

# **Modular and Hierarchical Key-Value Caching Architectures for Large Language Models: A Comprehensive Research Report**

## **1. Introduction: The Inference Memory Wall**

The rapid ascendancy of Large Language Models (LLMs) has fundamentally shifted the computational paradigm of artificial intelligence from training-centric bottlenecks to inference-centric constraints. While the training of models with hundreds of billions of parameters requires massive, synchronized compute clusters, the deployment and serving of these models introduce a distinct set of challenges rooted primarily in memory bandwidth and capacity. At the heart of this challenge lies the Key-Value (KV) cache—a critical component in autoregressive decoding that stores intermediate tensor states to prevent redundant computation. As context windows expand from the standard 4,096 tokens of early Llama models to the 1 million+ token capabilities of Gemini and Claude, the KV cache has transformed from a manageable auxiliary data structure into the primary consumer of high-bandwidth memory (HBM), often eclipsing the storage requirements of the model weights themselves.<sup>1</sup>

This report provides an exhaustive analysis of the architectural evolution of KV cache management, moving beyond ephemeral, in-memory storage toward modular, persistent, and file-backed systems. We explore the theoretical underpinnings of attention memory, the limitations of contiguous allocation, and the emergence of advanced paging and radix-tree structures such as those found in vLLM and SGLang. Furthermore, we investigate the rigorous engineering required to implement "modular caching"—the ability to serialize KV states to disk, retrieve them on demand, and composable stitch them together to form new contexts. This analysis encompasses the physics of data movement across PCIe interconnects, the mathematical complexities of position-dependent embeddings in modular composition, and the design of hierarchical multi-level caching (HLC) systems that seamlessly integrate GPU HBM, CPU DRAM, and NVMe storage.<sup>3</sup>

### **1.1 The Autoregressive Computational Primitive**

To understand the necessity of modular caching, one must first deconstruct the arithmetic operations of the Transformer decoder. The generation of text is an autoregressive process, meaning the probability distribution of the next token  $x_t$  is conditional on the entire history of the sequence  $x_{1:t-1}$ . In the self-attention mechanism, the model computes the relationship between the current token (the Query) and all preceding tokens (the Keys and

Values).

$$\$ \$ \text{Attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V \$ \$$$

In a naive implementation without caching, generating the token at step  $t$  would require re-computing the Key ( $K$ ) and Value ( $V$ ) projections for all tokens  $x_1 \dots x_{t-1}$ . This would result in a computational complexity of  $O(t^2)$  for the sequence, rendering real-time generation for long contexts impossible. The KV cache optimizes this by preserving the  $K$  and  $V$  tensors of past tokens in GPU memory. At step  $t$ , the model only projects the single new token  $x_t$  into  $q_t, k_t, v_t$ , and appends  $k_t, v_t$  to the cache. The attention operation then becomes a matrix-vector multiplication between  $q_t$  and the cached matrices  $K_{1:t}, V_{1:t}$ , reducing the per-step complexity to  $O(t)$  and the total sequence complexity to  $O(t^2)$ —but crucially, the redundant projection FLOPs are eliminated.<sup>5</sup> However, this conservation of FLOPs comes at the expense of memory. The KV cache is not a static asset like model weights; it is a dynamic, stateful tensor that grows linearly with sequence length and batch size. For high-throughput serving systems, where batching is essential to amortize the cost of reading model weights, the aggregate size of KV caches for all active requests can rapidly exhaust the available HBM. When HBM is depleted, the system must either stall, reject new requests, or evict existing caches—forcing expensive re-computation upon resumption. This creates a "memory wall" where the cost of inference is dominated not by the arithmetic logic units (ALUs) but by the capacity and bandwidth of the memory hierarchy.<sup>7</sup>

## 1.2 Mathematical Formalization of Cache Footprint

Designing a storage hierarchy requires precise estimation of the tensor shapes involved. The memory footprint of the KV cache is determined by the model architecture and the precision of the stored data. For a standard Transformer model, the size of the KV cache  $M_{KV}$  (in bytes) can be derived as follows:

Let:

- $L$ : Sequence length (number of tokens in context)
- $B$ : Batch size (number of concurrent sequences)
- $N_{\text{layers}}$ : Number of transformer layers
- $N_{\text{heads}}$ : Number of attention heads per layer
- $D_{\text{head}}$ : Dimension of each head
- $P_{\text{bytes}}$ : Precision in bytes (e.g., 2 for FP16/BF16, 1 for FP8, 0.5 for INT4)

The formula is:

$$M_{KV} = 2 \times B \times L \times N_{\text{layers}} \times N_{\text{heads}} \times D_{\text{head}} \times P_{\text{bytes}}$$

The factor of 2 accounts for storing both the Key matrix and the Value matrix.

