

On-device Semantic Selection Made Low Latency and Memory Efficient with Monolithic Forwarding

Jiahao Zhou[†] Chengliang Lin[†] Dingji Li[‡] Mingkai Dong[†] Haibo Chen[†]

[†]Shanghai Jiao Tong University [‡]Huawei

Abstract

Semantic top- K selection with cross-encoder rerankers underpins on-device AI services, such as retrieval-augmented generation, agent memory, and personalized recommendation. However, its latency and memory demands dominate end-to-end budgets on edge hardware. Revisiting the objective of top- K selection, we reveal that only *relative rankings* matter, not exact per-candidate scores. We further observe *sequence-level sparsity*: relative rankings progressively stabilize in intermediate layers, enabling early pruning prior to completing full inference.

Building on this insight, we propose *monolithic forwarding* and develop a training-free inference system, PRISM. By maintaining a global view of all candidates, it reduces latency through progressive cluster pruning. It also bounds peak memory usage by strategically overlapping I/O with computation via overlapped layer streaming and chunked execution. We evaluate PRISM against state-of-the-art baselines on rerankers from 0.6 B to 8 B parameters across Apple M2 and RTX 5070. PRISM consistently reduces latency by up to 89.2% and peak memory by up to 91.3% in microbenchmarks, without compromising precision. Across three real-world on-device AI applications, PRISM lowers latency by 11.6%–51.0% and peak memory by 18.6%–77.8%, demonstrating substantial improvements in efficiency and deployability.

CCS Concepts: • Information systems → Retrieval models and ranking; • Computer systems organization → Embedded systems.

Keywords: Semantic Selection, Edge Computing, Inference Optimization, Reranking

ACM Reference Format:

Jiahao Zhou, Chengliang Lin, Dingji Li, Mingkai Dong, and Haibo Chen. 2026. On-device Semantic Selection Made Low Latency and Memory Efficient with Monolithic Forwarding. In *European Conference on Computer Systems (EUROSYS '26)*, April 27–30, 2026, Edinburgh, Scotland Uk. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/3767295.3803572>



This work is licensed under a Creative Commons Attribution 4.0 International License.

EUROSYS '26, April 27–30, 2026, Edinburgh, Scotland Uk

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2212-7/26/04

<https://doi.org/10.1145/3767295.3803572>

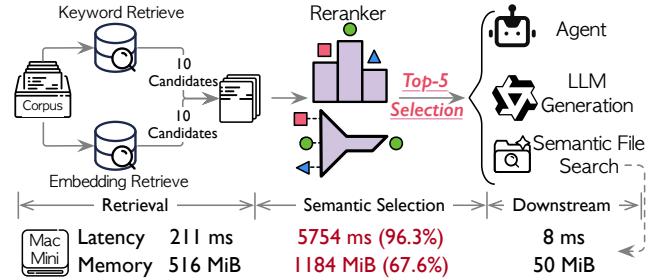


Figure 1. Typical on-device top- K selection pipeline and per-stage cost. Per-stage latency and peak memory under a representative on-device semantic file search is reported.

1 Introduction

Semantic selection—the process of identifying the most semantically relevant top- K items from a candidate pool—serves as a core component in a wide range of on-device artificial intelligence (AI) services running on commodity PCs and laptops, such as retrieval-augmented generation (RAG) [43, 44, 61, 65], AI agent memory [20, 37, 63, 68], and personalized recommendation [9, 25]. For example, in a typical semantic file search scenario illustrated in Figure 1, *keyword retrieval* and *embedding retrieval* select ten candidates respectively from a large corpus; a *reranker* then semantically selects the final top- K items to feed downstream components such as a large language model (LLM), a UI agent, or the user. The precision of this top- K selection directly governs downstream task quality and helps mitigate hallucinations [43].

To meet the stringent precision requirements, *cross-encoder rerankers* [2, 4, 13, 70] have emerged as the state-of-the-art technique for semantic selection [12, 21, 31]. Cross-encoder rerankers are transformers that process a query-candidate pair as a single, combined input and output their relevance score. Jointly processing the query-candidate pair enables a deep, token-level analysis of their relevance through attention mechanism [58], delivering substantial precision gains of 15% – 25% [26, 57] over traditional bi-encoders [64] that process query and candidate separately. However, this superior precision comes at a prohibitive resource cost for on-device deployment. For example, selecting top-5 from 20 candidates with a 0.6 B cross-encoder reranker incurs 5,754 ms latency and 1,184 MB peak memory consumption on a Mac Mini. The reranker alone contributes 96.3% of the latency and 67.6% of the memory footprint in the overall semantic file search pipeline, pushing the resource usage beyond typical mobile OS budgets (e.g., HarmonyOS guidance [10]) and severely harming user experience.

Unfortunately, existing optimizations fall short for reducing the high latency and large memory consumption of on-device rerankers due to fundamental workload mismatch. Decoding-centric optimizations [41, 56, 67, 69] provide limited benefit for rerankers, which are prefill-only. Methods based on *token-level sparsity* [30, 42, 46] are primarily designed for very long contexts (e.g., tens of thousands of tokens) and offer little advantage for the short, information-dense inputs typical of rerankers (e.g., up to 512 tokens). Other approaches, such as *model compression* [47] or early exit [66], require retraining and task-specific tuning, complicating the deployment. Consequently, rerankers’ high latency and resource demands severely limit the efficiency and applicability of on-device AI services.

In this paper, we revisit the core objective of semantic top- K selection: *instead of computing absolute scores for all candidates, it suffices to identify the top- K items via relative rankings*. By inspecting intermediate scores of all candidates across layers (Figure 2), we discover **sequence-level sparsity**: relative rankings of candidates progressively stabilize and converge to the final rankings in intermediate layers, and the stabilized candidates can be identified via clustering. This insight enables us to prune many candidates before full computation without sacrificing precision. Moreover, for memory, we observe **overlap window**: the computation time of current layer is sufficient to fully overlap with loading the next layer’s weights from SSD, enabling model weights offloading to SSD without latency overhead.

Driven by these insights, we propose **monolithic forwarding**, a novel inference paradigm tailored for cross-encoder rerankers. It processes all candidates as a single, monolithic batch, maintaining a global view throughout computation, rather than splitting them into isolated batches as in conventional systems.¹ This design offers opportunities to reduce inference latency via layer-wise pruning and lower memory footprint by on-demand weight loading.

Based on monolithic forwarding, we build PRISM, a training-free system for low-latency and memory-efficient semantic selection. PRISM introduces *progressive cluster pruning* to prune candidates without sacrificing precision. Progressive cluster pruning enables a dynamic, three-way routing strategy: dropping hopeless candidates, accepting the winners, and continuing computation only on the remaining uncertain candidates. For memory, PRISM proposes *overlapped layer streaming* to lower the memory footprint of layer weights, and *embedding table caching* to reduce the memory consumption of the embedding layer. Moreover, monolithic forwarding incurs the challenge of the memory explosion of the intermediate tensor. PRISM solves this challenge by *chunked execution*, which partitions candidates into chunks that fully utilize compute resources while keeping intermediate tensors within memory limits.

¹Vanilla systems often split inputs into multiple batches to balance computation and memory.

We implement and integrate all these techniques in PRISM atop HuggingFace Transformers [34] and evaluate various reranking models (0.6 B to 8 B) across platforms equipped with an RTX 5070 Laptop GPU and an Apple M2 SoC. We evaluate PRISM in microbenchmarks and three real-world on-device AI applications, comparing against HuggingFace Transformers [34] as the baseline. Our microbenchmark results show that PRISM consistently reduces latency by up to 89.2% while lowering peak memory by up to 91.3%. In real-world applications, PRISM delivers latency reductions of 11.6%–51.0% and peak memory savings of 18.6%–77.8%. These significant improvements in both latency and memory substantially advance the practicality for on-device AI applications.

In summary, this paper makes the following contributions:

- We identify top- K selection as a critical performance bottleneck in on-device AI applications and systematically analyze why existing optimizations fail for this workload.
- We reveal two underexploited opportunities: sequence-level sparsity and I/O-computation overlap, and demonstrate the unique challenges of leveraging them for top- K selection workloads.
- We design and implement PRISM, a training-free system with progressive cluster pruning, overlapped layer streaming, chunked execution, and embedding table caching for practical cross-encoder deployment on edge devices.
- Extensive experiments demonstrate that PRISM achieves up to 89.2% latency reduction and 91.3% memory reduction while maintaining precision across diverse benchmarks and device configurations.

2 Background and Motivation

2.1 The Bi-Encoder vs. Cross-Encoder Trade-off

Semantic top- K selection, or simply top- K selection, is a foundational component in modern on-device AI services, such as retrieval-augmented generation (RAG) [43, 44, 61, 65], AI agent memory [20, 37, 63, 68], and personalized recommendation [9, 25]. The reranking stage of this process is critical, as its precision directly dictates the quality of information provided to the user or consumed by an LLM.

Two primary architectures dominate the reranking landscape: *bi-encoders* and *cross-encoders*. Bi-encoders encode the query and each candidate document into separate, fixed-size embeddings, and then rank candidates using similarity scores between these embeddings [38, 71]. While efficient, this separation imposes a fundamental precision ceiling: because query and candidate representations are generated independently, the model cannot capture fine-grained, token-level interactions critical for precise relevance estimation [57, 64]. Late-interaction models like ColBERT [40] attempt to bridge this gap, but incur significant storage overhead by requiring per-token embeddings [60] and are not generally adopted for instruction-following or reasoning-intensive tasks [64].

In contrast, cross-encoders have emerged as the gold standard for reranking precision [12, 21, 31]. Existing cross-encoders can be categorized into two mainstream architectures: Encoder-only transformers with bidirectional self-attention (e.g., BERT-style) [3, 27], and Decoder-only transformers with causal self-attention (e.g., GPT-style) adapted for scoring [4, 18, 70]. Both architectures concatenate the query and a candidate into a single input sequence, process it through multiple transformer layers, and finally compute a scalar relevance score by applying a lightweight classifier head to the final hidden states. This joint encoding enables deep token-level attention between query and candidate across every transformer layer, capturing subtle semantic nuances and dependencies that bi-encoders inherently miss. Hence, cross-encoders yield substantial and consistent gains in retrieval performance [26, 57]. For on-device applications where output quality is critical, the superior precision of cross-encoders makes them indispensable.

2.2 The Prohibitive On-Device Cost

Despite their superior precision, cross-encoders face fundamental challenges on edge devices: their inference requires a full, compute-intensive forward pass for each query-candidate pair, a workload fundamentally misaligned with resource-constrained environments.

Compute-bound Latency. A cross-encoder’s latency is dominated by matrix multiplications in its transformer blocks. For an input sequence of length L (query + candidate) and a hidden dimension D , the complexity of self-attention and the feed-forward network (FFN) scales as $O(L^2 \cdot D)$ and $O(L \cdot D^2)$, respectively. Because this expensive computation must be executed independently for each of the N candidates, the total latency scales linearly with N . On edge devices with limited floating-point compute capabilities, this linear scaling results in delays that significantly degrade the user experience.

