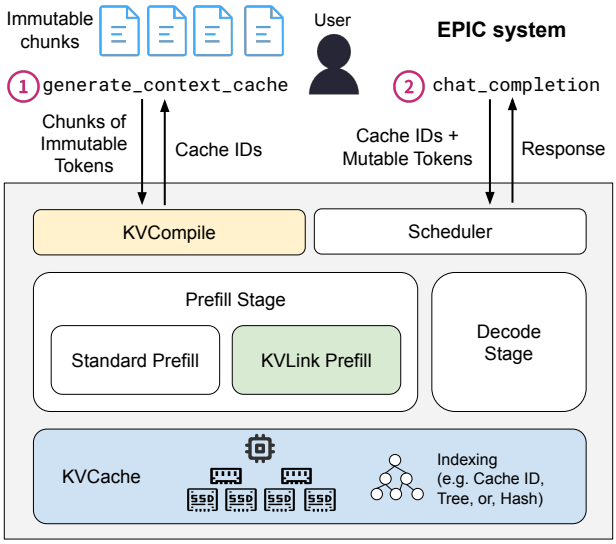

Figure 3. The architecture of EPIC serving system.

it compares the attention maps produced by KV_old_1 and KV_new_1, selecting the 15% of tokens that exhibit the most discrepancy. Fourth, it recomputes only these 15% of tokens in all subsequent layers[5]. Only 15% of tokens are necessary because attention exhibits **sparsity**—only a small subset of tokens significantly influence attention computation.

However, CacheBlend has two limitations. First, the time and resource complexities of the recomputation in the link step are the same as the original attention mechanism—$O(N^2)$, where N is the number of tokens in the prompt. Figure 1 shows that, although CacheBlend-15 dynamically selects 15% of tokens for recomputation, for very long prompts, common in many applications today, this $O(15\%N^2)$ complexity remains slow and prone to out-of-memory (OOM) errors (Figure 9). Second, CacheBlend relies on dynamic attention sparsity—recomputing all KV vectors in the first layer; this recomputation incurs heavy runtime overhead in addition to the $O(N^2)$ recomputation. Figure 10 shows that the runtime overhead of CacheBlend takes around 16.3% to 63.56% of Time-To-First-Token (TTFT).

3. System Overview

To support PIC, we develop EPIC (Efficient Position-Independent Caching), a serving system whose workflow aligns with the PIC framework described in Section 2 and consists of two main steps (Figure 3). First, in the compile step, ① users submit immutable chunks via the context caching API. The `KVCompile` component processes each chunk using a standard prefill pass to generate KV vectors, which are then stored in the `KVCache`. For each chunk, `KVCompile` returns a unique cache ID, which users can later reference to enable KV reuse. Second, in the link step, ② users submit requests containing mutable tokens (e.g., new instructions/queries) along with cache IDs (if any), using an extended chat completion API. The `Scheduler` component handles these requests by initiating a `KVLink` prefill. `KVLink` retrieves the relevant KV vectors from `KVCache` using the provided cache IDs, concatenates them, recomputes a subset of KV vectors to ensure correctness, and proceeds to the decode stage for subsequent token generation. The final response is then returned to users. At the core of `KVLink` is the *LegoLink* algorithm, which we detail in Section 4.

Discussion of implicit vs. explicit caching. Many existing systems (Kwon et al., 2023; Zheng et al., 2024; Hu et al., 2024a) adopt an implicit caching paradigm, where the system automatically manages cache generation, storage, and reuse through internal mechanisms such as hash tables or radix trees. In contrast, EPIC adopts an explicit caching paradigm, where users manage cache generation and reuse via explicit APIs that expose cache IDs. This paradigm, also used by systems such as Google Gemini and Mooncake (kim; gem, b), reduces indexing overhead and provides greater user control over cache management—particularly beneficial in RAG scenarios.

4. Algorithm Design

In this section, we analyze existing algorithms and propose a new algorithm *LegoLink* based on the analysis results.

4.1. Analysis of Existing Algorithms

We analyze the attention map of *Naive*, *FR*, and CacheBlend algorithms (Section 2), shown in the bottom right of Figure 4. First, in the *Naive*'s attention map, most attention scores concentrate on the initial tokens of each chunk, evident from the four bright vertical lines along the x-axis. This pattern arises because EPIC independently compiles each chunk with position IDs starting from zero. As a result, the initial tokens disproportionately absorb attention—a phenomenon known as "attention sink" (Xiao et al., 2024)—which prevents subsequent tokens from attending to the answer "Chrysan Company," located at the end of the third chunk (Chunk 1). Second, in the *FR*'s attention map, the initial tokens of each chunk release part of their attention to more relevant positions, including "Chrysan Company". However, they still retain relatively strong attention scores partly because they are special begin-of-sentence tokens, such as "<s>" in Llama models. Third, in the CacheBlend's attention map, the pattern closely resembles that of *FR*, reflecting its design goal of approximating *FR*'s attention map. Further analysis of CacheBlend's selected 15% of tokens shows that initial tokens of each chunk are frequently in-

[5]The recomputation rate might decrease in deeper layers.

