SPARSEENGINE: SPARSE-FIRST INFERENCE ENGINE

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-context LLM-based agents accumulate growing interaction histories, placing substantial pressure on KV cache memory and attention computation. Consequently, recent works have adopted sparse attention to reduce computational and memory overheads. However, due to the heterogeneous nature of sparse methods, they cannot be directly applied to existing inference engines. *Sparse inference engines* have therefore been proposed. However, existing serving abstractions tie method integration to particular cache layouts or computational workflows, limiting support for heterogeneous sparse methods. We present **SparseEngine**, a ground-up sparse-first inference engine built around a shared lifecycle contract. The contract lets each method control its KV representation and computational workflow while coordinating state transitions with shared serving infrastructure. SparseEngine supports **15** sparse methods across four major categories, demonstrating its generality. Furthermore, built upon its lifecycle contract, it enables higher-level cross-request state management: we propose **Chain Cache** to allow KV eviction methods to resume from retained history, and Controllable **Prefix-Cache Pruning** to let applications prune KV from selected history regions while preserving logical prefix matching. While preserving the quality of sparse methods, SparseEngine achieves over **10**× higher throughput with KV eviction methods and over **2.5**× decode throughput speedup under identical concurrency, compared to vLLM. It also achieves over **2**× end-to-end speedup on agent benchmarks. The code is available at `https://anonymous.4open.science/r/SparseEngine`.

1 INTRODUCTION

As LLMs evolve into autonomous, multi-turn agents, inference workloads become fundamentally stateful (Yao et al., 2022; Shinn et al., 2023). In iterative agent workflows (e.g., coding, research, and reasoning), context history expands rapidly through repeated tool use and reasoning chains. This creates dual GPU bottlenecks in KV cache capacity and attention compute latency. Sparse inference methods mitigate these costs via selective attention or compressed KV storage (Tang et al., 2024; Li et al., 2024; Chang et al., 2024; Liu et al., 2024; Hao et al., 2025). These methods differ in how they represent KV cache and when they update it, yet must operate within shared infrastructure for request scheduling and cache reuse. The key challenge is to define an abstraction that gives methods control over their state and computation while preserving these shared serving capabilities.

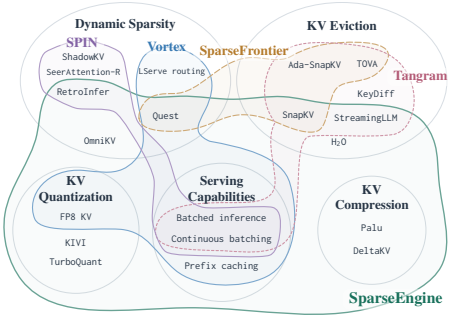


Figure 1: **Method coverage across sparse inference engines.** Contours enclose representative implementations.

Programming language design illustrates this challenge: abstractions that simplify some computational patterns can constrain others. MATLAB's matrix operations simplify numerical computation (Moler & Little, 2020), but irregular computations often require explicit control flow. SQL expresses relational queries (Astrahan & Chamberlin, 1975), but stateful iterative algorithms often require recursive queries or procedural extensions. Rust's ownership and borrowing support memory-safe systems programming (Jung et al., 2018), but constrain how mutable state can be shared. In

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

each case, assumptions about data or state determine which computations are natural to express. For sparse inference engines, assumptions about KV layouts and update workflows similarly determine which method-specific computations and state changes their interfaces can accommodate.

Current engines organize these interfaces around particular cache layouts or workflows. Vortex exposes page-centric operations (Chen et al., 2026b), while SPIN manages partitions through a GPU–CPU pipeline (Zhao et al., 2026). Tangram specializes in non-uniform head-wise KV retention (Kim et al., 2026). Such interfaces fit some methods but constrain others: Quest uses query-dependent page selection (Tang et al., 2024), whereas H_2O also needs persistent attention scores and physical KV eviction during generation (Zhang et al., 2023). Figure 1 compares method coverage with support for prefix caching, continuous batching and batched inference. This raises a central question:

Where should the abstraction boundary lie between shared serving infrastructure and method-specific control over KV state and computation?

We present **SparseEngine**, which places this boundary at a **shared lifecycle contract** that lets sparse methods define their own computational workflows and KV representations. Fine-grained hooks coordinate method-specific computation and state updates, while common interfaces integrate these methods with attention execution and scheduling. Building on this flexibility, we introduce **Chain Cache** to reuse retained KV across turns. **Controllable Prefix-Cache Pruning** further lets applications prune selected history intervals while preserving prefix matching. Our contributions are:

- **A general lifecycle abstraction for diverse sparse methods.** We establish a unified lifecycle contract by introducing fine-grained lifecycle. This allows diverse sparse attention methods to customize their own KV representations and computational workflows without modifying model implementations, enabling the integration of **15 methods** across four heterogeneous families.
- **Higher-level sparsity-based cache state management.** Building upon the lifecycle contract, we introduce higher-level cross-request cache management abstractions: *Chain Cache* enables eviction-based methods to resume directly from compacted historical state across interactions, while *Prefix-Cache Pruning* selectively reclaims physical KV capacity from application-specified history regions while preserving logical prefix matching.
- **Efficient serving and fast inference with faithful quality.** SparseEngine achieves over $10\times$ aggregate decode throughput compared to vLLM via physical KV eviction under large batch sizes, delivers over $2.5\times$ decode throughput speedup under identical concurrency, and attains up to $2.24\times$ end-to-end replay speedup on multi-turn agent benchmarks, all while faithfully preserving the original task quality of evaluated sparse methods.

2 Related Work

Sparse Attention. Existing methods reduce KV cache cost through four complementary strategies. **Dynamic sparse attention**, including Quest (Tang et al., 2024), InfLLM (Xiao et al., 2024a), and OmniKV (Hao et al., 2025), selects query-dependent context while typically preserving unselected history for future queries. NSA (Yuan et al., 2025) learns native sparse-attention patterns, whereas IndexCache (Bai et al., 2026) reuses selection indices across layers. **KV eviction**, including H_2O (Zhang et al., 2023), SnapKV (Li et al., 2024), and PyramidKV (Cai et al., 2024), discard entries to reduce memory and subsequent attention work. **KV compression** methods, including Palu (Chang et al., 2024), LoRC (Zhang et al., 2024), and DeltaKV (Hao et al., 2026), replace full-dimensional KV tensors with compact representations, while **KV quantization** methods, such as KIVI (Liu et al., 2024) and TurboQuant (Zandieh et al., 2026), reduce numerical precision.

These strategies require distinct state and execution semantics: dynamic selection maintains query-dependent indices and summaries; eviction changes the resident set and releases capacity; and compression or quantization changes the payload and may require specialized metadata or reconstruction. **SparseEngine** provides a shared lifecycle contract that supports heterogeneous sparse methods while preserving method control over KV representations, metadata, updates, and attention paths.

Inference Engine. Inference engines coordinate model execution, request scheduling, and KV cache management. Continuous batching updates the active batch between decoding steps, while chunked prefill divides long prompts into smaller scheduling units (Yu & Jeong, 2022; Agrawal et al., 2024). Paged caches map logical token positions to physical KV blocks (Kwon et al., 2023), and prefix caching reuses computed state across requests with shared prefixes (Zheng et al., 2024).

Sparse inference systems further coordinate sparse computation, data placement, and memory reclamation. **SparseFrontier** (Nawrot et al., 2026) evaluates the accuracy-efficiency trade-offs of sparse-attention methods. Among serving systems, **SPIN** (Zhao et al., 2026) maps different selection granularities to partitions backed by hierarchical GPU-CPU KV storage; **Vortex** (Chen et al., 2026b) provides programmable page-centric routing; and **Tangram** (Kim et al., 2026) manages non-uniform head-wise retention through budget reservation and Ragged Paging. **HiSparse** (Xie et al., 2026) bounds GPU residency by storing the complete KV history in host memory and fetching selected entries into a GPU cache, whereas **vToken** (Gao et al., 2026) reclaims token-level storage through logical-to-physical indirection and repacking.

These systems organize sparsity around specific access, placement, or retention patterns. In contrast, **SparseEngine** places the abstraction boundary at the method lifecycle: shared interfaces expose scheduling and capacity requirements, while methods control their physical representation and execution. This supports heterogeneous method families with continuous batching and method-compatible prefix caching (Figure 1). It also extends sparse-state management across requests: Chain Cache preserves compacted KV and metadata across turns, while controllable pruning reclaims history without losing logical-prefix reuse. See Appendix A for detailed background and notation.

3 SPARSEENGINE

Lifecycle Contract. Existing sparse inference engines organize methods around predefined computational workflows. For instance, SPIN follows *Index* $\rightarrow$ *Offload* $\rightarrow$ *Select* $\rightarrow$ *Retrieve* $\rightarrow$ *Attention* (Zhao et al., 2026), while Vortex follows *Cache Update & Summary Calculation* $\rightarrow$ *Page Selection* $\rightarrow$ *Attention* (Chen et al., 2026b). SparseEngine instead anchors its abstraction to a **key invariant**: the model's native module execution order. By exposing fine-grained hooks along this order, SparseEngine allows each sparse method to register callbacks and compose its own computational workflow without modifying the model implementation.

Section 3.1 develops the three components of this lifecycle contract: (1) fine-grained hooks exposed through `SparseController`; (2) a method-customized `CacheManager`; and (3) an `AttentionView` that connects method-owned state to attention execution. Building on these foundations, Section 3.2 extends method control across model executions: Chain Cache reuses retained KV and method-specific state across agent turns, while sparsity-based prefix-cache pruning removes selected physical KV entries without forfeiting logical-prefix matching or reuse.

3.1 ARCHITECTURAL FOUNDATIONS

Fine-Grained Lifecycle Hooks in `SparseController`. Figure 2 shows the hooks and their interactions with method-defined cache state and attention views. The hooks span both prefill and decoding, with extension points before and after attention and at the end of **every layer and execution step**. A `SparseMethodRuntime`, selected at initialization, implements the method-specific callbacks invoked through `SparseController`. These callbacks compose scoring, selection, and state-update operations across prefill chunks, decoding steps, and Transformer layers, allowing the same model implementation to execute different sparse methods.

During prefill, `finish_step` supports H_2O's chunk-wise eviction using accumulated attention scores and SnapKV's selection and eviction after the final prompt chunk. These hooks preserve chunked prefill without requiring method-specific model implementations such as `LlamaSnapKV` or `Qwen3H2O`. During decoding, `build_decode_selection` performs Quest's query-dependent page selection, while `finish_step` updates H_2O's cumulative scores and applies online eviction. Combining `on_layer_end` with `build_decode_selection` also enables OmniKV and DeltaKV to reuse observation-layer indices across subsequent layers.

Method-Customized `CacheManager`. The `CacheManager` owns the method-specific state and storage required by these workflows. Each method defines its KV representation, physical layout, position mapping, and allocation, write, update, and release operations. It also maintains persistent metadata alongside the corresponding KV state.

Lifecycle hooks invoke these operations to modify the physical cache. For example, SnapKV compaction and H_2O eviction jointly update retained positions, method statistics, and available

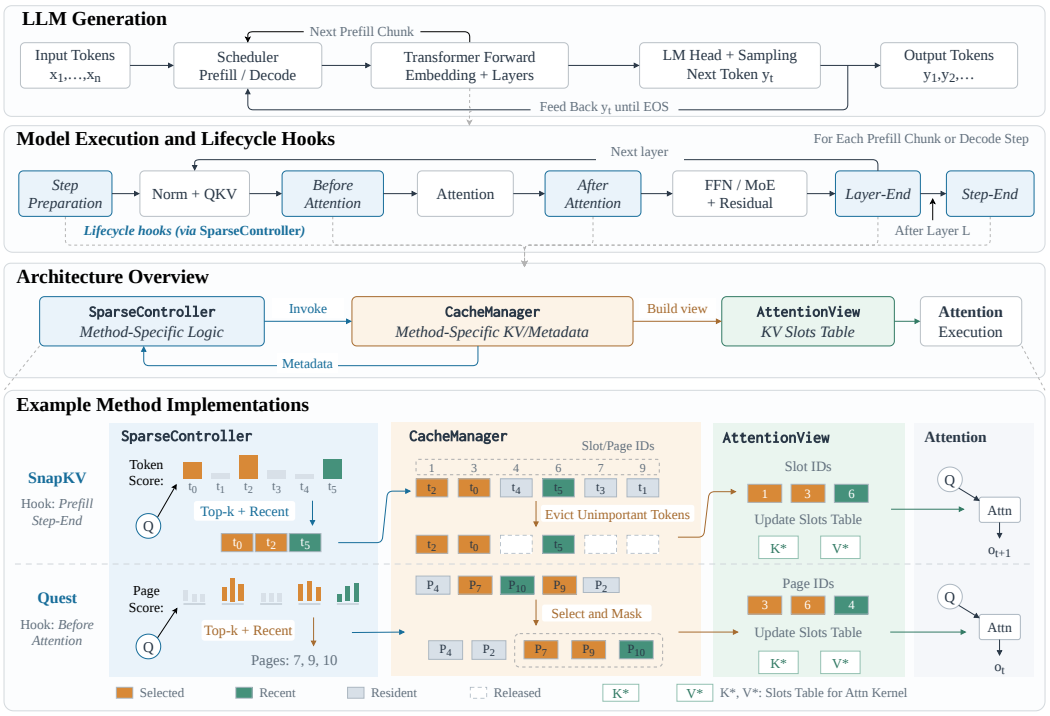


Figure 2: **Overview of SparseEngine.** From top to bottom, the figure shows the LLM generation loop; the Transformer execution path and lifecycle hooks invoked during prefill and decoding; the interactions among `SparseController`, the method-specific `CacheManager`, `AttentionView`, and attention execution; and representative implementations of SnapKV and Quest.

capacity. OmniKV applies cross-layer selections to resident KV entries, whereas KIVI manages quantized KV pages, quantization metadata, and an unquantized residual region. Physical eviction updates retained-slot mappings and context lengths before returning discarded slots to the allocator; logical selection changes only which resident entries attention reads. This separation allows methods with different storage and access semantics to share the same lifecycle.

