ObjectCache: Layerwise Object-Storage Retrieval for KV Cache Reuse

Yu Zhu
ETH Zurich
Switzerland
yu.zhu@inf.ethz.ch

Aditya Dhakal
HPE Labs
United States of America
aditya.dhakal@hpe.com

Yunming Xiao
The Chinese University of Hong Kong, Shenzhen
China
yunmingxiao@cuhk.edu.cn

Dejan Milojicic
HPE Labs
United States of America
dejan.milojicic@hpe.com

Gustavo Alonso
ETH Zurich
Switzerland
alonso@inf.ethz.ch

Abstract

Prefix KV caching has become a key mechanism in LLM serving: it reduces time to first token (TTFT) by avoiding redundant computation across requests that share a prefix (i.e., the system prompt). However, the accumulated KV cache is often larger than what GPU memory and local DRAM can hold. To preserve latency, current systems keep the KV cache in remote DRAM pools, increasing serving-cluster size and cost. In this paper, we explore a different approach: storing the KV cache in S3-compatible object storage so that capacity is no longer the constraint, while minimizing the impact on TTFT. We propose ObjectCache, which co-designs the storage protocol and transfer schedule so that the storage server delivers KV cache data in the order the GPU consumes it, overlapping data transfer with compute across concurrent requests. We prototype ObjectCache on a 100 Gbps RoCE cluster with NIXL (an inference library that abstracts storage and memory), Ceph RGW (an Object Gateway for clusters), and DAOS (an open source storage system). For 64K contexts, common in today's systems, ObjectCache adds only 5.6% latency over local DRAM; for 4K contexts, where less compute is available to mask transfer, ObjectCache adds 56–75 ms over the optimal local layerwise baseline. Under shared bandwidth caps, our scheduler reduces added TTFT by 1.2–1.8x compared with equal bandwidth sharing.

1 Introduction

Modern LLM serving increasingly relies on long and reusable input contexts, including system prompts, retrieval-augmented generation, code repositories, long conversations, and agent histories [1, 18, 69, 70] (Figure 1). These workloads make the prefill phase expensive because the model must materialize key–value (KV) tensors for every token and every layer before decoding can begin. Prefix KV-cache reuse avoids recomputing the shared prefix across requests and is therefore a primary mechanism for reducing TTFT. As context lengths and reuse opportunities grow, however, the aggregate KV-cache footprint that a serving cluster must retain also grows rapidly (Appendix Figure A1).

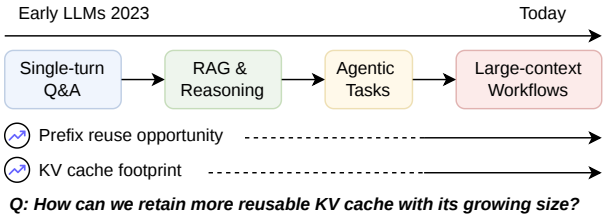


Figure 1. Long-context LLM tasks increasingly reuse long-lived prefixes, growing the aggregate KV cache footprint that a serving cluster must retain.

The challenge is that reusable KV cache is much larger than the memory capacity naturally available near GPUs. GPU memory is scarce and needed for model weights and active requests, while local DRAM only scales with the serving nodes that happen to produce or consume the cache. Recent systems therefore use remote DRAM pools or memory-centric KV-cache tiers to preserve low latency [27, 55]. Although effective, this approach makes cache capacity a provisioned part of the serving cluster: memory must be sized for the retained KV working set and remains coupled to the compute deployment, thus increasing cost and limiting how much long-lived reusable context the system can retain.

This motivates a different question: can S3-compatible object storage serve as a runtime backend for reusable KV cache? Object storage is attractive because prefix KV blocks are immutable after prefill, naturally addressable by content-derived prefix hashes, and reusable across users, sessions, and compute nodes. Unlike a remote-DRAM tier, an object store scales capacity independently of the GPU serving cluster and persists cache state beyond the lifetime of any particular worker. If object storage could be used on the serving path, prefill and decode workers would become largely stateless with respect to reusable prefixes: once KV blocks are committed, later requests could fetch them from the shared

object tier rather than returning to the node that produced them [52, 79].

The obstacle is not only transport bandwidth. Even with an RDMA data path like Nvidia cuObject [48, 49], the standard S3 abstraction remains mismatched to runtime KV-cache reuse. S3 exposes object-level operations: a request names one object, or one byte range of one object. In contrast, a prefix hit returns a set of many hash-addressed KV chunks, and the inference engine consumes the matched cache in layer order. Fetching each chunk or each layer range with independent S3 requests exposes fixed request overhead on the TTFT critical path, while fetching coarse objects sacrifices fine-grained prefix reuse. Thus, the gap between S3 and runtime KV-cache reuse is semantic: object storage moves objects, whereas inference needs prefix-selected KV blocks delivered in the order the GPU consumes them.

We present ObjectCache, a protocol-scheduling co-design that makes S3-compatible object storage suitable for runtime KV-cache reuse. On the protocol side, ObjectCache keeps KV cache stored as fine-grained, hash-addressed chunks, but extends the S3-compatible request with a compact descriptor that names the matched chunks, model layout, delivery order, and RDMA target. The storage server uses this descriptor to gather many chunk ranges, assemble one layer-major payload at a time, and deliver each layer directly to the serving node over RDMA. On the scheduling side, ObjectCache exploits the fact that layerwise transfer can overlap with per-layer GPU computation. Rather than sharing bandwidth equally or in proportion to bytes, ObjectCache allocates bandwidth according to each request's per-layer stall target, avoiding bandwidth assignments that do not further reduce TTFT.

We prototype ObjectCache on a 100 Gbps RoCE cluster using NIXL [50], Ceph RGW [5], and DAOS [12], and evaluate it with Llama 3.1 8B [44]. For 64K-token contexts, ObjectCache adds only 5.6% TTFT over an optimized local layerwise baseline. For shorter 4K-token contexts, where less compute is available to hide transfer, ObjectCache adds 56–75 ms over the local layerwise baseline. Under shared bandwidth caps, the ObjectCache scheduler reduces added TTFT by 1.2–1.8× compared with equal bandwidth sharing.

This paper makes the following contributions:

- We propose ObjectCache, a protocol-scheduling co-design for serving reusable KV cache from S3-compatible object storage.
- We design an S3-compatible protocol extension that supports prefix-selected multi-object aggregation and layerwise KV delivery while preserving fine-grained client-side prefix lookup.
- We introduce a bandwidth scheduler that allocates storage bandwidth according to per-layer compute-transfer overlap, reducing TTFT under contention.