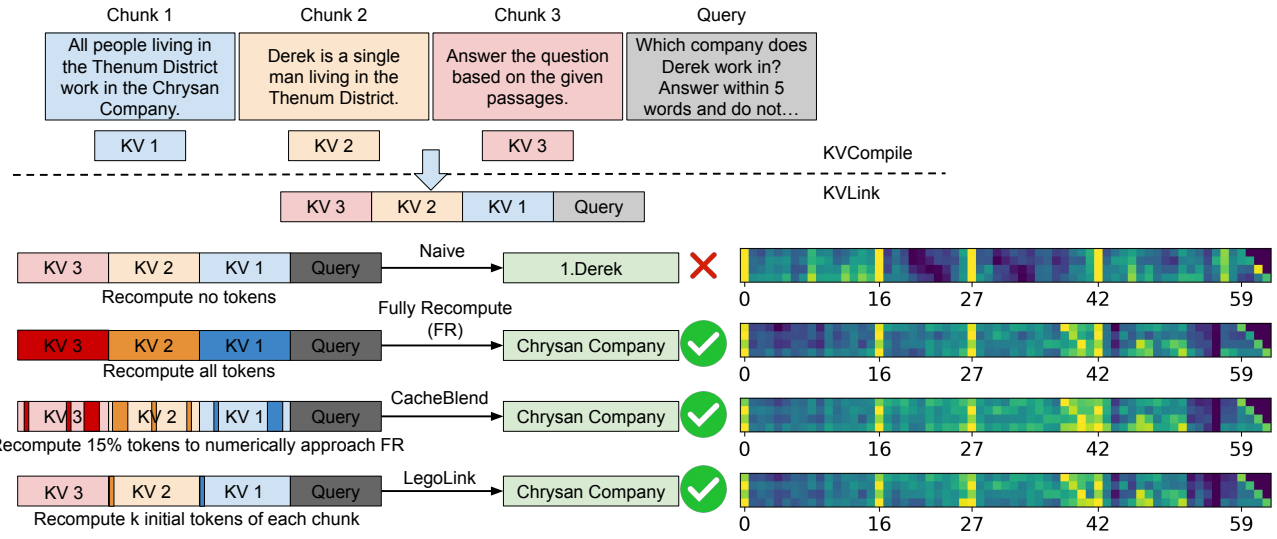

Figure 4. Comparison of PIC Algorithms. The area above the dashed line corresponds to the compile step, while the area below corresponds to the link step. `KVLink` recomputes a subset of tokens, highlighted in dark colors. Four algorithms include *Naive*, *Fully Recompute (FR)*, CacheBlend, and *LegoLink*. The bottom right visualizes attention maps (layer 5, head 5 of Llama 3.1 8B) for four decoded tokens. The x-axis marks the position ID of the first token of each chunk. To highlight the differences between attention maps, we normalize the QK^T results to the [0, 1] range using min-max scaling instead of Softmax.

cluded, reinforcing the importance of recomputing these tokens to improve accuracy.

4.2. The *LegoLink* Algorithm

Based on the preceding analysis, we propose *LegoLink*, which recomputes each chunk's first k tokens (except the first chunk), thus linking chunks like Lego pieces. By recomputing these initial tokens, *LegoLink* enables them to recognize their non-initial status, thereby mitigating their tendency to dominate attention and redirecting attention to relevant positions, as shown in *LegoLink*'s attention map in Figure 4. Evaluation results in Section 5 demonstrate that *LegoLink* consistently preserves accuracy across a wide range of datasets and models. See *LegoLink*'s details in the next paragraph.

Assuming that we have selected k' (k tokens from each chunk plus the user query) tokens from a total of N (prompt length) tokens, we recompute them as follows. First, we obtain the embedding matrix E (with shape (k', d)) of the k' tokens, where d is the hidden size. Second, at layer i, we compute the new K, Q, and V matrices (each with shape (k', d)) for these k' tokens: $Q = EW_Q$, $K = EW_K$, $V = EW_V$, where W_Q, W_K, and W_V are model parameters with shape (d, d)[6]. Third, we expand the K and V matrices by incorporating the cached KV vectors of the $N - k'$ unselected tokens at correct positions, forming K_{exp} and V_{exp} (both with shape (N, d)). Fourth, we compute the attention matrix A (with shape (k', N)) by multiplying Q (with shape (k', d)) with K_{exp}^T (with shape (d, N)), allowing the k' tokens to attend to all N tokens:

$$A = \text{softmax}(QK_{exp}^T \cdot \text{MASK}) \quad (1)$$

where MASK assures that the k' tokens attend to only tokens before them. Finally, we multiply A (with shape (k', N)) with V_{exp} (with shape (N, d)) to obtain the output (or input to the next layer, with shape (k', d)): $O = AV_{exp}W_O$, where W_O is a matrix with shape (d, d).

In addition to preserving accuracy and simple design/implementation, *LegoLink* offers two key advantages over CacheBlend. First, *LegoLink* reduces recomputation complexity to $O(kN) \sim O(N)$, where $k \ll N$ and increases with the number of immutable chunks instead of N. As described in Section 5, k could potentially become zero. Second, *LegoLink* relies on static attention sparsity, which selects each chunk's first k tokens to recompute beforehand, further improving performance.

5. Evaluation

We begin by describing the experimental setup, including implementation details, datasets, models, evaluation metrics, and software/hardware environment. We then present four key evaluation results.

[6]For notation simplicity, d represents all possible hidden dimension sizes, which may be further divided into the number of heads and head dimensions.

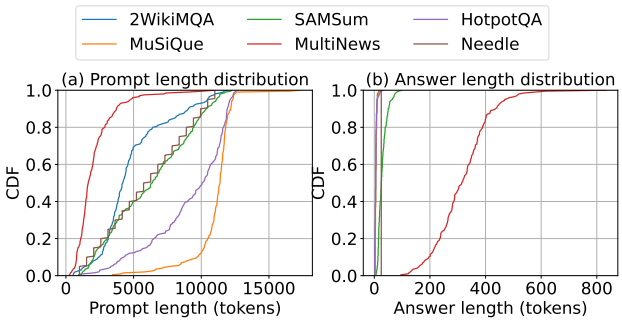

Figure 5. Prefill and decode length distribution.