Memory Footprint. Cross-encoders’ memory demands present an equally formidable challenge, comprising two main components:

- **Model Weights.** Even small sized rerankers (e.g., 0.6 B parameters) require several gigabytes of storage. These weights are dominated by the stacked Transformer layers. For instance, in Qwen3-Reranker-0.6B, 28 Transformer layers (15 M weights each layer) account for over 70% of the weight memory. Loading these weights into the limited DRAM or VRAM of an edge device consumes a substantial portion of the system’s memory budget.
- **Intermediate Tensors.** During inference, another memory issue comes from transient intermediate tensors (e.g., for query, key, value projections, attention scores, and FFN outputs). The peak memory consumption from these tensors scales with the number of candidates being processed in a batch, and can easily exceed the memory budget of an application.

2.3 Mismatch with Existing LLM Optimizations

While recent advances in LLM inference have introduced numerous optimization techniques, a systematic analysis reveals that they are poorly aligned with the unique workload characteristics of on-device cross-encoder reranking.

Decoding-centric Optimizations. A large body of research focuses on the autoregressive *decoding* phase of LLMs. Techniques such as speculative decoding and advanced KV-cache management are designed to accelerate the memory-bound, token-by-token generation process [14, 41, 56, 67]. Cross-encoder reranking, however, is a *prefill-only* workload: it performs a single, compute-bound forward pass to produce a score for each query-candidate pair. As there is no decoding phase, these optimizations are largely inapplicable.

Long-context Optimizations. Another major line of research focuses on efficient processing of extremely long input sequences. These methods often exploit token-level sparsity, such as sparse attention or dynamic token pruning, under the assumption that long contexts contain substantial informational redundancy [30, 42]. In contrast, reranking inputs are short and information-dense, typically consisting of a concise query and a highly relevant document chunk. In this setting, token sparsity offers negligible benefit and may even compromise the fine-grained relevance judgments that cross-encoders are designed to capture.

Training-based Compression. Techniques such as model pruning, quantization-aware training (QAT), and early exit can reduce inference costs, but they require expensive re-training or fine-tuning to preserve model precision [19, 47, 54, 66]. The associated cost and complexity pose a substantial barrier to rapid and reliable on-device deployment.

Post-training Quantization. While 4-bit weight quantization is a common baseline optimization [29, 45], achieving practical speedups through more aggressive sub-4-bit quantization on prefill workloads remains an open challenge. Beyond precision degradation, most edge devices also lack the specialized hardware and kernel support required for high-throughput sub-4-bit matrix multiplication, limiting real-world performance gains [50, 52].

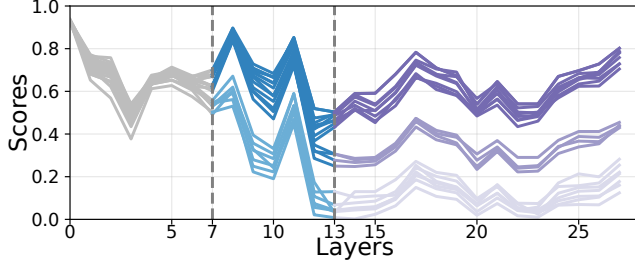
Taken together, these analyses reveal a clear research gap: a training-free, workload-specific optimization paradigm is required to make high-precision cross-encoder reranking feasible on edge devices.

3 PRISM Overview

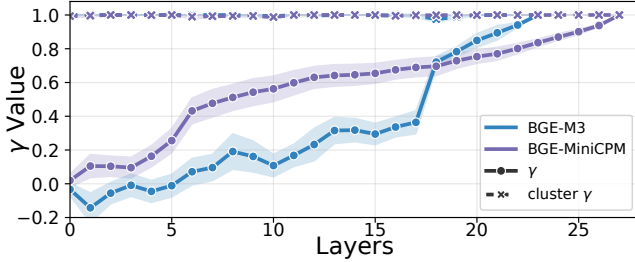
In this section, we introduce PRISM, a training-free inference system for cross-encoder rerankers on edge devices.

3.1 Key Insight: Sequence-level Sparsity

Our approach stems from reconsidering the fundamental objective of top- K selection: *identifying relative rankings among candidates, rather than computing precise absolute scores*. This insight motivates us to examine how candidate rankings evolve across transformer layers from a global perspective.



(a) **Score evolution across layers reveals sequence-level sparsity.** Each line presents the score evolution of a candidate (20 in total) on BGE-MiniCPM model. Candidates progressively diverge into statistically distinct clusters as they pass through layers, and relative rankings progressively stabilize and converge to the final rankings in intermediate layers.



(b) **Generality of sequence-level sparsity.** On 18 datasets and all mainstream model architectures, Goodman and Kruskal’s γ rises as layers deepen and cluster γ is consistently close to 1.0 across layers, indicating relative rankings stabilize early and the stabilized candidates can be identified by clustering and safely pruned without sacrificing precision.

Figure 2. Key observation of sequence-level sparsity.

In Figure 2(a), we compute candidate scores at each layer using the model’s original classifier and show score evolution across layers. We observe that candidate scores progressively diverge into statistically distinct clusters as they pass through layers. While the relative rankings of candidates within the same cluster remain in flux (lines of the same color), the relative rankings *between* different clusters stabilize early and converge to the final rankings (lines of different colors). This motivates our insight of **sequence-level sparsity**: relative rankings of candidates progressively stabilize and converge to the final rankings in intermediate layers, and the stabilized candidates can be identified by clustering and safely pruned without sacrificing precision.

We validate the generality of sequence-level sparsity across 18 datasets (detailed in §6.1) and all mainstream model architectures (encoder- and decoder-only). We quantify ranking convergence using Goodman and Kruskal’s γ [22], computed over candidate pairs by counting those whose relative order is preserved between an intermediate and the final layer (concordant, N_c) versus reversed (discordant, N_d): $\gamma = \frac{N_c - N_d}{N_c + N_d}$. To directly measure the stability of inter-cluster rankings, we further define *cluster γ* , which restricts the computation to only candidate pairs from different clusters. Figure 2(b) shows that the standard γ consistently rises as layers deepen (lines with circular markers), confirming that relative rankings progressively converge. More importantly, the cluster γ

remains consistently close to 1.0 across all layers (lines with cross markers), strongly validating that stabilized candidates can be identified by clustering and safely pruned without sacrificing precision.

We attribute the sequence-level sparsity to model’s coarse-to-fine understanding [35, 53]. Early layers capture broad semantic and separate candidates with clear relevance differences into distinct clusters; as layers deepen, these coarse clusters progressively split into finer ones to resolve subtle distinctions among highly similar candidates. As a result, inter-cluster rankings stabilize early, enabling us to safely prune while preserving precision.

3.2 Key Insight: Overlap Window

The prefill-only nature of reranking creates an opportunity for aggressive memory optimization. Unlike autoregressive generation that processes tokens iteratively, reranking performs a single forward pass over all tokens of a candidate, yielding high arithmetic intensity per layer. Meanwhile, although edge devices offer limited compute throughput, SSDs sustain high bandwidth. The combination of compute-heavy layers and fast storage opens an **overlap window**: the computation time of the current layer is sufficient to fully overlap with loading the next layer’s weights from SSD, enabling model weights offloading to SSDs without latency overhead.

3.3 Monolithic Forwarding

Motivated by these insights, we propose **monolithic forwarding**, a novel execution paradigm for cross-encoder rerankers. Instead of processing candidates in isolated batches as conventional systems do, monolithic forwarding consolidates all candidates into a single, unified batch that progresses through layers together.

This paradigm unlocks two critical opportunities. First, maintaining a global view of all candidates throughout execution enables dynamic pruning based on relative rankings at each layer. We thus can eliminate candidates that have no chance of reaching the top- K , reducing computation as the forward pass proceeds. Second, the large, consolidated batch creates substantial computation windows at each layer, sufficient to completely overlap I/O latency of loading the next layer’s weights from disk. This allows us to keep only two layers in memory, reducing the memory footprint of model weights.

3.4 System Overview of PRISM

PRISM realizes the main idea of monolithic forwarding through several complementary techniques as illustrated in Figure 3.

Before stepping forward to layer $i + 1$, PRISM leverages **progressive cluster pruning** (§4.1) to prune candidates. It first applies a clustering-based analysis to layer i ’s output scores using statistical properties of inter-cluster separation rather than absolute score gaps, to determine whether a stable relative ranking has emerged. After determining whether

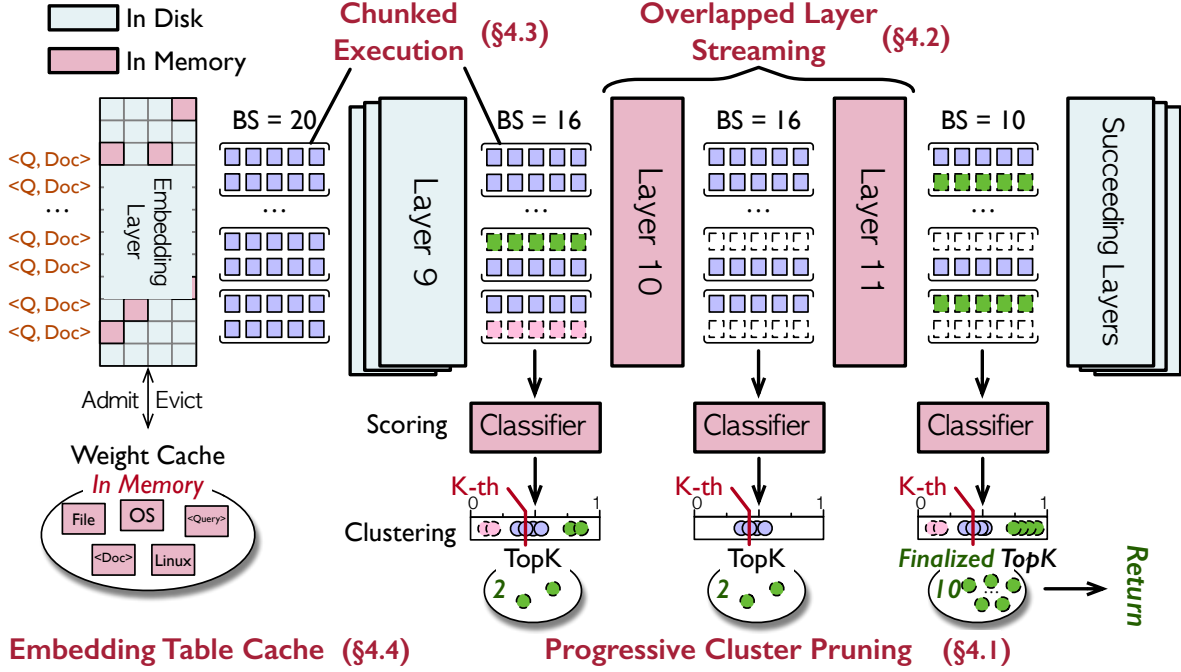


Figure 3. PRISM overview with a working example.

a stable relative ranking has emerged, it routes candidates to *selected*, *dropped*, or *deferred*.

Once layer $i+1$ starts to execute, PRISM utilizes **overlapped layer streaming** (§4.2) to minimize the memory footprint of model weights [33]. It immediately releases the weights of layer i from memory and starts prefetching the weights of layer $i+2$ from storage.

