PolarKV: Tier Locally, Serve Globally– A KV Cache over Cloud Memory and Storage

Yingqiang Zhang
Zhejiang University &
Alibaba Cloud Computing

Xinjun Yang
Alibaba Cloud Computing

Hao Chen*
Alibaba Cloud Computing

Feifei Li
Alibaba Cloud Computing

Shuwen Wang
Alibaba Cloud Computing

Chuan Sun
Alibaba Cloud Computing

Sai Wu
Zhejiang University

Ninglong Weng
Alibaba Cloud Computing

ABSTRACT

KV caching is essential for reducing redundant attention computation in LLM inference, yet existing memory-centric or storage-centric approaches operate inefficiently in cloud environments due to the bandwidth–capacity–cost coupling of cloud resources. Furthermore, conventional hierarchical cache designs treat each tier as a global pool and often incur substantial cross-tier data movement, which in cloud deployments becomes inter-node network traffic—an extremely scarce and expensive resource.

In this paper, we first analyze the bandwidth–capacity–cost characteristics of cloud memory and storage and derive practical guidelines for selecting and combining cloud resources for KV caching. Building on these insights, we present *PolarKV*, a cloud-native distributed KV cache pool deployed at scale in Alibaba Cloud. *PolarKV* organizes disaggregated memory and cloud block storage into a shard-paired, co-located hierarchy, treating memory as a bandwidth tier and storage as a capacity tier. *PolarKV* breaks the "tier-as-a-whole" abstraction: instead of migrating data between two global tiers, it partitions both the memory tier and the storage tier into aligned shards and performs tiering as shard-to-shard movement. By co-locating each memory shard with its corresponding storage shard on the same VM, *PolarKV* confines most tiering traffic to local paths rather than the network, removing additional VM-to-VM data movement network traffic. *PolarKV* is already commercially available as an independent KV cache service on Alibaba Cloud, enabling customers to connect inference engines such as *vLLM* and *SGLang* through a lightweight library. In our evaluation across controlled benchmarks and real production deployments, *PolarKV* reduces time-to-first-token (TTFT) by up to 80%.

PVLDB Reference Format:
Yingqiang Zhang, Xinjun Yang, Hao Chen, Feifei Li, Shuwen Wang, Chuan Sun, Sai Wu, and Ninglong Weng. PolarKV: Tier Locally, Serve Globally– A KV Cache over Cloud Memory and Storage. PVLDB, 19(12): 4480-4493, 2026.
doi:10.14778/3827998.3828047

*Hao Chen is the corresponding author (ch341982@alibaba-inc.com).

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.

1 INTRODUCTION

Large Language Models (LLMs) have rapidly become the foundation of modern AI applications [1, 34, 51, 52]. Central to their success is the Transformer architecture [42], whose attention mechanism enables effective modeling of long-range dependencies. Yet autoregressive decoding imposes substantial computational and memory overhead: each new token requires attending to the entire prefix. KV caching mitigates this cost by storing previously computed attention states [23, 36, 43], avoiding redundant computation and enabling prefix sharing across requests.

A growing body of prior work has extended KV caching beyond GPU HBM by storing KV in host DRAM, remote memory or distributed storage via distributed-file-system-based solutions [2, 11, 23, 36, 39, 55]. These designs demonstrate the usefulness of multi-tier KV placement and work well on traditional fixed clusters. However, they do not address the performance and cost characteristics unique to large-scale cloud environments, where bandwidth, capacity, and cost coupling, rather than raw capacity, dominate KV cache efficiency.

In production cloud platforms, we observe several fundamental operational constraints that limit existing KV cache designs. First, **storage is not always cheaper**: the bandwidth of cloud block storage (CBS) is tightly coupled to provisioned capacity, so achieving high throughput often requires reserving large volumes, making high-end storage tiers nearly as costly as memory. Moreover, CBS must be attached to VMs whose network bandwidth can sustain the storage bandwidth, and VM network bandwidth is itself tied to instance specifications; thus, using high-bandwidth CBS frequently forces provisioning oversized VM types with excessive CPU and memory resources, further inflating deployment cost. Second, **disaggregated memory can be significantly underutilized**, as its effective throughput is bounded by per-instance network bandwidth, which scales with the VM specification rather than with actual memory demand; guaranteeing sufficient remote-memory bandwidth often requires large instances, again increasing cost but with more memory resources. Third, **a hierarchical design is necessary, but classical hierarchies fail in the cloud:** given these constraints, neither memory-only nor storage-only designs offer a

doi:10.14778/3827998.3828047

viable performance–cost envelope. A hybrid memory–storage hierarchy becomes essential. However, existing hierarchical systems often treat memory and storage as two independent global pools. As a result, tiering operations frequently translate into inter-node data movement. In the cloud, such tiering traffic typically traverses VM-to-VM networks, where bandwidth is scarce, costly, and tied to instance specifications. Consequently, classical hierarchical designs suffer from severe network bottlenecks and require expensive over-provisioning, negating their intended benefits.

These observations highlight the need for a cloud-native KV cache architecture that jointly reasons about memory, storage, bandwidth, and VM-level constraints, rather than directly porting traditional hierarchical caching techniques into the cloud.

To address these challenges, we present *PolarKV*, a cloud-native distributed KV cache pool designed and deployed at scale on Alibaba Cloud. *PolarKV* adopts a **shard-paired, co-located hierarchical** architecture that unifies disaggregated memory and cloud block storage within each memory-node VM. In this design, memory serves as a **bandwidth tier**, while storage provides **elastic capacity**, enabling *PolarKV* to scale efficiently with cloud resources.

PolarKV breaks the "tier-as-a-whole" abstraction. Instead of treating memory and storage as two global pools and migrating data across tiers at cluster scope, *PolarKV* partitions both tiers into aligned shards and performs tiering as shard-to-shard movement. By co-locating each memory shard with its corresponding storage shard on the same memory-node VM (via directly attached cloud block storage volumes), *PolarKV* confines most tiering traffic to intra-VM data paths, largely removing network bandwidth from the critical tiering operations.

Architecturally, *PolarKV* externalizes KV management as an independent service, consisting of (i) a distributed memory layer, (ii) a co-located storage layer paired with memory shards, and (iii) a metadata service that tracks KV placement and lifecycle. Building on our prior distributed memory pool [8, 49], *PolarKV* adds production-grade metadata management and direct block-storage integration while avoiding distributed file-system overheads and dedicated storage VMs. Correctness is preserved by falling back to local recomputation on cache misses or failures.

By decoupling KV cache placement from inference execution, *PolarKV* enables KV cache to be shared across nodes, allowing successive requests of the same session to reuse KV cache even when scheduled on different nodes. It has been commercially deployed as an independent KV cache service on Alibaba Cloud and integrates with mainstream inference engines, including *vLLM* and *SGLang*, through a lightweight client library. Importantly, this integration is non-intrusive: users can enable *PolarKV* by simply installing the client library and configuring the service endpoint, without any changes to *vLLM* or *SGLang*. This drop-in design significantly lowers adoption cost and allows *PolarKV* to serve as a reusable KV cache backend for existing and evolving LLM serving stacks.

We summarize our main contributions as follows:

- We conduct an empirical analysis of cloud memory and block storage, showing how bandwidth–capacity–cost coupling fundamentally constrains KV cache performance. Based on these observations, we derive practical guidelines for cost-effective KV caching by appropriately combining cloud memory and storage.
- We introduce a shard-paired, co-located hierarchical architecture that treats memory as a bandwidth tier and storage as a capacity tier. The key idea is to break the "tier-as-a-whole" abstraction by partitioning both tiers into aligned shard pairs and performing tiering as local shard-to-shard movement within each VM, thereby keeping tiering traffic off the network-critical path.
- Building on this architecture, we implement *PolarKV* as a distributed KV cache system that materializes the shard-paired, co-located design by directly attaching cloud storage to memory-serving nodes. This implementation reduces network bandwidth overhead, simplifies tiering logic, and enables efficient scaling with cloud resource elasticity.
- We evaluate *PolarKV* under diverse workloads on Alibaba Cloud, including controlled microbenchmarks, public long-context benchmarks, and a representative case study from large-scale online deployments, demonstrating its performance and cost efficiency in large-scale LLM serving environments.

The rest of this paper is organized as follows. First, we present the background and motivation in Section 2. Then we provide *PolarKV*'s overview and detailed design in Section 3 and Section 4. Next, we evaluate *PolarKV* in Section 5 and review the related work in Section 6. Finally, we conclude the paper in Section 7.

2 BACKGROUND AND MOTIVATION

2.1 Background on LLM Inference

LLM inference. Most LLMs adopt decoder-only Transformer architectures [42] and generate tokens autoregressively: given a prompt, the model predicts the next token one step at a time until an end-of-sequence (EOS) symbol is reached. Figure 1 illustrates the LLM inference process, which typically consists of two phases: *prefill* and *decode*. During prefill, it processes all input tokens to generate the initial output token, which usually results in high computational complexity. Since the computational complexity of attention mechanisms scales quadratically with input length, the computation time in the prefill phase generally grows superlinearly with input length. In contrast, during decode, tokens are generated autoregressively one by one based on previously produced outputs until reaching the EOS marker.