5.1. Experiment Setup

Implementation. We implement EPIC based on vLLM 0.4.1 (Kwon et al., 2023), with 2K lines of code in Python. We incorporate the four PIC algorithms presented in Figure 4. We port CacheBlend from their public repository[7].

Dataset. Following CacheBlend, we evaluate on four LongBench datasets (Bai et al., 2024): *2WikiMQA* (multi-document question answering), *MuSiQue* (multi-document question answering), *SAMSum* (few-shot instruction following), and *MultiNews* (multi-document summarization). We also include *HotpotQA* (multi-document question answering) from LongBench, which identifies the two supporting documents containing the answer, enabling fine-grained analysis. To evaluate long-context retrieval, we include the *Needle in a Haystack* dataset (LLM), which tests the model's ability to locate and retrieve a single inserted fact from unrelated documents of varying lengths. All datasets contain 200 test cases, with the distribution of prompt (prefill) lengths and answer (decode) lengths shown in Figure 5. Immutable tokens constitute approximately 95%-99% of the prompt, while mutable tokens are fewer than 50.

Metrics. We use the following three metrics to evaluate performance and model accuracy. First, *Time-To-First-Token (TTFT)* (Kwon et al., 2023) (lower is better) is used to evaluate all datasets. This metric measures the prefill-stage time: the time from when users send a request to when users receive the first token; this time could be reduced by using context caching. Second, *F1 score* (Bai et al., 2024) (higher is better) is used to evaluate *2WikiMQA*, *MuSiQue*, *HotpotQA*, and *needle in a haystack*. This metric measures the similarity between LLMs' output and the ground-truth answer based on their common words. Third, *Rough-L score* (Lin, 2004) (higher is better) is used to evaluate *SAMSum* and *MultiNews*. This metric measures the similarity between LLMs' output and the ground-truth answer by calculating the length of their longest common subsequence.

[7]https://github.com/YaoJiayi/CacheBlend. Accessed in Sep 2024.

Models. We evaluate EPIC and *LegoLink* using three state-of-the-art open-source LLMs: Mistral 7B Instruct (Jiang et al., 2023), Llama 3.1 8B Instruct (Dubey et al., 2024), and Yi Coder 9B Chat (Young et al., 2024). These models represent diverse architectures and training recipes. Rather than employing quantized versions of larger models, we select smaller models to accommodate our limited GPU resources. Additionally, we do not choose models fine-tuned for the six specific task types, as the number of such models is extensive. While our chosen general-purpose base models may exhibit lower absolute accuracy on these tasks, the relative accuracy drop compared to the base model is sufficient to demonstrate the effectiveness of EPIC and *LegoLink*.

Baselines. We compare *LegoLink* with the other three recomputation algorithms in Figure 4: *FR*, *Naive*, and CacheBlend (Yao et al., 2025). Additionally, we evaluate different variants of CacheBlend, denoted as CacheBlend-r, where r represents the ratio of tokens recomputed. Similarly, we evaluate different variants of *LegoLink*, denoted as *LegoLink*-k, where k refers to each chunk's first k tokens.

Environment. We run experiments on a single NVIDIA A100 server with one A100-80GB GPU available. The server has 128-core Intel(R) Xeon(R) Platinum 8358P CPU@2.60GHz with 2 hyperthreading and 1TB DRAM. We use Ubuntu 20.04 with Linux kernel 5.16.7 and CUDA 12.6.

5.2. Workloads

We construct the following two kinds of workflows out of the six datasets.

Synchronous workload. To evaluate the accuracy–latency trade-off without interference from concurrent requests, we process test cases sequentially, ensuring that each completes before the next begins. First, for each test case, we compile all immutable chunks to obtain their corresponding cache IDs. For the LongBench dataset (excluding SAMSum), we treat each document as a chunk. For SAMSum and Needle-in-a-Haystack, we split all immutable tokens into 512-token chunks. Second, we send a request containing the cache IDs of cached chunks along with the query to obtain the response.

Asynchronous workload. To evaluate the latency and throughput of EPIC under varying request rates (requests per second), we simulate a PIC scenario as follows[8]. First, we select d test cases from *2WikiMQA* to simulate d active users. As the number of users increases, a larger portion of the GPU HBM is allocated to their position-independent cache.

[8]PIC is a relatively new approach and lacks publicly available traces or request arrival patterns. We try our best to mitigate potential bias in constructing this asynchronous workload.