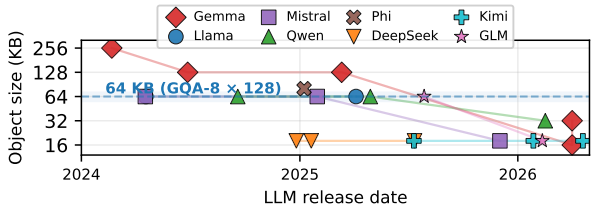


Figure 2. Per-layer KV payload for a 16-token chunk across recent open-weight LLM families. The 64 KB dashed line marks the grouped-query attention (GQA) baseline with 8 KV heads of 128 dimensions; multi-head latent attention (MLA) and smaller head counts push recent models below this threshold.

- We demonstrate on a 100 Gbps RoCE prototype that ObjectCache approaches local layerwise KV-cache performance for long-context workloads and improves TTFT under shared bandwidth limits.

2 Background and Motivation

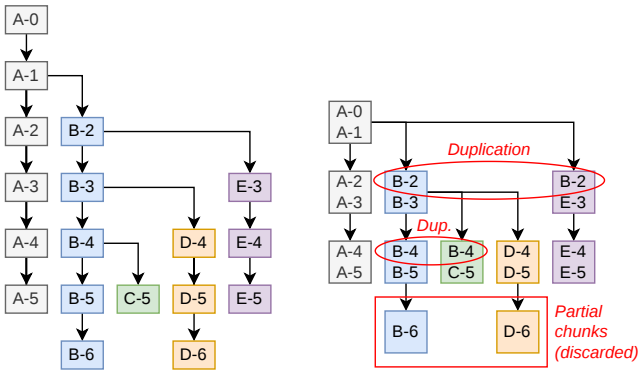
2.1 KV cache and Prefix Reuse

During the prefill phase, LLM inference processes the input prompts and materializes the per-layer key/value tensors to be used in transformer attention [11, 66]. During the decode phase, the model reuses these tensors instead of recomputing attention over the entire prefix each time by storing the context in a KV cache. To enable cross-query reuse and storage, serving systems partition the KV cache into fixed-size chunks corresponding to G consecutive tokens. For a model with L layers, n_{kv} KV heads, head dimension d, and element width p, the KV bytes per token and the per-layer bytes of a G-token chunk are:

$$\text{KV}_{\text{token}} = 2Ln_{\text{kv}}dp, \qquad S_{\text{layer,chunk}} = 2Gn_{\text{kv}}dp. \tag{1}$$

The prefix KV cache reuse relies on a radix tree or similar prefix index to find the longest cached match for a new request [21, 23, 32, 68, 73]. Each chunk can be identified by a rolling hash, $H_i = \text{Hash}(H_{i-1} \parallel \text{tokens}_i)$, that gives it a deterministic object key. Using H_i as the object key gives KV chunks the properties object storage is good at: immutable writes, content-addressed deduplication, and independent reuse across requests.

At fine chunk sizes, the per-layer KV payload becomes small enough that the fixed object-request overhead is visible even when the communication overhead is small (Figure 2 instantiates Equation 1 for current models). A 16-token chunk exposes only tens of KB per layer for many current models, and MLA-style layouts can reduce the layer slice to roughly 16 KB [35, 43]. This pressure remains even at coarser chunk sizes. For Llama 3.1 8B [44], a 256-token chunk is about 1 MB per layer; compact KV layouts, MLA-style representations, or shape-preserving compression can push



(a) Fine granularity preserves intermediate branch points.

(b) Coarse granularity merges branch points.

Figure 3. Prefix reuse under fine and coarse storage granularities. Coarse chunks reduce index depth but lose branch points where requests can diverge, forcing otherwise reusable tokens to be recomputed.

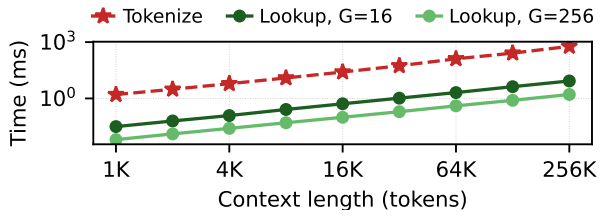


Figure 4. Prefix-hash lookup cost is small relative to tokenization, even at 16-token granularity. Fine-grained indexing is therefore not the request critical-path bottleneck.

that below the efficient object-transfer regime [6, 75]. Increasing the chunk granularity enables larger physical data transfers, but also coarsens prefix reuse. ObjectCache instead keeps the logical reuse granularity independent of the effective transfer granularity by aggregating chunks at the storage server.

Figures 3 and 4 separate two concerns that are often conflated. Figure 3 shows why fine chunks preserve more radix-tree branch points and avoid unnecessary recompute after a request diverges. Figure 4 shows that prefix-hash lookup remains small compared to tokenization, even at G=16 granularity. Fine-grained indexing does not add meaningful overhead to the request critical path. The system bottleneck is therefore not finding the prefix; it is delivering the matched KV chunks in the order the model can consume.

2.2 Serving-Path KV Access Pattern

In disaggregated serving, a prefill worker materializes KV cache for the input prompt, while a decode worker later uses the resulting KV state to generate tokens. When a new request shares a cached prefix, prefix lookup returns an ordered list of matched KV chunks that the prefill worker can reuse before computing the remaining suffix. Although these chunks are stored independently, the model consumes them layer by layer: it needs the layer 0 slice from all matched chunks before computing layer 0, then the layer 1 slice before computing layer 1, and so on. Thus, the storage-facing access pattern is an ordered multi-object, multi-range read rather than a single object read.

3 ObjectCache Design

Figure 5 shows how ObjectCache fits into a disaggregated LLM serving cluster. A central orchestrator receives a request, performs prefix matching, and assigns the remaining prefill work to a prefill node together with the matched prefix KV list. The prefill node fetches the reusable KV cache from the S3-compatible object tier, computes the remaining prompt suffix, and materializes the full KV state. The decode node can then load the full KV state and generate output tokens, while newly produced KV blocks are offloaded back to object storage for future reuse. This organization decouples prefill and decode workers from the machines that originally produced the cache, making reusable prefix state persistent and shared through the object-storage tier.

The key challenge is that this serving path needs more than ordinary object reads. ObjectCache is built around one design rule: *keep storage objects small enough for prefix reuse, but expose a serving interface that returns large, layer-ordered transfers.* The design is therefore an interface extension, not a new transport. The underlying S3-over-RDMA data path moves bytes; ObjectCache changes what the serving system can ask the object tier to do.