During the execution of layer $i+1$, consolidating all candidates into a single batch inflates intermediate tensor sizes, significantly increasing peak memory consumption. To address this challenge, PRISM adopts **chunked execution** (§4.3). It partitions the monolithic batch into smaller chunks and executes them sequentially within each layer, significantly reducing peak memory usage by only keeping one chunk's intermediate tensors in memory. Simultaneously, it provides sufficient computation window to overlap I/O of loading layer $i+2$'s weights. For extreme memory constraints, PRISM further supports dynamic offloading of hidden states.

Additionally, PRISM complements these techniques with **embedding table caching** (§4.4) that exploits token distribution sparsity before layer 0, which substantially lowers the memory consumption of the embedding layer (Figure 16).

4 Detailed Design

4.1 Progressive Cluster Pruning

To reduce the latency while preserving precision, we propose progressive cluster pruning based on sequence-level sparsity. In each layer, we calculate the candidate scores, which fall into clusters. Based on whether a cluster is in or

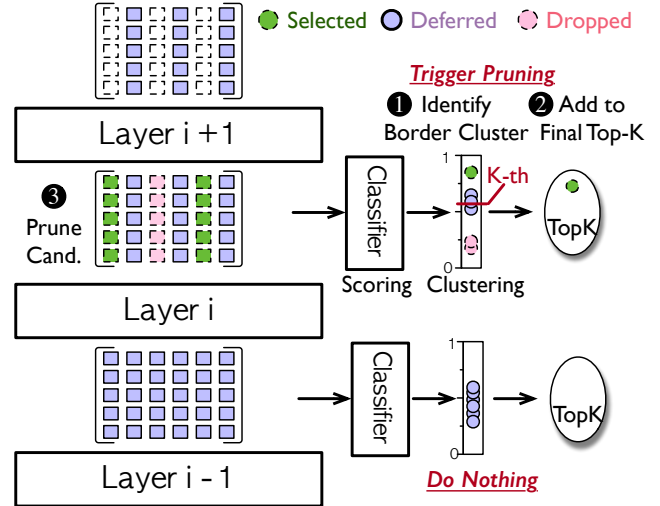


Figure 4. Progressive cluster pruning. Once the score dispersion exceeds the threshold during the layer progressing, we partition candidates into selected, deferred, and dropped clusters. Only the deferred candidates are retained and others are pruned.

out of the top-K set, we decide the fate of the candidates inside: either continue computation or prune them, greatly reducing computation and thus lowering latency.

Figure 4 illustrates the progressive cluster pruning. To realize this insight, we first identify if the stable relative ranking occurs in a layer. In details, we employ the model's classifier to calculate the current scores of candidates. We compute the coefficient of variation (CV) [23] of scores to quantify their dispersion: $CV = \frac{\text{std}(\text{scores})}{\text{mean}(\text{scores})}$. For layer $i-1$,

the CV does not exceed a predefined threshold (referred to as the dispersion threshold), we consider a stable relative ranking has not yet emerged. We do nothing and continue the forwarding of layer i . For layer i , the CV exceeds the dispersion threshold, we consider a stable relative ranking emerges and triggers the core clustering and pruning logic.

At the beginning, we perform K-Means [32] on CPU to partition candidates into clusters with negligible latency overhead (~ 1 ms). The pruning logic operates at a cluster granularity. The process pivots on identifying the boundary cluster, which contains the K -th ranked candidate. This boundary acts as a clear demarcation line, allowing us to classify all candidates into three groups: selected, deferred, and dropped. Selected candidates are those in clusters with scores higher than the boundary cluster’s. The selected candidates are safely included in the final top- K set and their computation ceases. Conversely, dropped candidates are those in lower-scoring clusters. The dropped candidates are pruned, as they have no chance of reaching the top- K . Consequently, only a small subset of candidates within the boundary cluster are deferred for continued processing in subsequent layers. Progressive cluster pruning allows the model to cease computation for the vast majority of candidates when stable relative rankings emerge, and the forward pass terminates completely if the number of deferred candidates is equal to the remaining top- K slots to be filled.

It’s worth mentioning that the dispersion threshold provides direct and intuitive control over the precision-latency trade-off. A lower threshold enables more aggressive pruning, maximizing performance at a potential precision cost, whereas a higher value preserves precision by being more conservative. Crucially, our system allows users to either manually tune this threshold or simply specify a minimum precision target. In the latter mode, our system automatically calibrates the threshold to the lowest possible value that meets the constraint, thereby maximizing performance under the given requirement. In detail, we sample requests at a frequency and log their top- K results. When the device is idle, we re-execute full inference (without pruning) to obtain the ground truth. We then compute the precision of the sampled requests against the ground truth. If the precision falls below the target precision, we raise the dispersion threshold for precision; otherwise, we lower it for performance.

In summary, progressive cluster pruning effectively reduces the latency while maintaining precision, and we provide the system ability to navigate the precision-latency spectrum.

4.2 Overlapped Layer Streaming

To reduce the memory footprint of model weights and overlap the I/O latency of weight loading, we introduce the overlapped layer streaming technique. This technique maintains at most two layers’ weights in memory — the current layer in

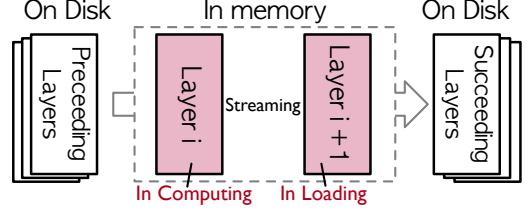


Figure 5. Overlapped layer streaming. Throughout the inference, we only reserve two memory buffers to stream model weights continuously. When layer i resident in one buffer is computing, the next layer $i+1$ is prefetched from storage into the other buffer to perfectly overlap I/O. Once layer i finishes computing, its buffer is released and recycled to prefetch layer $i+2$, sustaining a seamless streaming of weights that completely hides the load latency.

computing and the next layer in prefetching — and overlaps weight prefetching with computation to hide the I/O latency.

The core technique are depicted in Figure 5. While forwarding the layer i , we concurrently prefetch the weights for layer $i+1$ from disk into a dedicated memory buffer. Thanks to the key idea of monolithic forwarding, upon the completion of layer i ’s computation, the layer $i+1$ has already been loaded into memory and ready for forwarding. At this time, the weights of layer i are obsolete and immediately released from its memory buffer. This vacated buffer is then recycled for the prefetching of layer $i+2$. Now, we stream the model weights: we compute layer $i+1$ while prefetching layer $i+2$. Throughout, only two pre-allocated memory buffers are needed to hold weights in streaming manner, significantly reducing the memory footprint. Therefore, we perfectly overlap the computation of the current layer with the I/O of the next layer and hence incur no latency penalty.

In summary, the overlapped layer streaming minimizes the memory footprint of model weights with no latency penalty.

4.3 Chunked Execution

Monolithic forwarding incurs the challenge of the memory explosion of the intermediate tensor. Consolidating all candidates inflates intermediate tensor sizes proportionally. For 60 candidates with 512-token sequences on a 0.6 B model, intermediate tensors per layer increase peak memory by 473 MB (see Figure 16), which can cause out-of-memory issues on devices with strict memory constraints. This tension between the batch size needed for I/O overlap and the memory constraints of edge platforms must be carefully balanced.

To solve this challenge, we propose chunked execution. Our key observation is that I/O overlapping only depends on the total computation time of a layer, not necessitating executing all computations simultaneously. Thus, we split the monolithic batch into chunks, maintaining I/O overlap while keeping only one chunk’s intermediate tensors in memory, reducing peak memory usage.

As shown in Figure 6, the layer’s forward pass is executed sequentially on these chunks. Through this approach, we

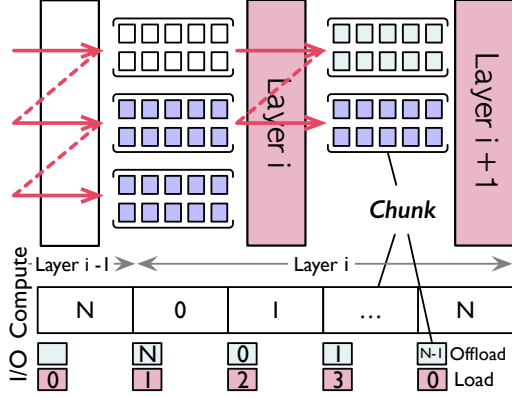


Figure 6. Chunked execution. For solving the memory explosion, we partition the monolithic batch into smaller chunks and execute their forwarding sequentially. To enable scalability with massive candidates, we support dynamic offloading of hidden states.

only need to allocate memory for the intermediate tensors of a single chunk and the hidden states of all chunks. This maintains minimal memory footprint compared to processing the monolithic batch at once. Notably, to fully exploit hardware computational capabilities, the chunk size has a lower bound. We dynamically determine the optimal chunk size considering device compute capability, model size, and input sequence length.

While chunked execution effectively manages intermediate tensors, the aggregated hidden states can become a memory bottleneck when candidate number scales. To address this, we support dynamic offloading of the hidden states. In the lower part of Figure 6, while computing the current chunk, we concurrently offload the completed hidden states from the previous chunk and prefetch the hidden states needed for the next chunk. This approach ensures at most three chunks reside in memory: one being computed, one being offloading, and one being prefetched. By bounding the memory footprint required for hidden states, we enable the scalability with massive candidates.

4.4 Embedding Table Caching

After the overlapped layer streaming and chunked execution significantly reduce the memory footprint of transformer layers, the embedding layer becomes the new dominant memory bottleneck. For further optimization, we propose embedding table cache that exploits the sparsity of the embedding layer.

Consider our optimized Qwen3-Reranker-0.6B, the active layers consume only 60 MB while the embedding table requires 296 MB, accounting for over 83% of the total memory footprint. To address this, we observed that the activation of the embedding layer weights is highly sparse. For the same 0.6 B model with a vocabulary of 151,669 tokens, a typical reranking task involving 20 documents with 512 sequence length accesses 10,240 unique tokens at most, merely 6.75% of the vocabulary. It indicates the activation of embedding layer is highly sparse and inspires our embedding table caching.

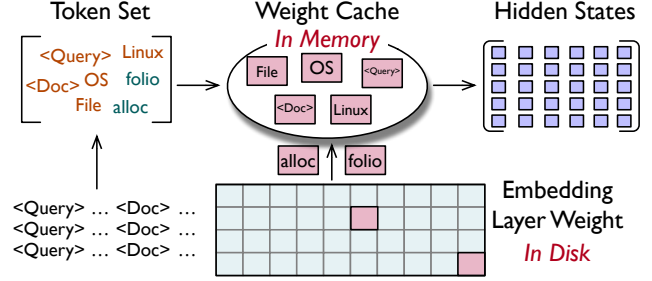


Figure 7. Embedding table caching. Based on the high sparsity of the embedding layer, we employ a small LRU cache to store a subset of the weights of the embedding layer. It significantly reduces the memory footprint with only a negligible latency.

Figure 7 illustrates our design. The core component is a small LRU cache residing in memory, which stores a subset of the embedding weights. During inference, we first collect the set of unique input tokens and lookup the cache for the activated weights of embedding layers. For any cache miss, the system triggers a synchronous read operation to fetch the missing weights from the disk and load them into cache. In practice, we set the cache size to only 10% of the vocabulary size, which significantly reduces memory consumption while maintaining high hit rates due to the skewed token distribution in natural language [72]. Besides, the cache miss incurs negligible latency due to the small data volume of the sparse activated weights and the effective LRU cache (see the ablation study in §6.4).