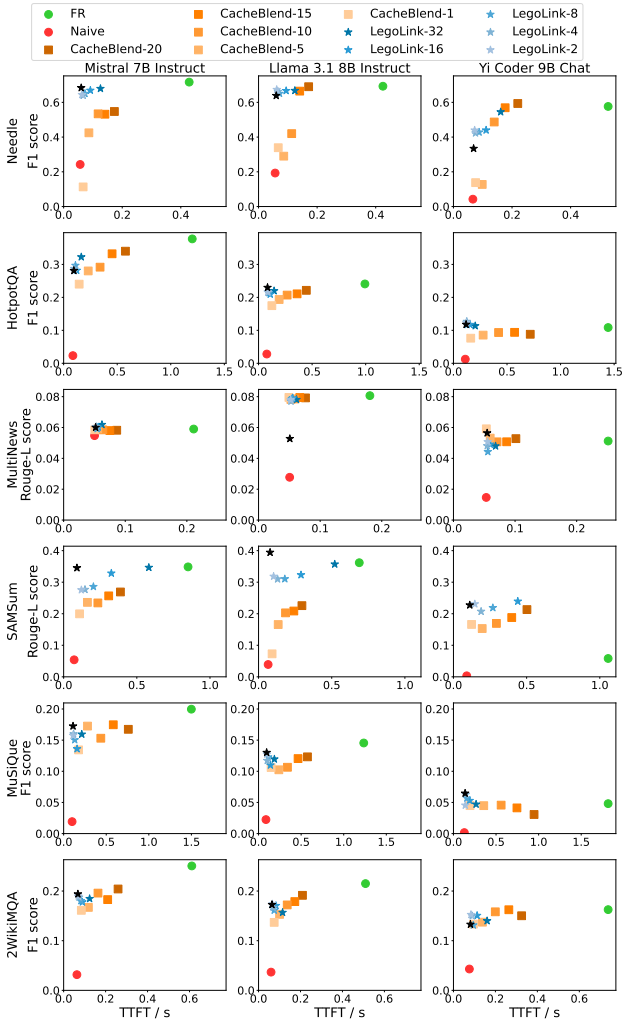

Figure 6. Accuracy vs. TTFT. Each point indicates the average *TTFT* and accuracy for running synchronous workloads of one dataset (row) on one model (column) using one specific algorithm (each legend label). The k in *LegoLink*-k denotes the number of recomputed initial tokens for each chunk, while the r in CacheBlend-r represents the ratio of all recomputed tokens. The black star represents *LegoLink*-0, a zero-linking algorithm.

This portion is defined as the Context Cache Ratio (CCR). Second, each user compiles all immutable chunks in the test case once and then repeatedly sends the same request (containing cache IDs of cached chunks along with the query) at a constant rate over a 40-second period. Although each user resends identical requests, this setup effectively simulates a user having different queries over the same document set, with all other context caching mechanisms, such as prefix caching, disabled. Third, we simulate request arrival times by sampling from a Poisson distribution.

5.3. Accuracy-Latency Trade-off of *LegoLink*

Using the synchronous workload described earlier, we draw three key insights from the results in Figure 6. First, *LegoLink* variants (a series of gradient blue stars) establish a new Pareto frontier, outperforming CacheBlend variants (a series of gradient orange rectangles) in most cases. Second, *LegoLink*-2 is sufficient to limit accuracy drops within 0 - 7% and reduces up to 300% *TTFT*, compared to the default CacheBlend-15 configuration. On the contrary, CacheBlend-1 or CacheBlend-5, which recomputes a similar number of tokens as most *LegoLink* variants (except on *SAMSum*), exhibits significant accuracy degradation—up to 80% worse than *FR*. Third, increasing the number of recomputed tokens in *LegoLink* yields diminishing accuracy gains. Recomputing only a small number of initial tokens suffices to restore most of the accuracy, whereas CacheBlend requires substantially more recomputation for marginal benefits.

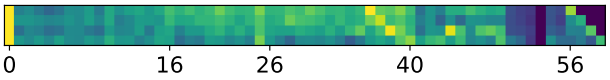

Figure 7. Attention map of *LegoLink*-0 using the example in Figure 4.

In addition, we also have three unusual observations that warrant further explanation. First, all algorithms, including *FR*, CacheBlend, and *LegoLink*, exhibit low accuracy in the Yi Coder model due to its poor handling of document understanding. This observation suggests that, to ensure that PIC algorithms perform optimally, robust models well suited to the task are required. Second, all approaches using all models perform poorly on the *MultiNews* dataset. This phenomenon can be attributed to the inherent difficulty of summarizing long documents with small models. Third, CacheBlend and *LegoLink* exhibit similar *TTFT* in the *MultiNews* and *SAMSum* datasets. Each document in these datasets is relatively short (around one hundred tokens), making the number of tokens recomputed (k tokens in *LegoLink*-k) equivalent to the ratio of tokens recomputed (r% in CacheBlend-r).

5.4. Algorithm Analysis

To further understand *LegoLink*, we introduce *LegoLink*-0, a variant that shifts all linking overhead to the compile step and comprises two different PIC steps. First, EPIC prepends four dummy tokens (e.g., begin-of-sentence tokens) to each immutable chunk during compilation, then discards their corresponding KV vectors. This removal eliminates "attention sink" tokens in advance, preventing them from interfering with subsequent attention computations. Second, in the link step, *LegoLink*-0 skips recomputation entirely, incurring zero runtime overhead.

Using the synchronous workload, we draw two key insights from the results in Figure 6 and Figure 7. First, *LegoLink*-0, despite its minimal link-time cost, *LegoLink*-0 preserves

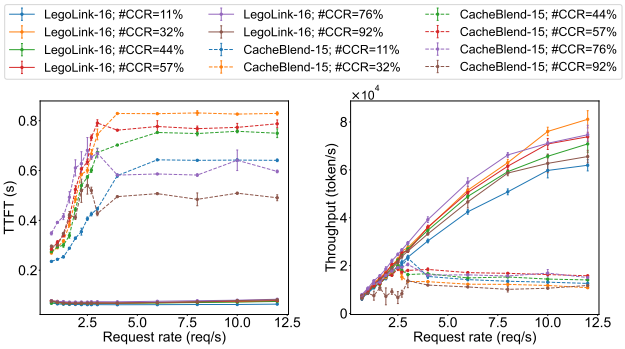

Figure 8. Latency and throughput comparison of *LegoLink*-16 and CacheBlend-15 under asynchronous workloads with varying request rates and context cache ratios (CCR). Each data point represents the average and standard deviation from five experiments. *LegoLink*-16 is shown using solid lines, while CacheBlend-15 is represented with dashed lines. Two algorithms with the same CCR are shown in the same color.