The resulting architecture has three roles (Figure 6). The *GPU serving node* issues S3-compatible requests and consumes KV cache data through the inference engine's layerwise interface. The *gateway* terminates the S3 control plane, parses the ObjectCache descriptor, forwards the multi-object request to the storage server, and upgrades the S3 control plane to express prefix-aware, layer-scheduled KV transfer. The *storage server* resolves chunk objects in the S3 namespace, performs range reads, assembles layer-major payloads, RDMA-writes them directly to the client buffer, and executes the layer-scheduled semantic at the object/range level. All runtime policy—whether a given request is served chunk-wise or via layerwise aggregation, and how concurrent tenants share the network cap—runs on the storage server, so the gateway and NIXL client both remain stateless with respect to scheduling decisions (Sections 3.4 and 3.6).

3.1 Required Semantics

ObjectCache requires four access semantics that no existing S3 primitive provides in combined form; this is the gap

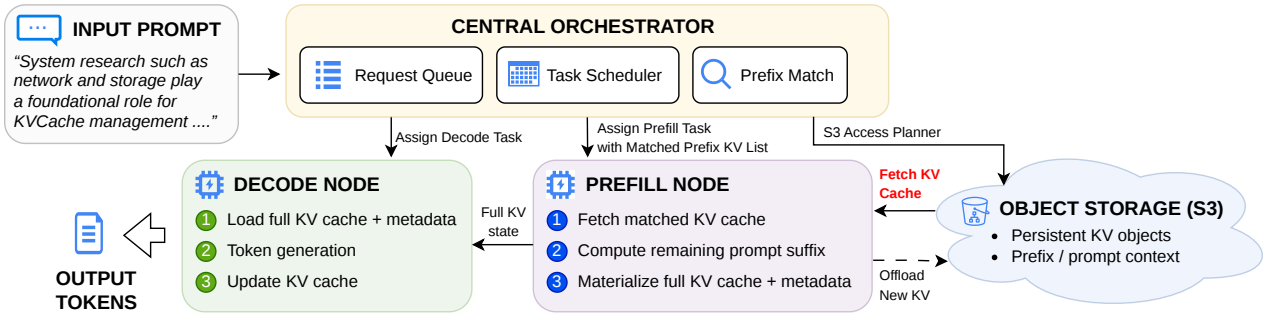


Figure 5. ObjectCache in a disaggregated cluster. Prefix KV caches are stored in an object-storage tier through an S3-compatible interface, decoupling prefill and decode workers from the machines that produced the cache. ObjectCache extends this interface so object storage can serve KV cache reuse with the granularity and layerwise delivery order expected by LLM serving systems.

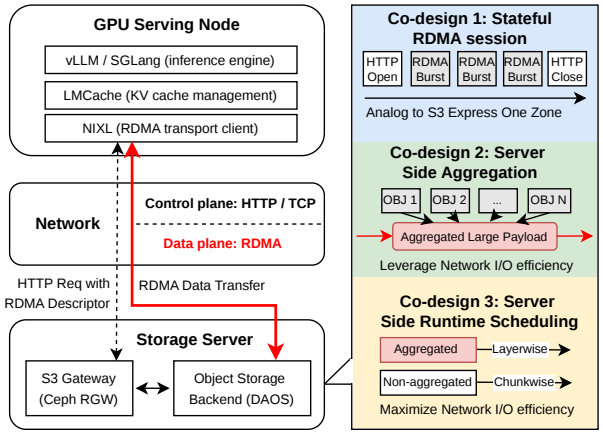


Figure 6. ObjectCache system design. HTTP preserves S3-compatible control, while the storage server aggregates matched chunk ranges into layerwise KV payloads and delivers them to the serving node over RDMA.

Table 1. ObjectCache request descriptor.

Field	Meaning
`chunk_keys`	Matched prefix chunks $[H_0, \ldots, H_{N-1}]$.
`num_layers`	L: Number of model layers.
`chunk_tokens`	G: Tokens per stored chunk.
`per_layer_chunk_bytes`	S: Slice size inside each chunk.
`delivery`	Layer-major order.
`rdma_target`	Client buffer address, key, and length.

between accelerating S3 through better transport protocols and making S3 suitable for prefix KV reuse.

(1) Multi-object batching. A prefix hit returns a list of matched KV chunks, not one object. If each chunk requires an independent S3 request, fixed request overhead dominates in the fine-grained regime that prefix caching prefers.

(2) Layerwise delivery. Inference consumes KV cache data layer by layer. Delivering complete chunks in chunk-major order forces the NIXL client to wait for the full matched prefix before starting layer 0. Delivery of layer 0 for all chunks first creates a compute window to transfer later layers.

(3) Hash-derived immutable keys. KV chunks must remain addressable by hashes so that requests reuse the same object when they share a prefix. This preserves the radix-tree storage semantics used by existing prefix caches.

(4) Bulk-transfer compatibility. Once the layer-major payload has been assembled, data should be transferred using RDMA because it provides the high-throughput transport that the higher-level aggregation semantics can use.

Standard S3 GET satisfies hash-derived keys but not batching or layerwise delivery. Range-GET can fetch one layer range, but only by issuing a request per chunk per layer. S3-over-RDMA satisfies the bulk-transfer requirement, but it inherits the same single-object request model. ObjectCache adds the missing server-side multi-object aggregation and layerwise delivery semantics while reusing the RDMA.

3.2 The ObjectCache Descriptor

ObjectCache adds an S3-compatible descriptor to a normal request. The descriptor names the matched chunk keys, the model layout, the delivery order, and the RDMA target buffer.

Table 1 lists the fields of the descriptor, which is intentionally arithmetic rather than manifest-heavy. Because every chunk in the same model deployment has the same per-layer size S, the byte range for layer ℓ in a chunk is $[\ell S,\ (\ell+1)S)$. Thus, the storage server can derive all fixed-shape layer ranges from the descriptor without a per-object manifest. Variable-size or compressed layouts can add a manifest later, but the common serving fast path keeps the descriptor compact.

3.3 Server-Side Layer Aggregation

The storage server executes the descriptor by assembling one payload per model layer. For each layer $\ell \in [0, L)$, it range-fetches $[\ell S, (\ell+1)S)$ from every matched chunk in parallel, appends the returned slices in prefix order, RDMA-writes the assembled payload to the client buffer, and notifies the serving node that the layer is ready (pseudocode in

Appendix Table A3). This notification is on the critical path of inference: it lets the GPU start layer computation without waiting for the whole prefix to arrive [36, 55].

The physical storage layout remains chunkwise even though the logical delivery layout is layerwise. In KV_L2TD format (**K**ey-**V**alue tensors stored in **L**ayer-major order, with the **2** matrices concatenated per layer, then **T**oken position, then hidden **D**imension), each immutable prefix-chunk object stores all layers sequentially, and each layer is arranged by token position and hidden dimension. Server-side aggregation changes only the readout order: one layerwise payload covers all matched chunks for a single model layer.