In summary, embedding table caching technique employs a small, in-memory LRU cache to hold only the active weights of embedding table, drastically reducing the memory footprint of the full embedding table.

4.5 A Working Example

Figure 3 illustrates our system’s end-to-end workflow using a typical re-ranking task as an example: identifying the top-10 most relevant documents from 20 candidates. Before forwarding, we consolidate all candidates into one monolithic batch. The forwarding begins at the embedding layer, where input tokens are converted into hidden states. To manage memory, we maintain a fixed-size cache for embedding weights. We identify unique input tokens not present in the cache, load them synchronously from disk, and evict existing entries using an LRU policy if the cache capacity is exceeded.

Subsequently, the hidden states are partitioned into chunks and processed through Transformer layers sequentially, where our system keeps an overlapped layer streaming across Transformer layers. We partition one monolithic batch with 20 candidates into ten chunks with two candidates and do the forwarding sequentially for each layer. During the forwarding of one layer, a dedicated I/O process begins prefetching the weights of the next layer from the disk in parallel. Upon the completion of the forward pass of a layer, its weights are immediately released from the memory. Assuming we have just completed forwarding of layer 9, we first immediately

deallocate its weights from the memory and the weights of layer 10 has already been loaded into memory for the subsequent forwarding.

Critically, before forwarding each Transformer layer, we perform a cluster-based pruning check to prune candidates whose final ranks are already evident. Continuing our example, before executing layer 10, we compute provisional scores for all 20 active candidates with the classifier layer. Their CV exceeds a predefined threshold, indicating significant score divergence and triggering a K-Means clustering to partition candidates into multiple clusters. To classify the clusters, we first identify the pivotal cluster containing the K-th ranked candidate (the 10th in this case). The clusters with a mean score higher than that of the pivotal cluster are selected clusters. Two candidates within these clusters are selected into the final top-10 and exit subsequent forwarding. Conversely, those clusters with a lower mean score are the dropped clusters and all two candidates are dropped. In this instance, the process identifies two selected and two dropped candidates. Consequently, we prune these four and proceed to layer 10 with only the remaining 16 deferred candidates.

After layer 10 completes, the CV of these 16 candidates falls below the threshold, so no pruning occurs, and all proceed to layer 11. After executing layer 11, however, the score CV once again surpasses the threshold, triggering another clustering. This time, a terminal condition is met: the number of candidates in the final deferred cluster, when added to the number of already included candidates, precisely equals the target $K = 10$. The system therefore terminates inference immediately, returning the combined set as the final result.

5 Implementation

We implement PRISM in ~5 k lines of Python and ~1.7 k lines of C. Our implementation builds on HuggingFace Transformers v4.52.4 [34] and HuggingFace Accelerate v1.6.0 [33].

To achieve high-performance, parallel I/O, we incorporate the following implementation optimizations. First, to bypass Python’s Global Interpreter Lock (GIL) [5, 24] and thus parallelize computation and I/O operations, we spawn a computation process and an I/O process. The two processes communicate with low latency via a shared memory buffer managed by Pytorch Multiprocessing [51]. Second, to saturate disk bandwidth, the I/O process leverages Libuv [8] to perform high-throughput asynchronous disk I/O. Finally, we enable the CUDA Multi-Process Service (MPS) [49] to facilitate efficient GPU sharing between two processes, minimizing the context-switching overhead.

6 Evaluation

The goal of evaluation is to answer three key questions:

- **Latency Reduction.** Can PRISM significantly reduce latency while preserving model precision?
- **Memory Efficiency.** Can PRISM substantially lower memory footprint without introducing latency overhead?

Table 1. The evaluated models.

Name	Model Size	Architecture
Qwen3-Reranker-0.6B	0.6 B	Decoder-only
Qwen3-Reranker-4B	4 B	Decoder-only
Qwen3-Reranker-8B	8 B	Decoder-only
Bge-Reranker-v2-MiniCPM	2 B	Decoder-only
Bge-Reranker-v2-M3	0.6 B	Encoder-only

- **Ablation Study.** What is the individual contribution of each of the four proposed techniques to the overall performance improvement?

6.1 Experiment Setup

Hardware configuration. Experiments are conducted on two distinct platforms, representing both unified and non-unified memory architectures:

- **NVIDIA Platform.** A laptop with an Intel(R) Ultra9-275HX processor, 32 GiB memory, NVIDIA RTX 5070 Laptop GPU with 8 GiB memory, and 1 TiB PCIe 4.0 SSD.
- **Apple Platform.** A Mac Mini with an Apple M2 SoC, 16 GiB unified memory, and 256 GiB PCIe 4.0 SSD.

Compared systems. We compare PRISM with these baselines in evaluation.

- **PRISM.** Our proposed system, which integrates all techniques: progressive cluster pruning, overlapped layer streaming, chunked execution, and embedding table caching.
- **HF.** The vanilla HuggingFace Transformers [34] with Pytorch backend. This baseline represents the standard, in-memory inference performance. We chose it over vLLM and SGLang due to its broader compatibility with diverse edge devices [15, 39].
- **HF Offload.** The vanilla HuggingFace Transformers [34] with the HuggingFace Accelerate [33] library’s disk offloading feature. All transformer layers are offloaded to disk and loaded right before execution.
- **HF Quant.** The state-of-the-art quantization method. We quantize the model in W4A16 with GPTQ [29].
- **PRISM Quant.** The state-of-the-art quantization method. We integrate our techniques with quantization techniques, which are orthogonal.

Models. As presented in Table 1, we evaluated a wide range of state-of-the-art models. These models vary in size from 0.6 B to 8 B and feature diverse architectures, from encoder-only (e.g., Bge-Reranker-v2-M3) to decoder-only (e.g., the Qwen3-Reranker series).

Workloads. We evaluate the compared systems in both microbenchmarks and real-world evaluations. In the microbenchmarks, we evaluate compared systems on 18 datasets: 15 datasets in BEIR benchmark [17, 57], LoTTE dataset [55], Wikipedia dataset [1] and CodeRAG dataset [62]. In real-world evaluations, we evaluate our system in three real-world scenarios, including RAG, Agent Memory, and LLM long context selection. The detailed descriptions are shown in Table 2.

Table 2. The description of real-world workloads.

Workload	Description
RAG	An on-device smart assistant that personalizes its model with user data. It combines vector and keyword searches, using a reranker to select the optimal final result.
Agent Memory [68]	An on-device agent leverages a reranker in its agent memory to cache actions, reducing costly model generations.
LLM Long Context Selection [36]	For on-device deployment of LLM handling extended contexts, a top- K selection mechanism is employed to identify the most related contextual segments, conforming to the model’s finite context window limitations.

Metrics. Our evaluation focuses on the following metrics:

- **Latency.** We measure the inference latency of the reranking models.
- **Precision.** We employ Precision@ K to evaluate the model precision. Precision@ K measures the ratio between the number of relevant items contained in the top- K results and K . When the ground truth is less than K , we take the ratio between the number of relevant items contained in the top- K and the number of ground truth.
- **Memory footprint.** We focus on both the mean and the peak memory footprint of the model in inference.

6.2 Microbenchmarks

In microbenchmarks, we extensively evaluate the latency, precision, and memory footprint of compared systems.

Latency and precision. Table 3 summarizes our latency and precision evaluation on five models across 18 datasets and two platforms, reporting the mean latency reduction and the maximum precision loss on Precision@ $K \in \{1, 5, 10\}$ out of 20 candidates compared to the baselines. Figure 8 further zooms in the latency and precision results on the Wikipedia dataset [1] across NVIDIA and Apple platforms. Each subplot corresponds to a specific model and top- K configuration, showing latency (left y-axis) and precision (right y-axis). Bars represent latency (purple: NVIDIA; pink: Apple), and the number atop each bar indicates the speedup relative to the HF Offload baseline. The line plot reports Precision@ K . As precision is platform-independent, both platforms share a single point. In Figure 8, we additionally plot PRISM/PRISM Quant under both low and high dispersion thresholds (see §4.1) to show the threshold’s configurability.

Overall, Table 3 shows that our systems deliver substantial mean latency reductions while preserving precision, strongly validating the effectiveness of our design. Quantitatively, our system’s benefits are significant. PRISM reduces latency up to 89.2% over HF Offload at Precision@1 on Bge-MiniCPM, while the maximum precision loss stays within -0.003. PRISM Quant also provides meaningful speedups over HF Quant (e.g., up to 72.2% at Precision@1), with the maximum precision loss bounded by -0.003. For larger models such as Qwen3-4B/8B, the HF baseline fails to run on our hardware platforms due to its large memory footprint (OOM). In contrast, our system enables low-latency inference for these

powerful models, further underscoring its practical utility and effectiveness.

Zooming in on the Wikipedia dataset, Figure 8 shows the detailed latency and precision on the compared systems and two platforms. PRISM consistently achieves the lowest latency with high precision, followed closely by PRISM Quant, which provides the next-best performance. Furthermore, the results highlight a clear trade-off configurable via the dispersion threshold: increasing the threshold from Low to High improves precision at the cost of a smaller latency reduction. Quantitatively, PRISM reduces latency up to 72% compared to HF and 88% compared to HF Offload, with no loss in precision. For the Qwen3-8B model, the PRISM with Low threshold setting abnormally increases precision significantly compared to the HF baselines. We attribute this to the overfitting of the Qwen3-8B model [70]; our low-threshold PRISM provides a regularizing effect by bypassing the later layers, thereby enhancing its generalization.

In summary, extensive latency microbenchmarks demonstrate our system achieves substantial latency reductions (up to 89.2%) while preserving precision. This effectiveness holds across a range of models, datasets, and hardware platforms. Crucially, our system enables low-latency inference on large models that are otherwise infeasible to run.

Memory Footprint. Figure 9 illustrates the inference memory footprint overtime of the compared systems across five different models. The benchmark was conducted on the NVIDIA platform with ranking top-10 out of 20 input candidates with an average sequence length of 500. The results on the Apple platform are similar to those on the NVIDIA platform, we do not elaborate further.

In each subfigure, the x-axis represents the timeline and the y-axis shows memory usage. Each line terminates upon inference completion, thus its length indicates the inference latency. We annotate the peak memory of PRISM in each subfigure and present the peak and average memory statistics in the table at the bottom right. To demonstrate the memory footprint of HF on Qwen3-4B and Qwen3-8B that cannot run in the NVIDIA platform due to the OOM error, we measure them on an NVIDIA A800 GPU. Therefore, the lengths of their corresponding lines do not represent valid latencies. Besides, we omit the curve for PRISM Quant because it nearly overlaps with that of PRISM, enhancing visual clarity.