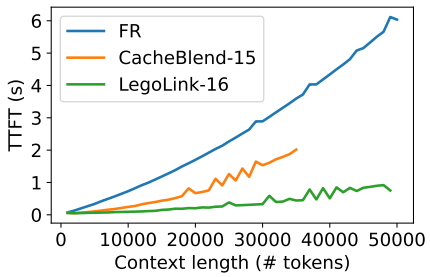

Figure 9. TTFT vs. context length of *FR*, CacheBlend-15, and *LegoLink*-16, using a fixed chunk size of 512 tokens. For *FR*, we do not compile context cache to display its full quadratic time complexity trend, as it would otherwise run out of memory earlier than CacheBlend-15 and *LegoLink*-16.

accuracy remarkably well. Second, the "attention sink" phenomenon disappears in the middle (Figure 7), reinforcing the importance of mitigating chunk-initial tokens' influence on subsequent attention computations. On the other hand, the previously raised concern in CacheBlend regarding limited cross-attention across chunks proves less impactful in practice. Query and decoded tokens can still attend to all earlier chunks, enabling effective aggregation of cross-chunk information.

Discussion of lengthy outputs in sparsity algorithms. *LegoLink* variants occasionally show reduced accuracy on cases such as (MultiNews, Llama 3.1) and (Needle, Yi). However, this drop stems not from incorrect answers but from unnecessarily lengthy outputs. For the example in Figure 4, *LegoLink*-0 correctly begins with "Chrysan Company" but continues with unrelated content such as "and that Derek is living in ...," which lowers F1 or ROUGE-L scores. We observe similar behaviors in other sparsity-based algorithms such as StreamingLLM (Xiao et al., 2024), H2O (Zhang et al., 2023), and Quest (Tang et al., 2024). Such behavior undermines the primary goals of sparsity—reducing latency and resource usage. We leave a more detailed investigation of this behavior to future work.

5.5. Latency and Throughput of EPIC

We employ the artificial asynchronous workloads on *LegoLink*-16 (16 is the block size of vLLM and a moderate number of tokens to recompute) and CacheBlend-15, presenting results in Figure 8. Notably, the numbers in this section should be interpreted cautiously when considering real-world scenarios.

Regarding *TTFT* versus request rates (left of Figure 8), we observe three key trends. First, *LegoLink*-16 achieves up to an $8\times$ reduction in *TTFT* compared to CacheBlend-15. Second, as the Context Cache Ratio (CCR) increases, *LegoLink*-16 remains stable *TTFT*, whereas CacheBlend-15 fluctuates around 0.5 seconds. This stability likely results from *LegoLink*-16 generating fewer intermediate results; higher CCR reduces available memory for intermediate computation, but *LegoLink*-16 incurs less recomputation overhead than CacheBlend-15. Third, *TTFT* plateaus as request rate increases, rather than growing exponentially. This plateau reflects vLLM's scheduling policy, which limits the number of concurrent running requests based on available memory. If we included the *TTFT* of all waiting (queued) requests, the average *TTFT* would approach infinity.

Regarding throughput versus request rates (right of Figure 8), we observe two notable facts. First, *LegoLink*-16 achieves a throughput that is up to $7\times$ higher than CacheBlend-15, as it recomputes fewer tokens, allowing more requests (about $7\times$) to be processed simultaneously. Second, as CCR increases, *LegoLink*-16's throughput continues to improve until the CCR reaches a threshold (approximately 30%), beyond which further increases in CCR lead to a reverse effect because requests start to severely interfere with each other. In contrast, CacheBlend-15's throughput remains constant as it becomes incapable of handling additional requests.

5.6. EPIC's Performance on Long Context

We send requests of varying context lengths with a fixed chunk size (512 tokens) synchronously to EPIC, yielding two observations from the results (Figure 9). First, as context length increases, the *TTFT* of both *FR* and CacheBlend-15 grows quadratically, while *LegoLink*-16 exhibits nearly linear growth. This difference arises because *FR* and CacheBlend-15 have time and resource complexities of $O(N^2)$, while *LegoLink*-16 operates with a complexity of $O(kN)$, where k represents the number of recomputed

tokens ($k \ll N$). Second, *LegoLink*-16 supports a longer context length compared to CacheBlend-15. Specifically, CacheBlend-15 encounters an out-of-memory (OOM) error at approximately 35,000 tokens, while *LegoLink*-16 avoids OOM until the context length reaches 50,000 tokens. This difference is due to CacheBlend-15's need to recompute more tokens and generate additional intermediate results, leading to higher GPU memory usage.

6. Related Work

This work formalizes Position-Independent Context Caching (PIC) and advances the state of the art in this emerging area. Below, we outline the broader design space relevant to our work.