Keeping storage fine-grained matters because coarsening sacrifices reuse. Larger KV blocks reduce object retrieval overhead but discard reuse when two requests diverge inside a block; increasing the lookup granularity from 16 to 512 tokens can force otherwise reusable tokens to be recomputed across models and hit boundaries (Appendix Table A6 quantifies this recompute cost). The right design point is therefore not "make objects as large as possible"; the system should keep lookup granularity small and transfer granularity large enough for efficient I/O.

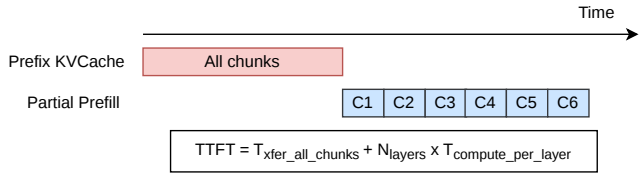
3.4 Server-Side Mode Selection

ObjectCache does not serve every cache hit using server-side layer aggregation. The KV cache is always stored in the chunkwise layout above, so the storage server chooses only the delivery mode. Each incoming descriptor names the matched chunks, from which the server derives the total matched payload $W = N \cdot L \cdot S$, where N is the number of matched chunks, L is the number of model layers, and S is the per-layer chunk size. It then applies a single size-threshold rule:

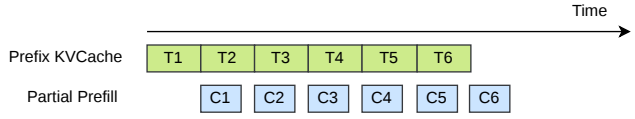
$$\text{mode}(W) = \begin{cases} \text{chunkwise} & W < \Theta, \\ \text{layerwise + aggregation} & W \geq \Theta. \end{cases} \quad (2)$$

The threshold Θ is a deployment knob: payloads below Θ use chunkwise delivery, and larger payloads use server-side aggregation with layerwise delivery. Below Θ, the layerwise schedule of Equation 3 still minimizes TTFT in theory, but the absolute saving is negligible relative to the aggregation overhead. Above Θ, the aggregation pipeline recovers its CPU and I/O-depth cost as measurable TTFT reduction while also amortizing per-object descriptor overhead across many small range reads.

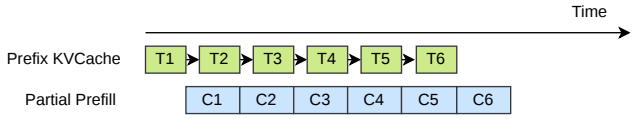
As an illustration, on our 100 Gbps prototype with Llama 3.1 8B we use $\Theta \approx 512$ MB, the payload size at which network transfer time at line rate becomes comparable to the prefill compute window. Under this rule, the 4K configurations in Section 5 fall on the chunkwise side and the 16K/64K configurations fall on the aggregated-layerwise side, matching where each mode wins in Figure 13. The appropriate value of Θ depends on the storage backend, SSD queue depth, and link rate; the dispatch rule itself does not. A full sensitivity analysis of Θ is left to future work.



(a) Chunkwise loading waits for the full cached prefix before compute can start.



(b) Transfer-bound layerwise prefetch.



(c) Compute-bound layerwise prefetch.

Figure 7. KV cache fetch scheduling. Chunkwise delivery serializes cache loading before prefill, while layerwise delivery exposes per-layer readiness so transfer can overlap compute when bandwidth is sufficient and otherwise appears as per-layer stall.

Equation 2 is also what decides which requests enter multi-tenant scheduling: chunkwise requests run independently against the object storage and consume only their fair share of object-store concurrency, while every layerwise request joins the shared bandwidth pool described next. The scheduling problem below is therefore scoped to layerwise requests under a shared bandwidth cap.

3.5 Layerwise Prefetch

Figure 7 explains why the delivery order matters. A chunkwise baseline waits for all matched chunks before compute can use any layer from the KV cache. Layerwise prefetch instead transfers layer 0 first; once layer 0 arrives, the GPU begins computing on it while the remaining layers transfer in parallel. Let X_ℓ be the transfer time for layer ℓ and C_ℓ be the compute time exposed by the miss tokens at layer ℓ. With one-layer prefetch, the TTFT model is

$$T_{\text{TTFT}} \approx X_0 + \sum_{\ell=0}^{L-2} \max(X_{\ell+1},\, C_\ell) + C_{L-1}, \quad (3)$$

where X_0 is the latency before the GPU can start (layer 0 must arrive in full), the summation captures $L-1$ stages in which transfer and compute overlap, and C_{L-1} is the final layer's compute after all transfers have finished. When $X_\ell > C_\ell$, the transfer time that exceeds the compute window appears as additional waiting time in the corresponding stage

of Equation 3. Section 5.3 evaluates this model with measured compute times and ObjectCache throughput.

3.6 Bandwidth-Aware Scheduling

Under the layerwise delivery of Section 3.5, each request's per-layer transfer can overlap with its per-layer compute window. This creates a bandwidth-scheduling problem specific to KV cache reuse: under a shared bandwidth cap B, policies proportional to matched bytes or equal-share misallocate. Matched-bytes allocation over-serves long-prefix requests whose per-layer transfer is already shorter than per-layer compute, so extra bandwidth yields no further benefit, while equal-share under-serves short-prefix requests whose compute window is small.

The key observation is that each layer of a request transfers roughly the same KV bytes and performs similar compute, so we characterise request i by its per-layer transfer size s_i and per-layer compute window c_i.[1] When the scheduler assigns bandwidth r_i to request i, the per-layer transfer time is s_i/r_i. Transfer that exceeds the compute window introduces additional stall:

$$\tau_i(r_i) \;=\; \max\!\Big(0,\; \frac{s_i}{r_i} - c_i\Big). \tag{4}$$

Each request's additional latency vanishes once r_i reaches its *zero-stall rate* $r_i^* = s_i/c_i$; any bandwidth beyond this point yields no further latency benefit. In an unconstrained setting, every request would receive r_i^*, but the total demand $\sum_i r_i^*$ may exceed the shared budget B. The scheduler must then distribute a total cut of $\sum_i r_i^* - B$ across requests. Let $\mathcal{R}$ denote the set of active layerwise requests sharing the storage link, and let $\Delta_i \geq 0$ denote the bandwidth cut applied to request i (so its allocated rate is $r_i = r_i^* - \Delta_i$). Allocating these cuts to minimize total stall leads to the *Stall-opt* problem:

$$\min_{\{\Delta_i\}} \sum_{i\in\mathcal{R}} \left(\frac{s_i}{r_i^* - \Delta_i} - c_i\right) \quad \text{s.t.} \quad \sum_{i\in\mathcal{R}} \Delta_i = \sum_{i\in\mathcal{R}} r_i^* - B,\; 0 \leq \Delta_i \leq r_i^*. \tag{5}$$