Attention Through `AttentionView`. `AttentionView` connects method-owned cache representations to attention execution. For a selection-based method,

$$\mathrm{View}_{\ell,t} = \mathrm{BuildView}(\mathcal{C}_{\ell,t}, \mathcal{I}_{\ell,t}), \quad o_{\ell,t} = \mathrm{Attention}(q_{\ell,t}, \mathrm{View}_{\ell,t}), \tag{1}$$

where t is an execution step, $\mathcal{C}_{\ell,t}$ is the persistent cache state, and $\mathcal{I}_{\ell,t}$ identifies the selected tokens or pages. Given this state and selection, `CacheManager` constructs a view that describes the accessible data and its physical layout for a compatible attention backend. The underlying state may contain explicit KV tensors or MLA latent and positional states together with method-specific metadata.

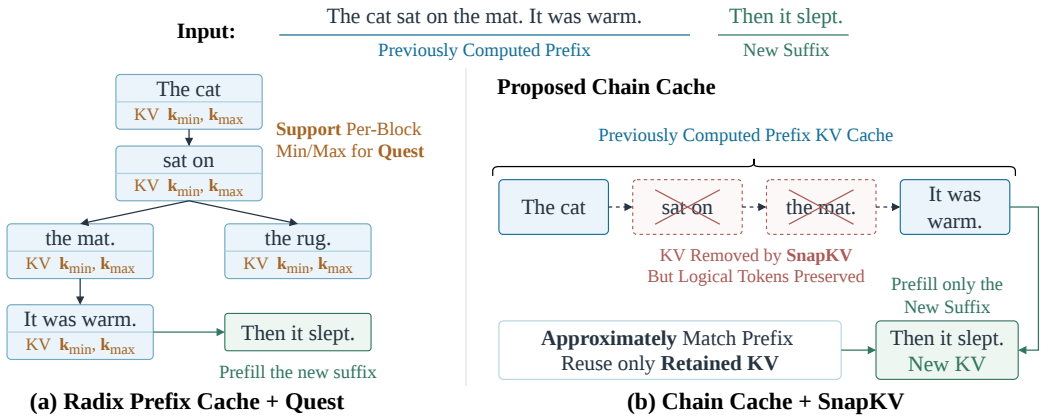
This interface supports attention over selected, compressed, quantized, or temporarily reconstructed KV while leaving persistent-state management to `CacheManager`. For example, a Quest view contains selected KV pages, an OmniKV view references resident KV according to cross-layer selections, and a DeltaKV view exposes KV reconstructed on demand. Thus, lifecycle hooks determine when state is selected or updated, `CacheManager` performs the corresponding storage operations, and `AttentionView` presents the resulting payload to attention. Appendix B details view payloads, temporary buffer lifetimes, and CUDA graph execution.

Execution Overview. Figure 2 traces inference from input tokens to generated output. The scheduler processes the prompt in prefill chunks and then feeds each sampled token back for the next decoding step. It queries `MemoryOracle` for method-specific capacity and reservation requirements and coordinates execution across GPU ranks (Appendix B.1). Within each Transformer layer, lifecycle hooks run the method-specific selection logic, and `CacheManager` exposes the required KV through an `AttentionView`. Layer-end hooks coordinate state across layers, while the step-end hook applies cache updates before the next forward pass.

3.2 Extended Sparsity-Based State Management

As agent contexts grow through repeated interactions, persistent cache management becomes as vital as fast execution. SparseEngine addresses this by extending method-owned state across requests to support compacted-history reuse and controlled pruning.

Chain Cache. Multi-turn agents accumulate long trajectories, making prefix caching important for reusing prior computation (Yao et al., 2022; Zheng et al., 2024; Wadlom et al., 2026; Pan et al., 2025). Dynamic sparse-attention methods (e.g., Quest, OmniKV, NSA, and DSA) retain full KV histories and thus support conventional prefix caching. In contrast, eviction-based methods (e.g., SnapKV, KVzip, and H_2O) violate the resident-KV assumption by discarding entries.



(a) Radix Prefix Cache + Quest **(b) Chain Cache + SnapKV**

Figure 3: **Radix prefix caching with Quest and Chain Cache with SnapKV.** Conventional radix prefix caching retains complete KV blocks and attaches the summaries required by Quest. After SnapKV evicts physical KV entries, Chain Cache preserves the logical token prefix and reuses the retained KV and method state, so a continuation prefills only the new suffix.

Chain Cache enables these eviction-based methods to reuse compacted state across turns. Its `ChainCacheIndex` associates a session with its retained per-layer KV, method-specific metadata, and logical token prefix. As illustrated in Figure 3, the logical prefix remains available for matching even when SnapKV has removed some of its physical KV entries. A continuation can therefore resume from the retained KV and method state instead of reconstructing the complete prefix. This is particularly important when different layers retain different token positions, as in H_2O and SnapKV, or when continuation requires persistent statistics such as H_2O's accumulated attention scores.

Each session receives a `chain_id` that identifies its retained state. A continuation must match the previously processed logical prefix, although not every matched token needs a resident KV entry. When a request completes, its chain becomes idle while `CacheManager` retains the per-layer KV and associated metadata. A matching continuation reattaches this state and prefills only the unprocessed suffix, preserving the method's previous cache decisions. When capacity is required, idle chains and their state are reclaimed in least-recently-used order. Appendix B.2 provides the implementation details.

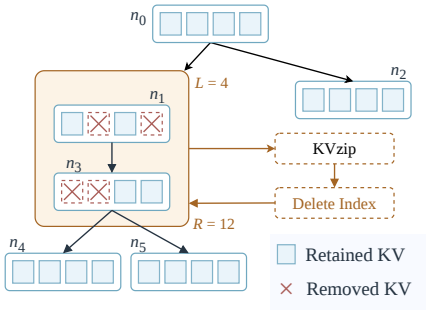


Figure 4: **Controllable prefix-cache pruning.** Within an application-specified history interval, a sparse policy selects retained KV entries; SparseEngine releases the rest while preserving the logical prefix for matching.

Sparsity-Based Prefix-Cache Pruning. Agent trajectories contain regions with different KV utility, including user inputs, tool results, reasoning traces, and outputs (Matam & Kim, 2026); for example, a tool response may lose value after its relevant information is summarized. SparseEngine therefore applies sparse scoring policies such as KVzip (Kim et al., 2025) to application-selected history regions, allowing the application to specify both the pruning interval and KV retention ratio. Concretely, the application submits the request's complete `token_ids` sequence and a pruning interval $[L, R)$ with zero-based boundaries, defining $\mathcal{T} = \{i \mid L \leq i < R\}$. A scoring

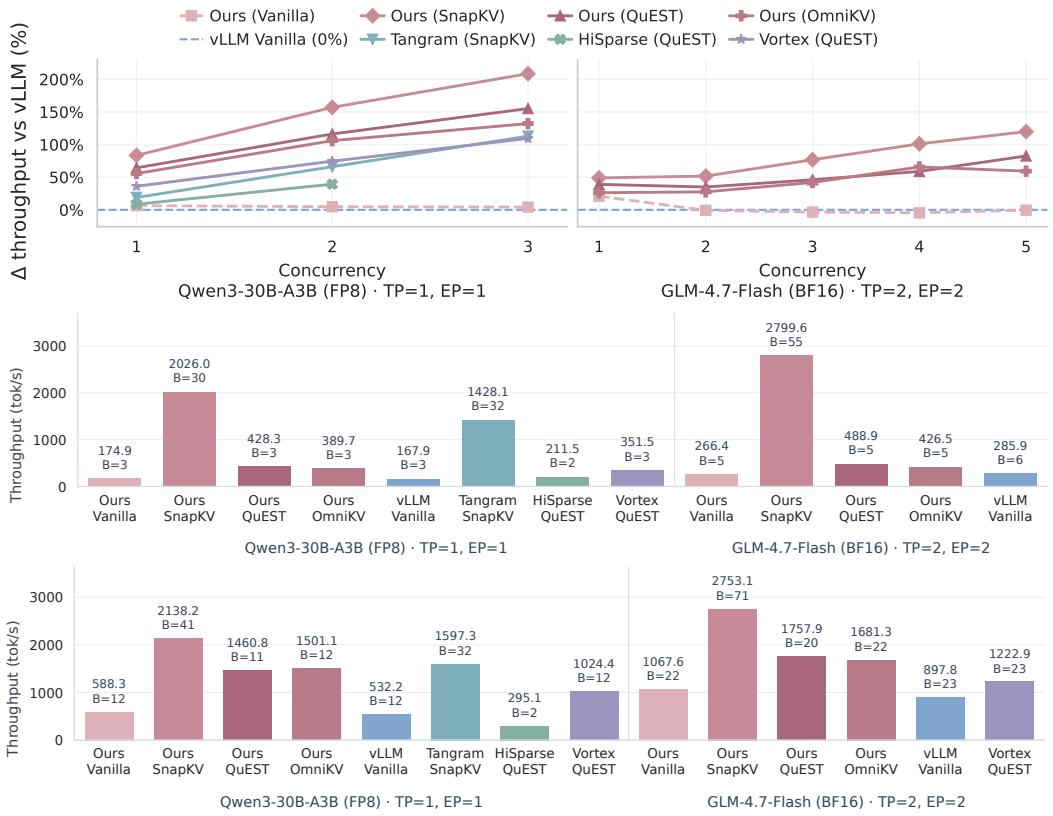


Figure 5: **Decode throughput on Qwen3-30B-A3B (left) and GLM-4.7-Flash (right).** The upper plot shows throughput improvement over vanilla vLLM at matched batch sizes with 128K-token inputs, with vanilla vLLM defining the 0% baseline. The middle and lower plots show absolute throughput at each system–method pair's largest successfully tested batch size with 128K- and 32K-token inputs, respectively. Bar annotations report throughput and batch size (B); the largest tested batch size need not be the system's maximum capacity. Vortex encounters OOM on GLM with 128K-token inputs even at batch size 1. HiSparse and Tangram do not support the evaluated MLA.

policy, such as SnapKV or KVzip, selects the positions to retain within $\mathcal{T}$ and returns a mask. The cache manager applies this mask only to idle blocks with no active references or transfers in flight.

Let P denote the logical token prefix and $\mathcal{R}$ the positions with resident KV. Retaining selected positions $\mathcal{I} \subseteq \mathcal{R} \cap \mathcal{T}$ yields:

$$P' = P, \qquad \mathcal{R}' = (\mathcal{R} \setminus \mathcal{T}) \cup \mathcal{I}. \tag{2}$$

The cache manager releases the physical slots excluded from $\mathcal{R}'$ but leaves the logical prefix P unchanged. Subsequent requests can therefore match the same prefix while reusing only its retained KV. Figure 4 illustrates KVzip-based selection over $[L, R)$ with $L = 4$ and $R = 12$. Appendix B.2 details the pruning metadata and cache operations.

4 Experiments

4.1 Experimental Setup

We evaluate whether SparseEngine preserves the quality of the original sparse methods on the long-context benchmarks LongBenchV1 (Bai et al., 2024) and LongBenchV2 (Bai et al., 2025), and assess reasoning quality on AIME (Maxwell-Jia, 2024). We further evaluate the impact of sparsity on multi-turn agent task quality using SWE-Bench Lite (Jimenez et al., 2024) and Claw-Eval (Ye et al., 2026). We follow the evaluation metrics defined by each benchmark.