LLM-serving optimizations. Numerous systems have recently emerged to improve LLM serving efficiency. vLLM (Kwon et al., 2023) introduces PagedAttention to achieve high throughput, while SGLang (Zheng et al., 2024) provides both a domain-specific frontend language and an optimized backend runtime. DeepFlow (Hu et al., 2025b) integrates the advantages of existing research work into a system running on Ascend accelerators at Huawei Cloud. In addition to full systems, researchers have proposed scheduling techniques such as disaggregated prefill and decode (Zhong et al., 2024; Hu et al., 2024a;b; Patel et al., 2024), continuous batching (Yu et al., 2022), and multi-LoRA integration (Sheng et al., 2024; Li et al., 2024). Storage-related optimizations such as KV-cache-centric inference systems (Qin et al., 2025; Hu et al., 2024a) also contribute to this space.

Context Caching (CC). Two primary types of context caching have emerged. First, prefix-based CC emerged in late 2023, represented by Pensieve (Yu et al., 2025), CacheGen (Liu et al., 2024), and SGLang (Zheng et al., 2024). Recently, vendors such as Kimi (kim) and Gemini (gem, b) have begun offering explicit CC APIs. Second, PIC emerged in mid-2024 and CacheBlend (Yao et al., 2025) represents the first attempt to tackle the PIC challenge, although it does not formally define the challenge. PromptCache (Gim et al., 2024) aims to support PIC, but its reuse remains position-dependent. In this paper, we formally define PIC and advance the state of the art by introducing *LegoLink*, a low-overhead or even zero-overhead linking algorithm.

Sparsity. Sparsity plays a crucial role in improving long-context inference and falls into two types: dynamic and static. First, dynamic sparsity (e.g., H2O (Zhang et al., 2023), Quest (Tang et al., 2024), ArkVale (Chen et al., 2024), RaaS (Hu et al., 2025a)) determines important tokens at runtime. Second, static sparsity (e.g., Longformer (Beltagy et al., 2020), StreamingLLM (Xiao et al., 2024)) relies on predefined sparse patterns. CacheBlend leverages dynamic sparsity while *LegoLink* leverages static sparsity to enable efficient linking.

Retrieval-Augmented Generation (RAG). RAG (Li et al., 2022; Jin et al., 2024; Gao et al., 2023; Jeong et al., 2024; Ram et al., 2023; Mao et al., 2021) enhances LLMs' capabilities by integrating external knowledge to improve factuality and relevance. For example, at the application level, Adaptive-RAG (Jeong et al., 2024) dynamically selects retrieval and generation strategies based on query complexity. At the system level, RAGCache (Jin et al., 2024) reduces latency by caching and reusing intermediate states from retrieved documents. PIC has strong potential in RAG scenarios, where reusing documents' KV cache across requests can yield significant performance gains.

7. Conclusion

In this paper, we have formalized the Positional-Independent-Cache (PIC) framework. Within this framework, we have proposed EPIC, a system that incorporates the *LegoLink* algorithm to address key limitations of existing approaches. By leveraging static attention sparsity, *LegoLink* significantly reduces recomputation complexity in the link step while maintaining accuracy. Extensive evaluation across six datasets and three LLM models has shown that EPIC achieves significant improvements in *TTFT* and throughput compared to existing systems, with minimal or no accuracy loss.

Acknowledgments

This work was partially supported by National Natural Science Foundation of China under Grant No. 92464301. We would also like to thank the anonymous reviewers for their insightful comments and suggestions, which helped improve the quality of this paper.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none of which we feel must be specifically highlighted here.

References

Needle In A Haystack. https://github.com/gkamradt/LLMTest_NeedleInAHaystack.

Gemini. https://gemini.google.com/, a.

Gemini context caching. https://ai.google.dev/gemini-api/docs/caching?lang=python, b.

Kimi context caching. https://platform.moonshot.cn/docs/api/caching.

LMCache. https://github.com/LMCache.

Bai, Y., Lv, X., Zhang, J., Lyu, H., Tang, J., Huang, Z., Du, Z., Liu, X., Zeng, A., Hou, L., Dong, Y., Tang, J., and Li, J. LongBench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the Sixty-Second Annual Meeting of the Association for Computational Linguistics*, pp. 3119–3137, 2024.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *CoRR*, 2020.

Chen, R., Wang, Z., Cao, B., Wu, T., Zheng, S., Li, X., Wei, X., Yan, S., Li, M., and Liang, Y. ArkVale: Efficient generative LLM inference with recallable key-value eviction. In *Proceedings of the Advances in Neural Information Processing Systems*, pp. 113134–113155, 2024.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., Rodriguez, A., Gregerson, A., Spataru, A., Rozière, B., Biron, B., Tang, B., Chern, B., Caucheteux, C., Nayak, C., Bi, C., Marra, C., McConnell, C., Keller, C., Touret, C., Wu, C., Wong, C., Ferrer, C. C., Nikolaidis, C., Allonsius, D., Song, D., Pintz, D., Livshits, D., Esiobu, D., Choudhary, D., Mahajan, D., Garcia-Olano, D., Perino, D., Hupkes, D., Lakomkin, E., AlBadawy, E., Lobanova, E., Dinan, E., Smith, E. M., Radenovic, F., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar, H., Xu, H., Touvron, H., Zarov, I., Ibarra, I. A., Kloumann, I. M., Misra, I., Evtimov, I., Copet, J., Lee, J., Geffert, J., Vranes, J., Park, J., Mahadeokar, J., Shah, J., van der Linde, J., Billock, J., Hong, J., Lee, J., Fu, J., Chi, J., Huang, J., Liu, J., Wang, J., Yu, J., Bitton, J., Spisak, J., Park, J., Rocca, J., Johnstun, J., Saxe, J., Jia, J., Alwala, K. V., Upasani, K., Plawiak, K., Li, K., Heafield, K., Stone, K., and et al. The Llama 3 herd of models. *CoRR*, 2024.