### Case Study: Llama-3-70B

Consider the Llama-3-70B model, a standard benchmark for high-performance open-weight models.

- $N_{\text{layers}} = 80$
- $N_{\text{heads}} = 8$  (Note: Llama-3 uses Grouped Query Attention (GQA), reducing the number of KV heads significantly compared to the 64 Query heads).
- $D_{\text{head}} = 128$
- $P_{\text{bytes}} = 2$  (FP16)

For a single request ( $B=1$ ) with a moderate context window of 32,000 tokens ( $L=32,768$ ):

$$M_{\text{KV}} = 2 \times 1 \times 32,768 \times 80 \times 8 \times 128 \approx 10.7 \text{ GB}$$

While 10.7 GB seems manageable on an 80GB NVIDIA H100, this is for a *single* user. In a production environment serving a batch size of 64, the requirement would theoretically balloon to 684 GB—far exceeding the capacity of a single GPU or even a standard 8-GPU node. Furthermore, if the context extends to the model's 128k limit, a single request consumes over 40 GB. This necessitates techniques like **PagedAttention** to handle fragmentation and **Hierarchical Caching** to offload inactive segments to CPU RAM or disk.<sup>1</sup>

### 1.3 The Latency-Throughput Trade-off in Memory Systems

The motivation for modular, file-backed caching is rooted in the latency characteristics of modern hardware. When a cache block is evicted from HBM to make room for active computation, the cost of bringing it back depends on the storage medium.

- **Re-computation Cost:** To regenerate the KV cache for a context of length  $L$ , the GPU must process all  $L$  tokens through the model. For deep models, this is a compute-bound operation.
- **I/O Load Cost:** To reload the KV cache from disk, the system is bound by the PCIe bandwidth and SSD read speeds.

If the time to load the data from storage ( $T_{\text{load}}$ ) is less than the time to recompute it ( $T_{\text{compute}}$ ), then offloading is viable.

$$T_{\text{load}} = \frac{M_{\text{KV}}}{\text{Bandwidth}_{\text{I/O}}}$$

$$T_{\text{compute}} = \frac{\text{FLOPs}(L)}{\text{FLOPS}_{\text{GPU}}}$$

For large batches or very long contexts,  $T_{\text{compute}}$  grows quadratically (in attention) or linearly with a large coefficient (in feed-forward layers), while  $T_{\text{load}}$  grows strictly linearly with data size. As sequence lengths increase, the advantage of loading pre-computed modular caches from persistent storage becomes overwhelmingly positive, provided the I/O pipeline is optimized.<sup>10</sup>

## 2. Evolution of Cache Management: From Contiguous to Modular

Before implementing a file-backed system, one must understand the in-memory structures

that allow for modularity. The transition from monolithic tensors to block-based management is the software enabler for file persistence.

## 2.1 The Limitations of Contiguous Memory Allocation

Early LLM inference engines (e.g., the original Hugging Face Transformers implementation) managed the KV cache as a single, contiguous tensor pre-allocated for the maximum possible sequence length. If a model supported 2048 tokens, the system would allocate a `` tensor at initialization.

- **Internal Fragmentation:** If a user prompt was only 100 tokens, over 95% of the allocated memory was wasted "dark silicon."
- **External Fragmentation:** In dynamic serving, requests of different lengths finish at different times. The memory allocator (like cudaMalloc) might have sufficient *total* free memory, but if it is fragmented into small gaps, it cannot service a new request requiring a large contiguous block.
- **Serialization Difficulty:** Saving a contiguous cache for a partial sequence requires slicing and copying, and merging two contiguous caches (e.g., prefix + suffix) requires allocating a new, larger tensor and copying both inputs—an  $\$O(L)$  operation that stalls the GPU.<sup>12</sup>

## 2.2 PagedAttention: The Virtualization of KV Memory

The breakthrough in modular caching came with **vLLM's PagedAttention** algorithm. Drawing inspiration from operating system virtual memory paging, vLLM divides the KV cache into fixed-size blocks (e.g., 16 or 32 tokens).

- **Logical vs. Physical:** The system maintains a "Block Table" that maps the logical token indices (e.g., tokens 0-15, 16-31) to non-contiguous physical blocks in HBM.
- **Modularity Enabler:** PagedAttention is the prerequisite for file-backed modularity. Because the cache is already broken into discrete blocks, individual blocks can be evicted to disk, saved as files, or loaded independently without requiring a large contiguous memory allocation. This allows for "streaming" the cache: the inference engine can start processing the first block while the second block is still being DMA-transferred from the SSD.<sup>13</sup>
- **Memory Efficiency:** By allocating memory only when a new block is needed, vLLM reduces waste to near zero (<4%). This density allows for larger batch sizes, which in turn increases the throughput of the entire system.<sup>14</sup>

## 2.3 RadixAttention: Tree-Based Structural Reuse

While PagedAttention optimizes memory allocation, **SGLang's RadixAttention** optimizes the reuse of cached content. In scenarios like agentic workflows or few-shot learning, multiple requests often share the same prefix (e.g., a lengthy system prompt or a set of examples).