Since c_i does not depend on Δ_i, substituting $r_i = r_i^* - \Delta_i$ reduces the objective to minimising total transfer time under the bandwidth budget:

$$\min_{\{r_i\}} \sum_{i\in\mathcal{R}} \frac{s_i}{r_i} \quad \text{s.t.} \quad \sum_{i\in\mathcal{R}} r_i = B,\; 0 < r_i \leq r_i^*. \tag{6}$$

Because s_i/r_i is convex in r_i, Equation 6 is a convex program. When $\sum_i r_i^* \leq B$, every request receives its zero-stall rate and all additional TTFT vanishes; when the budget is tight, the optimal allocation admits a closed-form solution. In practice, the analytical r_i^* falls on the slope of the TTFT curve rather than at the start of the flat region (see Section 5.7). *Calibrated Stall-opt* shifts the target by a small positive offset δ to ensure the operating point lies on the plateau:

$$\hat{r}_i \;=\; r_i^* + \delta, \tag{7}$$

Calibrated Stall-opt solves the bandwidth allocation of Equation 5 with corrected targets $\hat{r}_i$ as upper bounds. When the total demand $\sum_i \hat{r}_i$ exceeds the budget B, the closed-form solution distributes the deficit to minimize total stall across concurrent requests (pseudocode in Appendix Table A4). During each scheduling epoch, a batch of active layerwise requests is admitted under a fixed total bandwidth budget, and each request receives a stable target layer-delivery rate for the duration of its KV load. If one request finishes early, its released bandwidth returns to the pool for the next scheduling epoch rather than being redistributed immediately to in-flight requests. This conservative rule makes per-request transfer times predictable, so the serving node does not need to react to unexpected bandwidth changes during an epoch.

4 Implementation

Our prototype integrates ObjectCache into a three-node serving/storage stack. The serving node runs the LLM and issues KV cache reads. The gateway terminates S3-compatible requests, parses ObjectCache descriptors, and forwards them to the storage server, which executes the multi-object aggregation. Concretely, the prototype uses NIXL as the asynchronous transfer library [50], Ceph RGW as the S3-compatible gateway [5], and DAOS as the storage server behind the gateway [12]. The implementation keeps the gateway thin: the serving node issues normal S3-compatible requests with additional headers, the gateway handles only S3 control, and the DAOS storage server performs the cross-object range reads and layer-major assembly.

4.1 S3-Compatible Paths

The prototype includes five S3-compatible paths. The first three isolate transport choices for a single object; the last two exercise the ObjectCache interface:

- **S3TCP**: standard S3 request; object sent via HTTP/TCP.
- **S3RDMA Buffer**: S3 request naming one object; the gateway stages the payload before the RDMA transfer to the NIXL client.
- **S3RDMA Direct**: S3 request naming one object; the data goes through the object-store RDMA path without gateway staging.
- **S3RDMA Batch**: S3 request naming multiple objects; one S3 request carries one HTTP header followed by an RDMA burst, thereby amortizing per-object overhead.
- **S3RDMA Agg**: S3 request naming multiple objects; the storage server assembles layer-major payloads and RDMA-writes them to the NIXL client.

[1] Both are approximately constant across layers because every layer has the same number of KV heads and the same attention/feed-forward network (FFN) structure.

Table 2. Interface-level positioning. Existing systems accelerate either the memory tier or the S3 byte path; ObjectCache adds KV-aware multi-object, layer-major aggregation inside an S3-compatible request.

Interface	Primary abstraction	S3 Namespace	Multi-object	Layer-major	Rate target
Remote KV transfer [27, 55]	Memory-resident KV blocks	No	Runtime-specific	Yes	Runtime-specific
S3-over-RDMA [45, 47–49, 65]	One S3 object or byte range	Yes	No	No	No
RDMA scatter-gather	Registered memory segments	No	No	No	No
ObjectCache	Hash-addressed KV chunk objects	Yes	Yes	Yes	Yes

We use the following names in the end-to-end TTFT evaluation:

- **Local-DRAM-CW** and **Local-DRAM-LW**: local CPU-DRAM KV cache baselines with chunkwise and layerwise delivery, respectively.
- **S3Batch-CW**: the S3-backed chunkwise batched path.
- **S3Agg-LW**: ObjectCache's server-side aggregated layerwise path.

4.2 GPU Serving Node

The serving node uses the NIXL object backend to issue S3 operations. For ordinary objects, the backend can use TCP, an RDMA-buffer mode, or the RDMA-direct path used in our microbenchmarks. For ObjectCache requests, LMCache [40] provides the matched prefix chunk keys and the model layout parameters (L, G, and S). NIXL serializes these fields into the HTTP descriptor and attaches an RDMA target token for the receive buffer. The LLM inference framework, such as vLLM [67], waits for layer-ready notifications and lets the model proceed as soon as the next layer's KV has arrived. This is the same execution pattern used by DRAM-backed layerwise KV loading, but the source is now object storage.

4.3 Gateway and Storage Server Execution

Ceph RGW is used as the gateway. It parses the ObjectCache descriptor after normal S3 request handling, then forwards the descriptor to the DAOS storage server via a server-side extension. The DAOS server maps each chunk key to the corresponding DAOS object, fetches the per-layer record extent from each chunk in parallel, concatenates the slices in chunk order, and RDMA-writes the assembled layer payload directly into the client's registered buffer.

The gateway keeps the S3 abstraction in the control path: request headers, authentication, bucket/object naming, and access control are still handled at the S3 layer. The data path is split: HTTP carries control, while RDMA carries the assembled layer payload directly between the storage server and the GPU serving node. This mirrors the S3-over-RDMA control/data-plane separation [47–49], but ObjectCache adds multi-object range selection and layer-major assembly at the storage server before the RDMA transfer.