For quality evaluation, we compare against the original sparse-method implementations where available to assess whether SparseEngine preserves their task quality, and against full attention

Table 1: **Sparse-method quality on LongBench V1 and V2.** Deltas below scores report SparseEngine minus the reference implementation in percentage points. Following official evaluation metrics, average score is the six-category average for V1 and accuracy over all examples for V2.

Method	LongBenchV1							LongBenchV2						
	S-Doc	M-Doc	Summ.	F-Shot	Synth.	Code	Avg.	S-Doc	M-Doc	Hist.	Learn.	Struct.	Code	Avg.
GLM-4.7-Flash†	41.6	47.1	27.8	71.0	52.5	65.7	**51.0**	34.3	28.8	20.5	29.6	21.2	46.0	**31.4**
SnapKV	41.0	47.2	26.9	69.8	52.5	65.2	**50.4**	36.0	31.2	20.5	32.1	30.3	42.0	**33.2**
OmniKV	41.7	47.1	27.8	70.6	52.5	65.1	**50.8**	35.4	28.0	23.1	33.3	42.4	42.0	**33.4**
Quest	41.4	47.2	27.7	71.2	52.5	65.4	**50.9**	34.9	29.6	23.1	40.7	27.3	42.0	**33.8**
StreamingLLM	34.7	41.7	26.4	69.2	53.0	64.1	**48.2**	34.3	32.0	23.1	34.6	30.3	32.0	**32.4**
PyramidKV	39.1	46.4	26.2	70.1	53.0	64.6	**49.9**	35.4	26.4	25.6	34.6	24.2	36.0	**31.6**
Llama-3.1-8B	43.3	46.6	28.8	69.3	55.8	60.2	**50.7**	33.7	31.2	10.3	25.9	27.3	36.0	**29.8**
	-0.1	+0.0	+0.0	-0.1	+0.8	+0.7	+0.2	-1.2	+1.6	-5.1	+2.5	-3.0	+2.0	+0.0
SnapKV	43.4	46.6	27.0	68.4	55.9	60.1	**50.2**	34.3	31.2	10.3	24.7	27.3	36.0	**29.8**
	+0.0	+0.0	-0.1	-0.5	+0.8	+0.1	+0.1	+0.6	+0.8	-2.6	+0.0	-3.0	+0.0	+0.0
OmniKV	43.2	46.7	29.0	69.1	56.2	60.1	**50.7**	33.1	30.4	10.3	24.7	27.3	38.0	**29.4**
	-0.2	+0.5	-0.1	+0.0	+1.9	+0.0	+0.4	+0.0	+0.8	-2.6	-1.2	-3.0	+2.0	-0.2
Quest	43.1	46.1	28.6	68.8	55.4	59.0	**50.2**	29.1	28.8	12.8	24.7	24.2	32.0	**27.0**
	+0.2	-0.2	-0.3	-0.2	+0.5	-0.1	+0.0	-2.3	-0.8	+7.7	+1.2	+3.0	+0.0	+0.0
RetroInfer	43.6	46.5	28.9	69.4	55.9	60.5	**50.8**	32.6	30.4	10.3	24.7	27.3	38.0	**29.2**
	+0.2	+0.1	-0.1	+0.0	+0.3	+0.5	+0.2	-0.6	+0.0	+0.0	+1.2	-3.0	-2.0	-0.4
StreamingLLM	33.9	40.1	25.5	67.0	52.3	59.4	**46.4**	33.7	32.8	10.3	19.8	30.3	34.0	**29.2**
	+0.1	+0.5	+0.1	-0.1	+0.5	+0.3	+0.2	-0.6	+1.6	-2.6	-4.9	+3.0	+2.0	-0.4
PyramidKV	43.4	46.7	27.1	69.0	55.9	59.0	**50.2**	32.6	28.8	10.3	29.6	33.3	38.0	**30.0**
	-0.1	+0.2	-0.1	+0.0	+0.8	-0.6	+0.0	+0.0	-2.4	-2.6	+2.5	+3.0	+4.0	+0.2
DeltaKV	43.5	46.2	27.8	69.5	55.6	59.7	**50.4**	32.6	34.4	10.3	22.2	30.3	32.0	**29.4**
	+0.0	+2.0	-0.2	+1.7	+1.8	-0.8	+0.8	+0.6	+2.4	-5.1	-9.9	+3.0	+2.0	-0.8
Palu	36.6	36.4	25.2	66.4	47.0	39.0	**41.8**	32.0	28.8	18.0	28.4	30.3	18.0	**28.0**
	+0.3	+0.4	+0.0	+0.6	+0.0	-0.2	+0.2	+1.7	+0.0	-2.6	+2.5	+0.0	+2.0	+1.0
TurboQuant	43.5	46.4	29.1	68.8	54.6	59.6	**50.3**	36.0	27.2	12.8	27.2	39.4	38.0	**31.0**
	+0.3	-0.7	+0.0	+0.0	+0.1	-0.6	-0.1	+3.4	-6.4	+0.0	+4.9	+12.1	+8.0	+2.0

to measure the quality impact of sparsity. For serving performance, we compare SparseEngine against Vortex (Chen et al., 2026b), HiSparse (Xie et al., 2026), Tangram (Kim et al., 2026), and vLLM (`v0.26.0`) (Kwon et al., 2023). Appendix C gives the benchmark protocols and hardware configurations.

For grouped-query attention (GQA), we evaluate dense models represented by Llama-3.1-8B-Instruct (Dubey et al., 2024) and Qwen3-4B-Thinking-2507 (Yang et al., 2025), as well as the MoE model Qwen3-30B-A3B-Instruct-2507 (Yang et al., 2025). For multi-head latent attention (MLA), we evaluate the MoE model GLM-4.7-Flash (Zeng et al., 2025). We also evaluate Qwen3.6-27B (Yang et al., 2025), a dense hybrid model that combines linear attention with GQA. Appendix C.8 lists the full set of supported models and sparse methods. The hardware platforms used across our experiments include NVIDIA H100, H20, RTX PRO 6000, 4090, and 5090 GPUs.

4.2 Overall Serving Performance and Quality

Long-Context Decode Performance. SparseEngine improves long-context decode efficiency. Physical KV eviction lets SparseEngine sustain substantially larger tested decode batches on the same GPU configuration. At each system–method pair's largest successfully tested batch size (Figure 5, middle), SparseEngine with SnapKV delivers approximately $10\times$ the aggregate decode throughput of vanilla vLLM on both models. Dynamic sparse attention reduces attention computation by accessing only a selected subset of the KV cache at each decode step, improving throughput at matched batch sizes. With Quest and OmniKV, SparseEngine reaches roughly $1.5\times$–$2.6\times$ the decode throughput of vanilla vLLM at several matched batch sizes across the two evaluated models (upper). These results capture complementary benefits of sparse inference: lower attention cost and higher concurrency within the same GPU memory budget. The lower plot in Figure 5 shows that the throughput benefit from larger tested batches also extends to 32K-token inputs. Appendix C.6 also reports the absolute decode throughput of Figure 6. Appendix C.7 details the 32K settings and complementary results.

Efficiency within Specialized Systems' Supported Methods. We further conduct fair, method-specific comparisons against baselines to examine whether SparseEngine's more general implementation comes at the expense of efficiency. Quest directly matches **Vortex** and **HiSparse**'s page-selection abstraction, and SnapKV directly matches **Tangram**'s KV-retention abstraction (Chen et al., 2026b; Kim et al., 2026; Xie et al., 2026). At a matched batch size of two on Qwen3 with 128K-token inputs (Figure 5, upper), SparseEngine with Quest delivers $1.24\times$ Vortex's and $1.55\times$ HiSparse's decode throughput. With SnapKV, SparseEngine delivers $1.55\times$ Tangram's throughput at the same batch size. SparseEngine remains ahead at every other common measured batch size in the upper plot. These method-matched results show that SparseEngine's broader method support is compatible with efficient execution even for methods that directly fit the competing systems' abstractions. Next, we show that our acceleration does not sacrifice the quality of the original sparse methods.

Sparse-Method Quality This experiment evaluates whether sparse methods retain task quality when implemented within SparseEngine. We evaluate diverse sparse methods on Llama-3.1-8B and GLM-4.7-Flash, using full attention (Vanilla) and the original implementations as references for each model and method. Table 1 reports the results, with method-specific settings in Appendix C.1. Across the 20 overall-score comparisons on LongBench V1 and V2, SparseEngine achieves a mean difference of $+0.17$ points from the reference implementations, with a population variance of 0.32 squared points. These small aggregate differences indicate that SparseEngine closely preserves the task quality of the evaluated methods. Notably, because top-p sampling introduces randomness and the number of instances in LongBench V2 subtasks is relatively small, individual subtask scores show noticeable fluctuations, although the average score remains stable.

Table 2: **AIME 2024 results.** Acc. (%) averages 60 responses (two per problem). Speedup is relative to Qwen3-4B-Thinking.

Method	Acc.	E2E Speedup
Qwen3-4B	81.7	1.00×
OmniKV	80.0	1.51×
Quest	76.7	1.28×
Quest (`Vortex`)	75.0	0.91×
SnapKV (`Low`)	58.3	1.66×
SnapKV (`Mid`)	80.0	1.33×
SnapKV (`High`)	85.0	1.05×
PyramidKV	53.3	1.66×
R-KV	51.7	1.30×
StreamingLLM	18.3	3.36×

End-to-End Reasoning Performance and Efficiency. Table 2 evaluates task accuracy and end-to-end generation speed on AIME 2024 with Qwen3-4B-Thinking-2507. In SnapKV, `Low`, `Mid`, `High` represent sparsity budgets of 4k, 8k, and 16k, respectively. These results also reveal a broad quality-efficiency trade-off: methods offering larger speedups generally suffer greater accuracy drops. The speedup here is lower than that in Figure 5 because the evaluated contexts are shorter. The shorter contexts in this workload reduce attention's share of execution time relative to the long-context decode benchmark, leaving less computation for sparse attention to eliminate. Appendix C.2 details the evaluation protocol and the method-specific concurrency limits, which reflect different GPU memory requirements.

Table 3: **SWE-bench Lite resolved tasks (%) with mini-SWE-agent.**

Model	Vanilla	SnapKV	H_2O	Quest	OmniKV
GLM-4.7-Flash	25.0	24.7	13.0	25.3	26.0
Qwen3-30B-A3B	5.3	8.3	–	7.3	5.3

Table 4: **Claw-Eval quality on Qwen3.6-FP8.**

Method	Avg. score (%)	Pass@1 (%)
Vanilla	67.4	50.5
OmniKV	68.0	51.1

4.3 Performance of Cache State Management

Chain Cache Quality on Agent Tasks We evaluate chain cache with mini-SWE-agent on SWE-bench Lite and with Claw-Eval on text-only agent tasks. SnapKV and H_2O retain their compacted KV and method state through Chain Cache, while Quest and OmniKV use radix prefix caching. Table 3 reports task success alongside a full-attention reference for each model, using the model-specific settings in Appendix C.3. SnapKV with Chain Cache achieves a task success rate close to the full-attention reference on GLM and resolves more tasks than the reference on Qwen3, demonstrating multi-turn agent execution with retained compacted state. Table 4 reports Qwen3.6 results on Claw-Eval. OmniKV maintains task quality close to the reference, with settings in Appendix C.5. Experiments demonstrate that, as long as an overly aggressive sparsity budget is not used, Chain Cache is able to maintain robust quality on complex multi-turn agent tasks.

Quality of Prefix Pruning We evaluate prefix-cache pruning during mini-SWE-agent execution on SWE-bench Lite (Table 5). Using KVzip-based global scoring to retain 20% of the aligned KV tokens in targeted tool results incurs a slight drop in performance. However, when we delay pruning with `lag=4` (i.e., when a new tool result arrives, it prunes only the fifth most recent tool-result round, leaving the latest four rounds intact), the original performance is immediately restored. This policy achieves a 25.0% task success rate, illustrating the potential of adapting pruning to an agent's use of recent evidence, as agent dialogues indeed often contain many low-value tool results. The result motivates further exploration of pruning thresholds and selection policies within the same cache-management abstraction. Appendix C.4 details the protocol and configurations.

Table 5: **SWE-bench Lite resolved tasks (%) with mini-SWE-agent and tool-result prefix pruning.** All runs use 300 tasks; settings are in Appendix C.4.