- **Radix Tree Structure:** SGLang manages the KV cache indices using a Radix Tree (a compressed prefix tree). Each node in the tree represents a sequence of tokens. Edges represent the transition tokens.

- **Automatic Composition:** When a new request arrives, the scheduler traverses the Radix Tree to find the longest matching prefix that already exists in memory. This is, in effect, an automatic "stitching" of cached modules. If the system prompt "You are a coding assistant..." is at Node A, and the user query is "Write a Python script...", the system simply branches from Node A.
- **Eviction Policy:** The Radix Tree enables intelligent eviction. SGLang uses an LRU (Least Recently Used) policy on the tree nodes. If memory is full, it evicts leaf nodes (specific user queries) while retaining root nodes (shared system prompts), maximizing the global hit rate.<sup>16</sup>

**Implications for File-Backed Systems:** The Radix Tree provides the "file system" logic for our modular cache. A file-backed implementation can map the Radix Tree nodes to disk paths. A "cache file" effectively becomes a serialized node (or chain of nodes) from this tree. To compose a cache over time, one essentially mounts a saved branch of the Radix Tree onto the current active tree.<sup>19</sup>

---

### 3. Modular Caching Architecture: Storing and Composing State

The core of this research addresses the user's request for a "modular cache (cache <-> file) that we can compose over time." This requires moving beyond simple swapping to a system where KV blocks are treated as independent, composable assets.

#### 3.1 The "Stitching" Problem: Positional Embeddings

The most significant technical barrier to modular caching is the handling of Positional Embeddings. In modern LLMs (Llama, Mistral, Qwen), position is encoded using **Rotary Positional Embeddings (RoPE)**.

RoPE rotates the Key (\$K\$) and Query (\$Q\$) vectors by an angle  $\theta$  proportional to their absolute position  $m$  in the sequence.

$$x_m = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} x^{(1)} \\ x^{(2)} \end{pmatrix}$$

##### The Conflict:

Imagine we have two modular cache files:

1. **Module A (System Prompt):** "You are a helpful assistant." (Tokens 0-5). The Keys in this file are rotated by positions \$0, 1, 2, 3, 4, 5\$.
2. **Module B (User Context):** "Here is a document about caching..." (Tokens 0-100). This module was generated independently, so its Keys are rotated by positions \$0 \dots 100\$.

If we attempt to stitch them to form the sequence A + B, the tokens in Module B are now at logical positions \$6 \dots 106\$. However, the cached Keys in Module B are rotated for positions \$0 \dots 100\$.

- **Mismatch:** The attention mechanism will compute dot products between a Query at position  $P$  and a Key that "thinks" it is at position  $0$  but is actually at position  $6$ . The geometric relationship encoded by RoPE is destroyed, and the attention scores will be noise.<sup>21</sup>

## 3.2 Solution Strategies for Modular Composition

### 3.2.1 Strategy A: Raw (Pre-RoPE) Caching

The most robust solution for modularity is to store the Keys before the RoPE rotation is applied.

- **Mechanism:** Modify the model's forward pass to intercept the Key tensor  $K$  immediately after the linear projection layer but before the `apply_rotary_pos_emb` function. Save this "raw" tensor to disk.
- **Composition:** When stitching Module B after Module A (length  $L_A$ ), load the raw Keys for B. Then, applying the RoPE function on-the-fly using the offset positions ( $L_A \backslash dots L_A + L_B$ ).
- **Trade-off:** This adds a computational step during the "load" phase (applying RoPE). However, RoPE is an element-wise operation and is computationally cheap ( $O(1)$ ) compared to the matrix multiplications of the attention mechanism. This ensures mathematical correctness for any arbitrary composition.<sup>24</sup>

### 3.2.2 Strategy B: Inverse Rotation (Re-computation)

If we only have access to cached files that are already rotated (e.g., from a standard inference run), we must perform an "inverse rotation."