4.4 Tracing and Rate Control

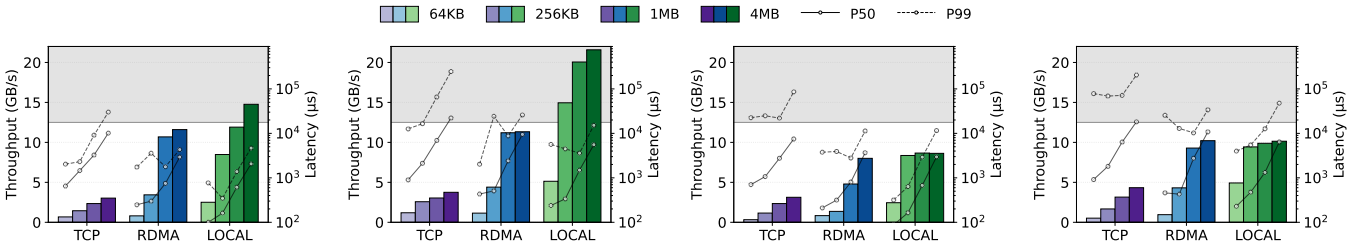
To understand request overhead, the prototype propagates request identifiers from the NIXL client through RGW into DAOS traces, allowing the evaluation to split latency into client, gateway, storage, and RDMA-bulk components. For the multi-tenant scheduling experiments, we add a layerwise pacing control that limits the effective layer delivery rate. This lets us emulate shared bandwidth caps while keeping the same layerwise access pattern used by ObjectCache.

4.5 Testbed

The evaluation uses a 100 Gbps RoCEv2 (RDMA over Converged Ethernet, v2) cluster with a GPU serving node for inference, an S3 gateway, and a DAOS storage node. The GPU is an A100 80 GB GPU (CUDA 12.8, PyTorch 2.10.0, vLLM v0.19.0, the modified LMCache v0.4.2, the modified NIXL v1.0.0); the gateway runs the modified Ceph RGW v18.2.7; the storage node runs DAOS v2.7.102 with its default UCX [64] transport, equipped with 4 KIOXIA enterprise NVMe SSDs. In this stack, vLLM is the inference runtime, LMCache manages KV cache reuse on the GPU serving node, NIXL provides the unified storage/memory transfer interface used by both the serving node and the gateway, Ceph RGW terminates S3 requests, DAOS is the underlying object-storage backend, and UCX is the RDMA transport beneath DAOS. DAOS stripes data across all 4 SSDs; however, because layerwise range reads access non-contiguous offsets within each chunk object, the per-SSD access pattern is random rather than sequential. The measurement of the same micro-benchmark for H100 80GB is in Appendix.

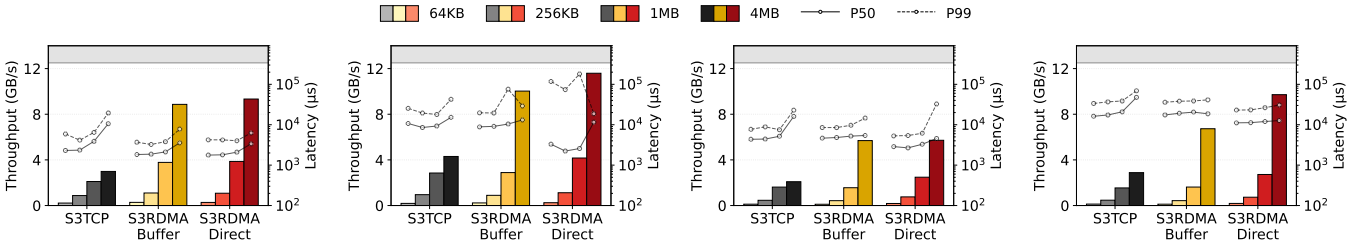
5 Evaluation

Our evaluation follows the same mechanisms introduced in the design. First, the storage and network substrate must transfer data fast enough. Second, the S3 control path must be measured separately from the RDMA data path. Third, ObjectCache's server-side aggregation must achieve high throughput for fine-grained KV chunks. Fourth, layerwise delivery must overlap with GPU compute in configurations where measured compute and transfer rates make overlap feasible. Finally, under a shared bandwidth cap, bandwidth-aware scheduling should minimize TTFT.



(a) READ, C=8 **(b)** READ, C=32 **(c)** WRITE, C=8 **(d)** WRITE, C=32

Figure 8. Raw object-storage interface baseline. The gray region marks throughput above the 100 Gbps link capacity; points in this region are limited by local storage and host execution rather than the network.



(a) S3 GET, C=8 **(b)** S3 GET, C=32 **(c)** S3 PUT, C=8 **(d)** S3 PUT, C=32

Figure 9. S3-compatible interface baseline. S3RDMA Direct preserves high large-object throughput, while S3TCP and S3RDMA Buffer expose protocol and staging bottlenecks.

5.1 Raw Storage Baseline

Setup. We measure DAOS throughput as seen by the NIXL object client without the Ceph RGW gateway, isolating the storage backend from S3 protocol overhead (Figure 8). The benchmark ranges block sizes from 64 KB to 4 MB and client concurrency $C \in \{8, 32\}$, where C=8 uses a single thread with 8 in-flight requests and C=32 uses 4 threads each with 8 in-flight requests. Writes populate deterministic per-thread key spaces; a large scrub workload flushes the DAOS cache before measuring the cold-read throughput.

Local DAOS reads exceed the 100 Gbps NIC capacity at 256 KB block size with C=32, and saturate the SSD throughput at 4 MB block size with C=32. DAOS over RDMA approaches the 100 Gbps hardware limit on cold reads at 1 MB block size with C=8, while TCP lags consistently (Figure 8). We use the saturated RDMA configurations as the reference point for later S3 and ObjectCache experiments: any remaining gap is attributable to the S3 interface, gateway processing, GPU landing, or aggregation policy rather than the DAOS backend alone.

5.2 S3 Transport Baseline

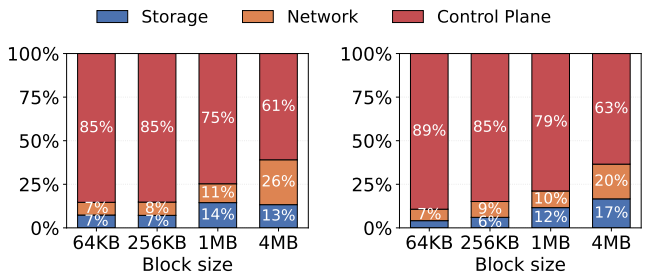
Setup. We next measure S3 PUT and GET throughput across block sizes from 64 KB to 4 MB. The benchmark compares S3TCP, S3RDMA Buffer that uses a gateway staging buffer, and S3RDMA Direct where the S3 request carries control information data going through the DAOS RDMA data path; Section 4 gives the corresponding path names. The two concurrency points, C=8 and C=32, match the canonical raw-storage configurations in Figure 8. Figure 9 shows that S3RDMA Direct approaches the NIC capacity at 4 MB block size with C=32. S3TCP and S3RDMA Buffer suffer from different bottlenecks. S3TCP is limited by the gateway's streaming HTTP path, while S3RDMA Buffer pays a server-side staging cost. Figure 9 also shows why transport alone is not sufficient: the single-object overhead remains visible for small KV chunks.