Overall, our system achieves the lowest memory footprint among all baselines while simultaneously delivering the lowest latency. This result demonstrates that our technique enables models to run faster with substantially less memory, a dual benefit not offered by competing approaches. In contrast, other memory-saving baselines like HF Offload and HF Quant trade latency for lower memory. Quantitatively, our system reduces peak memory by $5.34\times - 11.45\times$ compared to HF, $1.34\times - 3.83\times$ compared to HF Offload, and $2.77\times - 4.83\times$ compared to Quant. This substantial memory

Model	System	Baseline	Precision@1		Precision@5		Precision@10	
			Lat. Reduction Range (Mean)	Prec. Loss Mean / Max	Lat. Reduction Range (Mean)	Prec. Loss Mean / Max	Lat. Reduction Range (Mean)	Prec. Loss Mean / Max
Qwen3 0.6B	PRISM	HF	10.5–53.9% (33.1%)	0.003 / -0.003	9.0–51.7% (32.9%)	0.002 / -0.003	8.5–52.3% (32.2%)	0.000 / -0.003
		HF Offload	32.2–67.4% (47.6%)		32.5–66.2% (47.6%)		32.0–65.8% (47.1%)	
	PRISM Quant	HF Quant	5.6–50.1% (20.3%)	0.001 / -0.001	5.7–48.0% (18.3%)	0.002 / -0.004	5.3–48.6% (19.6%)	0.000 / -0.004
Qwen3 4B	PRISM	HF	OOM	-0.001 / -0.003	OOM	-0.001 / -0.005	OOM	0.000 / -0.004
		HF Offload	12.8–65.7% (38.2%)		10.7–65.4% (37.7%)		6.2–64.1% (35.3%)	
	PRISM Quant	HF Quant	5.1–43.5% (18.7%)	0.000 / -0.001	5.2–41.8% (17.9%)	0.000 / -0.003	5.7–40.7% (17.1%)	0.000 / -0.004
Qwen3 8B	PRISM	HF	OOM	0.039 / -0.001	OOM	0.048 / -0.004	OOM	0.025 / -0.006
		HF Offload	23.2–80.7% (54.6%)		23.4–80.5% (53.6%)		23.5–80.1% (53.3%)	
	PRISM Quant	HF Quant	6.5–64.3% (36.8%)	0.040 / -0.006	7.1–62.6% (36.1%)	0.056 / -0.001	9.6–62.2% (36.0%)	0.031 / -0.008
Bge M3	PRISM	HF	8.1–41.6% (24.6%)	0.000 / -0.005	6.2–41.7% (22.4%)	0.000 / -0.004	4.5–43.2% (20.5%)	0.002 / -0.007
		HF Offload	52.4–80.0% (70.9%)		50.9–79.4% (70.1%)		49.9–79.7% (69.3%)	
	PRISM Quant	HF Quant	4.3–44.9% (25.6%)	0.002 / -0.006	1.3–41.6% (22.5%)	0.004 / -0.006	0.5–41.8% (21.1%)	0.003 / -0.006
Bge MiniCPM	PRISM	HF	9.2–72.8% (44.3%)	0.000 / -0.003	6.1–66.9% (35.5%)	-0.003 / -0.006	5.6–62.4% (29.6%)	-0.002 / -0.005
		HF Offload	13.7–89.2% (56.7%)		14.2–87.0% (50.7%)		15.4–84.3% (47.6%)	
	PRISM Quant	HF Quant	6.1–72.2% (36.7%)	0.000 / -0.003	3.2–66.8% (28.2%)	-0.002 / -0.006	2.3–59.1% (23.7%)	-0.002 / -0.006

Table 3. Summary of latency and precision evaluation on 5 models, 2 platforms, and 18 datasets. For each precision@K, we report our system’s mean latency reduction range (with mean) and mean/max precision loss compared to baselines across 18 datasets × 2 platforms.

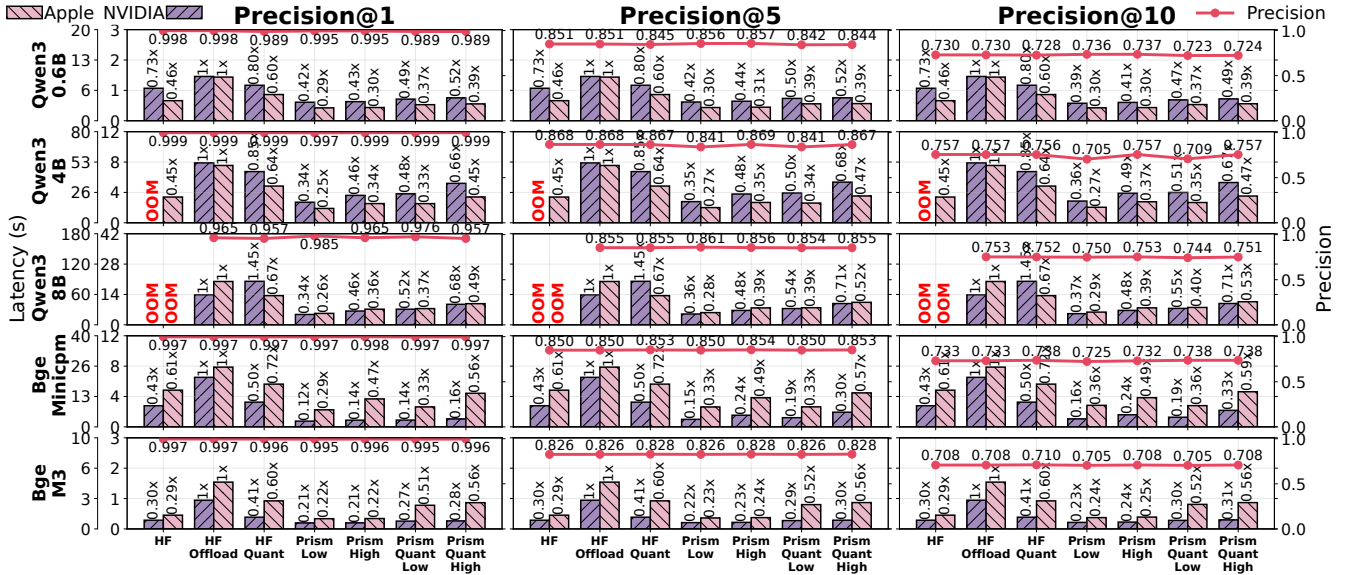


Figure 8. The detailed latency and precision evaluation on the Wikipedia dataset [1].

saving is particularly critical for resource-constrained edge devices, as it alleviates memory pressure and enables robust co-location of multiple applications.

In summary, our system substantially reduces both memory footprint and inference latency. This unique combination of memory efficiency and high performance is crucial for resource-constrained edge devices.

Tuning the latency-precision trade-off. Figure 10 demonstrates our system’s ability to navigate the latency-precision spectrum by tuning the dispersion threshold. We evaluate this capability on five models under Precision@1/5/10. As a general trend, increasing the threshold improves precision at the cost of higher latency. This result validates that our system can be configured to operate at different points

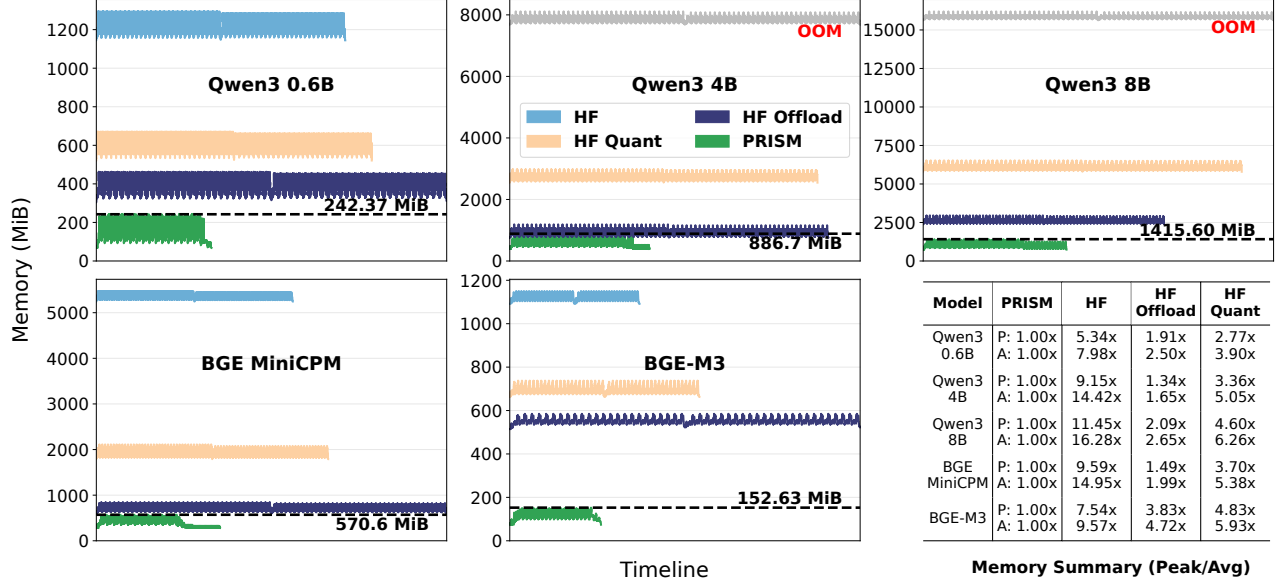


Figure 9. The memory footprint in microbenchmarks.

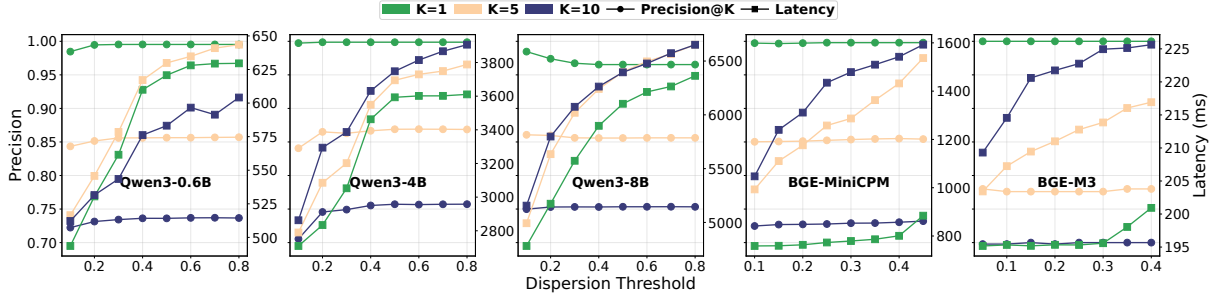


Figure 10. Tuning the latency and precision trade-off.

on the latency-precision spectrum to meet diverse application requirements. Notably, Qwen3-8B exhibits an inverse trend, achieving peak precision at the lowest threshold. We attribute this to the overfitting of this model, which is also shown in the official benchmark [11]; the pruning mechanism acts as a form of regularization, mitigating this effect.

6.3 Real-world Evaluations

Retrieval-Augmented Generation (RAG). We evaluate our system on the RAG-based personal assistant scenario. During an offline indexing phase, user’s personal data is converted into vector embeddings by the embedding model and stored in the vector database. When a user query arrives, we perform a hybrid search using both dense retrieval (i.e., vector search) and sparse retrieval (i.e., keyword search) to find the top-10 relevant documents, respectively. Then, a reranking model consolidates the results and selects the top-10 documents, which are then sent to an LLM for generation.

We employ the DiskANN-based Milvus [48, 59] as our vector database, the Qwen3-Embedding-0.6B for embedding. For reranking, we employ Qwen3-Reranker-0.6B on the Apple platform and Bge-Minicmpm on the NVIDIA platform.