KV cache. At the heart of Transformer inference lies the self-attention mechanism, which enables the model to capture long-range dependencies across tokens. Given an input sequence of hidden states $X = [x_1, x_2, \ldots, x_n]$, each attention layer computes:

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{1}$$

where $Q = XW_Q$, $K = XW_K$, and $V = XW_V$ are the query, key, and value matrices, respectively, and d_k is the key dimension. During autoregressive decoding, for each newly generated token, the model must form a new query Q_t and compute its attention with all previous keys $[K_1, K_2, \ldots, K_{t-1}]$ and values $[V_1, V_2, \ldots, V_{t-1}]$ to predict the next token.

Figure 2 illustrates the attention process during inference. Given input tokens x_1, x_2, x_3, the model first enters the *prefill* phase, processing the full input sequence to compute the corresponding query, key, and value matrices and generates the next token x_4. In the

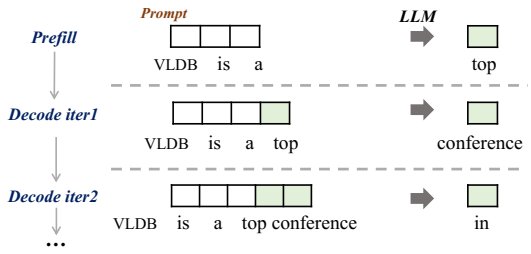


Figure 1: Overview of the LLM inference pipeline

first iteration of the *decode* phase, the newly generated token x_4 is appended to the previous context to form $[x_1, x_2, x_3, x_4]$, which the model uses to predict the next token. In each step, the model attends to all previous tokens by recomputing K and V for earlier positions, even though these representations remain unchanged. This leads to substantial redundant computation—for instance, $K_1, K_2, K_3, V_1, V_2, V_3$ are repeatedly recomputed at every decoding step. As the sequence grows longer, the inference cost increases quadratically with the sequence length.

To improve efficiency, modern LLMs cache the key and value matrices in GPU memory. In subsequent decoding steps, when generating a new token x_t, the model only needs to compute the current token's $Q_t = x_t W_Q$, $K_t = x_t W_K$, and $V_t = x_t W_V$, while directly reusing the cached $\{K_i\}_{i=1}^{t-1}$ and $\{V_i\}_{i=1}^{t-1}$ from previous tokens. This *KV cache* mechanism eliminates redundant computation and significantly accelerates autoregressive generation. By reusing historical keys and values, it reduces the per-token computation complexity from $O(n^2)$ to $O(n)$, enabling fast, scalable, and memory-efficient inference—particularly critical for long-context applications.

Prefix cache. While the standard KV cache avoids redundant computation during the decoding process of a single request, it cannot exploit redundancy across different requests. In real-world LLM serving, requests often share long common prefixes, such as system prompts, shared contexts, or dialogue histories. However, without cross-request reuse, each request must still perform prefill computation for the shared prefix, leading to inefficiency.

To address this limitation, recent systems introduce *prefix cache*. Instead of caching only the keys and values generated during the current session, a prefix cache stores reusable attention states (i.e., precomputed K and V tensors) corresponding to frequently occurring prefix segments. When a new request begins with a known prefix, the model can directly load the corresponding cached KV and skip the expensive prefill computation for that segment. This approach extends the benefits of KV caching from intra-request reuse to inter-request reuse, significantly improving end-to-end throughput and latency for workloads with shared context. Prefix caching has become a fundamental optimization in modern LLM inference engines such as *vLLM* and *SGLang*, and has also attracted increasing attention in the academic community [10, 26, 47].

KV cache stores. The capacity of GPU high-bandwidth memory (HBM) is inherently limited and is primarily consumed by model weights. As LLM context lengths continue to grow [22, 40], the size of the KV cache increases proportionally, quickly exceeding the remaining HBM capacity. This has motivated a line of work exploring alternative KV cache storage layers on host memory, remote memory, or storage to provide larger capacity at lower cost.

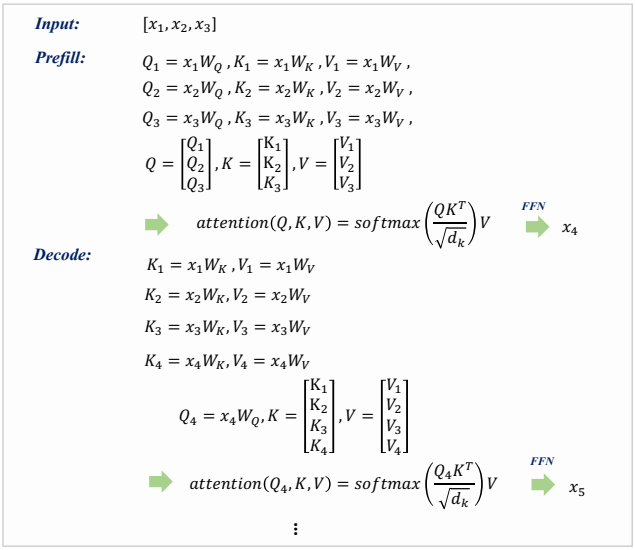


Figure 2: Attention computation in prefill and decode stages

Mooncake [36] pools host memory across inference nodes to create a shared KV cache pool. AIBrix [41] further extends this idea by placing KV caches on disaggregated remote memory. LMCache [11] uses host memory as its default KV store and exposes interfaces to integrate external stores such as Mooncake. Other systems leverage distributed file systems (e.g., 3FS [39]) to implement SSD-based KV caches for improved cost efficiency, while AlayaDB [14] stores KV entries inside a database. Although effective in fixed, on-premises cluster environments, these systems do not account for the unique performance and cost characteristics of large-scale cloud deployments. Moreover, beyond GPU HBM and local host memory, most designs adopt a single-layer approach—either disaggregated memory or storage—rather than a hierarchy that jointly leverages both. As a result, existing solutions fall short in balancing capacity, performance, and cost under cloud resource constraints.

2.2 Characteristics of Cloud-Based KV Cache

Cloud storage is not always cheaper. Although cloud block storage (CBS) is often viewed as a low-cost alternative for KV cache placement, especially compared to memory, our analysis shows that this intuition breaks down in real cloud environments. CBS offerings such as AWS EBS [3], Azure Ultra Disks [4] and Alibaba Cloud ESSD [12] exhibit a strong bandwidth–capacity coupling: higher bandwidth is available only when provisioning proportionally larger storage volumes. As shown in Table 1 (using Alibaba ESSD as an illustrative example), achieving multi-GB/s throughput may require reserving terabytes of capacity, even in scenarios where the KV cache itself is far smaller.

For many modern LLM deployments, especially during high-throughput decoding, memory and I/O bandwidth—rather than raw capacity—often becomes the dominant performance bottleneck [13, 18, 38]. KV cache access is highly bandwidth-intensive, requiring sustained high-throughput reads/writes from memory (and, when offloaded, from remote memory/storage), especially as context lengths and concurrency increase, and as model sizes grow, inflating the per-token KV footprint [23, 31]. In these cases, the

Table 1: Specifications of Alibaba Cloud ESSD Block Storage Tiers

Metric	PL3	PL2	PL1	PL0
Provisioned Bandwidth	$\min(120 + 0.5 \times C,\ 4000)$	$\min(120 + 0.5 \times C,\ 750)$	$\min(120 + 0.5 \times C,\ 350)$	$\min(100 + 0.25 \times C,\ 180)$
Maximum Bandwidth (MB/s)	4000	750	350	180
Saturation Capacity (GB)	7760	1260	460	320
Cost (CNY/GB/month)	4	2	1	0.5
Cost at Max Bandwidth (CNY/month)	31040	2520	460	160
Cost per GB/s of Bandwidth (CNY/month)	7946	3441	1346	910

C denotes the storage capacity in GB.

Table 2: Configuration and cost of selected Alibaba Cloud ECS instance types

Metric	Memory-optimized Family					Compute-optimized Family				
	r9i.large	r9i.xlarge	r9i.2xlarge	…	r9i.48xlarge	c8i.large	c8i.xlarge	c8i.2xlarge	…	c8i.48xlarge
vCPU count	2	4	8	…	192	2	4	8	…	192
Memory (GB)	16	32	64	…	1536	4	8	16	…	512
Network bandwidth (GB/s)	0.31	0.50	0.75	…	8.00	0.31	0.50	0.75	…	12.50
Monthly cost (CNY)	317.48	634.95	1269.91	…	32001.67	195.70	391.39	782.78	…	19618.40
Cost per GB/s of Bandwidth (CNY/month)	1024.13	1269.90	1693.21	…	4000.21	631.29	782.78	1043.71	…	1569.47
Cost per GB (CNY/month)	19.84	19.84	19.84	…	20.83	48.93	48.92	48.92	…	38.32

effective cost of storage is governed not by the per-GB price but by the amortized cost per GB/s of usable bandwidth. Table 1 and Table 2 show that this bandwidth-normalized cost for CBS can be comparable to, or sometimes exceed, the bandwidth cost associated with the VM instances that provide memory resources.

Additionally, CBS cannot be used standalone: block storage must be attached to VM instances with sufficient network bandwidth to match storage throughput and maximize performance. Since network bandwidth scales with VM specification, high-bandwidth storage deployments require high-end VM types, as shown in Table 2, adding further cost. For example, 10 GB/s ESSD bandwidth requires an ECS instance sustaining 10 GB/s network bandwidth, substantially increasing total cost.