- **Mechanism:** Apply the inverse rotation matrix  $R^{-1}_{\{m\_old\}}$  to strip the original position encoding, recovering the raw Key. Then, apply the new rotation  $R_{\{m\_new\}}$ .
- **Complexity:** This requires knowing the exact position index  $m_{old}$  used during the creation of the cache file. It introduces floating-point error accumulation but allows reusing standard cache outputs.<sup>21</sup>

### 3.2.3 Strategy C: Position-Independent Context Caching (PIC)

Recent research, such as the **Epic** system, proposes algorithms to bypass exact position dependence.

- **LegoLink Algorithm:** Epic identifies "anchor" tokens (usually the first few tokens, due to the "Attention Sink" phenomenon) and "body" tokens. It stitches contexts by maintaining the positional integrity of anchors while allowing the body tokens to float or use relative positioning.
- **Efficiency:** Epic demonstrates up to **8x improvement in Time-To-First-Token (TTFT)** by stitching contexts compared to recomputing, with negligible accuracy loss. This method effectively modularizes the KV cache by making the majority of blocks position-agnostic.<sup>21</sup>

### 3.3 File Formats and Serialization

To support high-performance modular caching, the file format on disk must support **Zero-Copy** loading.

- **The Problem with Pickle:** Python's standard `torch.save` uses pickle, which executes arbitrary code (security risk) and requires CPU deserialization. The data is copied from Disk → OS Buffer → Python Object → Tensor memory, incurring high CPU overhead and latency.
- **Safetensors:** The industry standard for modular tensor storage is **Safetensors**.
  - **Architecture:** A Safetensors file consists of a fixed-size JSON header containing metadata (dtype, shape, offsets) followed by a monolithic binary byte buffer.
  - **Memory Mapping (mmap):** The operating system can map the file directly into the virtual address space. When moving to GPU, the driver can theoretically transfer data directly from the page cache to HBM (or via GDS), bypassing the CPU copy loop.
  - **Lazy Loading:** A user can load only specific tensors (e.g., "Layer 12, Head 4") without reading the entire file, which is ideal for streaming modular blocks.<sup>28</sup>

#### Proposed Modular File Schema (JSON Header):

JSON

```
{  
    "__metadata__": {  
        "model": "llama-3-70b",  
        "rope_status": "raw", // Critical: Indicates Pre-RoPE storage  
        "chunk_id": "uuid-v4",  
        "semantic_tags": ["python", "sorting_algo"]  
    "layer_0.key": { "dtype": "F16", "shape": , "data_offsets": },  
    "layer_0.value": { "dtype": "F16", "shape": , "data_offsets": }  
    ...  
}
```

---

## 4. Hierarchical and Multi-Level Caching (HLC)

A single layer of file-backed cache is insufficient for high-throughput systems. To balance the massive capacity of disk with the blistering speed of HBM, we must implement a **Hierarchical Cache** analogous to the L1/L2/L3 cache in CPUs, but distributed across the inference cluster.

## 4.1 The Physics of Data Movement

The viability of HLC is determined by the bandwidth bottlenecks between tiers.

Tier	Medium	Latency	Bandwidth	Capacity	Role
L1	GPU HBM3	~100 ns	3,350 GB/s (H100)	80 GB	Active Decoding
L2	Host RAM (DDR5)	~100 ns	~600 GB/s	1-4 TB	Fast Swap / Staging
L3	Local NVMe SSD	~10-100 µs	14-28 GB/s (Gen5)	10-30 TB	Persistence / Modular Storage
L4	Network / S3	ms - sec	12-50 GB/s (400Gbps)	\$\infty\$	Global Sharing

30

### The PCIe Bottleneck:

The connection between L1 (GPU) and L2/L3 (Host) is the PCIe bus. A PCIe Gen5 x16 link has a theoretical bandwidth of 64 GB/s. In practice, overhead reduces this to ~50-58 GB/s.

- **Impact:** Loading a 10 GB KV module (Llama-3-70B, 32k tokens) takes \$10 / 50 \approx 0.2\\$ seconds.
- **Comparison:** Recomputing 32k tokens on H100 might take 2-5 seconds depending on batch size. Thus, even with the PCIe bottleneck, loading is **10-25x faster** than recomputing. This validates the HLC approach.<sup>10</sup>

## 4.2 LMCache: The Knowledge Delivery Network

**LMCache** is a pioneering system designed to implement this hierarchy explicitly for LLMs. It functions as a middleware between the inference engine (vLLM) and the storage backend.