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Guo, Q., Wang, M., and Wang, H. Retrieval-augmented generation for large language models: A survey. *CoRR*, 2023.

Gim, I., Chen, G., Lee, S., Sarda, N., Khandelwal, A., and Zhong, L. Prompt cache: Modular attention reuse for low-latency inference. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems*, pp. 325–338, 2024.

Hu, C., Huang, H., Hu, J., Xu, J., Chen, X., Xie, T., Wang, C., Wang, S., Bao, Y., Sun, N., and Shan, Y. MemServe: Context caching for disaggregated LLM serving with elastic memory pool. *CoRR*, 2024a.

Hu, C., Huang, H., Xu, L., Chen, X., Xu, J., Chen, S., Feng, H., Wang, C., Wang, S., Bao, Y., Sun, N., and Shan, Y. Inference without interference: Disaggregate LLM inference for mixed downstream workloads. *CoRR*, 2024b.

Hu, J., Wang, C., Huang, H., Luo, H., Jin, Y., Deng, Y., and Xie, T. Predicting compilation resources for adaptive build in an industrial setting. In *Proceedings of the Thity-Eighth IEEE/ACM International Conference on Automated Software Engineering*, pp. 1808–1813, 2023.

Hu, J., Huang, W., Wang, W., Li, Z., Hu, T., Liu, Z., Chen, X., Xie, T., and Shan, Y. Efficient long-decoding inference with reasoning-aware attention sparsity. *CoRR*, 2025a.

Hu, J., Xu, J., Liu, Z., He, Y., Chen, Y., Xu, H., Liu, J., Zhang, B., Wan, S., Dan, G., Dong, Z., Ren, Z., Meng, J., He, C., Liu, C., Xie, T., Lin, D., Zhang, Q., Yu, Y., Feng, H., Chen, X., and Shan, Y. DeepFlow: Serverless large language model serving at scale. *CoRR*, 2025b.

Jeong, S., Baek, J., Cho, S., Hwang, S. J., and Park, J. Adaptive-RAG: Learning to adapt retrieval-augmented large language models through question complexity. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 7036–7050, 2024.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de Las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mistral 7b. *CoRR*, 2023.

Jin, C., Zhang, Z., Jiang, X., Liu, F., Liu, X., Liu, X., and Jin, X. RAGCache: Efficient knowledge caching for retrieval-augmented generation. *CoRR*, 2024.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the Twenty-Ninth Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Li, H., Su, Y., Cai, D., Wang, Y., and Liu, L. A survey on retrieval-augmented text generation. *CoRR*, 2022.

Li, S., Lu, H., Wu, T., Yu, M., Weng, Q., Chen, X., Shan, Y., Yuan, B., and Wang, W. CaraServe: CPU-assisted and rank-aware LoRA serving for generative LLM inference. *CoRR*, 2024.

Lin, C.-Y. Rouge: A package for automatic evaluation of summaries. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pp. 74–81, 2004.

Liu, Y., Li, H., Cheng, Y., Ray, S., Huang, Y., Zhang, Q., Du, K., Yao, J., Lu, S., Ananthanarayanan, G., Maire, M., Hoffmann, H., Holtzman, A., and Jiang, J. Cachegen: KV cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pp. 38–56, 2024.

Mao, Y., He, P., Liu, X., Shen, Y., Gao, J., Han, J., and Chen, W. Generation-augmented retrieval for open-domain question answering. In *Proceedings of the Fifty-Ninth Annual Meeting of the Association for Computational Linguistics and the Eleventh International Joint Conference on Natural Language Processing*, pp. 4089–4100, 2021.

Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S., and Bianchini, R. Splitwise: Efficient generative LLM inference using phase splitting. In *Proceedings of the Fifty-First Annual International Symposium on Computer Architecture*, pp. 118–132, 2024.

Qin, R., Li, Z., He, W., Cui, J., Ren, F., Zhang, M., Wu, Y., Zheng, W., and Xu, X. Mooncake: Trading more storage for less computation - A KVCache-centric architecture for serving LLM chatbot. In *Proceedings of the Twenty-Third USENIX Conference on File and Storage Technologies*, pp. 155–170, 2025.

Ram, O., Levine, Y., Dalmedigos, I., Muhlgay, D., Shashua, A., Leyton-Brown, K., and Shoham, Y. In-context retrieval-augmented language models. *Transactions of the Association for Computational Linguistics*, pp. 1316–1331, 2023.

Sheng, Y., Cao, S., Li, D., Hooper, C., Lee, N., Yang, S., Chou, C., Zhu, B., Zheng, L., Keutzer, K., Gonzalez, J., and Stoica, I. SLoRA: Scalable serving of thousands of lora adapters. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems*, pp. 296–311, 2024.

Tang, J., Zhao, Y., Zhu, K., Xiao, G., Kasikci, B., and Han, S. QUEST: query-aware sparsity for efficient long-context LLM inference. In *Proceedings of the Forty-First International Conference on Machine Learning*, pp. 47901–47911, 2024.

Xiao, G., Tian, Y., Chen, B., Han, S., and Lewis, M. Efficient streaming language models with attention sinks. In *Proceedings of the Twelfth International Conference on Learning Representations*, 2024.