These characteristics are not unique to Alibaba Cloud; other major providers offer storage with similar bandwidth–capacity relationships. Overall, the operational cost of achieving high-bandwidth KV access on CBS is significantly higher than implied by per-GB pricing alone, and storage-only KV cache designs may not provide the expected cost advantage for bandwidth-intensive workloads.

Cloud memory may be underutilized. Disaggregated memory pools in the cloud are typically built by aggregating DRAM across VMs. While DRAM itself provides high raw bandwidth, its effective bandwidth in a distributed memory pool is limited by the hosting VM's network interface. As shown in Table 2, memory-rich instances often offer only modest network throughput. This mismatch means that even multi-hundred-GB or terabyte-scale DRAM configurations can become bandwidth-limited rather than capacity-limited, leaving substantial portions of memory underutilized when serving KV cache traffic from multiple inference engines.

Conversely, instances with higher network bandwidth often provide higher memory capacities and higher per-GB cost. Thus, memory capacity and memory bandwidth do not scale proportionally in the cloud—an important but frequently overlooked operational constraint. The cost-effective way to build a large DRAM pool is not always to purchase the largest-memory instances; in many cases, multiple smaller instances with higher aggregate bandwidth offer better effective throughput for KV access.

Overall, these observations highlight that cloud memory, much like cloud storage, does not offer a simple performance–cost trade-off. A KV cache that naively treats DRAM as a uniformly high-performance tier risks severe underutilization and unnecessary cost. The cloud-native KV cache design must instead reason jointly about memory capacity, usable bandwidth, and instance-level costs, rather than treating memory as a monolithic high-bandwidth resource.

Memory–storage hierarchy becomes necessary. The above observations reveal a fundamental asymmetry in cloud resources: memory is a **bandwidth resource**, while storage is a **capacity resource**. Cloud memory offers the only practical way to achieve multi-GB/s to tens-of-GB/s effective bandwidth (by combining some small VMs) for hot KV access, but scaling it for capacity is prohibitively expensive and often leads to bandwidth underutilization. Conversely, cloud block storage provides elastic capacity at low per-GB cost, but achieving sufficient bandwidth requires substantial over-provisioning of both storage volume and high-bandwidth VM.

Because KV cache workloads may simultaneously require high bandwidth for a small hot set and large capacity for the cold set, neither memory-only nor storage-only designs can provide a satisfactory performance–cost operating point. Instead, the natural architectural consequence is a memory–storage hierarchy, where

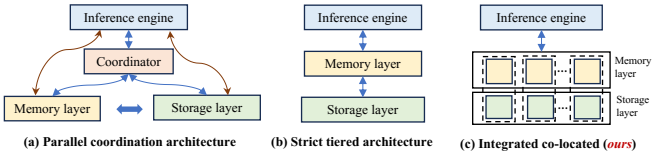


Figure 3: Mainstream hierarchical cache architectures *vs.* our co-located integrated design

the **memory tier serves as a bandwidth layer** for hot KV data, and the **storage tier serves as a capacity layer** for the colder KV entries. This division aligns each cloud resource with its strengths and avoids the inefficiencies of single-tier designs.

2.3 Hierarchical KV Caching Challenges

While hierarchical caching is a long-standing concept in storage and memory systems [21, 32, 38, 44], applying it directly to cloud-based KV caching introduces unique challenges. As shown in Figure 3(a) and Figure 3(b), existing designs can be abstracted into two patterns: parallel-coordination architectures, where memory and storage operate as peer layers and placement is mediated by a controller, and strict tiered architectures, where storage serves as an overflow (backing) tier for memory and misses trigger fetch-and-fill from storage. Despite their differences, both abstractions treat each tier as a global pool and rely on frequent cross-tier data movement, which in cloud deployments typically traverses the network.

These designs implicitly assume cheap cross-tier data movement, an assumption that breaks down in cloud environments, especially for KV cache workloads where multi-GB transfers are common and network bandwidth is the dominant bottleneck. As shown earlier, cloud network bandwidth scales with VM specifications, not data demand, making frequent migration between memory and storage expensive and slow. Consequently, naively applying classical hierarchical designs leads to substantial performance loss and inflated operational cost.

These limitations highlight a deeper insight: **hierarchy alone is insufficient**—the tiers must be organized in a way that avoids cross-node movement and removes dependence on network bandwidth for tiering operations. This motivates our co-located, cloud-native memory–storage organization, shown in Figure 3(c). Instead of operating two globally disjoint tiers, *PolarKV* breaks the "tier-as-a-whole" abstraction by partitioning both memory and storage into aligned units and co-locating each memory unit with its corresponding storage capacity within the same VM. Concretely, each memory-node VM attaches its own cloud block storage volumes, allowing memory–storage transfers (e.g., fill and eviction/writeback) to occur locally within a single VM rather than across the network.

This architectural shift turns the expensive inter-node tiering problem into an efficient intra-node data movement path, eliminating additional network overhead on VM-to-VM data movement and enabling the hierarchy to scale smoothly with cloud resources. Conceptually, *PolarKV* preserves the semantic benefits of hierarchical caching while avoiding the performance penalties that arise when tiering traffic must traverse the network.

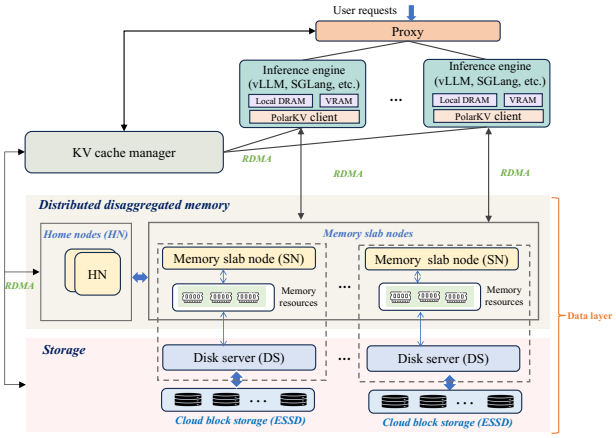


Figure 4: Overview of *PolarKV*

3 OVERVIEW

Rationale. With the increasing deployment of LLM services on cloud platforms, the primary goal of *PolarKV* is to build a cost-effective and high-performance KV cache system that fully leverages cloud-native infrastructure. KV caching requires both high bandwidth and large capacity to accommodate the growing attention states of modern LLMs. In traditional monolithic settings, these requirements are typically satisfied by local DRAM or NVMe devices. In the cloud, however, bandwidth to disaggregated memory and cloud block storage is usually lower than on-node interconnects and is coupled with instance specifications—for example, the bandwidth of block storage often scales with its provisioned capacity. *PolarKV* is designed with these constraints in mind and adopts a shard-paired, co-located hierarchy that balances cost, capacity, and performance by localizing tiering within each VM.

Overview. Figure 4 illustrates the architecture of *PolarKV*. The system organizes its distributed KV cache pool into two hierarchical layers: a memory layer, realized through distributed disaggregated memory, and a storage layer, backed by cloud block storage. Above this data layer, a KV manager server manages resource metadata, coordinates memory and storage allocation, and exposes a unified interface to inference engines. During runtime, the inference engine contacts the manager server to obtain remote memory addresses corresponding to the target KV entries and subsequently performs direct RDMA-based read/write operations to the disaggregated memory nodes. To support large-scale multi-node inference, a proxy component is deployed alongside each inference node to schedule user requests, balance load, and exploit data locality when possible.

Data layer. The data layer combines disaggregated memory with cloud block storage to form a hierarchical caching substrate. It breaks the "tier-as-a-whole" abstraction: rather than operating two global tiers and migrating data across them at cluster scope, *PolarKV* partitions both tiers into aligned shards and treats tiering as shard-to-shard movement within each memory-node VM. Each memory-serving VM hosts a memory shard and attaches its own cloud block storage volumes, forming a co-located shard pair that confines most data movement traffic to intra-VM paths. The memory tier is implemented using our distributed memory pool (DMP), introduced in our previous work [8, 49], which provides elastic, high-bandwidth

remote memory accessible via RDMA. Within DMP, the Home Node (HN) handles resource management and metadata coordination, while the Slab Nodes (SNs) contribute the actual physical memory resources. The inference engine connects directly to SNs through the RDMA network, achieving low-latency and high-throughput access. The storage tier extends capacity using cloud block storage (Alibaba ESSD in our product). Each SN runs a lightweight disk server (DS) process that asynchronously offloads cold KV cache entries to ESSD and reloads them when required.

KV manager. The KV manager is responsible for managing both memory and storage resources and maintaining the global KV cache metadata. It communicates with the HN of the disaggregated memory system to allocate or reclaim large contiguous memory chunks. Once the HN returns available regions and their corresponding SN locations, the KV manager exposes these as addressable spaces to inference engines. Before performing read or write operations, inference engines query the KV manager for address mapping, ensuring efficient memory utilization and consistent metadata synchronization. Communication between the inference engines and the KV cache manager uses RDMA-based RPC to achieve low latency.

Proxy. The Proxy component operates at the inference-node level to coordinate requests and maintain cache locality. It balances incoming traffic across inference nodes and routes each request to the node with the highest cache affinity, thereby improving hit rates and reducing remote access. Since inference engines typically maintain a small local DRAM-resident KV cache, the proxy leverages this locality information to prioritize requests that can be served directly from local memory, further minimizing latency and inter-node communication.