- **Architecture:** LMCache introduces a "Connector" abstraction. The inference engine requests KV blocks via a key (e.g., hash of the prompt text). The Connector checks L2 (RAM), then L3 (Disk), then L4 (Remote).
- **Pipelining:** To hide the latency of lower tiers, LMCache employs aggressive pipelining. While the GPU is decoding the first token of a retrieved context, LMCache is asynchronously fetching the next required chunks from disk into CPU RAM, and DMA-transferring the subsequent chunks to GPU.
- **Disaggregated Prefill:** LMCache enables "Prefill-Decode Disaggregation." One set of heavy-duty GPUs can be dedicated to "Prefill" (computing KV caches from prompts). They stream the generated caches via LMCache (over RDMA/Network) to lighter-weight "Decode" GPUs that serve the users. This decouples the bursty prefill workload from the latency-sensitive decode workload.<sup>31</sup>
- **Benchmarks:** LMCache demonstrates up to **15x improvement** in throughput for multi-round QA workloads by avoiding recomputation. In local deployments, it reduces

TTFT by 75% for long-context documents.<sup>34</sup>

### 4.3 Implementing HLC in SGLang (HiCache)

SGLang implements hierarchy through **HiCache**, which integrates deeply with its Radix Tree.

- **Consistency:** A major challenge in distributed HLC is consistency. If running Tensor Parallelism (TP=8), the KV cache is sharded across 8 GPUs. When loading a modular file, all 8 workers must load their specific shard simultaneously. SGLang coordinates this via a deterministic mapping in the Radix Tree.
- **Disk Offload:** HiCache supports evicting cold tree branches to local NVMe. It uses an LRU policy not just for memory, but for the disk cache itself, promoting frequently accessed modules to RAM.<sup>36</sup>

### 4.4 FlexGen: Optimization via Linear Programming

**FlexGen** takes a different approach, optimizing for **throughput** on memory-constrained hardware (e.g., running a 175B model on a 16GB GPU).

- **Zigzag Scheduling:** Instead of processing one request at a time, FlexGen computes a schedule using linear programming. It determines exactly which tensor blocks to swap to CPU/Disk and when, creating a "zigzag" pattern of computation that maximizes the overlap between Compute and I/O.
- **Relevance:** For modular caching, FlexGen's scheduling algorithms provide the theoretical bounds for how efficiently one can stream modular caches from disk without stalling the GPU.<sup>3</sup>

---

## 5. Semantic and Composable Caching: Beyond Exact Matches

The ultimate evolution of modular caching is **Semantic Caching**. Standard systems (vLLM, SGLang) use hash-based or exact-string matching. If a user changes one word in the prompt, the hash changes, and the cache is missed. Semantic caching allows fetching relevant KV modules based on meaning.

### 5.1 ChunkKV and Semantic Retrieval

**ChunkKV** treats the KV cache as a database of semantic units rather than a monolithic block.

- **Chunking:** The context is divided into chunks (e.g., sentences or paragraphs).
- **Indexing:** The system computes an embedding for each chunk's text. This embedding is stored in a vector database (e.g., FAISS or Milvus), pointing to the file path and offset of the corresponding KV tensor on disk.
- **Retrieval:** When a new query arrives, the system retrieves the top-\$k\$ most relevant chunks from the vector DB, loads their pre-computed KV tensors, stitches them together (using Pre-RoPE caching), and feeds them to the model.
- **Result:** This turns the KV cache into a RAG (Retrieval Augmented Generation) system

that retrieves *activations* instead of *text*. This skips the prefill phase entirely for retrieved knowledge, achieving massive speedups.<sup>38</sup>

## 5.2 Industry Implementations: Prompt Caching

Major providers have begun exposing these capabilities via API, validating the modular caching paradigm.

- **Anthropic (Claude) & Gemini:** Both offer "Prompt Caching" or "Context Caching."
  - **Explicit Caching (Anthropic):** Users set "breakpoints" in the API call. The system caches the prefix up to that point. This effectively creates a named module (e.g., "Legal\_Docs\_V1") that can be reused.
  - **Implicit Caching (Gemini):** The system automatically detects repeated prefixes. If a prompt exceeds a token threshold (e.g., 32k), it is hashed and stored. Subsequent requests with the same prefix trigger a cache hit.
  - **Pricing:** These providers offer significant discounts (up to 90%) for cached input tokens, reflecting the reduced computational cost of loading vs. computing.<sup>40</sup>

## 5.3 Security Implications

Modular caching introduces new security surfaces.

- **Information Leakage:** If Module A (User A's private data) is stitched with Module B (User B's query) in a shared inference server, there is a risk of cross-talk if the memory is not strictly zeroed or isolated.
- **Poisoning:** A malicious actor could "poison" a shared semantic cache module (e.g., a common system library module) with activations that induce hallucinations or jailbreaks in any user session that stitches that module.<sup>43</sup>

---

# 6. Future Directions and Hardware Convergence

The separation between "Cache" and "File" is an artifact of current hardware architectures. Emerging technologies promise to dissolve this boundary.