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

Young, A., Chen, B., Li, C., Huang, C., Zhang, G., Zhang, G., Li, H., Zhu, J., Chen, J., Chang, J., Yu, K., Liu, P., Liu, Q., Yue, S., Yang, S., Yang, S., Yu, T., Xie, W., Huang, W., Hu, X., Ren, X., Niu, X., Nie, P., Xu, Y., Liu, Y., Wang, Y., Cai, Y., Gu, Z., Liu, Z., and Dai, Z. Yi: Open foundation models by 01.AI. *CoRR*, 2024.

Yu, G., Jeong, J. S., Kim, G., Kim, S., and Chun, B. Orca: A distributed serving system for transformer-based generative models. In *Proceedings of the Sixteenth USENIX Symposium on Operating Systems Design and Implementation*, pp. 521–538, 2022.

Yu, L., Lin, J., and Li, J. Stateful large language model serving with Pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 144–158, 2025.

Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C. W., Wang, Z., and Chen, B. H2O: heavy-hitter oracle for efficient generative inference of large language models. In *Proceedings of the Advances in Neural Information Processing Systems*, pp. 34661–34710, 2023.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C. W., and Sheng, Y. SGLang: Efficient execution of structured language model programs. In *Proceedings of the Advances in Neural Information Processing Systems*, pp. 62557–62583, 2024.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. DistServe: Disaggregating prefill and decoding for goodput-optimized LLM serving. In *Proceedings of the Eighteenth Symposium on Operating Systems Design and Implementation*, pp. 193–210, 2024.

Zhou, Z., Ning, X., Hong, K., Fu, T., Xu, J., Li, S., Lou, Y., Wang, L., Yuan, Z., Li, X., Yan, S., Dai, G., Zhang, X., Dong, Y., and Wang, Y. A survey on efficient inference for large language models. *CoRR*, 2024.

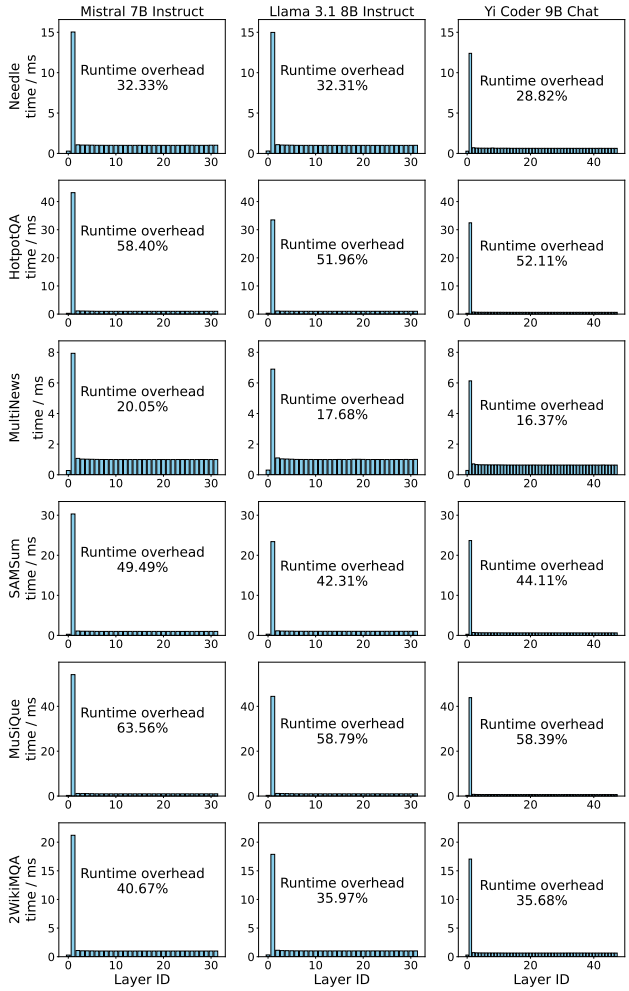

Figure 10. TTFT breakdown of CacheBlend-15.

A. Runtime Overhead of CacheBlend

We evaluate the runtime overhead of CacheBlend-15, using the synchronous workloads described in Section 5. Figure 10 presents the average time-to-first-token (*TTFT*) across 200 requests, with a breakdown of computational costs. The second transformer layer—where 15% of tokens for recomputation are dynamically selected—contributes $16.37\% - 63.56\%$ of the total *TTFT*. This finding underscores the substantial overhead introduced by dynamic sparsity in CacheBlend. In contrast, static sparsity, which predefines the recomputed tokens, significantly reduces this overhead, as detailed in Section 4.

B. Implementation Details

We implement `KVCache`[9] based on vLLM's original memory management and prefix caching subsystem with the following changes. First, we add a cache-ID-based indexing mechanism using the sequence group ID as the cache ID. Second, since the original vLLM manages historical KV cache residing in HBM only, we extend it to include DRAM and local filesystem, akin to Mooncake (Qin et al., 2025). Third, we modify the scheduler to retain block tables and memory for PIC compile requests. Fourth, we also implement helper APIs that allow users to manage the lifecycle of KV cache, such as `expire_cache(cache_id)`.

We implement `KVCompile` as a standalone module that handles CC APIs that are similar to those in Kimi (kim) and Gemini (gem, a). `KVCompile` forwards a request to `Regular Prefill` with maximum generation token set to 0.

We implement `KVLink` as a parallel module of the `Regular Prefill`. First, we adapt the model architecture to support masked attention across tokens scattered in different positions. Second, we modify the attention backends to handle data placement, movement, and the computational steps required by PIC algorithms, to ensure efficient recomputation.

[9] As building a highly efficient `KVCache` is not the core focus of this paper, we build a minimal working system.