4 DESIGN AND IMPLEMENTATION

4.1 Client-side library

The client-side library of *PolarKV* serves as the bridge between the LLM inference engine and the distributed KV cache pool. It provides an abstraction that hides the underlying complexity of remote memory management and RDMA communication, allowing inference frameworks to store and retrieve KV states transparently. As shown in Figure 5, the library provides two core components: a chunk-based hashing module and an RDMA client.

Chunk-based hashing scheme. The inference engine invokes `store_kv(token_ids)` and `load_kv(token_ids)` to store or retrieve KV cache entries. Upon receiving a request, the library first divides the input token sequence into multiple chunks and computes a unique hash key for each chunk. These hash keys are then used to index and locate KV entries across remote memory and storage nodes. To enable prefix-aware reuse, *PolarKV* employs a chunk-based prefix-hash chain, where each chunk's hash is computed by concatenating its token IDs with the hash of its predecessor, following common practice in existing LLM caching systems [11, 23]. This recursive chaining ensures that two chunks share the same prefix key only if all preceding tokens are identical, preserving semantic correctness for prefix matching. Through this mechanism, *PolarKV* supports fine-grained cache management and efficiently reuses overlapping KV segments across requests.

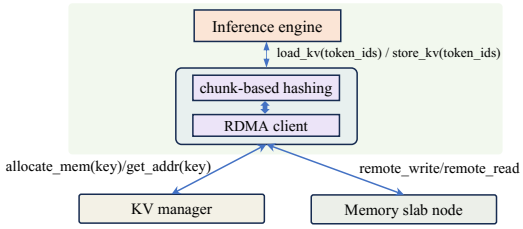


Figure 5: Client-side overview of *PolarKV*

RDMA client. The RDMA client manages the communication between the inference node, the KV manager, and the slab nodes that constitute the disaggregated memory pool. It uses an RDMA-based RPC protocol to allocate remote memory regions and obtain the physical addresses associated with specific hash keys. Once the addresses are resolved, one-sided RDMA read and write operations are used to directly access memory on remote nodes, ensuring low latency and minimal CPU involvement.

4.2 KV cache manager

The KV cache manager is *PolarKV*'s core control component. It manages the lifecycle of KV cache across both memory and storage and serves client requests from inference nodes. As illustrated in Figure 6, communication between clients and the KV cache manager is conducted via an RDMA-based RPC protocol. The manager employs an RPC handler to process incoming requests and coordinate cache operations such as memory allocation, metadata lookup, and data migration. Internally, the KV cache manager consists of three main submodules: the memory manager, which handles disaggregated memory resources; the storage manager, which coordinates cache offloading to cloud storage; and the metadata module, which tracks the location and status of all cached entries. The following subsections describe each of these components in detail.

Memory manager. The memory manager is responsible for managing raw memory resources provided by the distributed disaggregated memory. At initialization, it allocates a set of fixed-size memory chunks from the disaggregated memory pool and records their metadata, including node ID, starting address, and size, in a hash table. When a client requests memory for storing a KV cache entry, the memory manager scans the available chunks to locate one with sufficient free space, splits the chunk as needed, and updates the metadata table accordingly. The remaining space in the chunk is retained for future allocations.

Each allocated KV cache entry is also registered in both the metadata hash table and the LRU list for cache replacement management. When an entry is evicted from the LRU list, its associated memory region is released back to the memory manager and merged with the original chunk from which it was allocated. Since the chunk size is fixed on the client side, the size of each KV cache entry is also constant, which simplifies memory allocation, reduces fragmentation, and streamlines the overall management logic.

To balance load across nodes, the memory manager uses round-robin allocation among memory nodes, distributing requests evenly and avoiding resource hotspots.

Metadata management. The metadata manager maintains global metadata for all KV cache entries, including their locations and

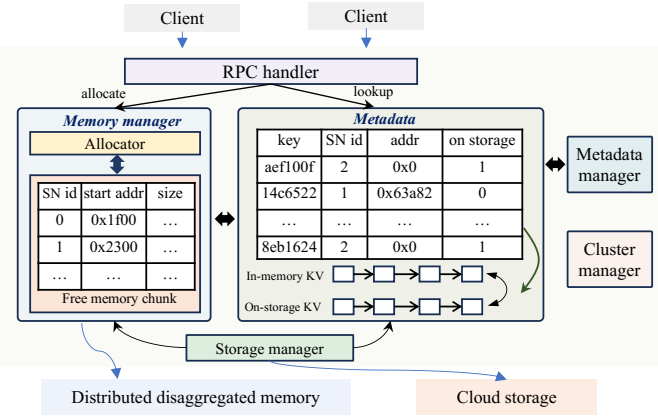


Figure 6: Overview of KV cache manager

states. It stores the mapping between each key and its corresponding memory address in a hash table. Since some KV caches may reside on cloud storage, an additional flag is used to indicate whether an entry is stored in memory or on storage. For on-storage entries, the corresponding memory address is set to zero.

To track data access patterns and support eviction decisions, the metadata manager maintains two LRU lists, one for in-memory KV caches and the other for on-storage caches. Each hash-table entry maintains a pointer to its node in the corresponding LRU list, allowing quick updates when a lookup request hits an existing entry. When a client requests to load a KV cache, it first sends an RPC to the KV manager, which queries the metadata table to locate the entry and returns the memory node ID and the corresponding address. Once the client obtains the address, it can directly perform a one-sided RDMA read from the memory node.

To migrate data between memory and storage, the metadata manager employs a background thread that periodically issues offload requests to the storage. When KV caches are moved from memory to storage, their metadata entries are also migrated from the in-memory LRU list to the on-storage list. When the total storage usage reaches a predefined threshold, the system evicts the least recently used on-storage entries to free up capacity.

To improve fault tolerance, the metadata manager can persist the global metadata (e.g., key-to-address mappings, location flags, and LRU states) to a cloud database. This enables fast recovery after KV manager server failures: if the KV manager crashes, it can reconstruct the metadata table by reloading from the cloud database, thereby restoring the locations and states of all KV cache entries. To avoid impacting the critical path, metadata persistence is performed asynchronously (e.g., via a background flush thread and batched updates), so normal lookup and RDMA access remain high-performance while durability is provided in the background.

Write/read workflow. Based on the preceding components, Figure 7 illustrates the end-to-end KV cache write and read workflow in *PolarKV* when accessing disaggregated memory. The handling of on-storage data will be introduced in the next subsection.

When a client initiates a write operation, it first sends an RPC request to the KV manager to allocate remote memory, which is handled by the memory manager (❶). After the allocation, the

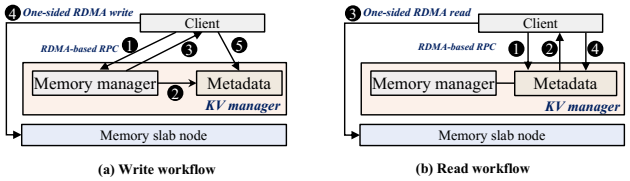


Figure 7: Workflow of in-memory data access

key and its associated memory information are recorded in the metadata table (❷), and the key is temporarily write-locked to prevent concurrent access or background eviction before the data is ready. The KV manager then returns the allocated address to the client (❸). Upon receiving the address, the client performs a one-sided RDMA write to transfer the KV data directly into remote memory (❹). After the write completes, the client issues an RPC to release the write lock (❺), finalizing the operation.

For a read operation, the client first requests the corresponding address from the KV manager (❶). The KV manager looks up the metadata table and, if the entry is found, temporarily read-locks it to prevent concurrent writes or background eviction, then returns the memory-node ID and address to the client (❷); otherwise, an invalid address is returned to indicate a cache miss. Once a valid address is returned, the client performs a one-sided RDMA read from the remote memory (❸). Finally, after the data is fetched, the client sends an RPC to unlock the key (❹).

PolarKV's RPC is implemented over RDMA, enabling request handling within tens of microseconds. For data transfer, *PolarKV* fully exploits one-sided RDMA operations, allowing direct remote memory access without involving remote CPUs. Together, these designs enable highly efficient data access.

Concurrency control. *PolarKV* uses per-key locks in the KV manager to handle concurrent accesses. For a write, the manager allocates memory and installs a write lock only if the key does not already exist. The entry becomes readable only after the client finishes the one-sided RDMA write and sends a write-finish RPC. For a read, the manager returns the remote address only when the entry exists and is not write-locked; otherwise, the request is treated as a cache miss or retried according to the client policy. Thus, readers observe either a fully written KV entry or a cache miss, but never partially written data. If multiple clients concurrently write the same key, only the first writer proceeds, while later writers receive a duplicate/in-progress status and skip the write. This is safe because KV entries for the same model and prefix are deterministic, and it avoids duplicate allocation and RDMA traffic.

Scalability. The KV manager is involved only in metadata operations, such as address lookup, memory allocation, lock management, and metadata state transitions. It is not on the large-data transfer path: after an address is resolved, KV tensors are transferred directly between the inference engine and memory slab nodes through one-sided RDMA read/write operations. Therefore, the manager handles small metadata RPCs rather than large KV data movement. The key-address table is hash-based, and the LRU lists can be partitioned by key range to reduce contention under high concurrency. Since KV entries are naturally identified by prefix/hash keys, the manager design is also amenable to sharding by assigning disjoint key ranges to different manager instances for larger-scale deployments.