For generation, we deploy a Qwen3-32B model on a server with two NVIDIA A800 GPU.

Figure 11(a) compares the latency and precision of HF and PRISM. Our system achieves significant performance gains, reducing latency by 51% on NVIDIA platform and 31% on Apple platform, respectively. Crucially, these improvements come with almost no loss in model precision. Figure 11(b) and Figure 11(c) shows the memory footprint of HF and PRISM. Our system also substantially lowers peak memory by up to 77.8% and average memory by 92.3%. This large reduction in average memory stems from the aggressive memory optimization of the reranking phase, which dominates the overall execution time.

Agent Memory (AM). We evaluate our system in an agent memory application [68]. This application optimizes GUI-based agent by caching past successful action trajectories to bypass expensive and redundant Vision-Language Model (VLM) [7] inference. The core of the agent memory lies in selecting the most semantic relevant trajectories [6], which is performed by a reranker. We employ Qwen3-Reranker-0.6B reranker and evaluate our system in the NVIDIA platform, the VLM is serving on two NVIDIA A800 server. The test

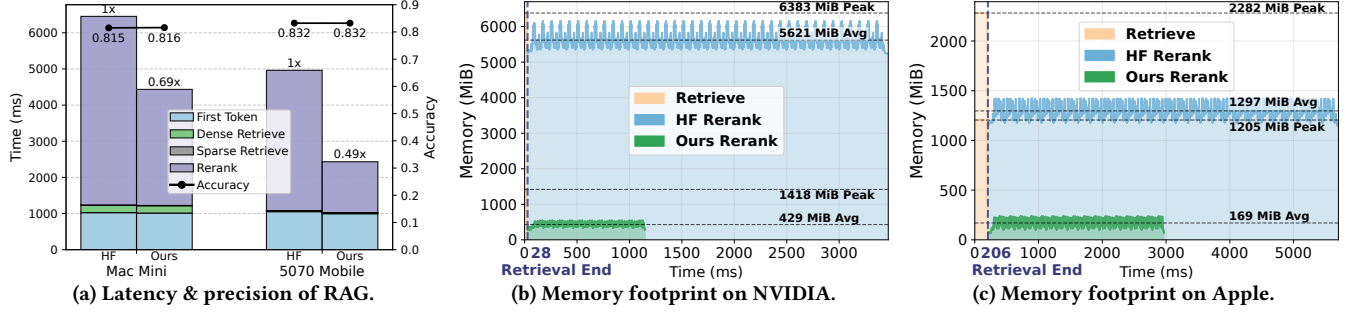
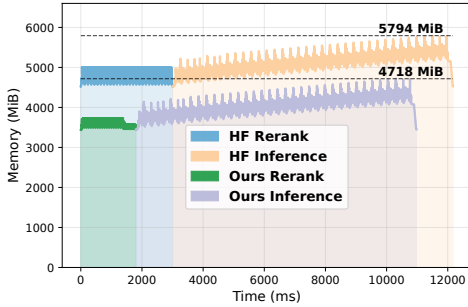
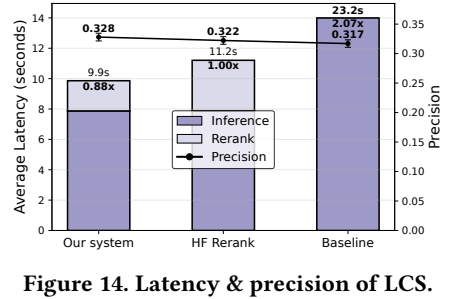
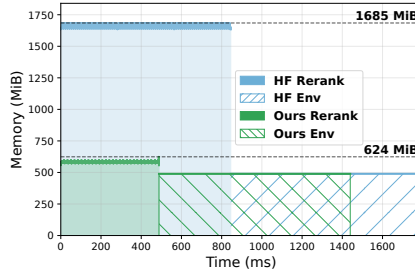
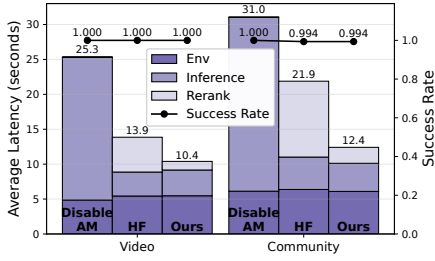


Figure 11. The latency, precision, and memory footprint of RAG.



results on the Apple platform are similar to those on the NVIDIA platform, so we will not elaborate further.

As shown in Figure 12, we evaluate the task completion latency and the task success rate in two different workloads. Our system significantly reduces the latency by 25.2% in the video workload and 43.4% in the community scenario, respectively. Crucially, these improvements come with no loss in the task success rate. Figure 13 shows the memory footprint in the period of a single click action performing by the agent. Compared to HF, PRISM reduces the peak memory usage by 63.0%. Such substantial savings are particularly valuable on resource-constrained edge devices, where lower memory usage directly increases the keep-alive rate of applications and improves the users' experience.

LLM Long Context Selection (LCS). LLM long context selection aims to select the most relevant information in an ultra long context, thereby accelerating inference. In this real-world scenario, we employ a Qwen3-Reranker-0.6B reranker to select the most relevant information and then

feed them to a quantized Qwen3-4B-Instruct for generation. The evaluations are conducted on the NVIDIA platform with the LongBench2 [16] benchmark. We compare the latency and precision of three systems: PRISM, HF Reranker, and No Reranker.

Figure 14 reports the end-to-end latency and precision. Generally, PRISM achieves a latency reduction of 11.6% compared to HF Reranker and 57.3% compared to No Reranker, with even marginal precision increasing. For precision, the two systems with reranker surpass the No Reranker, which is distracted by the irrelevant information. Besides, the slight precision gain observed in PRISM over HF Reranker fall within normal variation. Figure 15 compares the memory footprint of HF Reranker and PRISM in one generation. PRISM reduces peak memory by about 1 GiB compared to the HF Reranker.

In summary, PRISM consistently outperforms the baselines in latency and memory footprint.

6.4 Ablation Study

We conduct an ablation study to show the contributions of our four proposed techniques by applying them incrementally. We measure the latency and memory footprint running the Qwen3-Reranker-0.6B on the NVIDIA platform to rank 60 candidates with an average length of 500.

Figure 16 illustrates how memory footprint and latency change as each technique is applied incrementally. Starting from the baseline, we first apply progressive cluster pruning. It reduces the latency by 49.0%, but increases the peak memory by 44.8% due to the monolithic batch. Then, we apply the chunked execution to reduce this memory overhead to 7.2%. The remaining overhead stems from storing hidden states for all chunks, a requirement for the monolithic forwarding

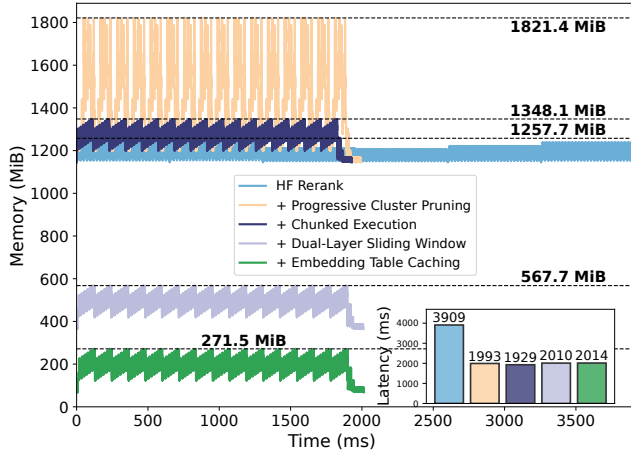


Figure 16. Memory & latency ablation of four techniques.

scheme. Next, we apply the overlapped layer streaming, significantly reduces memory usage by 57.8%. This optimization incurs a modest 81 ms latency overhead because the reduced computation time from pruning no longer fully hides the I/O latency. Finally, embedding table caching eliminates the last dominant memory bottleneck, reducing the peak memory usage to 271 MiB with a negligible 4 ms latency overhead. When combined, PRISM achieves a 78.4% reduction in peak memory and a 48.5% reduction in latency compared to the baseline, demonstrating the effectiveness of our techniques when working in concert.

These results validate that each technique successfully achieves its intended optimization goal, and their integration yields a system optimized for both memory and latency.

7 Discussion

Flexibility for diverse application needs. PRISM’s design provides the flexibility to cater to diverse application requirements. In most scenarios, such as selecting documents for a RAG pipeline, the primary goal is to identify the top- K candidates regardless of their precise internal ranking. For such cases, the pruning both winners and hopeless candidates can maximize latency reduction as demonstrated in our evaluation. Furthermore, PRISM is equally capable of handling applications where the exact rank order or the final scores are critical. PRISM supports this by only pruning the hopeless candidates while allowing the top contenders to undergo full inference. This adaptability allows developers to tune the system for their specific latency budget and application-level quality requirements.

Generality beyond evaluated models. While our evaluation focused on a representative set of state-of-the-art rerankers, we have also observed the core insight of sequence-level sparsity is a general characteristic of cross-encoder architectures. The hierarchical nature of transformers, where earlier layers capture broader contextual features and later layers refine nuanced semantic relationships, naturally leads to the progressive emergence of stable relative rankings. Our

preliminary experiments with other cross-encoder models, including LLM as rerankers (e.g., Qwen3-4B-Instruct), confirm this pattern. This suggests that the principles behind PRISM are not limited to specialized reranker models but can likely be extended to a broader class of transformer-based models performing semantic selection tasks.

Discussion on related work. For related works on model compression, PRISM is a training-free system and is orthogonal to a wide array of model compression techniques. As shown by our “PRISM Quant” evaluation, its benefits seamlessly compound with post-training quantization methods [29, 45]. Moreover, our approach can be readily applied to models that have already undergone training-based compression [19, 47, 54]. This orthogonality allows PRISM to be integrated with existing and future model compression advancements to further push the efficiency frontier of on-device AI. For related works on memory optimization, PrefillOnly [28] proposes chunking batches to reduce the memory overhead of intermediate tensors. PRISM additionally proposes dynamic offloading of hidden states to further reduce peak memory. Furthermore, tailored for edge scenarios, PRISM introduces overlapped layer streaming and embedding table cache, significantly reducing the memory footprint of model weights.

8 Conclusion

We introduce PRISM, a training-free inference system that re-frames reranking to focus on relative rankings, enabling a highly efficient monolithic forwarding architecture. Our system uses progressive cluster pruning and a series of memory optimizations to significantly reduce latency by up to 89.2% and peak memory by up to 91.3%, substantially advancing on-device semantic selection.

Acknowledgments

We sincerely thank our shepherd, Supawit Chockchowwat, and the anonymous reviewers for their insightful comments. This work is supported in part by the National Natural Science Foundation of China (No. 62132014), the Fundamental Research Funds for the Central Universities, Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM113), and Huawei Technologies. The corresponding authors are Mingkai Dong (mingkaidong@sjtu.edu.cn) and Dingji Li (lidingji1997@hotmail.com).