Method	Resolved (%)
Vanilla	23.7
Quest	21.3
OmniKV	22.3
OmniKV (Lag=4)	25.0

End-to-End Agent Serving Performance Table 6 complements the agent quality evaluation with end-to-end replay of SWE-bench Lite trajectories on GLM-4.7-Flash and a dedicated Gasai agent trace on Qwen3-30B-A3B FP8. Gasai is used only to evaluate system efficiency. Replay fixes recorded outputs and includes prefill, decoding, and scheduling, together with simulated tool waits. All methods use the same GPU configuration, with concurrency selected for each method's KV memory requirements. SnapKV with an 8K cache budget and Chain Cache reach a **2.24**× end-to-end speedup over Vanilla on Gasai. Its memory savings support higher concurrency, while Chain Cache retains compacted state for reuse across turns. OmniKV also accelerates both workloads at matched concurrency, showing that dynamic sparse attention remains effective with cross-turn prefix reuse. Together, these results show how SparseEngine translates both sparse computation and KV memory savings into end-to-end serving efficiency.

5 Discussions

Agent-Assisted Method Integration. Agent-assisted development creates an opportunity to combine broad method support with specialized execution. Frameworks often reduce implementation effort by routing methods through a common representation or workflow. As coding agents lower the cost of specialized implementations, framework design can give each method greater freedom over its state and computation. SparseEngine provides such a boundary through explicit lifecycle hooks and method-owned cache state. Building on this contract, an integration harness could guide agents through implementing new methods using reference modules and automated checks for method fidelity and cache lifecycle correctness. Agents could then use test feedback to refine their implementations within a clearly defined scope. This development model would accelerate method integration while preserving method-specific optimizations and access to shared serving capabilities.

Table 6: **End-to-end agent trace replay performance.** Speedup is relative to each workload's Vanilla at the listed batch sizes. Gasai measures system efficiency only.

Method	Approx. Max BS	Time (min)	E2E Speedup	E2E Out. tok/s
SWE-lite: GLM-4.7-Flash				
Vanilla	40	98.1	1.00×	302.0
Quest	32	92.2	1.06×	321.3
OmniKV	40	62.4	1.57×	474.5
SnapKV (High)	52	82.0	1.20×	361.2
H_2O	80	48.5	2.02×	610.8
Gasai: Qwen3-30B-A3B FP8				
Vanilla	32	49.9	1.00×	346.3
Quest	32	58.9	0.85×	293.5
OmniKV	32	30.6	1.63×	564.2
SnapKV (Mid)	72	22.2	2.24×	777.3
SnapKV (High)	52	30.0	1.66×	575.9
H_2O	80	25.0	1.99×	690.2

Serving Natively Sparse Models. Native sparse attention makes sparse-state management an increasingly central part of model serving. DeepSeek-V3.2 incorporates sparse attention into the model architecture (DeepSeek-AI et al., 2025). Cross-layer index reuse, explored by OmniKV for inference-time sparsification, also underlies IndexCache's acceleration of learned sparse attention (Hao et al., 2025; Bai et al., 2026). These mechanisms fit naturally with SparseEngine's lifecycle hooks, which expose when selection state is produced and reused while preserving method-specific cache representations. Native sparse execution also opens opportunities for complementary KV optimizations. For selection-based models that retain full KV histories, eviction can control cache growth, while quantization and compression can reduce the storage cost of retained state. SparseEngine provides a common foundation for exploring these combinations and extending their state reuse across agent turns.

AI Use Statement

We used AI tools to polish the language of the manuscript and assist with implementing portions of the code. The authors take full responsibility for the manuscript and the accompanying implementation.

Ethics Statement

This work studies efficient inference and cache management for large language models. Its intended benefit is to reduce the computational and memory costs of long-context applications. The proposed system does not address the safety or biases of the underlying models, and deployment should retain appropriate safeguards for the intended application.

Reproducibility Statement

Appendix C documents the evaluation protocols, with hardware configurations and method-specific hyperparameters provided in the corresponding subsections. Appendix B describes the lifecycle interfaces and cache-management mechanisms used in SparseEngine. The implementation and reproduction materials are available in an anonymous repository: https://anonymous.4open.science/r/SparseEngine.

References

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. In *USENIX Symposium on Operating Systems Design and Implementation*, pp. 117–134, 2024. URL https://api.semanticscholar.org/CorpusID:268249103.

M. M. Astrahan and D. D. Chamberlin. Implementation of a structured English query language. *Communications of the ACM*, 18(10):580–588, 1975. doi: 10.1145/361020.361215. URL https://doi.org/10.1145/361020.361215.

Yu Bai, Qian Dong, Tingyu Jiang, Xin Lv, Zhengxiao Du, Aohan Zeng, Jie Tang, and Juanzi Li. Indexcache: Accelerating sparse attention via cross-layer index reuse. *ArXiv*, abs/2603.12201, 2026. URL https://api.semanticscholar.org/CorpusID:286493297.

Yushi Bai, Xin Lv, Jiajie Zhang, Hong Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juan-Zi Li. Longbench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 3119–3137, 2024. URL https://api.semanticscholar.org/CorpusID:261245264.

Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, Jie Tang, and Juan-Zi Li. Longbench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3639–3664, 2025. URL https://api.semanticscholar.org/CorpusID:274859535.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, and Wen Xiao. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *ArXiv*, abs/2406.02069, 2024. URL https://api.semanticscholar.org/CorpusID:270226243.

Zefan Cai, Wen Xiao, Hanshi Sun, Cheng Luo, Yi-Kai Zhang, Ke Wan, Yucheng Li, Yeyang Zhou, Li-Wen Chang, Jiuxiang Gu, Zhen Dong, Anima Anandkumar, Abedelkadir Asi, and Junjie Hu. R-kv: Redundancy-aware kv cache compression for reasoning models. *Advances in neural information processing systems*, 38:60980–61005, 2025. URL https://api.semanticscholar.org/CorpusID:279070957.

Chi-Chih Chang, Wei-Cheng Lin, Chien-Yu Lin, Chong-Yan Chen, Yu-Fang Hu, Pei-Shuo Wang, Ning-Chi Huang, Luis Ceze, and Kai-Chiang Wu. Palu: Compressing kv-cache with low-rank

projection. *ArXiv*, abs/2407.21118, 2024. URL https://api.semanticscholar.org/CorpusID:271571616.

MiniMax Aili Chen, Aonian Li, Baichuan Zhou, Bangwei Gong, Binyan Jiang, Bo Dan, Chang Yu, Chao Wang, Chengnuo Ma, Chengzhi Zhong, Chen Zhu, Chengjun Xiao, Chengyi Yang, Chengyu Du, Chenyang Zhang, Chi Zhang, Chuang Huang, Chunhao Zhang, Chunhui Du, Chunyu Zhao, Cong Guo, Da Chen, Deming Ding, Dianjun Sun, Dongyu Zhang, Enhui Yang, Fei Richard Yu, Guangxuan Zheng, et al. The minimax-m2 series: Mini activations unleashing max real-world intelligence. *ArXiv*, abs/2605.26494, 2026a. URL https://api.semanticscholar.org/CorpusID:288672090.

Yaoqi Chen, Jin-Kai Zhang, Baotong Lu, Qianxi Zhang, Chengruidong Zhang, Jing Liu, Jingjia Luo, Di Liu, Huiqiang Jiang, Qi Chen, Bailu Ding, Xiao Yan, Jia-Wei Jiang, Chen Chen, Mingxing Zhang, Cheng Li, Yuqing Yang, Fan Yang, and Mao Yang. Retroinfer: A vector storage engine for scalable long-context llm inference. *Proc. VLDB Endow.*, 19:1016–1031, 2025. URL https://api.semanticscholar.org/CorpusID:287782427.

Zhuoming Chen, Xin Zhong, Qilong Feng, Ranajoy Sadhukhan, Yang Zhou, Michael Qizhe Shieh, Zhihao Jia, and Beidi Chen. Vortex: Efficient and programmable sparse attention serving for ai agents. *ArXiv*, abs/2606.06453, 2026b. URL https://api.semanticscholar.org/CorpusID:288977094.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Ban Lin, Bing Xue, Bing-Li Wang, Bin Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Erhang Li, Fangqi Zhou, Fangyun Lin, Fucong Dai, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Li, Hao Liang, Haoran Wei, Haowei Zhang, Hao sheng Luo, Haozhe Ji, Honghui Ding, Hongxuan Tang, Huan Cao, Huazuo Gao, Huixian Qu, Hui Zeng, Jialiang Huang, Jiashi Li, Jiaxin Xu, et al. Deepseek-v3.2: Pushing the frontier of open large language models. *ArXiv*, abs/2512.02556, 2025. URL https://api.semanticscholar.org/CorpusID:283448719.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, A. S. Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany M. Biron, Binh Tang, Bobbie Chern, Char lotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. URL https://api.semanticscholar.org/CorpusID:271571434.

Qihang Fan, Huaibo Huang, Zhiying Wu, Bingning Wang, and Ran He. Flashprefill v2: Block-sparse prefill attention for long-context llm serving. *arXiv preprint arXiv:2608.19758*, 2026. URL https://api.semanticscholar.org/CorpusID:291257315.

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S. Kevin Zhou. Ada-kv: Optimizing kv cache eviction by adaptive budget allocation for efficient llm inference. *Advances in Neural Information Processing Systems*, 38:113152–113188, 2025. URL https://api.semanticscholar.org/CorpusID:271218006.

Yuanhang Gao, Xiangrui Yang, Yuanfeng Chen, Hongjia Chen, Qianru Lv, Wenfei Wu, and Dongsheng Li. vtoken: Token-level virtualization for reclaimable kv caches. *arXiv preprint arXiv:2608.13263*, 2026. URL https://api.semanticscholar.org/CorpusID:291084792.

Jitai Hao, Yuke Zhu, Tian Wang, Jun Yu, Xin Xin, Bo Zheng, Zhaochun Ren, and Sheng Guo. Omnikv: Dynamic context selection for efficient long-context llms. In *International Conference on Learning Representations*, volume 2025, pp. 87443–87464, 2025. URL https://api.semanticscholar.org/CorpusID:278601790.

Jitai Hao, Qiang Huang, Yaowei Wang, Min Zhang, and Jun Yu. Deltakv: Residual-based kv cache compression via long-range similarity. *ArXiv*, abs/2602.08005, 2026. URL https://api.semanticscholar.org/CorpusID:285454299.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shun-Yu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024. URL https://api.semanticscholar.org/CorpusID:263829697.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2018. doi: 10.1145/3158154. URL https://doi.org/10.1145/3158154.

Hyungmin Kim, Minsoo Kim, Hongseok Kim, and Jungwook Choi. Tangram: Unlocking non-uniform kv cache compression for efficient multi-turn llm serving. *ArXiv*, abs/2606.06302, 2026. URL https://api.semanticscholar.org/CorpusID:288976234.

Jang-Hyun Kim, Jinuk Kim, Sangwoo Kwon, Jae W. Lee, Sangdoo Yun, and Hyun Oh Song. Kvzip: Query-agnostic kv cache compression with context reconstruction. *Advances in Neural Information Processing Systems*, 38:167563–167591, 2025. URL https://api.semanticscholar.org/CorpusID:278996887.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Haotong Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023. URL https://api.semanticscholar.org/CorpusID:261697361.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr F. Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems*, 37:22947–22970, 2024. URL https://api.semanticscholar.org/CorpusID:269303164.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. In *International Conference on Machine Learning*, 2024. URL https://api.semanticscholar.org/CorpusID:267413049.

Venkatesha Matam and Keonwoo Kim. Memdecay: Region-aware kv cache eviction for efficient llm agent inference. *ArXiv*, abs/2607.10582, 2026. URL https://api.semanticscholar.org/CorpusID:290135698.

Maxwell-Jia. AIME 2024 Dataset. Hugging Face, 2024. URL https://huggingface.co/datasets/Maxwell-Jia/AIME_2024. 30 problems from the 2024 American Invitational Mathematics Examination (AIME I and II).

Cleve Moler and Jack Little. A history of MATLAB. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–67, 2020. doi: 10.1145/3386331. URL https://doi.org/10.1145/3386331.

Piotr Nawrot, Robert Li, Renjie Huang, Sebastian Ruder, Kelly Marchisio, and E. Ponti. The sparse frontier: Sparse attention trade-offs in transformer llms. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 38667–38701, 2026. URL https://api.semanticscholar.org/CorpusID:278033756.

Zai-Feng Pan, Ajjkumar Patel, Yipeng Shen, Zheng-Ding Hu, Yue Guan, Wanlu Li, Lian-Hui Qin, Yida Wang, and Yu-Fei Ding. Kvflow: Efficient prefix caching for accelerating llm-based multi-agent workflows. *Advances in Neural Information Processing Systems*, 38:126246–126265, 2025. URL https://api.semanticscholar.org/CorpusID:280298721.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023. URL https://api.semanticscholar.org/CorpusID:258833055.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference. *ArXiv*, abs/2406.10774, 2024. URL https://api.semanticscholar.org/CorpusID:270559146.