## 6.1 CXL: The End of "Offloading"

**Compute Express Link (CXL)** introduces a cache-coherent interconnect for memory expansion.

- **CXL.mem:** Allows attaching Terabytes of DRAM to the CPU/GPU complex that is byte-addressable.
- **Impact:** A modular KV cache stored in a CXL memory pool is technically "off-chip," but it can be accessed by the GPU with load/store semantics rather than DMA block transfers. This reduces the latency of "stitching" modules to near-native DRAM speeds, effectively granting LLMs infinite memory capacity.<sup>44</sup>

## 6.2 GPU Direct Storage (GDS) and NVMe-oF

NVIDIA's **Magnum IO** stack enables **GPU Direct Storage**, allowing the GPU to read files directly from NVMe SSDs without CPU intervention.

- **Application:** An LLM inference kernel could issue read requests for modular KV files directly to the SSD. This frees up the CPU for request scheduling and reduces the latency of the stitching operation by removing the bounce buffer in system RAM.
  - **Integration:** Future versions of LMCache and vLLM are expected to leverage GDS to maximize the utilization of PCIe Gen5 bandwidth.<sup>45</sup>
- 

## 7. Conclusion

The transition to modular, file-backed KV caching represents a maturation of LLM infrastructure. We are moving from a paradigm of "stateless compute" to "stateful management," where the KV cache is treated as a valuable, persistent asset.

To achieve the user's goal of a composable, modular cache:

1. **Adopt a Block-Based Manager:** Use vLLM's PagedAttention or SGLang's RadixAttention to virtualize memory.
2. **Implement Pre-RoPE Caching:** Store Keys in their raw, unrotated state to allow for mathematical correctness when stitching modules at arbitrary positions.
3. **Deploy Hierarchical Storage:** Utilize LMCache or similar middleware to tier data across HBM, DRAM, and NVMe, maximizing the cost/performance ratio.
4. **Standardize Formats:** Use Safetensors for zero-copy serialization to ensure that the I/O pipeline does not become the bottleneck.

By treating the KV cache not as ephemeral waste but as a modular building block, we unlock the potential for "Infinite Context" agents that can instantly recall and compose knowledge from vast libraries of pre-computed experiences.

---

## Selected Tables and Data Comparisons

**Table 1: Comparative Analysis of Caching Systems**

Feature	vLLM (Native)	SGLang (Radix)	LMCache	FlexGen
<b>Memory Structure</b>	Block Table (Paged)	Radix Tree (Trie)	Connector Middleware	Linear Prog. Schedule
<b>Persistence</b>	Swap-only (CPU)	Local Disk (HiCache)	Multi-backend (Disk/Net/S3)	Aggressive Offloading
<b>Modularity</b>	Block-level	Tree-branch level	File/Object level	Tensor-level
<b>Stitching Logic</b>	Hash-based Prefix	Automatic Tree Traversal	Via Connector API	Manual / Scheduling
<b>Primary Goal</b>	High Throughput Serving	Complex Agents / Structure	Distributed / KDN	Single GPU Capacity

**Table 2: Estimated Latency for 32k Token Context Retrieval (Llama-3-70B)**

Medium	Bandwidth	Load Time (10.7 GB)	Speedup vs Compute
<b>GPU HBM3</b>	3,350 GB/s	0.003 s	N/A (Baseline)
<b>CPU DRAM (PCIe 5)</b>	58 GB/s	0.18 s	~20x
<b>NVMe RAID 0 (4x)</b>	28 GB/s	0.38 s	~10x
<b>Single NVMe (Gen4)</b>	7 GB/s	1.53 s	~2.5x
<b>Network (100 Gbps)</b>	12.5 GB/s	0.85 s	~4.5x
<b>Re-computation</b>	(Compute Bound)	~3.0 - 5.0 s	1x