5.3 Layerwise Overlap Model with Hit Rate

What matters for inference is not whether the full prefix arrives quickly, but whether each layer of cached KV arrives before the GPU needs it. For a context length P (in tokens) with prefix hit rate r (the fraction of the P context tokens that are covered by the cached prefix), the matched KV bytes per layer are $D^{(\ell)} = 2\, n_{\text{kv}}\, d\, p\, (Pr)$. Given the measured per-layer compute time $t^{(\ell)}$, perfect overlap requires transfer throughput $B_{\text{req}} = D^{(\ell)}/t^{(\ell)}$. These expressions connect the byte layout from Equation 1 to the TTFT model in Equation 3. If ObjectCache's effective per-layer throughput exceeds B_{req}, only the first-layer transfer latency is visible in the TTFT; otherwise, each layer whose transfer exceeds its compute window adds directly to the total TTFT.

5.4 Per-Request Overhead and Overlap Feasibility

Figures 10 and 11 first isolate the two interface costs that ObjectCache must address: fixed per-request control cost



(a) GET. **(b)** PUT.

Figure 10. Per-request latency breakdown of S3RDMA Direct. Storage is backend object I/O, Network is RDMA data-plane transfer, and Control Plane is S3 frontend request and metadata processing. For small objects, fixed control-plane work dominates the remaining latency after RDMA removes TCP data movement.

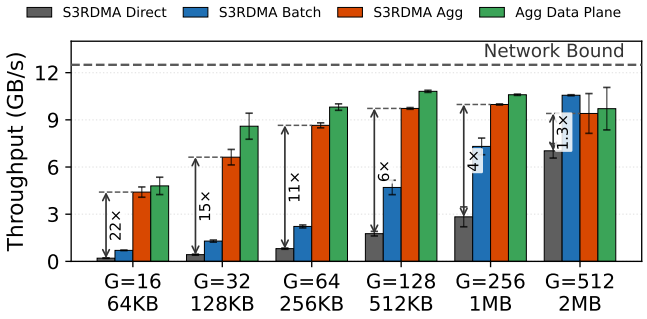
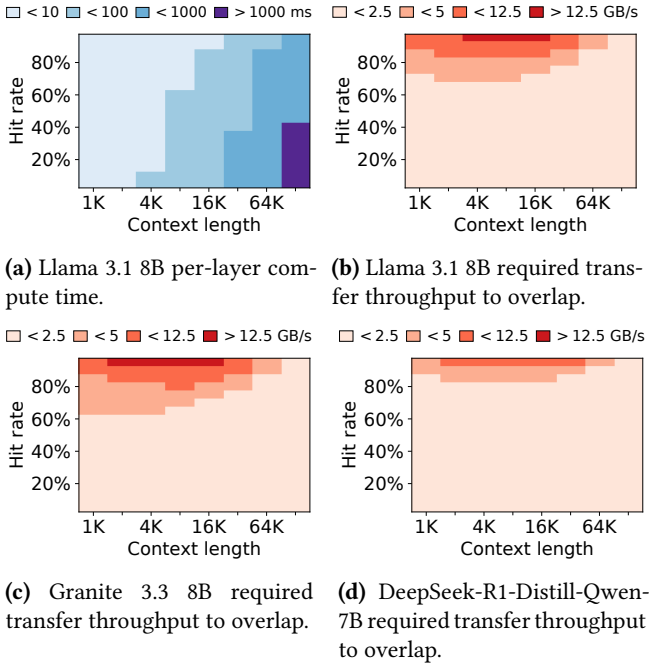


Figure 11. Server-side aggregation amortizes per-object overhead and achieves high speedups at small chunk granularities.

and low throughput for fine-grained objects. After RDMA reduces transfer overhead, HTTP and RGW metadata work dominate the remaining per-request cost at small objects (Figure 10). Figure 11 shows that batching and server-side aggregation turn many small object reads into larger layerwise transfers and recover high throughput for KV cache loads. Together, these measurements show why ObjectCache needs both pieces: batching amortizes fixed request overhead, while aggregation makes fine-grained KV chunks efficient to transfer.

Figure 12 connects aggregation throughput to serving. The first heatmap reports measured per-layer compute time for Llama 3.1 8B, while the remaining heatmaps report the transfer throughput required for Llama, Granite [30], and DeepSeek [13] models. Configurations requiring less bandwidth than the ObjectCache layer throughput are compute-bound; configurations above that boundary suffer from added latency. The counter-intuitive takeaway is that longer contexts can relax the transfer budget: although they carry more cached



(a) Llama 3.1 8B per-layer compute time. **(b)** Llama 3.1 8B required transfer throughput to overlap.

(c) Granite 3.3 8B required transfer throughput to overlap. **(d)** DeepSeek-R1-Distill-Qwen-7B required transfer throughput to overlap.

Figure 12. A100 layerwise overlap requirements. Most evaluated configurations require less than 2.5GB/s of layerwise transfer bandwidth, so ObjectCache can hide cached KV transfer under prefill compute in the dominant regimes.

KV bytes, they also create a larger per-layer compute window in which ObjectCache can hide layerwise transfer. ObjectCache also reduces many cross-object range reads per layer to a single RDMA payload per layer (Appendix Table A7).

5.5 End-to-End TTFT

We evaluate end-to-end TTFT for Llama 3.1 8B across two context lengths, 4K and 64K, three prefix hit rates, 12.5%, 50%, and 87.5%, and three chunk granularities, $G \in \{16, 64, 256\}$. For each workload configuration, the baseline is **opt-local-LW**: a pre-aggregated KV cache in layer-major order in pinned host-memory, so inference requires only host-to-device transfers and incurs no runtime aggregation cost. For all other TTFT measurements, we store KV cache data chunkwise with the specified G in local DRAM or on the remote DAOS server. Figure 13 reports the TTFT overhead of each configuration relative to this baseline.