Gemma Team, Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, Ian Ballantyne, Cormac Brick, Victor Cărbune, Michelle Casbon, et al. Gemma 4 technical report. *ArXiv*, abs/2607.02770, 2026. URL https://api.semanticscholar.org/CorpusID:289923375.

Jiayi Tian, Seyedarmin Azizi, Yequan Zhao, Erfan Baghaei Potraghloo, Sean McPherson, Sharath Nittur Sridhar, Zhengyang Wang, Zheng Zhang, Massoud Pedram, and Souvik Kundu. Skipkv: Selective skipping of kv generation and storage for efficient inference with large reasoning models. *Proceedings of Machine Learning and Systems*, 8:1496–1514, 2026. URL https://api.semanticscholar.org/CorpusID:283711700.

Noppanat Wadlom, Junyi Shen, and Yao Lu. Efficient llm serving for agentic workflows: A data systems perspective. *Proc. ACM Manag. Data*, 4:169:1–169:29, 2026. URL https://api.semanticscholar.org/CorpusID:286579910.

Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, Song Han, and Maosong Sun. Infllm: Training-free long-context extrapolation for llms with an efficient context memory. *Advances in neural information processing systems*, 37:119638–119661, 2024a. URL https://api.semanticscholar.org/CorpusID:267523068.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, volume 2024, pp. 21875–21895, 2024b. URL https://api.semanticscholar.org/CorpusID:263310483.

Zhiqiang Xie, Zhangheng Huang, Ting-Jun Huang, Ziyi Xu, Ruiyang Ma, and Christos Kozyrakis. Hisparse: Scaling sparse-attention decoding with hierarchical kv cache management. *arXiv preprint arXiv:2608.07009*, 2026. URL https://api.semanticscholar.org/CorpusID:290978977.

An Yang, An-Feng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bo-Wen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxin Yang, Jingren Zhou, Jingren Zhou, Junyan Lin, Kai Dang, Keqin Bao, Ke-Pei Yang, Le Yu, Li-Chun Deng, Mei Li, Min Xue, Ming-Ze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tian-Yi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yi-Chao Zhang, Ying-Er Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhen-Ru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://api.semanticscholar.org/CorpusID:278602855.

Qwen An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bo-Wen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxin Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yi-Chao Zhang, Yunyang Wan, Yuqi Liu, Zeyu Cui, Zhen-Ru Zhang, Zihan Qiu, Shanghaoran Quan, and Zekun Wang. Qwen2.5 technical report. *ArXiv*, abs/2412.15115, 2024. URL https://api.semanticscholar.org/CorpusID:274859421.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *ArXiv*, abs/2210.03629, 2022. URL https://api.semanticscholar.org/CorpusID:252762395.

Bowen Ye, Rang Li, Qibin Yang, Yuanxin Liu, Linli Yao, Hanglong Lv, Zhihui Xie, Chenxin An, Lei Li, Lingpeng Kong, Qi Liu, Zhifang Sui, and Tong Yang. Claw-eval: Towards trustworthy evaluation of autonomous agents. *arXiv preprint arXiv:2604.06132*, 2026. URL https://api.semanticscholar.org/CorpusID:287209548.

Gyeong-In Yu and Joo Seong Jeong. Orca: A distributed serving system for transformer-based generative models. In *USENIX Symposium on Operating Systems Design and Implementation*, 2022. URL https://api.semanticscholar.org/CorpusID:251734964.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Y. X. Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 23078–23097, 2025. URL https://api.semanticscholar.org/CorpusID:276408911.

Amir Zandieh, Majid Daliri, Majid Hadian, and Vahab S. Mirrokni. Turboquant: Online vector quantization with near-optimal distortion rate. In *International Conference on Learning Representations*, volume 2026, pp. 56418–56439, 2026. URL https://api.semanticscholar.org/CorpusID:278165621.

GLM-4.5 Team Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, Kedong Wang, Lucen Zhong, Mingdao Liu, Rui Lu, Shulin Cao, Xiaohan Zhang, Xuancheng Huang, Yao Wei, Yean Cheng, Yifang An, Yilin Niu, Yuan Wen, Yu Bai, Zhengxiao Du, Zi-Han Wang, Zilin Zhu, Bohan Zhang, Bosi Wen, Bowen Wu, et al. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *ArXiv*, abs/2508.06471, 2025. URL https://api.semanticscholar.org/CorpusID:280561359.

Rongzhi Zhang, Kuan Wang, Liyuan Liu, Shuohang Wang, Hao Cheng, Chao Zhang, and Yelong Shen. Lorc: Low-rank compression for llms kv cache with a progressive compression strategy. *ArXiv*, abs/2410.03111, 2024. URL https://api.semanticscholar.org/CorpusID:273162265.

Zhenyu (Allen) Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark W. Barrett, Zhangyang Wang, and Beidi Chen. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in neural information processing systems*, 36:34661–34710, 2023. URL https://api.semanticscholar.org/CorpusID:259263947.

Zihan Zhao, Baotong Lu, Shengjie Lin, Yizou Chen, Jing Liu, Yanqi Zhang, Ziming Miao, Ming-Chang Yang, Haiying Shen, Qi Chen, and Fan Yang. Unifying sparse attention with hierarchical memory for scalable long-context llm serving. *ArXiv*, abs/2604.26837, 2026. URL https://api.semanticscholar.org/CorpusID:287902044.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christoforos E. Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024. URL https://api.semanticscholar.org/CorpusID:266174771.

A Detailed Background and Notations

This appendix expands the background introduced in Section 2 with a common notation for attention and cache state, followed by the selection rules of four representative sparse methods. The equations describe their core mechanisms, with small examples to illustrate how their cache behavior differs.

A.1 Attention and Cached Representations

Notation. Consider one request with a prompt of n tokens. We use i for a query's token position and $j \le i$ for a key's position. These positions differ from the execution-step index t in Section 3: a prefill step can process several positions, while a standard decode step processes one position per request. Table 7 summarizes the notation.

Attention over Selected Positions. For one layer and head, write vectors as rows and let d_k be the key dimension. Given a nonempty set $\mathcal{S}_i \subseteq [i]$, attention computes

$$z_{i,j} = \frac{q_i k_j^\top}{\sqrt{d_k}}, \quad \alpha_{i,j}(\mathcal{S}_i) = \frac{\exp(z_{i,j})}{\sum_{u \in \mathcal{S}_i} \exp(z_{i,u})}, \quad o_i = \sum_{j \in \mathcal{S}_i} \alpha_{i,j}(\mathcal{S}_i) v_j. \tag{3}$$

Dense causal attention uses $\mathcal{S}_i = [i]$. Sparse attention restricts this set and normalizes over the selected positions. Physical eviction additionally changes $\mathcal{R}_i$, determining which entries remain

Table 7: **Notation for the background equations.** Layer and head indices are omitted when discussing a single layer and head.

Symbol	Meaning
ℓ, a	Transformer layer and query-head indices.
n, i, j	Prompt length, current query position, and a key position.
$[i]$	The causal history $\{1, \cdots, i\}$, including the current token.
$q_{\ell,i}^{(a)}$	Query vector for head a at position i.
$k_{\ell,j}^{(a)}, v_{\ell,j}^{(a)}$	Logical key and value used by query head a at position j.
$z_{i,j}, \alpha_{i,j}$	Query–key logit and normalized attention weight.
$\mathcal{S}_i, \mathcal{R}_i$	Positions read by attention at i, and positions retained after processing i.
$\mathrm{TopK}_b(s; \mathcal{A})$	Indices of the b largest scores $s(j)$ among candidates $j \in \mathcal{A}$, or all candidates if fewer than b exist.
$B, \mathcal{W}_i$	Retained-token budget and protected recent-token window.

available to later queries. For example, reading positions $\{1, 4, 8\}$ from an eight-token cache can leave all eight entries stored, whereas retaining only these positions prevents later queries from recovering the other five entries from that cache.

MHA, MQA, and GQA. Let H_q and H_{kv} be the numbers of query heads and KV heads, and let $g(a)$ map query head a to its KV head. The logical vectors above satisfy $k_{\ell,j}^{(a)} = k_{\ell,j}^{g(a)}$ and $v_{\ell,j}^{(a)} = v_{\ell,j}^{g(a)}$. Multi-head attention (MHA) uses a separate KV head for each query head. Multi-query attention (MQA) shares one KV head across all query heads, while grouped-query attention (GQA) shares each KV head within a group. For L layers with uniform head dimensions and N cached tokens, explicit KV contains $LNH_{kv}(d_k + d_v)$ scalar elements, excluding metadata. Sharing KV across heads changes storage cost while preserving the attention computation for each query head.

MLA. Multi-head latent attention represents the content keys and values through a shared low-dimensional latent vector. Suppressing the layer index, let h_j be the hidden state and define

$$c_j = h_j W^{DKV}, \qquad k_j^{C,(a)} = c_j U_a^K, \qquad v_j^{(a)} = c_j U_a^V. \tag{4}$$

Here W^{DKV} projects to the latent dimension, while U_a^K and U_a^V recover a head's content key and value. MLA also stores a positional key k_j^R, with a corresponding positional query $q_i^{R,(a)}$. The query's content component is $q_i^{C,(a)}$. Writing d_C and d_R for their dimensions, the logit and output can be evaluated as

$$z_{i,j}^{(a)} = \frac{(q_i^{C,(a)}(U_a^K)^\top)c_j^\top + q_i^{R,(a)}(k_j^R)^\top}{\sqrt{d_C + d_R}}, \quad o_i^{(a)} = \left(\sum_{j \in \mathcal{S}_i} \alpha_{i,j}^{(a)}(\mathcal{S}_i) c_j\right) U_a^V. \tag{5}$$

This rearrangement computes with cached (c_j, k_j^R) without explicitly storing the full per-head content keys and values. With latent dimension d_c, these tensors contain $LN(d_c + d_R)$ scalar elements under uniform layer dimensions. Thus the logical attention rule does not prescribe the physical cache format. In SparseEngine, the cache manager owns that format and exposes it to a compatible backend through an attention view (Section 3.1).

A.2 Inference Execution and Cache Management

Prefill and Decode. Prefill evaluates the prompt with a causal mask, producing the state needed to sample the first output token. Each later decode step feeds the last sampled token through the model to obtain the next token distribution. Consequently, the last sampled token has no KV until it is processed in a subsequent forward pass. Chunked prefill partitions prompt positions into consecutive intervals. A chunk attends to the preceding cached context and the causally available positions within that chunk. The scheduler can interleave these chunks with decode work from other requests, while continuous batching updates the participating requests between steps.

Logical Positions and Physical Blocks. Paged cache management (Kwon et al., 2023) allocates KV in fixed-capacity physical blocks and maintains a block table for each request. For blocks of p tokens, an uncompressed token position j has logical block number $\lfloor (j-1)/p \rfloor$ and offset $(j-1) \bmod p$. The block table resolves this logical block to its physical allocation. This indirection allows noncontiguous storage and lets attention kernels locate a request's KV. Sparse compaction additionally needs retained-position mappings because physical offsets need no longer correspond to consecutive logical tokens.

Reuse and Capacity. Prefix caching (Zheng et al., 2024) matches a request's token prefix against previously processed history and reuses compatible cached state. The scheduler still needs to reserve capacity for the uncached suffix and subsequent generation. An attention budget measures how much context a computation reads, while a storage budget measures how much state remains allocated. Quest and OmniKV can read a small subset while retaining the complete history, potentially across memory tiers. H_2O and SnapKV also reduce retained KV through eviction. This distinction motivates the separate selection and cache-management interfaces in Section 3.1.

A.3 Representative Sparse Methods

The following equations describe one request. For H_2O, SnapKV, and Quest, layer and head indices are suppressed to emphasize the selection rule. Each method uses a budget with its own meaning: retained tokens for eviction, or tokens/pages read by a sparse attention operation.

H_2O: Accumulated Scores and Online Eviction. H_2O (Zhang et al., 2023) treats tokens receiving substantial accumulated attention as heavy hitters. Before processing position i, the available positions are $\mathcal{A}_i = \mathcal{R}_{i-1} \cup \{i\}$. Attention reads this set and adds its weights to persistent scores:

$$u_i(j) = u_{i-1}(j) + \alpha_{i,j}(\mathcal{A}_i), \qquad j \in \mathcal{A}_i, \tag{6}$$

where the new token starts with score zero before this update. For a cache budget B and a protected recent window $\mathcal{W}_i \subseteq \mathcal{A}_i$ of at most B positions, the retained set is

$$\mathcal{R}_i = \mathcal{W}_i \cup \mathrm{TopK}_{B-|\mathcal{W}_i|}(u_i; \mathcal{A}_i \setminus \mathcal{W}_i). \tag{7}$$

KV outside $\mathcal{R}_i$ is evicted, while the scores of retained entries persist for later updates. For example, with $B = 4$ and two protected recent tokens, the other two slots hold the highest-scoring older tokens. A recent token that leaves the protected window must compete on its accumulated score to remain cached. The recurrence explains why continuing H_2O requires both retained KV and aligned score state.