References

- [1] 2023. ellamind/wikipedia-2023-11-retrieval-multilingual-corpus · Datasets at HuggingFace. <https://huggingface.co/datasets/ellamind/wikipedia-2023-11-retrieval-multilingual-corpus>
- [2] 2024. BAAI/bge-reranker-v2-gemma · HuggingFace. <https://huggingface.co/BAAI/bge-reranker-v2-gemma>.
- [3] 2024. BAAI/bge-reranker-v2-m3 · HuggingFace. <https://huggingface.co/BAAI/bge-reranker-v2-m3>.
- [4] 2024. BAAI/bge-reranker-v2-minicpm-layerwise · HuggingFace. <https://huggingface.co/BAAI/bge-reranker-v2-minicpm-layerwise>.

- [5] 2025. Global Interpreter Lock - Python Wiki. <https://wiki.python.org/moin/GlobalInterpreterLock>
- [6] 2025. IPADS-SAI/MobiAgent: The Intelligent GUI agent for mobile Phones. <https://github.com/IPADS-SAI/MobiAgent>
- [7] 2025. IPADS-SAI/MobiMind-Decider-7B · HuggingFace. <https://huggingface.co/IPADS-SAI/MobiMind-Decider-7B>
- [8] 2025. libuv/libuv: Cross-platform asynchronous I/O. <https://github.com/libuv/libuv>
- [9] 2025. Magic Cue on Pixel 10 Series Phones: smart, contextual assistance across apps, emails & more. <https://store.google.com/intl/en/ideas/articles/magic-cue/>.
- [10] 2025. Memory Usage, Huawei HarmonyOS Developer. <https://developer.huawei.com/consumer/cn/doc/harmonyos-guides/performance-memory-usage> [Online; accessed 2025-09-18].
- [11] 2025. Qwen/Qwen3-Reranker-8B · HuggingFace. <https://huggingface.co/Qwen/Qwen3-Reranker-8B>
- [12] 2025. Rerank | Boost Enterprise Search and Retrieval | Cohere. <https://cohere.com/rerank>.
- [13] 2025. Reranking for Vertex AI RAG Engine | Generative AI on Vertex AI | Google Cloud. <https://cloud.google.com/vertex-ai/generative-ai/docs/rag-engine/retrieval-and-ranking> [Online; accessed 2025-09-12].
- [14] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2025. Taming throughput-latency tradeoff in LLM inference with sarathi-serve. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) (OSDI'24). USENIX Association, USA, Article 7, 18 pages.
- [15] Avinashsingh. 2025. [Feature]: Add support for Apple MPS(Metal Performance Shaders). <https://github.com/vllm-project/vllm/issues/22629>
- [16] Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2024. LongBench v2: Towards Deeper Understanding and Reasoning on Realistic Long-context Multitasks. *arXiv preprint arXiv:2412.15204* (2024).
- [17] Beir-Cellar. 2025. beir-cellar/beir: A Heterogeneous Benchmark for Information Retrieval. Easy to use, evaluate your models across 15+ diverse IR datasets. <https://github.com/beir-cellar/beir>
- [18] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. <https://arxiv.org/abs/2005.14165>. arXiv:2005.14165 [cs.CL]
- [19] Mengzhao Chen, Wenqi Shao, Peng Xu, Jiahao Wang, Peng Gao, Kaipeng Zhang, Yu Qiao, and Ping Luo. 2024. EfficientQAT: Efficient Quantization-Aware Training for Large Language Models. *arXiv preprint arXiv:2407.11062* (2024).
- [20] Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. 2025. Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory. arXiv:2504.19413 [cs.CL] <https://arxiv.org/abs/2504.19413>
- [21] Cohere. 2024. Introducing Rerank 3 on Microsoft Azure AI | Cohere blog. <https://cohere.com/blog/introducing-rerank-3-on-microsoft-azure-ai>.
- [22] Wikipedia contributors. 2022. Goodman and Kruskal's gamma. https://en.wikipedia.org/wiki/Goodman_and_Kruskal%27s_gamma
- [23] Wikipedia contributors. 2025. Coefficient of variation. https://en.wikipedia.org/wiki/Coefficient_of_variation
- [24] Wikipedia contributors. 2025. Global interpreter lock. https://en.wikipedia.org/wiki/Global_interpreter_lock
- [25] Mike Darling. 2025. 4 ways Pixel's Magic Cue can help you save time. <https://blog.google/products/pixel/google-pixel-magic-cue-ai-feature/>.
- [26] Gabriel de Souza P. Moreira, Ronay Ak, Benedikt Schifferer, Mengyao Xu, Radek Osmulski, and Even Oldridge. 2024. Enhancing Q&A Text Retrieval with Ranking Models: Benchmarking, fine-tuning and deploying Rerankers for RAG. arXiv:2409.07691 [cs.IR] <https://arxiv.org/abs/2409.07691>
- [27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. <https://arxiv.org/abs/1810.04805>. arXiv:1810.04805 [cs.CL]
- [28] Kuntai Du, Bowen Wang, Chen Zhang, Yiming Cheng, Qing Lan, Hejian Sang, Yihua Cheng, Jiayi Yao, Xiaoxuan Liu, Yifan Qiao, Ion Stoica, and Junchen Jiang. 2025. PrefillOnly: An Inference Engine for Prefill-only Workloads in Large Language Model Applications. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) (SOSP '25). Association for Computing Machinery, New York, NY, USA, 399–414. <https://doi.org/10.1145/3731569.3764834>
- [29] Elias Frantar, Saleh Ashkboos, Torsten Hoeftler, and Dan Alistarh. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. <https://arxiv.org/abs/2210.17323>. arXiv:2210.17323 [cs.LG]
- [30] Qichen Fu, Minsik Cho, Thomas Merth, Sachin Mehta, Mohammad Rastegari, and Mahyar Najibi. 2024. LazyLLM: Dynamic Token Pruning for Efficient Long Context LLM Inference. arXiv:2407.14057 [cs.CL] <https://arxiv.org/abs/2407.14057>
- [31] Michael Glass, Gaetano Rossiello, Md Faisal Mahbub Chowdhury, Ankita Naik, Pengshan Cai, and Alfio Gliozzo. 2022. Re2G: Retrieve, Rerank, Generate. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Marine Carpuat, Marie-Catherine de Marneffe, and Ivan Vladimir Meza Ruiz (Eds.). Association for Computational Linguistics, Seattle, United States, 2701–2715. <https://doi.org/10.18653/v1/2022.naacl-main.194>
- [32] John A Hartigan and Manchek A Wong. 1979. Algorithm AS 136: A k-means clustering algorithm. *Journal of the royal statistical society, series c (applied statistics)* 28, 1 (1979), 100–108.
- [33] HuggingFace. 2025. Accelerate: A simple way to launch, train, and use PyTorch models on almost any device and distributed configuration, automatic mixed precision (including fp8), and easy-to-configure FSDP and DeepSpeed support. <https://github.com/huggingface/accelerate>
- [34] HuggingFace. 2025. Transformers: the model-definition framework for state-of-the-art machine learning models in text, vision, audio, and multimodal models, for both inference and training. <https://github.com/huggingface/transformers>
- [35] Ganesh Jawahar, Benoît Sagot, and Djamel Seddah. 2019. What Does BERT Learn about the Structure of Language?. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Anna Korhonen, David Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, Florence, Italy, 3651–3657. <https://doi.org/10.18653/v1/P19-1356>
- [36] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. LongLLMLingua: Accelerating and Enhancing LLMs in Long Context Scenarios via Prompt Compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 1658–1677. <https://doi.org/10.18653/v1/2024.acl-long.91>

- [37] Jiazheng Kang, Mingming Ji, Zhe Zhao, and Ting Bai. 2025. Memory OS of AI Agent. arXiv:2506.06326 [cs.AI] <https://arxiv.org/abs/2506.06326>
- [38] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 6769–6781. <https://doi.org/10.18653/v1/2020.emnlp-main.550>
- [39] Khan-Yin. 2025. [Feature] Support Apple Silicon (M2/M3...). <https://github.com/sgl-project/sglang/issues/5767>
- [40] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 39–48. <https://dl.acm.org/doi/10.1145/3397271.3401075>
- [41] 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 (Koblenz, Germany) (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. <https://doi.org/10.1145/3600006.3613165>
- [42] Xunhao Lai, Jianqiao Lu, Yao Luo, Yiyuan Ma, and Xun Zhou. 2025. FlexPrefix: A Context-Aware Sparse Attention Mechanism for Efficient Long-Sequence Inference. In *The Thirteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=OfjllbelT>
- [43] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (Vancouver, BC, Canada) (NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 793, 16 pages.
- [44] Jiaxing Li, Chi Xu, Lianchen Jia, Feng Wang, Cong Zhang, and Jiangchuan Liu. 2025. EACO-RAG: Towards Distributed Tiered LLM Deployment using Edge-Assisted and Collaborative RAG with Adaptive Knowledge Update. arXiv:2410.20299 [cs.DC] <https://arxiv.org/abs/2410.20299>
- [45] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. <https://arxiv.org/abs/2306.00978>. arXiv:2306.00978 [cs.CL]
- [46] Lingkun Long, Rubing Yang, Yushi Huang, Desheng Hui, Ao Zhou, and Jianlei Yang. 2025. SlimInfer: Accelerating Long-Context LLM Inference via Dynamic Token Pruning. arXiv:2508.06447 [cs.CL] <https://arxiv.org/abs/2508.06447>
- [47] Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. LLM-Pruner: On the Structural Pruning of Large Language Models. In *Thirty-seventh Conference on Neural Information Processing Systems*. <https://openreview.net/forum?id=J8Ajf9WfXP>
- [48] Milvus. 2024. Milvus | The High-Performance Vector Database built for Scale. <https://milvus.io/>
- [49] NVIDIA. 2025. NVIDIA Multi-Process Service. <https://docs.nvidia.com/deploy/mps/index.html>
- [50] Authors of BitNet b1.58. 2025. BitNet b1.58: 1.58-bit Large Language Models. arXiv:2504.12285 [cs.CL] <https://arxiv.org/abs/2504.12285>
- [51] PyTorch. 2025. Multiprocessing package - torch.multiprocessing. <https://docs.pytorch.org/docs/2.8/multiprocessing.html>
- [52] Yuwei Ren, Yuhui Ding, Lijun Wu, Shujian Huang, Lei Li, and Qun Liu. 2024. BitNet a4.8: 1-bit Weight 4-bit Activation LLMs. arXiv:2411.04965 [cs.CL] <https://arxiv.org/abs/2411.04965>
- [53] Hassan Sajjad, Nadir Durrani, Fahim Dalvi, Firoj Alam, Abdul Khan, and Jia Xu. 2022. Analyzing Encoded Concepts in Transformer Language Models. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Marine Carpuat, Marie-Catherine de Marneffe, and Ivan Vladimir Meza Ruiz (Eds.). Association for Computational Linguistics, Seattle, United States, 3082–3101. <https://doi.org/10.18653/v1/2022.naacl-main.225>
- [54] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. arXiv:1910.01108 [cs.CL] <https://arxiv.org/abs/1910.01108>
- [55] Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. 2022. ColBERTv2: Effective and Efficient Retrieval via Lightweight Late Interaction. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Marine Carpuat, Marie-Catherine de Marneffe, and Ivan Vladimir Meza Ruiz (Eds.). Association for Computational Linguistics, Seattle, United States, 3715–3734. <https://doi.org/10.18653/v1/2022.naacl-main.272>
- [56] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2024. PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU. arXiv:2312.12456 [cs.LG] <https://arxiv.org/abs/2312.12456>
- [57] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. 2021. BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*. <https://openreview.net/forum?id=wCu6T5xFje>
- [58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (Long Beach, California, USA) (NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.
- [59] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 International Conference on Management of Data (Virtual Event, China) (SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2614–2627. <https://doi.org/10.1145/3448016.3457550>
- [60] Yichuan Wang, Shu Liu, Zhifei Li, Yongji Wu, Ziming Mao, Yilong Zhao, Xiao Yan, Zhiying Xu, Yang Zhou, Ion Stoica, Sewon Min, Matei Zaharia, and Joseph E. Gonzalez. 2025. LEANN: A Low-Storage Vector Index. arXiv:2506.08276 [cs.DB] <https://arxiv.org/abs/2506.08276>
- [61] Zijie J. Wang and Duen Horng Chau. 2024. MeMemo: On-device Retrieval Augmentation for Private and Personalized Text Generation. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval (Washington DC, USA) (SIGIR '24)*. Association for Computing Machinery, New York, NY, USA, 2765–2770. <https://doi.org/10.1145/3626772.3657662>
- [62] Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025. CodeRAG-Bench: Can Retrieval Augment Code Generation?. In *Findings of the Association for Computational Linguistics: NAACL 2025*, Luis Chiruzzo, Alan Ritter, and Lu Wang (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, 3199–3214. <https://doi.org/10.18653/v1/2025.findings-naacl.176>
- [63] Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2025. Agent Workflow Memory. In *Forty-second International Conference on Machine Learning*. <https://openreview.net/forum?id=NTAhi2JEE>