10

## Works cited

1. Transformers Key-Value Caching Explained - Neptune.ai, accessed January 25, 2026, <https://neptune.ai/blog/transformers-key-value-caching>
2. Efficiently Scaling Transformer Inference - arXiv, accessed January 25, 2026, <https://arxiv.org/pdf/2211.05102.pdf>
3. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU - OpenReview, accessed January 25, 2026, <https://openreview.net/pdf?id=RRntzKrBTp>
4. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management - arXiv, accessed January 25, 2026, <https://arxiv.org/html/2406.19707v1>
5. KV Caching Explained: Optimizing Transformer Inference Efficiency - Hugging Face, accessed January 25, 2026, <https://huggingface.co/blog/not-lain/kv-caching>
6. LLM Inference Series: 3. KV caching explained | by Pierre Lienhart | Medium, accessed January 25, 2026, <https://medium.com/@plienhar/llm-inference-series-3-kv-caching-unveiled-048152e461c8>
7. Efficient LLM Inference with Activation Checkpointing and Hybrid Caching - arXiv, accessed January 25, 2026, <https://arxiv.org/html/2501.01792v1>
8. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management - USENIX, accessed January 25, 2026, <https://www.usenix.org/system/files/osdi24-lee.pdf>
9. \name: KV Cache Compression and Streaming for Fast Language Model Serving The original version was uploaded on 11 Oct 2023 - arXiv, accessed January 25, 2026, <https://arxiv.org/html/2310.07240v4>
10. Scaling AI Inference with KV Cache Offloading - Samsung, accessed January 25, 2026, [https://download.semiconductor.samsung.com/resources/white-paper/scaling\\_ai\\_inference\\_with\\_kv\\_cache\\_offloading.pdf](https://download.semiconductor.samsung.com/resources/white-paper/scaling_ai_inference_with_kv_cache_offloading.pdf)
11. TIL: For long-lived LLM sessions, swapping KV Cache to RAM is ~10x faster than recalculating it. Why isn't this a standard feature? : r/LocalLLaMA - Reddit, accessed January 25, 2026,

[https://www.reddit.com/r/LocalLLaMA/comments/1olouiw/til\\_for\\_longlived\\_llm\\_sessions\\_swapping\\_kv\\_cache/](https://www.reddit.com/r/LocalLLaMA/comments/1olouiw/til_for_longlived_llm_sessions_swapping_kv_cache/)

12. How PagedAttention resolves memory waste of LLM systems - Red Hat Developer, accessed January 25, 2026,  
<https://developers.redhat.com/articles/2025/07/24/how-pagedattention-resolves-memory-waste-llm-systems>
13. vLLM and PagedAttention: A Comprehensive Overview | by Abonia Sojasingarayar, accessed January 25, 2026,  
<https://medium.com/@abonia/vllm-and-pagedattention-a-comprehensive-overview-20046d8d0c61>
14. Introduction to vLLM and PagedAttention | Runpod Blog, accessed January 25, 2026, <https://www.runpod.io/blog/introduction-to-vllm-and-pagedattention>
15. Paged Attention - vLLM, accessed January 25, 2026,  
[https://docs.vllm.ai/en/latest/design/paged\\_attention/](https://docs.vllm.ai/en/latest/design/paged_attention/)
16. SGLang: Fast Serving Framework for Large Language and Vision-Language Models on AMD Instinct GPUs, accessed January 25, 2026,  
<https://rocm.blogs.amd.com/artificial-intelligence/sqlang/README.html>
17. SGLang Deep Dive: Inside SGLang - SugiV Blog, accessed January 25, 2026,  
<https://blog.sugiv.fyi/sqlang-deep-dive-inside-sqlang>
18. Inside SGLang: LMSys New Framework for Super Fast LLM Inference | by Jesus Rodriguez, accessed January 25, 2026,  
<https://jrodthoughts.medium.com/inside-sqlang-lmsys-new-framework-for-super-fast-llm-inference-77e67b8933ce>
19. SGLang: Efficient Execution of Structured Language Model Programs - arXiv, accessed January 25, 2026, <https://arxiv.org/pdf/2312.07104>
20. [Bug] HiCacheController gets stuck when testing with multiple long text documents · Issue #3998 · sgl-project/sqlang · GitHub, accessed January 25, 2026, <https://github.com/sql-project/sqlang/issues/3998>
21. Epic: Efficient Position-Independent Context Caching for Serving Large Language Models, accessed January 25, 2026, <https://arxiv.org/html/2410.15332v1>
22. Joint Encoding of KV-Cache Blocks for Scalable LLM Serving | OpenReview, accessed January 25, 2026, <https://openreview.net/forum?id=M9SgtgvF7I>
23. How positional encoding affects KV caching : r/learnmachinelearning - Reddit, accessed January 25, 2026,  
[https://www.reddit.com/r/learnmachinelearning/comments/1j8671r/how\\_positional\\_encoding\\_affects\\_kv\\_caching/](https://www.reddit.com/r/learnmachinelearning/comments/1j8671r/how_positional_encoding_affects_kv_caching/)
24. A Survey on Large Language Model Acceleration based on KV Cache Management - arXiv, accessed January 25, 2026, <https://arxiv.org/html/2412.19442v2>
25. My journey understanding: KV-Cache. Clarifying and correcting relevant sources. - Medium, accessed January 25, 2026, <https://medium.com/@kshitijkhode/my-journey-understanding-kv-cache-clarifying-and-correcting-relevant-sources-d0e5479b830b>
26. Epic: Efficient Position-Independent Context Caching for Serving Large Language Models, accessed January 25, 2026, <https://arxiv.org/html/2410.15332v2>