4.3 Storage layer

PolarKV employs a storage layer to provide large-capacity backing for the disaggregated KV cache. Instead of deploying a distributed file system across multiple storage nodes to construct a global namespace, *PolarKV* breaks the "tier-as-a-whole" abstraction and adopts a shared-nothing, shard-paired design at the storage layer. Concretely, it partitions storage capacity into per-SN storage shards and pairs each shard with a corresponding SN, localizing tiering operations to the SN. As illustrated in Figure 8, each storage shard is co-located with its paired SN and is responsible for offloading in-memory KV blocks from that SN to its local cloud block storage, and reloading them on demand. This design eliminates global coordination overhead and keeps most tiering traffic off the network-critical path. Because KV allocation is balanced across SNs, the resulting storage footprint is naturally balanced as well, avoiding cross-node synchronization for storage placement.

Disk server. Data movement between memory and storage is handled by a lightweight disk server (DS). In *PolarKV*, each DS is co-located with its corresponding SN on the same VM, as shown in Figure 8. The DS shares the memory space of its SN, allowing it to directly access in-memory KV data during offloading without additional network hops or serialization overhead.

To fully leverage cloud infrastructure, *PolarKV* uses cloud block storage as the storage medium. As summarized in Table 1, cloud block storage is typically offered in multiple performance tiers with different bandwidth, capacity, and pricing characteristics. Given that KV caching is sensitive to both bandwidth and capacity, *PolarKV* favors storage options with low cost per GB and low cost per provisioned bandwidth (e.g., per GB/s), and then aggregates bandwidth across devices. In our implementation, we construct a RAID-0 array from multiple cost-efficient, low-bandwidth Alibaba ESSDs (PL0) to achieve high aggregate throughput. A native file system (e.g., ext4) is mounted atop the array, enabling the DS to access the ESSDs through standard file interfaces with minimal software overhead.

The DS does not maintain additional metadata; instead, it directly uses the KV key, a fixed-size hash value typically around 64 bytes, as the filename, simplifying lookup and management. This does not create one file per token: *PolarKV* stores KV cache at the chunk granularity, where each chunk contains multiple tokens and typically corresponds to tens to hundreds of MB of KV data for modern LLMs. When the DS receives an offload request from the KV manager, the request includes the key, which the DS uses as the filename when writing the data to storage. Similarly, when reloading data, the DS reads the corresponding file directly by key name. Since each storage node uses multiple ESSDs organized in RAID-0, a write for a single KV cache entry can span multiple ESSDs, fully utilizing the underlying bandwidth. Hence, further chunking of KV items into smaller files is unnecessary. Communication between the DS and KV manager is also implemented via RDMA-based RPC, and each DS contains an RPC handler to process requests for offload, load, and deletion operations.

Workflow. The KV manager orchestrates data movement between memory and storage through the DS. It periodically scans the LRU list and issues offload requests for recently unused KV entries. For each selected KV item, the KV manager sends a request to the DS

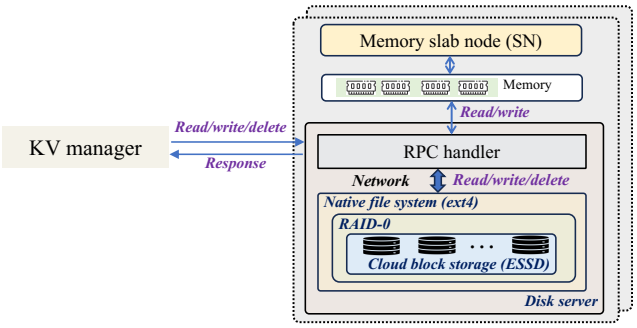


Figure 8: The architecture of the storage layer

corresponding to the SN where the item currently resides. The request includes the key, memory address, and size. Upon receiving the request, the DS launches a background thread to asynchronously write the specified memory region to storage. Once the operation completes, the DS notifies the KV manager, which updates the metadata to mark the entry as stored on disk.

When a read request targets a KV item that resides on storage rather than in memory, the KV manager first allocates a new memory region from the SN paired with the corresponding DS. It then sends a load request containing the key and target address to the DS, which reads the data from storage back into memory and confirms completion. Meanwhile, the KV manager monitors the storage utilization on each node and instructs the DS to evict the least recently used on-storage KV entries once storage utilization reaches a predefined threshold.

4.4 High availability

In production cloud environments, performance alone is insufficient. As an independent KV cache service on the critical path of LLM inference, *PolarKV* must tolerate failures, scale dynamically, and remain continuously available under node churn and partial outages. These requirements motivate high availability (HA) as a first-class design concern.

Failure model and design philosophy. *PolarKV* operates in a cloud environment where VM restarts and transient network disruptions may occur. Rather than enforcing strong availability guarantees through synchronous replication or coordination-heavy protocols, *PolarKV* adopts a pragmatic HA model aligned with the role of KV caching in LLM inference. Specifically, KV cache entries are treated as performance optimizations rather than correctness-critical state. If KV data becomes unavailable, inference correctness is unaffected: the LLM engine can always recompute the corresponding attention states. This design prioritizes forward progress and graceful degradation over strict cache preservation, achieving high availability with minimal operational overhead.

Client-side failure handling. When the KV manager or a memory/storage node becomes temporarily unavailable, the client library detects the failure and transparently reports a cache miss to the inference engine. If the failure occurs while a client RPC is in flight, the unfinished operation is also treated as a cache miss. The inference engine then recomputes the required KV states as part of normal execution. Meanwhile, the client continues to retry

connections to the KV service and automatically resumes KV cache usage once the service recovers. This approach ensures that failures in *PolarKV* never block inference progress; they only reduce cache effectiveness temporarily.

Metadata persistence and manager recovery. To prevent the KV manager from becoming a single point of failure, *PolarKV* persists KV metadata in an external cloud database. This metadata includes mappings between logical KV entries and their physical locations in distributed memory or storage. Upon a KV manager crash or restart, the metadata is recovered from persistent storage, allowing the manager to reconstruct the global KV cache metadata and resume serving requests. Because the underlying KV data stored on memory or storage nodes remains intact, recovery does not require rebuilding cached states from scratch.

Memory-node failures. Failures of individual memory nodes are handled independently. When a memory node becomes unavailable, the KV manager invalidates only the KV entries residing on that node. KV entries stored on surviving nodes remain accessible and continue serving inference requests without disruption. This design confines failures to a limited subset of cached data and avoids cascading impact across the cluster. Since KV entries are recomputable, lost entries simply result in temporary cache misses rather than service interruption.

Discussion. Together, these mechanisms ensure that *PolarKV* remains continuously available under common cloud failure scenarios while bounding the complexity and overhead of HA mechanisms. By treating KV caching as an optional acceleration layer rather than a correctness-critical component, *PolarKV* achieves a practical balance between availability, simplicity, and performance—well aligned with the operational characteristics of large-scale cloud-based LLM serving.

5 EVALUATION

5.1 Experimental setup

Setup. In this section, we evaluate *PolarKV* within two mainstream inference engines, *vLLM* and *SGLang*, across multiple model families (DeepSeek, Qwen, and GLM). Since they provide flexible interfaces for external KV cache stores, *PolarKV* can be easily integrated with them via a lightweight client-side library. All experiments are conducted on the Alibaba public cloud, including the GPU, memory and storage resources. For the GPU setup, each GPU instance is equipped with eight H20 96GB GPUs.

Baselines. We compare *PolarKV* with two representative external KV cache solutions: Mooncake, which represents memory-centric KV caching, and a DFS-based KV cache built on 3FS, which represents storage-centric KV caching. Although 3FS is a general-purpose distributed file system rather than a KV-cache-specific system, it is supported as a KV cache backend by mainstream inference frameworks such as SGLang. Since we are not aware of an open-source external KV cache system with the same hybrid memory–storage design point as *PolarKV*, we use these baselines to compare three practical design choices for external KV cache management: storage-centric caching, memory-centric caching, and *PolarKV*'s hybrid memory–storage caching.

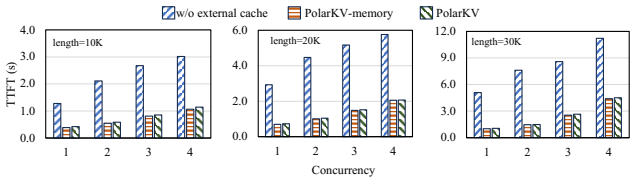


Figure 9: Overall performance of *PolarKV*

Workloads. We first use a synthetic microbenchmark to study the improvement of *PolarKV* under the upper-bound benefits of KV reuse. We then evaluate *PolarKV* on two public benchmarks, LooGLE [25] and SCBench [28], which capture realistic long-context dependency and heterogeneous inference workloads, respectively. Finally, we report a production case study from Alibaba Cloud, where *PolarKV* is deployed under real online traffic to quantify its impact on end-to-end latency and GPU provisioning efficiency.

5.2 Microbenchmark performance

We begin by evaluating the efficiency of *PolarKV* using a microbenchmark on the DeepSeek-R1 model with *vLLM*, varying prompt lengths from 10K to 30K tokens. This experiment measures the upper-bound benefits of KV cache reuse by focusing on time-to-first-token (TTFT), which is dominated by the prefill phase. We keep concurrency up to 4 to avoid TTFT being dominated by inference-engine queueing. In this evaluation, each session first issues a warm-up request consisting of a long prompt (10K–30K tokens). After warm-up, they concurrently issue a second request whose input consists of the same prefix used by that session plus a short continuation, producing a 200-token output. Thus, prefix reuse occurs within each session, allowing us to isolate and measure the benefits of KV cache reuse under concurrent workloads.