SnapKV: Prompt Selection from an Observation Window. SnapKV (Li et al., 2024) estimates which prompt entries will matter during generation by inspecting attention from the prompt's final w positions, $\mathcal{O} = \{n-w+1, \cdots, n\}$. For each earlier prompt position, it accumulates attention from these observation queries and pools neighboring scores:

$$u(j) = \sum_{i \in \mathcal{O}} \alpha_{i,j}([i]), \qquad s = \mathrm{Pool}_\kappa(u), \qquad j \in [n] \setminus \mathcal{O}. \tag{8}$$

The one-dimensional pooling operator has width κ and promotes neighborhoods around important positions. For example, max pooling sets $s(j) = \max_{r \in \mathcal{N}_\kappa(j)} u(r)$, where $\mathcal{N}_\kappa(j)$ is the local prefix neighborhood centered at j. For a prompt-cache budget $B \geq w$, the retained prompt is

$$\mathcal{R}_{\mathrm{prompt}} = \mathcal{O} \cup \mathrm{TopK}_{B-w}(s; [n] \setminus \mathcal{O}). \tag{9}$$

For example, an eight-token prompt with $w = 2$ and $B = 4$ retains positions 7 and 8 plus two earlier positions selected by the pooled scores. This selection is computed separately for attention heads, so their retained positions can differ. The selected prompt KV remains fixed during decoding, while newly generated tokens add their own KV. The observation window must therefore be available before prompt compaction, including when prefill is chunked.

Quest: Query-Dependent Page Selection. Quest (Tang et al., 2024) groups token positions into pages $\mathcal{P}_b$ and stores a minimum and maximum key value for each channel c:

$$m_{b,c} = \min_{j \in \mathcal{P}_b} k_{j,c}, \qquad M_{b,c} = \max_{j \in \mathcal{P}_b} k_{j,c}. \tag{10}$$

For query q_i, a page's importance estimate is

$$\widehat{z}_{i,b} = \frac{1}{\sqrt{d_k}} \sum_{c=1}^{d_k} \max\{q_{i,c} m_{b,c},\ q_{i,c} M_{b,c}\}. \tag{11}$$

This quantity upper-bounds the query–key logit of every token in the page. It need not equal any individual token's logit because different channels can attain their extrema at different positions. Quest ranks pages by this estimate and performs attention using the actual KV in the selected pages. For example, an eight-token history stored in four two-token pages can be searched through four page summaries before loading two selected pages for attention. Unselected pages remain stored and can be selected by a later query. New keys update the extrema of their pages, so the page summaries persist alongside the KV history.

OmniKV: Selection Reuse across Layers. OmniKV (Hao et al., 2025) exploits the similarity of important context positions across nearby layers during decoding. Designated observation (filter) layers inspect the context and produce token indices that subsequent sparse layers reuse. For illustration, a current-query selector can rank each position by its largest query–key logit across heads:

$$s_{f,i}(j) = \max_a z^{(a)}_{f,i,j}, \qquad \mathcal{J}_{f,i} = \mathrm{TopK}_b(s_{f,i}; [i]), \tag{12}$$

where f is an observation layer and b is the selected-token budget. This is a current-query scoring example from the method's released implementation. For a sparse layer ℓ assigned to observation layer $f(\ell)$, the defining reuse relationship is

$$\mathcal{I}_{\ell,i} = \mathcal{J}_{f(\ell),i}, \qquad o^{(a)}_{\ell,i} = \sum_{j \in \mathcal{I}_{\ell,i}} \alpha^{(a)}_{\ell,i,j}(\mathcal{I}_{\ell,i}) v^{(a)}_{\ell,j}. \tag{13}$$

Thus layers reuse token positions while computing their own attention weights over their own KV. For example, if an observation layer selects $\{1, 4, 8\}$, its associated sparse layers each read those positions from their respective caches. A later decode query can select a different set because historical KV has been retained. The engine needs to pass selection indices between layers while preserving each layer's cache state.

These mechanisms illustrate the lifecycle requirements used in Section 3.1. H_2O updates persistent statistics as inference advances, while SnapKV compacts prompt state after observation-window scoring. Quest consumes the current query together with persistent page summaries, and OmniKV carries selection results across layers.

B Lifecycle Implementation Details

Attention View Payloads. Separate prefill and decode view types pair a physical payload with request indices, context lengths, and token-slot or page-table coordinates. Payload types distinguish explicit KV tensors from MLA latent and positional tensors, allowing attention implementations to declare which representations they support. The cache manager resolves selections into the required physical inputs, including temporary reconstruction when needed. Reconstruction slots are released after the layer consumes them, while persistent KV remains under the cache manager's ownership.

CUDA Graph Execution. For supported CUDA Graph paths, cache managers, method runtimes, and attention implementations provide buffers whose addresses remain stable across replay. Step preparation updates slot mappings, lengths, and other mutable inputs in these buffers. The model runner handles capture and replay, while each component keeps its referenced storage alive. Graph and layout compatibility are checked when the execution path is configured. This division allows method-specific cache representations and update logic to participate in graph execution through the same lifecycle interfaces.

B.1 Scheduling Interfaces

MemoryOracle. The scheduler queries the runtime state on rank 0, whose cache manager provides the capacity reference for admission and step reservations. A sparse method's attention budget alone does not determine its memory requirement. Quest may read a small set of pages while retaining

a much larger cache. An eviction method may temporarily hold prompt KV before compaction, and a compressed method may need reconstruction space in addition to its persistent storage. The scheduler therefore queries method-specific capacity and reservation costs through `MemoryOracle`, implemented by the runtime state and cache-manager interfaces.

For prompt admission, the interface returns a set of named resource budgets and a request's cost in each budget. Let B_j be the remaining budget for resource j and $c_j(r)$ the reservation required by request r. The scheduler admits r only when

$$c_j(r) \leq B_j \quad \text{for every reported resource } j, \tag{14}$$

and deducts each reservation before considering another request. The cache manager defines the resources and their units. For example, PyramidKV can report a separate retained-slot budget for each layer, while DeltaKV reports budgets for full-attention layers, reference entries, and retained raw KV, with space reserved for reconstruction. This allows methods with heterogeneous storage to expose multiple capacity constraints to the same admission loop.

Step Reservations and Batching. Admission reserves capacity for the request, and each execution step also checks the space required for its next prefill chunk or decode token. The interface reports step costs, available slots, and prefill execution requirements. These requirements distinguish chunked prefill, full-prompt prefill, and prefill paths that use raw-KV offload. A compatibility key groups requests that can use the same prefill path. The scheduler builds a batch within the reported capacity and token limits, reserving space as requests are added. Requests that cannot fit remain queued or follow the configured failure policy. When execution requires reclamation or preemption, `RuntimeState` releases the associated payload through its owner before the capacity is reused.

B.2 State Reuse and Pruning Details

Chain Records and Continuation Boundaries. A chain record contains an identifier, its active or idle status, a method/configuration fingerprint, and the token count and digest at the processed boundary. A continuation must match the fingerprint and the exact logical prefix through that boundary before the retained cache can be attached. The processed boundary excludes the final sampled token when that token has not yet passed through the model, so its KV is computed as part of the continuation. The driver retains compact logical token IDs to preserve token identity for text-based continuation. Cancellation, failure, or preemption invalidates the chain and releases its payload and metadata together. Under tensor parallelism, the driver distributes the admission and victim plan so every rank applies the same state transition.

Pruning Metadata and Cache Operations. Applications can inspect radix prefixes, assign retention priorities, and request deletion of eligible subtrees. Interval pruning accepts a retained-token budget and a scoring policy for a block-aligned history interval. A zero budget removes all KV entries in that interval. After pruning, the cache manager records retained offsets within each logical block, and a pruning record marks affected continuations as using compressed history. Device capacity accounting and subsequent transfers follow the retained slots. The original logical token history remains the prefix-matching key, while attention uses the reduced physical payload.

C Experimental Settings

C.1 Sparse-Method Quality

Models and Hardware. Table 1 evaluates GLM-4.7-Flash and Llama-3.1-8B-Instruct with a maximum model length of 131,072 tokens, except for Llama Palu and RetroInfer on LongBench v1, which use 121,000 tokens. Tensor parallelism is 2 for GLM on LongBench except StreamingLLM and PyramidKV, which use TP1, and 1 for all other model-benchmark combinations.

Evaluation Protocol. LongBench v1 reports category scores and their six-category macro-average, using temperature 0, top-p 1, and top-k 1. The LongBench overall difference is their six-category **macro-average**. For the workbook-derived comparisons, the HF FullKV reference corresponds to Vanilla in the table. LongBenchV2 uses all examples with the zero-shot direct-answer prompt and the model's chat template. Inputs exceeding 120,000 tokens undergo middle truncation before chat

Table 8: **Method-specific settings for LongBench results in Table 1.** Budgets and windows are in tokens. The method-budget column retains each method's budget definition.

Method	Sink	Recent	Method budget	Other settings
SnapKV	0	32	2,016 selected	32 scoring window; pooling kernel 7
Quest	–	–	2,048 selected	16-token pages; first two layers dense
OmniKV	0	32	2,048 selected	Full layers: GLM $\{0, 3, 8, 16, 19, 25, 31, 37, 44\}$; Llama $\{0, 2, 7, 13, 16, 26\}$
H_2O	–	–	2,048 prefill, 2,048 decode	Prefill chunk eviction; recent ratio 0.5; requested prefill scoring window 128
StreamingLLM (GLM)	8	4,096	–	–
PyramidKV (GLM)	64	512	4,096 selected	Layer ratios 0.6 to 0.01 (linear); 32 scoring window
StreamingLLM (Llama)	4	2,044	–	–
PyramidKV (Llama)	–	8	3,978 base selected	Layer ratios 1 to 0.02564 (linear), averaging approximately 2,048; 8 scoring window; pooling kernel 7
DeltaKV	8	128	2,048 decode; 4,096 prefill	Full layers $\{0, 2, 7, 13, 16, 26\}$; latent dimension 512; center ratio 0.1; 4-bit latent and full-layer KV, group size 32
RetroInfer	4	64	–	Retrieval ratio 0.018; estimation ratio 0.232; average cluster size 16
Palu	–	–	All tokens	Grouped SVD (group size 1); K/V rank 96; BF16 factors
TurboQuant	–	–	All tokens	4-bit K/V; Gaussian codebook (SparseEngine), Lloyd–Max keys and uniform values (vLLM)

Table 9: **Method-specific settings for LongBenchV2 results in Table 1.** Budgets and windows are in tokens. The method-budget column retains each method's budget definition. SnapKV sink values are GLM/Llama.

Method	Sink	Recent	Method budget	Other settings
SnapKV	0/64	256	4,096 selected	32 scoring window; pooling kernel 7
Quest	64	256	4,096 selected (4,416 total)	16-token pages; first two layers dense
OmniKV	64	256	4,096 selected	Full-attention layers: GLM $\{0, 3, 8, 16, 19, 25, 31, 38, 44\}$; Llama $\{0, 2, 7, 13, 16, 26\}$
H_2O	–	–	8,192 prefill; 4,096 configured decode	128 prefill scoring window; recent ratio 0.5; decode eviction disabled
StreamingLLM (GLM)	8	4,096	–	–
PyramidKV (GLM)	64	512	4,096 selected	Layer ratios 0.6 to 0.01 (linear); 32 scoring window
StreamingLLM (Llama)	512	4,096	–	–
PyramidKV (Llama)	64	256	4,096 base selected	Layer ratios 0.6 to 0.01 (linear); 32 scoring window; pooling kernel 1
DeltaKV	8	128	2,048 decode; 4,096 prefill	Full layers $\{0, 2, 7, 13, 16, 26\}$; latent dimension 512; center ratio 0.1; 4-bit latent and full-layer KV, group size 32; HF sparse-reference FP8 (SparseEngine off)
RetroInfer	4	64	–	Retrieval ratio 0.018; estimation ratio 0.232; average cluster size 16
Palu	–	–	All tokens	Grouped SVD (group size 1); K/V rank 96; BF16 factors
TurboQuant	–	–	All tokens	4-bit K/V; Gaussian codebook (SparseEngine), Lloyd–Max keys and uniform values (vLLM)

formatting, with a prompt-token ceiling of 130,944 and at most 128 generated tokens. Its overall score is **sample-level accuracy**, including unparsed responses in the denominator.

LongBench Sparse Configurations. Table 8 summarizes settings for the methods listed there from Table 1. SnapKV, Quest, OmniKV, H_2O, and PyramidKV use probability-based prefill scoring with FP32 scores.