27. ICML Poster EPIC: Efficient Position-Independent Caching for Serving Large Language Models, accessed January 25, 2026,  
<https://icml.cc/virtual/2025/poster/43926>
28. Serialization - Hugging Face, accessed January 25, 2026,  
[https://huggingface.co/docs/huggingface\\_hub/v0.24.1/package\\_reference/serialization](https://huggingface.co/docs/huggingface_hub/v0.24.1/package_reference/serialization)
29. SafeTensors: Efficient Serialization Format for Deep Learning | by Nishtha kukreti | Medium, accessed January 25, 2026,  
<https://medium.com/@nishthakukreti.01/safetensors-efficient-serialization-format-for-deep-learning-57364317be43>
30. MultiPath Transfer Engine: Breaking GPU and Host-Memory Bandwidth Bottlenecks in LLM Services - arXiv, accessed January 25, 2026,  
<https://arxiv.org/html/2512.16056>
31. LMCache/LMCache: Supercharge Your LLM with the Fastest KV Cache Layer - GitHub, accessed January 25, 2026, <https://github.com/LMCache/LMCache>
32. Integration - LMCache, accessed January 25, 2026,  
[https://docs.lmcache.ai/developer\\_guide/integration.html](https://docs.lmcache.ai/developer_guide/integration.html)
33. LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference - arXiv, accessed January 25, 2026, <https://arxiv.org/html/2510.09665v2>
34. Benchmarking - LMCache, accessed January 25, 2026,  
[https://docs.lmcache.ai/getting\\_started/benchmarking.html](https://docs.lmcache.ai/getting_started/benchmarking.html)
35. Open-Source LLM Inference Cluster Performing 10x FASTER than SOTA OSS Solution, accessed January 25, 2026,  
<https://blog.lmcache.ai/en/2025/03/06/open-source-lm-inference-cluster-performing-10x-faster-than-sota-oss-solution/>
36. How to turn on the kv cache disk? #10017 - sgl-project/sglLang - GitHub, accessed January 25, 2026, <https://github.com/sql-project/sglLang/issues/10017>
37. Alibaba Cloud Tair Partners with SGLang to Build HiCache: Constructing a New Cache Paradigm for "Agentic Inference", accessed January 25, 2026,  
[https://www.alibabacloud.com/blog/alibaba-cloud-tair-partners-with-sqlang-to-build-hicache-constructing-a-new-cache-paradigm-for-agentic-inference\\_602767](https://www.alibabacloud.com/blog/alibaba-cloud-tair-partners-with-sqlang-to-build-hicache-constructing-a-new-cache-paradigm-for-agentic-inference_602767)
38. ChunkKV: Semantic-Preserving KV Cache Compression for Efficient Long-Context LLM Inference | OpenReview, accessed January 25, 2026,  
<https://openreview.net/forum?id=20JDhbJqn3>
39. ChunkKV: Semantic-Preserving KV Cache Compression for Efficient Long-Context LLM Inference - arXiv, accessed January 25, 2026,  
<https://arxiv.org/html/2502.00299v1>
40. Don't Break the Cache: An Evaluation of Prompt Caching for Long-Horizon Agentic Tasks, accessed January 25, 2026, <https://arxiv.org/html/2601.06007v1>
41. Prompt Caching Explained | DigitalOcean, accessed January 25, 2026,  
<https://www.digitalocean.com/community/tutorials/prompt-caching-explained>
42. Prompt Caching with OpenAI, Anthropic, and Google Models - PromptHub, accessed January 25, 2026,  
<https://www.promphub.us/blog/prompt-caching-with-openai-anthropic-and-go>

## ogle-models

43. Structuring Applications to Secure the KV Cache | NVIDIA Technical Blog, accessed January 25, 2026,  
<https://developer.nvidia.com/blog/structuring-applications-to-secure-the-kv-cache/>
44. Breaking Through the Memory Wall: How CXL Transforms RAG and KV Cache Performance, accessed January 25, 2026,  
<https://www.asteralabs.com/breaking-through-the-memory-wall-how-cxl-transforms-rag-and-kv-cache-performance/>
45. How to Reduce KV Cache Bottlenecks with NVIDIA Dynamo | NVIDIA Technical Blog, accessed January 25, 2026,  
<https://developer.nvidia.com/blog/how-to-reduce-kv-cache-bottlenecks-with-nvidia-dynamo/>
46. Inside vLLM's New KV Offloading Connector: Smarter Memory Transfer for Maximizing Inference Throughput, accessed January 25, 2026,  
<https://blog.vllm.ai/2026/01/08/kv-offloading-connector.html>