To isolate the impact of *PolarKV*'s storage hierarchy, we evaluate two *PolarKV* variants. The default *PolarKV* setup uses the hybrid memory–storage hierarchy and is deployed with 30 memory nodes (shards), providing 480 GB of aggregate memory. Each node is provisioned with about 1.2 TB of cloud block storage, mainly to provide sufficient aggregate bandwidth rather than to match the working-set capacity. Although this bandwidth-driven provisioning introduces extra storage capacity, cloud memory is about 40× more expensive per GB than storage in our setting. Therefore, the hybrid setup remains much more cost-effective than the *PolarKV*-memory setup, which uses 3 TB of distributed memory without the storage tier. This comparison allows us to quantify how close the practical hybrid design comes to the ideal memory-only ceiling under KV-intensive, concurrent workloads.

As shown in Figure 9, both *PolarKV*-memory and *PolarKV* dramatically outperform native *vLLM* without external KV caching, reducing TTFT by up to 80% across all prompt lengths. The speedup comes from eliminating redundant attention computation: loading a 20K-token KV cache from *PolarKV* takes only ~0.5 seconds, compared with ~3 seconds for recomputation. Notably, *PolarKV* achieves TTFT that is nearly identical to *PolarKV*-memory. This indicates that, in our setting, the hybrid system already operates close to the memory-only performance ceiling. The reason is that KV loading is primarily limited by network bandwidth rather than by the peak bandwidth of DRAM or storage media. The GPU host

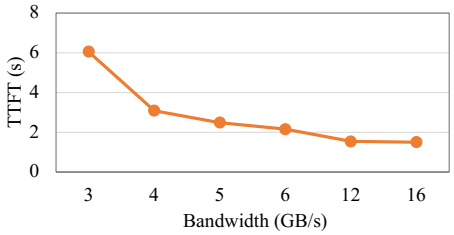


Figure 10: Impact of KV cache bandwidth on TTFT

has two 100-Gbps NICs (200-Gbps aggregate), and the storage tier is provisioned with sufficient aggregate throughput such that the end-to-end data path saturates at the network. As a result, the hybrid *PolarKV* design can match the memory-only configuration for KV reuse in this microbenchmark. This experiment highlights an important insight for cloud KV cache design: once network bandwidth dominates, a carefully provisioned hybrid hierarchy can deliver memory-like KV loading latency in practice.

5.3 Bandwidth sensitivity analysis

To further understand the impact of storage bandwidth on KV cache performance, we vary the provisioned bandwidth of *PolarKV* and measure the resulting TTFT under the microbenchmark configuration (20K-token input). As shown in Figure 10, TTFT decreases sharply as bandwidth increases from 3 GB/s to 12 GB/s. This result clearly indicates that KV loading is strongly bandwidth-bound in low-bandwidth regimes, and insufficient storage bandwidth directly inflates prefill latency. Beyond 12 GB/s, further bandwidth increases yield only marginal improvements. At this point, storage bandwidth is no longer the dominant bottleneck, and overall latency becomes constrained by GPU-side network limits and other overheads. These results reinforce our central observation that bandwidth, rather than raw storage capacity, is the primary determinant of KV cache efficiency. Proper bandwidth provisioning is therefore essential for achieving predictable and low-latency LLM inference.

5.4 Performance under realistic workloads

While the microbenchmark isolates the upper-bound benefits of KV reuse under controlled conditions, we next evaluate *PolarKV* on two public benchmarks, LooGLE [25] and SCBench [28], using *SGLang* with the Qwen3-235B-A22B model. As a baseline, *SGLang* uses a local prefix cache backed by the remaining GPU HBM plus an extra 50 GB of host memory per GPU. In this evaluation, we further compare *PolarKV* with two representative external KV cache solutions: Mooncake, which provides a remote-memory-based KV cache, and a DFS-based KV cache built on 3FS.

Configuration. To ensure a fair comparison, we fix a total cloud-resource budget for constructing each KV cache pool. In all configurations, memory capacity/bandwidth is provided by VMs, while storage capacity/bandwidth is provided by cloud block storage. As shown in Figure 10, achieving good performance requires approximately 12 GB/s KV loading bandwidth. Therefore, we provision each system to provide at least approximately 12 GB/s of aggregate bandwidth under a comparable cloud-resource budget. As shown in Table 3, for 3FS we stripe 67 320-GB PL0 ESSD volumes to reach 12 GB/s aggregate bandwidth (about 21 TB total capacity). Since it

Table 3: Configurations of 3FS, Mooncake and *PolarKV* under a fixed cloud-resource budget

	Memory Capacity	Storage Capacity	Bandwidth	Total cost (per month)
3FS	–	21.4 TB	12.09 GB/s	18,352.30 CNY
Mooncake	928 GB	–	17.98 GB/s	18,413.84 CNY
PolarKV	156 GB	21.4 TB	12.09 GB/s	18,352.30 CNY

The pricing details of memory and ESSD are summarized in Table 1 and Table 2.

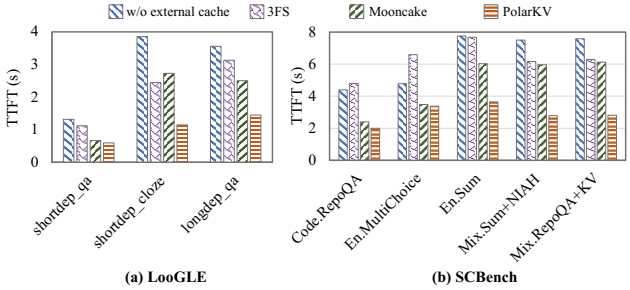
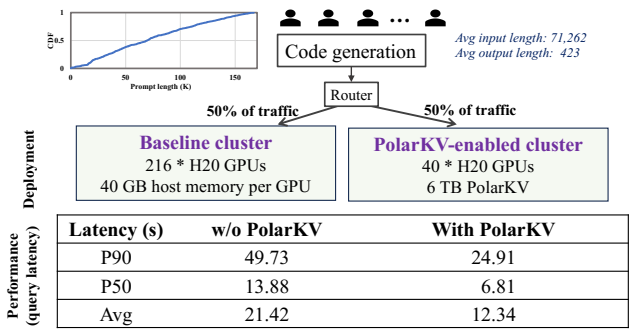


Figure 11: The performance on LooGLE and SCBench datasets

requires dedicated VMs to host these ESSDs, and the VMs' aggregate network bandwidth must match the ESSDs' throughput, this configuration requires 39 VMs. We then allocate the same budget to provision resources for Mooncake. Since Mooncake's remote memory primarily serves as a capacity resource, we choose a VM type with the lowest cost per GB (e.g., *r9i.large*) to minimize capacity cost. Under the fixed budget, this results in 58 VMs for Mooncake, providing an aggregate memory capacity of 928 GB and an aggregate network bandwidth of 17.98 GB/s. In contrast, for 3FS and *PolarKV*, the hosting VMs are provisioned primarily for network bandwidth, so we choose a VM type with low cost per provisioned bandwidth (e.g., *c8i.large*, which offers the lowest cost per GB/s of bandwidth).

Performance. Figure 11 shows TTFT results on the LooGLE and SCBench benchmarks. Due to space limitations, we evaluate *PolarKV* on a representative subset of SCBench tasks and the remaining tasks exhibit similar trends. Across all evaluated datasets, *PolarKV* consistently and significantly outperforms the other configurations, including Mooncake and 3FS.

For `shortdep_qa`, the total KV footprint is smaller than in the other workloads, leading to a higher local prefix-cache hit rate. As a result, the baseline *SGLang* local cache already performs well, leaving less room for additional gains from *PolarKV*. In this workload, Mooncake's memory capacity can hold most, or even all, reusable KV entries, so it achieves TTFT close to *PolarKV*, with *PolarKV* reducing TTFT by only 11%. In contrast, 3FS remains slower because it cannot effectively leverage the host memory on storage-hosting VMs and introduces distributed-file-system overheads such as metadata management and replication. Across the evaluated workloads, *PolarKV* reduces TTFT by 47%–53% over 3FS. For `shortdep_cloze` and `longdep_qa`, the KV footprint is substantially larger, making Mooncake's capacity limitation more significant. Mooncake even performs worse than 3FS in `shortdep_cloze`. However, *PolarKV* continues to reduce TTFT by 41%–58% over both 3FS and Mooncake on these large-footprint workloads, benefiting from higher effective cache capacity and a more efficient tiering design.



Latency (s)	w/o PolarKV	With PolarKV
P90	49.73	24.91
P50	13.88	6.81
Avg	21.42	12.34

Figure 12: Production results during *PolarKV*'s partial (canary) rollout on an online code-generation workload

On SCBench, we observe trends consistent with LooGLE. In most cases, 3FS slightly reduces TTFT compared to the no-external-cache baseline, but remains worse than Mooncake and *PolarKV* due to its higher system overhead. Mooncake further improves over 3FS but still falls behind *PolarKV* because of its limited cache capacity. Overall, *PolarKV* reduces TTFT by 47%–58% and 3%–58% relative to 3FS and Mooncake, respectively, across these datasets.