LongBenchV2 Sparse Configurations. Table 9 summarizes method-specific settings for Table 1. Prefill chunk eviction is enabled for the reported H_2O configurations on both models. These methods use FP32 scores and a prefill chunk size of 8,192, with GPU memory utilization set to 0.95.

Table 10: **Method budgets for the AIME 2024 results in Table 2.** Budgets and windows are in tokens. Selected budgets exclude separately listed sink and recent tokens.

Method	Sink	Recent	Method budget	Other settings
OmniKV	16	64	2,048 selected	Model-profile full-attention layers
Quest (SparseEngine)	16	64	2,992 selected (3,072 total)	16-token pages; first two layers dense
Quest (Vortex)	16	64	2,048 selected (128 pages; 2,128 total)	First two layers dense
SnapKV	16	64	4,096 selected (4,176 total)	16 observation window; probability-based prefill scores; no full-attention layers; eviction every 1,024 decode steps
H_2O	–	–	4,096 prefill, 4,096 decode	128 prefill scoring window; recent ratio 0.2; probability-based prefill scores; eviction every 1,024 decode steps
PyramidKV	16	64	7,987 configured selected	32 observation window; layer ratios 0.6 to 0.0154712 from layer 0; eviction every 1,024 decode steps
R-KV	16	64	4,096 selected	Compression interval 1,024; 8 observation tokens; $\alpha = 0.1$; pooling kernel 7
StreamingLLM	64	2,048	–	–

Reference Implementations and Hardware. For the corresponding Llama rows in Table 1, DeltaKV and Palu use the authors' HF implementations, StreamingLLM uses KVCache-Factory, RetroInfer uses the authors' GPU implementation, and TurboQuant uses upstream vLLM 0.20.2 as references.

C.2 AIME Reasoning Performance

Model and Hardware. All methods in Table 2 use Qwen3-4B-Thinking-2507 in BF16 on one NVIDIA H100 80GB HBM3 GPU per run, with tensor-, expert-, and data-parallel sizes of 1.

Evaluation Protocol. Table 2 evaluates all 30 problems in the Maxwell-Jia/AIME_2024 dataset's train split with two attempts per problem, for 60 generated responses per method. Mean accuracy is the number of correct responses divided by 60, equivalently the average accuracy of the two attempts. Sampling uses temperature 0.6, top-p 0.95, top-k 20, min-p 0, and seed 42, with at most 40,960 generated tokens and a context limit of 41,984 tokens.

Concurrency and Sparse Configurations. Concurrency limits differ across methods to accommodate their GPU memory requirements. Vanilla and OmniKV use a limit of 24 concurrent requests. SparseEngine Quest and Vortex Quest both use a limit of 20, with total token budgets of 3,072 and 2,128, respectively. SnapKV, H_2O, PyramidKV, R-KV, and StreamingLLM use a concurrency limit of 64. All runs disable prefix caching and enable CUDA Graph, with a 4,096-token prefill chunk and a 65,536-token batch budget.

Table 10 summarizes the method-specific budgets and retention settings for Table 2.

C.3 Multi-Turn Agent Quality

Evaluation Protocol. Table 3 reports GLM-4.7-Flash in BF16 and Qwen3-30B-A3B-Instruct-2507-FP8. We use mini-SWE-agent with the canonical 300 test tasks of SWE-bench Lite and the official Docker evaluator.

Thinking is enabled and retained across turns. Scores are the number of resolved tasks divided by 300; empty patches and failed tasks remain in the denominator. Each entry represents one assembled 300-task evaluation, rather than an average over repeated campaigns. The context limit is 202K tokens, with at most 16,384 generated tokens per response, temperature 0.7, top-p 0.8, and top-k 20. Table 11 summarizes the GLM and Qwen3 cache settings for Table 3.

Table 11: **Cache settings for the agent-quality results in Table 3.** Budgets are in tokens and retain each method's configured meaning.

Method	Sink	Recent	Method budget	Prefix cache	Other settings
GLM-4.7-Flash					
SnapKV	64	512	15,808 selected	Chain	32 scoring window; probability-based prefill scores; no full-attention layers
H_2O	–	–	16,384 prefill; 8,192 decode	Chain	128 prefill scoring window; recent ratio 0.5; FP32 logit scoring; decode eviction disabled
Quest	64	512	1,472 selected (2,048 total)	Radix	16-token pages; first two layers dense
OmniKV	64	512	1,472 selected (2,048 total)	Radix	Model-profile full-attention layers
Qwen3-30B-A3B					
Vanilla	–	–	Full KV	Radix	–
SnapKV	0	32	8,192 selected	Chain	32 scoring window; pooling kernel 7
Quest	0	32	8,192 selection budget	Radix	16-token pages; no full-attention layers
OmniKV	0	32	8,192 selected	Radix	Full-attention layers $\{0, 3, 9, 18, 22, 27, 43\}$

C.4 Tool-Result Prefix Pruning Quality

Table 5 uses GLM-4.7-Flash with mini-SWE-agent and all 300 SWE-bench Lite test tasks, scored by the official evaluator. Each configuration has one closed-loop campaign and generates its own tool trajectory. For the three immediate-pruning runs, the agent allows 80 steps and 7,200 seconds per task, with preserved thinking, at most 16,384 generated tokens per response, temperature 0.7, and top-p 1.

The three immediate-pruning runs use BF16 on two NVIDIA H20 96GB GPUs with tensor and expert parallel sizes of 2 and data parallel size 1. Agent and decode concurrency are both 24, with at most 64 resident sequences and `favor_min_decoding_seqs=18`. The model context limit is 202,752 tokens, the batch-token budget is 65,536, and the prefill chunk is 4,096 tokens. GPU memory utilization is 0.95, CUDA Graph captures batches up to 24, and CPU prefix-cache offload is disabled.

The harness targets tool-result bodies only and submits eligible aligned token ranges after each turn containing new tool results. KVzip-based global scoring selects a shared mask over those ranges with a requested keep ratio of 0.2; Quest rounds its keep budget down to whole 16-token pages. System instructions, user input, assistant reasoning, and assistant output are not pruning targets. All three methods use radix prefix caching without Chain Cache. Vanilla uses dense decode attention. Quest uses 64 sink, 512 recent, and 1,472 selected tokens, 16-token pages, and two full-attention layers. OmniKV uses the same token budgets and its model-profile full-attention layers.

The delayed OmniKV policy changes the pruning lag to `lag=4` while retaining the 20% keep ratio. Each new tool result triggers pruning of only the fifth most recent tool-result round. The latest four rounds remain intact, and older rounds retain their previously pruned state. This run uses FP8 on four NVIDIA RTX 4090 GPUs with four TP1 replicas and concurrency 24.

The earlier GLM agent results in Table 3 used prefix caching without pruning. Their hardware and concurrency settings differ from this campaign, so the two tables do not form a matched pruning ablation.

C.5 Claw-Eval Agent Quality

Table 4 uses 182 text-only tasks selected from the 199 general Claw-Eval tasks. Another 17 tasks with visual files are skipped by policy. Each task has one trial, with DeepSeek-V4-Flash as the judge. The model is Qwen3.6-27B-FP8 on one NVIDIA H100 80GB HBM3 GPU, with tensor-parallel size 1 and eight concurrent workers. The context limit is 65,536 tokens, with at most 4,096 generated tokens per response. Both methods use prefix caching, a 4,096-token prefill chunk, a 65,536-token batch budget, at most four sequences per batch, and at most eight decoding sequences. GPU memory utilization is 0.9. OmniKV retains 2,048 selected tokens and 32 recent tokens, with no sink tokens, and keeps layers $\{3, 19, 35, 51\}$ dense. The average scores in Table 4 are the recorded means over tasks with valid scores, converted to percentages. One Vanilla task returned a model endpoint error after visual tool output, so its mean covers 181 tasks. OmniKV's mean covers all 182 selected tasks.

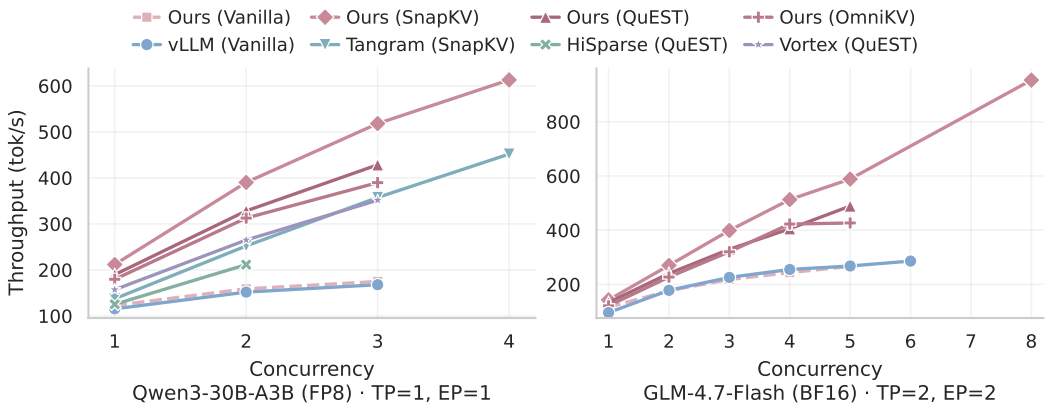


Figure 6: **Absolute decode throughput.** Under 128K-token inputs and 2K-token outputs on Qwen3-30B-A3B (left) and GLM-4.7-Flash (right). Ours denotes SparseEngine. Available external baselines are retained; Vortex encounters OOM on GLM at batch size 1, and HiSparse and Tangram do not support the evaluated GLM MLA configuration.

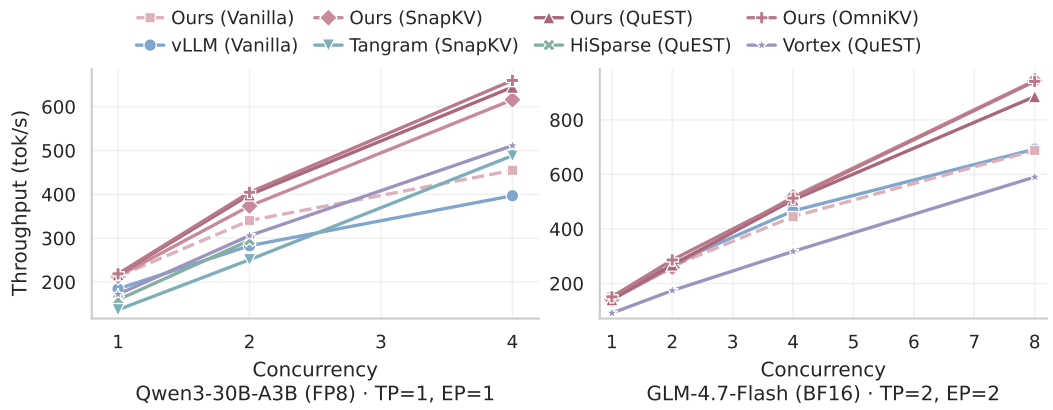


Figure 7: **Absolute decode throughput.** Under 32K-token inputs and 2K-token outputs on Qwen3-30B-A3B (left) and GLM-4.7-Flash (right). Ours denotes SparseEngine. HiSparse and Tangram are omitted for GLM because they do not support the evaluated MLA configuration.

Pass@1 divides the passed-task counts, 92 and 93 respectively, by all 182 selected tasks, including the Vanilla error.

C.6 Overall Serving Performance

Long-context decode benchmark. Figures 5 and 6 evaluate Qwen3-30B-A3B-Instruct-2507-FP8 on one NVIDIA H100 and GLM-4.7-Flash in BF16 on two H100s. The tensor- and expert-parallel sizes are both 1 for Qwen3 and both 2 for GLM. Each request uses 131,072 input tokens and 2,048 output tokens, with GPU memory utilization set to 0.9.

Table 12 lists the sparse-method configurations used for SparseEngine on both models in Figure 5. Parameter names follow the runtime configuration to make the settings explicit; their budgets retain each method's own semantics. SnapKV uses a scoring window of 32 and probability-based prefill scoring. For the 128K SnapKV runs, requests are admitted in waves of two with one decode step between waves, using an 8,192-token prefill chunk and batch-token budget; measurement begins only after the full batch is resident.