- [64] Orion Weller, Michael Boratko, Iftekhhar Naim, and Jinhyuk Lee. 2025. On the Theoretical Limitations of Embedding-Based Retrieval. arXiv:2508.21038 [cs.IR] <https://arxiv.org/abs/2508.21038>
- [65] Menglin Xia, Xuchao Zhang, Camille Couturier, Guoqing Zheng, Saravan Rajmohan, and Victor Rühle. 2024. Hybrid-RACA: Hybrid Retrieval-Augmented Composition Assistance for Real-time Text Prediction. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, Franck Dernoncourt, Daniel Preotiu-Pietro, and Anastasia Shimorina (Eds.). Association for Computational Linguistics, Miami, Florida, US, 120–131. <https://doi.org/10.18653/v1/2024.emnlp-industry.11>
- [66] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. 2020. DeeBERT: Dynamic Early Exiting for Accelerating BERT Inference. arXiv:2004.12993 [cs.CL] <https://arxiv.org/abs/2004.12993>
- [67] Zhenliang Xue, Yixin Song, Zeyu Mi, Xinrui Zheng, Yubin Xia, and Haibo Chen. 2024. PowerInfer-2: Fast Large Language Model Inference on a Smartphone. arXiv:2406.06282 [cs.LG] <https://arxiv.org/abs/2406.06282>
- [68] Cheng Zhang, Erhu Feng, Xi Zhao, Yisheng Zhao, Wangbo Gong, Jiahui Sun, Dong Du, Zhichao Hua, Yubin Xia, and Haibo Chen. 2025. MobiAgent: A Systematic Framework for Customizable Mobile Agents. arXiv:2509.00531 [cs.MA] <https://arxiv.org/abs/2509.00531>
- [69] Yanqi Zhang, Yuwei Hu, Runyuan Zhao, John C. S. Lui, and Haibo Chen. 2024. Unifying KV Cache Compression for Large Language Models with LeanKV. *CoRR* abs/2412.03131 (2024). <https://doi.org/10.48550/arXiv.2412.03131>
- [70] Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. 2025. Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models. <https://arxiv.org/abs/2506.05176>. arXiv:2506.05176 [cs.CL]
- [71] Wayne Xin Zhao, Jing Liu, Ruiyang Ren, and Ji-Rong Wen. 2024. Dense Text Retrieval Based on Pretrained Language Models: A Survey. *ACM Trans. Inf. Syst.* 42, 4, Article 89 (Feb. 2024), 60 pages. <https://doi.org/10.1145/3637870>
- [72] George Kingsley Zipf. 1949. Human behavior and the principle of least effort. (1949).

A Artifact Appendix

A.1 Abstract

The artifact includes the source code of PRISM and the base-lines, and the scripts to reproduce the results in the paper.

A.2 Description & Requirements

A.2.1 How to access The artifact is available at: https://ipads.se.sjtu.edu.cn:1312/opensource/monolithic_forwarding and archived at <https://doi.org/10.5281/zenodo.18809731>.

A.2.2 Hardware dependencies

- **GPU:** NVIDIA GPU with at least 8 GB VRAM or M-series Apple Silicon.
- **RAM:** 16 GB minimum.
- **Disk:** ~50 GB for datasets and model checkpoints.

Experiments in the paper use an NVIDIA RTX 5070 Laptop (8 GiB) and an Apple Mac Mini M2 (16 GiB unified memory). Reproduction on equivalent or better NVIDIA hardware is supported; Apple platform results can be reproduced on M-series Macs with sufficient memory.

A.2.3 Software dependencies

- **OS:** Linux (tested on Ubuntu 22.04) and macOS (tested on Sequoia 15.1).
- **Environment:** Conda (Miniconda or Anaconda).
- **CUDA:** 12.1 or newer (for NVIDIA).
- **Docker:** Required for Milvus-based RAG experiments (Section 6.3).

A.2.4 Benchmarks

- **Microbenchmarks (Section 6.2):** 18 datasets and reranker models (Qwen3-Reranker-0.6B/4B/8B, Bge-Reranker-v2-MiniCPM, Bge-Reranker-v2-M3) downloaded via provided `download_models.sh` from HuggingFace.
- **Real-world (Section 6.3):** RAG pipeline (with Milvus); Agent Memory (video and community scenarios); Long Context Selection (LongBench-style workloads). Data and workloads are included or fetched by the artifact scripts.

A.3 Set-up

1. Clone the artifact repository and enter its root directory.
2. Run `bash install_dependencies.sh` to create the Conda environment, install Python dependencies, and build C extensions.
3. Run `bash download_models.sh` to fetch model checkpoints from HuggingFace.
4. Run `bash run_demo.sh` to perform kick-the-tires verification with a small example. Successful completion indicates the environment is ready.

For detailed steps, see `quickstart.md` in the repository.

A.4 Evaluation workflow

A.4.1 Major Claims

- **(C1) Microbenchmark latency:** PRISM reduces latency (e.g., up to 89.2%) versus HF vanilla/offload/quant without compromising precision. Supported by experiments (E1.1); results in Section 6.2 and **Figure 8**.
- **(C2) Microbenchmark memory:** PRISM reduces peak GPU memory by $5.34\times$ – $11.45\times$ vs. HF, $1.34\times$ – $3.83\times$ vs. HF Offload, and $2.77\times$ – $4.83\times$ vs. Quant. Supported by (E1.2); results in Section 6.2 and **Figure 9**.
- **(C3) Latency–precision trade-off:** Increasing the threshold improves precision at the cost of higher latency. Supported by (E1.3); results in Section 6.2 and **Figure 10**.
- **(C4) RAG pipeline:** PRISM reduces latency by 51% on NVIDIA and 31% on Apple, and reduces peak memory by 77.8% and average memory by 92.3%, with no precision loss. Supported by (E2.1); results in Section 6.3 and **Figure 11**.
- **(C5) Agent Memory:** PRISM reduces latency by 25.2% (video) and 43.4% (community) with no loss in task success rate, and reduces peak memory by 63.0%. Supported by (E2.2); results in Section 6.3 and **Figures 12 & 13**.
- **(C6) Long Context Selection:** PRISM reduces latency by 12% vs. HF Reranker and 57.3% vs. No Reranker with marginally better precision, and reduces peak memory by ~1 GiB vs. HF Reranker. Supported by (E2.3); results in Section 6.3 and **Figures 14 & 15**.
- **(C7) Ablation:** Progressive cluster pruning, chunked execution, dual-layer sliding window, and embedding table caching each achieve their intended optimization goals. Supported by (E3.1); results in Section 6.4 and **Figure 16**.

A.4.2 Experiments Below are the major procedures to reproduce experiments. **Please refer to** `experiments/{experiment_name}/README.md` **for complete instructions**.

- **(E1.1) Latency & Precision (Figure 8)** [~10 human-minutes + ~2 compute-hours]. To evaluate the latency and precision of PRISM vs. baselines:
 - *[Execution]*
`cd experiments/Latency_and_Precision/
bash ./run_latency_experiments.sh
bash ./run_precision_experiments.sh`
 - *[Results]*
`python experiments/Latency_and_Precision/plot.py`
- **(E1.2) Memory Footprint (Figure 9)** [~5 human-minutes + ~30 compute-minutes]. To evaluate the peak memory comparison:
 - *[Execution]*
`bash experiments/Memory_Footprint/run.sh`
 - *[Results]*
`python experiments/Memory_Footprint/plot.py`

- **(E1.3) Latency–Precision Trade-off (Figure 10)** [~ 5 human-minutes + ~ 1 compute-hour]. To reproduce threshold vs. latency/precision curves:
 - *[Execution]*
`bash ./experiments/Latency_Precision_Tradeoff/run.sh`
 - *[Results]*
`python experiments/Latency_Precision_Tradeoff/plot.py`
- **(E2.1) RAG Pipeline (Figure 11)** [~ 5 human-minutes + ~ 1 compute-hours]. To evaluate RAG latency and memory:
 - *[Execution]*
`cd experiments/RAG_Pipeline/`
`bash ./run_latency_experiments.sh`
`bash ./run_memory_experiments.sh`
 - *[Results]*
`cd experiments/RAG_Pipeline/`
`python plot_rag_latency.py`
`python plot_rag_memory.py`
- **(E2.2) Agent Memory (Figures 12 & 13)** [~ 5 human-minutes + ~ 2 compute-hours]. To evaluate Agent Memory latency and memory:
 - *[Execution]*
`cd experiments/Agent_Memory/`
`bash ./run_latency_experiments.sh`
`bash ./run_memory_experiments.sh`
 - *[Results]*
`cd experiments/Agent_Memory/`
`python plot_agent_latency.py`
`python plot_agent_memory.py`
- **(E2.3) Long Context Selection (Figures 14 & 15)** [~ 5 human-minutes + ~ 1.5 compute-hours]. To evaluate long-context latency and memory:
 - *[Execution]*
`cd experiments/Long_Context_Selection/`
`bash ./run_latency_experiments.sh`
`bash ./run_memory_experiments.sh`
 - *[Results]*
`cd experiments/Long_Context_Selection/`
`python plot_latency.py`
`python plot_memory.py`
- **(E3.1) Ablation (Figure 16)** [~ 5 human-minutes + ~ 1 compute-hours]. To evaluate the ablation study:
 - *[Execution]*
`cd experiments/Ablation_Study/`
`bash ./run_latency_ablation.sh`
`bash ./run_memory_ablation.sh`
 - *[Results]*
`python experiments/Ablation_Study/plot.py`