5.5 Production case study

We further evaluate *PolarKV* in a real-world production deployment on Alibaba Cloud: a coding-agent service operated by a customer in the autonomous driving domain. The service runs *SGLang* as the inference engine and serves the GLM-4.7 model. All measurements reported in this section are collected online during normal production operation, rather than through offline trace replay.

Application characteristics. This application exemplifies agent-style interactive LLM workloads, such as coding assistants and development agents. Within each session, the agent incrementally extends a growing context and repeatedly issues follow-up requests that reuse the entire accumulated prefix, making intra-session KV reuse a dominant performance factor. In this deployment, the per-request prompt length ranges from 192 to 160K tokens. The average input length observed by the model reaches 71K tokens, while the average output length is 423 tokens. The distribution of prompt lengths is shown in the top-left panel of Figure 12.

Baseline deployment and canary rollout. Prior to adopting *PolarKV*, the customer operated a homogeneous inference cluster with 256 H20 GPUs, where KV states were cached using a conventional hybrid design: remaining GPU HBM was used as the primary KV cache, and each GPU was further backed by 40 GB of local host-memory KV caching. In total, this configuration used approximately 10 TB of host memory cluster-wide for KV caching. However, the host-memory KV cache is node-local and cannot be shared across nodes, resulting in a relatively low KV cache hit rate.

To evaluate *PolarKV* in production, the customer first conducted a canary rollout. A small *PolarKV*-backed cluster consisting of 40 H20 GPUs connected to 6 TB of storage was introduced. During this phase, 50% of the incoming traffic was migrated to the *PolarKV*-enabled cluster, while the remaining 50% continued to be served by the remaining 216-GPU cluster without external KV caching.

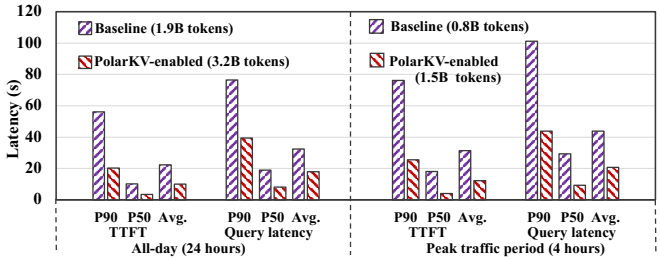


Figure 13: Performance before (256 H20 GPUs) and after (160 H20 GPUs) full migration to *PolarKV*-enabled cluster

Importantly, this rollout was performed under non-saturated operating conditions, with sufficient capacity headroom preserved on both clusters to ensure service stability. As a result, the comparison focuses on per-request latency rather than maximum throughput.

Figure 12 reports the end-to-end latency comparison between the two configurations. Despite serving the same fraction of traffic with less than one-fifth of the GPUs, the *PolarKV*-enabled cluster achieves significantly lower latency. Specifically, *PolarKV* reduces median (P50) latency from 13.88 s to 6.81 s and tail latency (P90) from 49.73 s to 24.91 s, representing nearly a 2× improvement. The average latency is also reduced by 42%. This improvement is primarily driven by higher KV cache reuse. It stems not only from the larger KV cache capacity provided by *PolarKV*, but also from the cross-node KV sharing it enables. In the baseline deployment, the effective KV cache hit rate was approximately 30%, since KV states were confined to node-local GPU and host memory. With *PolarKV*, the KV cache hit rate increases to about 85%, allowing most requests to bypass expensive prefill recomputation and instead load KV states directly from the shared cache.

Full migration. After the canary rollout validated *PolarKV*'s stability and effectiveness, the customer proceeded with a full traffic migration. During this period, the production workload intensity increased substantially, with the total number of served tokens growing by approximately 68%. To accommodate the increased demand, the service was consolidated from the original 256-GPU cluster to a *PolarKV*-backed deployment with 160 H20 GPUs and 20 TB of storage, while still serving full production traffic.

Figure 13 compares the end-to-end request latency distributions served by the two clusters. Despite using 37.5% fewer GPU resources and serving 68% higher traffic, the *PolarKV*-enabled deployment delivers markedly better performance across both all-day traffic and peak period. Over the full day, *PolarKV* reduces average TTFT by 55% and average end-to-end latency by 45%, while during a four-hour peak traffic period, *PolarKV* reduces average TTFT by 61% and average end-to-end latency by 53%. Beyond the mean, *PolarKV* also significantly reduces median latency and tail latency (P90), by 48–78%. More importantly, these latency improvements are achieved while the system processes a significantly higher volume of requests and tokens. As shown in Figure 13, the total number of served tokens increases from 1.9B to 3.2B over the full day, and from 0.8B to 1.5B during peak period, reflecting a substantial increase in effective throughput under real production load.

These results indicate that *PolarKV* not only reduces per-request latency but also fundamentally improves cluster efficiency: by eliminating redundant prefill computation through external KV reuse,

a smaller GPU cluster can sustain higher overall load than a much larger baseline deployment. To our knowledge, this workload provides one of the first public characterizations of KV cache behavior in a production coding-agent deployment.

6 RELATED WORK

Prefix KV cache. Recent LLM serving systems exploit prefix caching to reuse precomputed attention states (KV) across requests that share a common prompt prefix, reducing redundant prefill compute and KV memory. vLLM [23] and SGLang [55] implement such prefix caching primarily within GPU HBM, using mechanisms such as PagedAttention and RadixAttention to improve decoding efficiency. In practice, these built-in prefix caches are confined to local GPU HBM or host memory, whose capacity is typically limited to tens of gigabytes per GPU. Moreover, cached KV states are not shared across GPUs or inference nodes, which can significantly reduce cache effectiveness when LLM services scale to large multi-GPU or multi-node deployments. To extend KV cache capacity beyond GPU memory, several systems explore host and remote memory. LMCache [11] utilizes host DRAM as a KV cache and exposes a general interface that allows inference engines to integrate external KV stores. However, KV states in LMCache remain local to each inference node and cannot be shared across machines, limiting cache effectiveness at scale. Mooncake [36] further pools DRAM across multiple hosts to provide a shared remote-memory-based KV cache, improving cross-node reuse but relying exclusively on disaggregated memory. Other efforts leverage distributed storage systems, such as using 3FS [39], to build SSD-backed KV caches. These designs offer lower capacity cost compared to memory-based approaches but incur higher access latency and metadata overhead, resulting in a performance–cost trade-off.

Overall, existing systems place KV caches in GPU memory, host memory, remote memory, or distributed storage, but treat each resource in isolation. They do not address the combined bandwidth, capacity, and operational constraints of cloud environments, nor provide an integrated design that jointly leverages memory and storage for scalable, cost-efficient KV caching on cloud.

Disaggregated memory systems. A large body of systems research has explored disaggregated and remote memory as a way to expand memory capacity beyond a single machine [5, 16, 21, 35, 37, 48]. They primarily aim to provide general-purpose memory abstraction or extend memory for specific applications, focusing on transparency, low latency, fault tolerance, and OS integration. In contrast, *PolarKV* targets a workload-specific KV cache for LLM inference, where access patterns, data lifetime, and performance requirements are explicitly known. Rather than exposing a generic memory abstraction, *PolarKV* uses disaggregated memory as a bandwidth tier within a hierarchical KV cache and jointly optimizes data placement with cloud storage. This workload-aware design allows *PolarKV* to avoid many of the performance and cost pitfalls that arise when applying general-purpose remote memory systems directly to LLM serving.

CXL-based memory pooling [20, 24, 48] is a promising direction and may offer different bandwidth–cost trade-offs from RDMA-based remote memory. However, current CXL pooling remains limited in public-cloud deployment because its bandwidth is tied to PCIe links and pooling domains are often switch-local or rack-scale. Thus, we view CXL as a future substrate, while *PolarKV* targets currently deployable cloud infrastructure using RDMA-accessible disaggregated memory and cloud block storage.

Hierarchical caching and tiered storage. Hierarchical caching and tiered storage have long been studied in storage and database systems, where fast but expensive memory is combined with slower, cheaper storage to balance performance and cost. Examples include DRAM–SSD buffer caches in databases [7, 15], multi-tier key–value stores [9, 17, 33], and storage systems that transparently migrate data across memory and persistent tiers [56]. Most existing hierarchical designs implicitly assume that data movement across tiers is relatively inexpensive, often occurring within a single node or over high-bandwidth, low-latency interconnects. However, this assumption does not hold for cloud-based LLM serving. As shown in our analysis, cross-node data movement in the cloud is constrained by per-instance network bandwidth and tightly coupled to resource provisioning and cost. For KV cache workloads, where multi-gigabyte data transfers are common, naively applying classical hierarchical designs can lead to severe performance degradation and inflated operational cost. *PolarKV* revisits hierarchical caching from a cloud-native perspective. By co-locating memory and storage within the same VM and eliminating cross-node tiering traffic, *PolarKV* preserves the semantic benefits of a memory–storage hierarchy while avoiding the cross-node network bottlenecks.