External sparse baselines. Vortex uses Quest with 16-token pages: 4 sink pages, 32 recent pages, and 92 selected pages, with layers 0 and 1 kept dense. HiSparse uses Quest with 16-token pages, 32 recent pages, and a sparsity ratio of $1/85$. Tangram uses SnapKV with a uniform 8,192-token

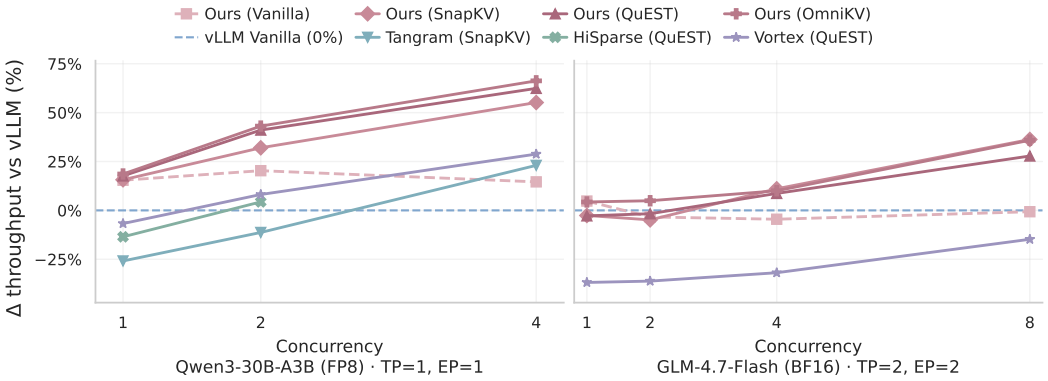


Figure 8: **Decode throughput improvement over vanilla vLLM at matched batch sizes with 32K-token inputs and 2K-token outputs.** Ours denotes SparseEngine; vanilla vLLM defines the 0% baseline. HiSparse and Tangram do not support the evaluated GLM MLA configuration and are omitted for that model. Hardware, hyperparameters, and the measurement protocol are specified in Appendix C.7.

compression budget, 64 sink tokens, a 512-token recent window, a scoring window of 32, a pooling kernel of 7, and a prefill chunk size of 4,096.

Measurement protocol. After one discarded workload, we measure three repetitions, each with 32 warmup decode steps followed by a contiguous 256-step full-batch decode window. Decode throughput is the total number of decoded tokens across these windows divided by their total elapsed time, with GPU synchronization only at each window's boundaries; prefill is excluded. Each request completes the full 2,048-token output, and a batch size is accepted only when all requests remain resident throughout the measured window and the run completes successfully. The upper plot in Figure 5 reports $100(T/T_{\text{vLLM}} - 1)$ at matched batch sizes, where T denotes decode throughput. Relative throughput uses only exactly matched measured batch sizes, without interpolation. The line plots display batch sizes up to 4 on Qwen3 and 8 on GLM; larger measured batches determine the bars. The middle plot reports absolute throughput at each system–method pair's largest successfully tested batch size, rather than its peak throughput over the sweep. A lower-bound batch annotation indicates that maximum capacity has not been established. Figure 6 gives the absolute throughput corresponding to the low-batch comparison in the upper plot of Figure 5.

C.7 Serving Performance with 32K-token Inputs

Experiments Setup. The lower plot in Figure 5 and Figures 7 and 8 use 32,768 input tokens and 2,048 output tokens per request. The hardware and model configurations follow Appendix C.6: Qwen3-30B-A3B-Instruct-2507-FP8 runs on one NVIDIA H100 80GB HBM3 GPU with tensor- and expert-parallel sizes of 1, and GLM-4.7-Flash in BF16 runs on two such GPUs with both parallel sizes set to 2. GPU memory utilization is 0.9 for all configurations. SparseEngine uses the budgets in Table 12, including SnapKV's 32-token scoring window and probability-based prefill scoring. Vortex retains the same Quest page settings as the 128K benchmark. HiSparse uses 16-token pages, 32 recent pages, and a sparsity ratio of $1/21$. Tangram uses an 8,192-token uniform compression budget, 64 sink tokens, a 512-token recent window, a 32-token scoring window, a pooling kernel of 1, and a prefill chunk size of 4,096. We use the same measurement protocol as Appendix C.6: one discarded workload followed by three measured repetitions, each with 32 warmup steps and a contiguous 256-step decode window, synchronized only at its boundaries.

Results. SparseEngine maintains higher Quest decode throughput than Vortex at the matched batch sizes shown for both models in Figure 8. The SnapKV results in the lower plot of Figure 5 further demonstrate how physical KV eviction enables higher serving concurrency within the same GPU memory budget. Together with the 128K results, these experiments support efficient execution across different context lengths under SparseEngine's shared lifecycle contract.

Table 12: **SparseEngine hyperparameters for Figure 5.** Token-budget entries are in tokens.

Parameter	SnapKV	Quest	OmniKV
`sink_keep_tokens`	64	64	64
`recent_keep_tokens`	512	512	512
`decode_keep_tokens`	7,616	1,472	1,472
`full_attention_layers`	–	–	`auto`

C.8 Supported Models and Sparse Methods

Table 13 summarizes the model variants currently supported by SparseEngine, covering different attention mechanisms and FFN architectures. Table 14 further summarizes the supported sparse methods and groups them into the four categories introduced in Figure 1, illustrating the breadth of model and method coverage enabled by the proposed abstraction.

D Abstraction Compatibility Criteria

Scope of the assessment. Table 15 concerns extensibility through each system's documented boundary, not the number of shipped methods. A compatible mapping may add an algorithm module, kernels, and method-owned state behind that boundary; it may not replace a storage layout, allocator, or transport contract that the abstraction assigns to the shared system. A checkmark requires a mapping for the method's defining computation, state updates, and storage semantics. A triangle identifies a partial mapping, such as selecting the same tokens without physically reclaiming the discarded KV, or providing eviction without the required online score updates.

Method requirements. Quest requires persistent page summaries and query-dependent selection. RetroInfer additionally requires variable-sized clusters, GPU–CPU retrieval, and attention estimation over unselected content (Chen et al., 2025). SnapKV selects prompt KV using an observation window and physically retains the selected entries; Ada-SnapKV (Feng et al., 2025) also requires non-uniform head budgets. H_2O requires cumulative attention statistics for retained tokens, online updates, and physical eviction as decoding advances. Palu and LoRC replace full-dimensional KV storage with low-rank representations and matching reconstruction/computation; KIVI requires asymmetric K/V quantization, grouped scales, and a residual full-precision region. Storing auxiliary summaries alongside an unchanged full KV cache does not implement those representation-level memory savings.

HiSparse. The published design (Xie et al., 2026) keeps the complete history in host memory and swaps selected entries into a GPU hot buffer. For algorithm extensibility we also inspect SGLang revision `f618022`, including its sparse-algorithm interface: algorithms construct/update representations and return selected indices, which a backend adaptor maps to attention metadata. Quest fits this selected-index design; this assessment does not assert that every Quest/HiSparse deployment path is implemented. Token-retention methods can supply selection policies, but hot-buffer LRU replacement does not delete their host-backed history. RetroInfer's selection component fits, while its cluster-aware transport and estimation require additional contracts. The inspected interfaces also do not establish method-owned Palu/LoRC payloads or KIVI's packed payload and quantization-state lifecycle.

Tangram. Tangram's scorer, budget-scope, and eviction abstractions support SnapKV and Ada-SnapKV directly (Kim et al., 2026). At revision `6fa551f`, the scorer contract is stateless; cached-position rescoring exposes cached keys/values, rather than the current query and accumulated attention state required by H_2O. Its eviction regime supplies retention machinery, hence H_2O receives a partial mark. The compression boundary chooses which KV entries remain; it does not expose query-time retrieval of discarded entries or replacement of the retained KV payload with method-defined low-rank/quantized storage.

Vortex. Vortex exposes cache-side auxiliary computation and query-side page selection (Chen et al., 2026b). Its vFlow contract (revision `ab9ac68`) reserves the standard K/V entries; `create_cache` declares additional tensors, and `forward_indexer` writes routing indices for a paged attention

Table 13: **Supported models and their attention and FFN architectures.** GQA denotes grouped-query attention, MLA denotes multi-head latent attention, and MoE denotes mixture of experts. Hybrid linear attention combines GQA with Gated DeltaNet layers.

No.	Model	Attention Architecture	FFN Architecture
1	Qwen2.5 (Yang et al., 2024)	GQA	Dense
2	Qwen3 Dense (Yang et al., 2025)	GQA	Dense
3	Qwen3 MoE (Yang et al., 2025)	GQA	MoE
4	Qwen3.5 Dense	Hybrid linear attention	Dense
5	Qwen3.5 MoE	Hybrid linear attention	MoE
6	Qwen3.6 Dense	Hybrid linear attention	Dense
7	Qwen3.6 MoE	Hybrid linear attention	MoE
8	Qwen3.8	Hybrid linear attention	Dense
9	GLM-4.7-Flash (Zeng et al., 2025)	MLA	MoE
10	Gemma 4 Dense (Team et al., 2026)	Sliding-window / global attention	Dense
11	Gemma 4 MoE (Team et al., 2026)	Sliding-window / global attention	MoE
12	Llama 3 (Dubey et al., 2024)	GQA	Dense
13	Llama 3.1 (Dubey et al., 2024)	GQA	Dense
14	MiniMax-M2.7 (Chen et al., 2026a)	GQA	MoE

backend. This supports Quest and parts of retention/statistics policies, including the paper's H_2O example, without exposing physical reclamation of the standard KV allocation. RetroInfer additionally needs cluster-aware storage/movement and its estimation operator. Backend FP8 or MLA support does not by itself establish an extension path for Palu, LoRC, or KIVI's distinct storage semantics.

SPIN. Sections 4-5 of SPIN (Zhao et al., 2026) expose algorithm-defined Index, Select, and Attention, with system-managed Offload and Retrieve. A partition may represent one token, a page, or a variable-sized cluster. Quest maps to page summaries and selection; RetroInfer is explicitly integrated, including its custom estimation-aware attention. The same selection interface covers parts of retention policies, but the documented pipeline retrieves subsets of a full tiered KV history and does not establish permanent algorithm-driven deletion. Its system-owned head-wise KV pages and transport also do not establish a payload-replacement contract for the three representation methods.

SparseEngine. The mappings follow the lifecycle boundary and implementation interfaces at revision `8ea8f80`. Quest uses page summaries and query-aware compute views. A RetroInfer integration places cluster maps, summaries, and tiered payloads behind a method-owned cache manager, with a specialized attention provider and explicit memory requirements. SnapKV, Ada-SnapKV, and H_2O place retained indices, head budgets, and persistent scores under the runtime/cache-manager lifecycle, including physical compaction and release. Palu/LoRC integrations can own latent storage and reconstruction, while KIVI owns packed K/V, scales, and residual storage; providers consume the corresponding compute views. These are integration mappings, not claims that all eight complete methods are already shipped.

Table 14: **Supported sparse methods grouped into four categories.** The inventory includes 15 cache methods and FlashPrefill-v2 for sparse prefill. Each method is assigned to one category according to its primary mechanism.

No.	Method	Sparsity Category
1	Quest (Tang et al., 2024)	Dynamic Sparsity
2	OmniKV (Hao et al., 2025)	Dynamic Sparsity
3	RetroInfer (Chen et al., 2025)	Dynamic Sparsity
4	FlashPrefill-v2 (Fan et al., 2026)	Dynamic Sparsity
5	StreamingLLM (Xiao et al., 2024b)	KV Eviction
6	SnapKV (Li et al., 2024)	KV Eviction
7	H_2O (Zhang et al., 2023)	KV Eviction
8	PyramidKV (Cai et al., 2024)	KV Eviction
9	R-KV (Cai et al., 2025)	KV Eviction
10	SkipKV (Tian et al., 2026)	KV Eviction
11	KVzip (Kim et al., 2025)	KV Eviction
12	Palu (Chang et al., 2024)	KV Compression
13	DeltaKV (Hao et al., 2026)	KV Compression
14	KIVI (Liu et al., 2024)	KV Quantization
15	TurboQuant (Zandieh et al., 2026)	KV Quantization
16	FP8 KV	KV Quantization

Table 15: **Abstraction-level compatibility with representative sparse methods.** ✓: the method's defining computation and KV-state semantics fit the documented extension boundary; △: only part of the method fits; –: a mapping is not established by that contract. These are design assessments, not implementation or performance results. New method modules are allowed; changes to system-owned storage/transport contracts are not. Physical eviction must release KV storage, rather than merely mask reads or offload entries. Appendix D gives the mapping criteria and evidence.

	Query-dependent retrieval		*Physical KV eviction*			*Compressed KV representations*		
System	**Quest**	**RetroInfer**	**SnapKV**	**Ada-SnapKV**	**H_2O**	**Palu**	**LoRC**	**KIVI**
SparseEngine (Ours)	✓	✓	✓	✓	✓	✓	✓	✓
HiSparse	✓	△	△	△	△	–	–	–
Tangram	–	–	✓	✓	△	–	–	–
Vortex	✓	△	△	△	△	–	–	–
SPIN	✓	✓	△	△	△	–	–	–