LLM serving systems and inference optimization. To reduce the memory footprint and data movement cost of KV caches, a growing body of work explores KV compaction [53], compression [6, 19, 29, 30], and sparsification techniques [27, 45, 46, 50, 54]. These approaches exploit redundancy or structure in attention states, or selectively retain only the most influential tokens, to reduce KV size, improve cache efficiency, and mitigate compute and memory overhead during inference. These techniques primarily operate at the algorithmic level within a single inference instance, trading off attention completeness for reduced resource usage. In contrast, *PolarKV* addresses a complementary systems-level challenge: how KV states are stored, shared, and reused across requests and across inference nodes in cloud environments. Rather than altering KV representations, *PolarKV* focuses on scalable, cloud-native KV management, and its modular cache and metadata design allows existing compaction or sparsification techniques to be transparently integrated within either the memory or storage tiers.

7 CONCLUSION

KV caching is a critical determinant of performance and cost in cloud-based LLM inference, yet existing designs fail to account for the bandwidth, capacity, and operational constraints of cloud platforms. In this paper, we present *PolarKV*, a cloud-native distributed KV cache system that unifies disaggregated memory and cloud block storage through a shard-paired, co-located hierarchical design, treating memory as a bandwidth tier and storage as a capacity tier. By breaking the "tier-as-a-whole" abstraction and turning tiering into local shard-to-shard movement, *PolarKV* removes network traffic from the critical tiering path. *PolarKV* has been deployed as an independent KV cache service on Alibaba Cloud, integrating with production inference engines.

REFERENCES

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

[2] Wei An, Xiao Bi, Guanting Chen, Shanhuang Chen, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Wenjun Gao, Kang Guan, et al. 2024. Fire-Flyer AI-HPC: A Cost-effective Software-Hardware Co-design for Deep Learning. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–23.

[3] AWS. 2025. Amazon Elastic Block Store. https://aws.amazon.com/ebs/.

[4] Azure. 2025. Azure Ultra Disks. https://learn.microsoft.com/en-us/azure/virtual-machines/disks-enable-ultra-ssd.

[5] Daniel S Berger, Yuhong Zhong, Fiodar Kazhamiaka, Pantea Zardoshti, Shuwei Teng, Mark D Hill, and Rodrigo Fonseca. 2025. Octopus: Scalable Low-Cost CXL Memory Pooling. *arXiv preprint arXiv:2501.09020* (2025).

[6] Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, et al. 2024. PyramidKV: Dynamic KV Cache Compression based on Pyramidal Information Funneling. *arXiv preprint arXiv:2406.02069* (2024).

[7] Mustafa Canim, George A Mihaila, Bishwaranjan Bhattacharjee, Kenneth A Ross, and Christian A Lang. 2010. SSD Bufferpool Extensions for Database Systems. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1435–1446.

[8] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 International Conference on Management of Data*. 2477–2489.

[9] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 17–32.

[10] Weijian Chen, Shuibing He, Haoyang Qu, Ruidong Zhang, Siling Yang, Ping Chen, Yi Zheng, Baoxing Huai, and Gang Chen. 2025. IMPRESS: An Importance-Informed Multi-Tier Prefix KV Storage System for Large Language Model Inference. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 187–201.

[11] Yihua Cheng, Yuhan Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. 2025. LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference. *arXiv preprint arXiv:2510.09665* (2025).

[12] Alibaba Cloud. 2025. Alibaba Cloud ESSD. https://www.alibabacloud.com/help/en/ecs/user-guide/essds.

[13] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FLASHATTENTION: Fast and Memory-Efficient Exact Attention with IO-Awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.

[14] Yangshen Deng, Zhengxin You, Long Xiang, Qilong Li, Peiqi Yuan, Zhaoyang Hong, Yitao Zheng, Wanting Li, Runzhong Li, Haotian Liu, et al. 2025. AlayaDB: The Data Foundation for Efficient and Effective Long-context LLM Inference. In *Companion of the 2025 International Conference on Management of Data*. 364–377.

[15] Jaeyoung Do, Donghui Zhang, Jignesh M Patel, David J DeWitt, Jeffrey F Naughton, and Alan Halverson. 2011. Turbocharging DBMS Buffer Pool Using SSDs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 1113–1124.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 401–414.

[17] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 65–78.

[18] Yunhua Fang, Rui Xie, Asad Ul Haq, Linsen Ma, Kaoutar El Maghraoui, Naigang Wang, Meng Wang, Liu Liu, and Tong Zhang. 2025. Accelerating LLM Inference via Dynamic KV Cache Placement in Heterogeneous Memory System. *IEEE Computer Architecture Letters* (2025).

[19] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2023. Model Tells You What to Discard: Adaptive KV Cache Compression for LLMs. *arXiv preprint arXiv:2310.01801* (2023).

[20] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory Pooling with CXL. *IEEE Micro* 43, 2 (2023), 48–57.

[21] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[22] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. 2024. KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization. *Advances in Neural Information Processing Systems* 37 (2024), 1270–1303.

[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th symposium on operating systems principles*. 611–626.

[24] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: CXL-based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 574–587.

[25] Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. 2024. LooGLE: Can Long-Context Language Models Understand Long Contexts?. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 16304–16333.

[26] Yuhang Li, Rong Gu, Chengying Huan, Zhibin Wang, Renjie Yao, Chen Tian, and Guihai Chen. 2025. HotPrefix: Hotness-Aware KV Cache Scheduling for Efficient Prefix Sharing in LLM Inference Systems. *Proceedings of the ACM on Management of Data* 3, 4 (2025), 1–27.

[27] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. SnapKV: LLM Knows What You Are Looking for before Generation. *Advances in Neural Information Processing Systems* 37 (2024), 22947–22970.

[28] Yucheng Li, Huiqiang Jiang, Qianhui Wu, Xufang Luo, Surin Ahn, Chengruidong Zhang, Amir H Abdi, Dongsheng Li, Jianfeng Gao, Yuqing Yang, et al. 2024. SCBENCH: A KV CACHE-CENTRIC ANALYSIS OF LONG-CONTEXT METHODS. *arXiv preprint arXiv:2412.10319* (2024).

[29] Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Gholamreza Haffari, and Bohan Zhuang. 2024. MiniCache: KV Cache Compression in Depth Dimension for Large Language Models. *Advances in Neural Information Processing Systems* 37 (2024), 139997–140031.

[30] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 38–56.

[31] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024. KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache. *arXiv preprint arXiv:2402.02750* (2024).

[32] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. 1970. Evaluation Techniques for Storage Hierarchies. *IBM Systems journal* 9, 2 (1970), 78–117.

[33] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. 2021. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the ACM SIGOPS 28th symposium on operating systems principles*. 243–262.

[34] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2025. A Comprehensive Overview of Large Language Models. *ACM Transactions on Intelligent Systems and Technology* 16, 5 (2025), 1–72.

[35] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. 2015. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 1–55.

[36] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. MOONCAKE: Trading More Storage for Less Computation–A KVCache-centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 155–170.

[37] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 69–87.

[38] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *International Conference on Machine Learning*. PMLR, 31094–31116.

[39] DeepSeek Team. 2024. Fire-Flyer File System. https://github.com/deepseek-ai/3FS.

[40] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530* (2024).

[41] The AIBrix Team, Jiaxin Shan, Varun Gupta, Le Xu, Haiyang Shi, Jingyuan Zhang, Ning Wang, Linhui Xu, Rong Kang, Tongping Liu, et al. 2025. AIBrix: Towards Scalable, Cost-Effective Large Language Model Inference Infrastructure. *arXiv preprint arXiv:2504.03648* (2025).

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *Advances in neural information processing systems* 30 (2017).

[43] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. 2025. KVCache Cache in the Wild: Characterizing and Optimizing KVCache Cache at a Large Cloud

Provider. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '25)*. USENIX Association, USA, Article 28, 18 pages.

[44] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. 2019. Autoscaling Tiered Cloud Storage in Anna. *Proc. VLDB Endow.* 12, 6 (2019), 624–638.

[45] Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. 2024. InfLLM: Training-Free Long-Context Extrapolation for LLMs with an Efficient Context Memory. *Advances in Neural Information Processing Systems* 37 (2024), 119638–119661.

[46] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient Streaming Language Models with Attention Sinks. *arXiv preprint arXiv:2309.17453* (2023).

[47] Dongsheng Yang, Austin Li, Kai Li, and Wyatt Lloyd. 2025. Learned Prefix Caching for Efficient LLM Inference. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

[48] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Gerry Fan, Yang Kong, Bo Wang, Jing Fang, Yuhui Wang, Tao Huang, et al. 2025. Unlocking the Potential of CXL for Disaggregated Memory in Cloud-native Databases. In *Companion of the 2025 International Conference on Management of Data*. 689–702.

[49] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. 2024. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory. In *Companion of the 2024 International Conference on Management of Data*. 295–308.

[50] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*. 94–109.

[51] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629* (2022).

[52] Shukang Yin, Chaoyou Fu, Sirui Zhao, Ke Li, Xing Sun, Tong Xu, and Enhong Chen. 2024. A Survey on Multimodal Large Language Models. *National Science Review* 11, 12 (2024), nwae403.

[53] Yanqi Zhang, Yuwei Hu, Runyuan Zhao, John CS Lui, and Haibo Chen. 2025. DiffKV: Differentiated Memory Management for Large Language Models with Parallel KV Compaction. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 431–445.

[54] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2023. H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. *Advances in Neural Information Processing Systems* 36 (2023), 34661–34710.

[55] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. SGLang: Efficient Execution of Structured Language Model Programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.

[56] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 207–219.