The results show that layerwise delivery is important regardless of where the KV cache is stored. Local-DRAM-LW consistently outperforms Local-DRAM-CW, indicating that delivering KV blocks in layer order enables better overlap between transfer and GPU computation. In contrast, S3Batch-CW incurs the highest overhead in most configurations because it combines chunkwise delivery with S3 control-plane and network transfer costs. S3Agg-LW substantially reduces this overhead and performs close to Local-DRAM-LW in

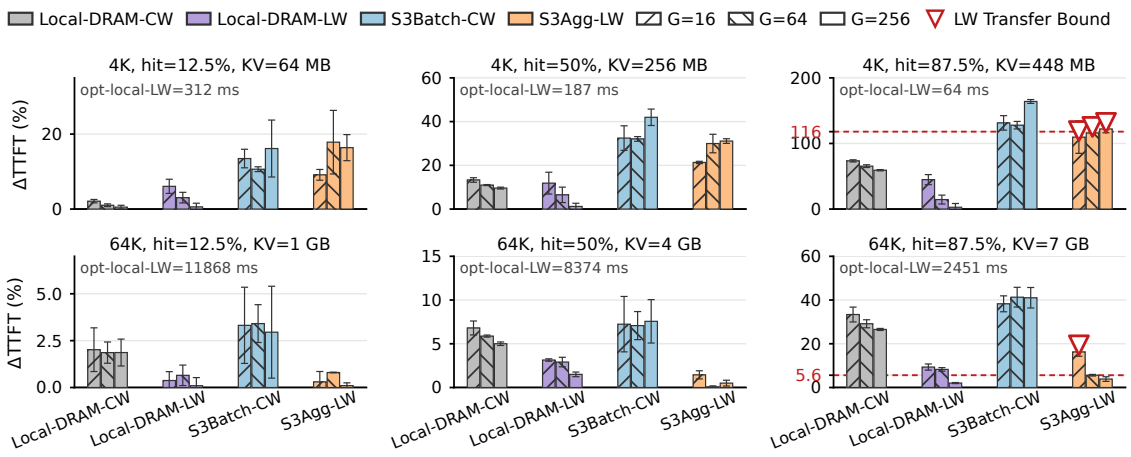


Figure 13. TTFT overhead for Llama 3.1 8B relative to the measured optimal local layerwise baseline for each workload configuration. CW denotes chunkwise delivery and LW denotes layerwise delivery; S3Agg-LW stays close to the local baseline except in the transfer-bound corner where layer delivery cannot be hidden by compute.

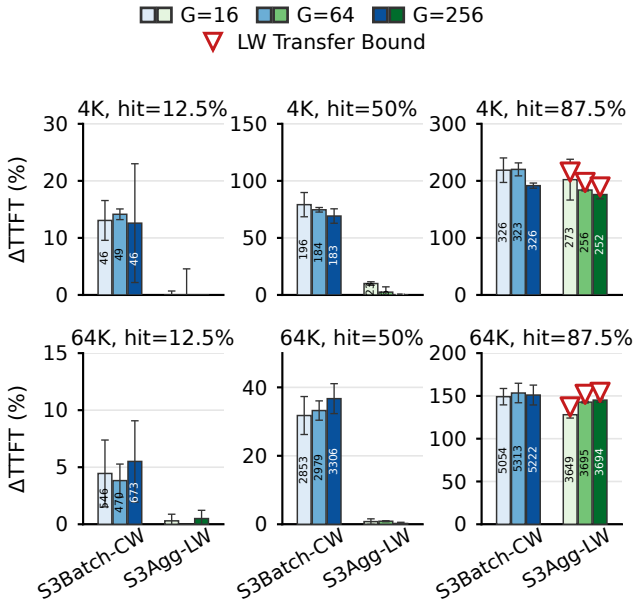


Figure 14. Sensitivity of S3-backed KV loading to bandwidth changes for Llama 3.1 8B. Each bar reports the relative TTFT increase when the same path and granularity are capped at 10 Gbps, using its 100 Gbps result as the baseline.

most cases. In several configurations, S3Agg-LW even achieves lower TTFT than Local-DRAM-LW. We interpret this as an observed resource-isolation effect: server-side aggregation uses dedicated CPU cores and parallel SSD reads, whereas Local-DRAM-LW consumes client-side CPU and memory bandwidth that may also be needed by the inference engine.

The benefit of S3Agg-LW is most evident for long-context workloads. At 64K context length, the per-layer compute window is sufficiently large to hide most transfer latency, and S3Agg-LW remains within 0.1–5.6% of opt-local-LW for $G = 64$ across all 64K configurations. Even at the highest hit rate, where the KV payload reaches 7 GB, the predicted overlap throughput is 3.1 GB/s (Appendix Table A8), which is below ObjectCache's sustained aggregation bandwidth. However, $G = 16$ shows noticeably higher overhead than $G = 64$ and $G = 256$, suggesting that small chunk granularity prevents the pipeline from fully exploiting the available server-side aggregation throughput of approximately 5 GB/s.

The short-context 4K workloads are more challenging because the compute window is much smaller. In these cases, the main $G = 64$ S3Agg-LW configuration adds 56–75 ms over opt-local-LW, and its TTFT can become comparable to S3Batch-CW. The limiting case is the 4K, 87.5% hit-rate workload, where the baseline compute time is only 64 ms and the required overlap throughput rises to 7.4 GB/s (Appendix Table A8). Although the KV payload is only 448 MB, fixed costs such as RDMA session setup, first-layer transfer, and control-plane exchange consume a significant fraction of the available compute window. As a result, all three granularities converge to similar TTFT overhead in this configuration, indicating that the bottleneck is the short compute window rather than steady-state transfer throughput. We therefore classify all three granularities in this configuration as transfer-bound: layer delivery, not steady-state bandwidth efficiency, determines the tail. Chunk granularity has a modest effect on chunkwise methods. For Local-DRAM-CW, larger G improves efficiency by reducing chunk management overhead. For S3Batch-CW, the effect is weaker because even $G = 16$ produces objects large enough to achieve reasonably high network transfer efficiency. These 4K results are consistent with the thresholded mode selection in Section 3.4: small-payload regimes can be dominated by fixed

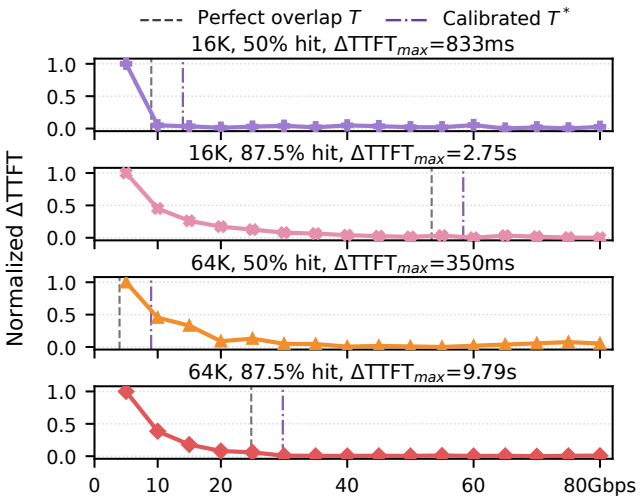


Figure 15. Sensitivity of layerwise TTFT to throttled transfer throughput with S3Agg-LW. Each panel normalizes TTFT increase relative to its best measured point. Dashed lines show the perfect-overlap bandwidth estimate; dash-dot lines show the calibrated scheduler target.

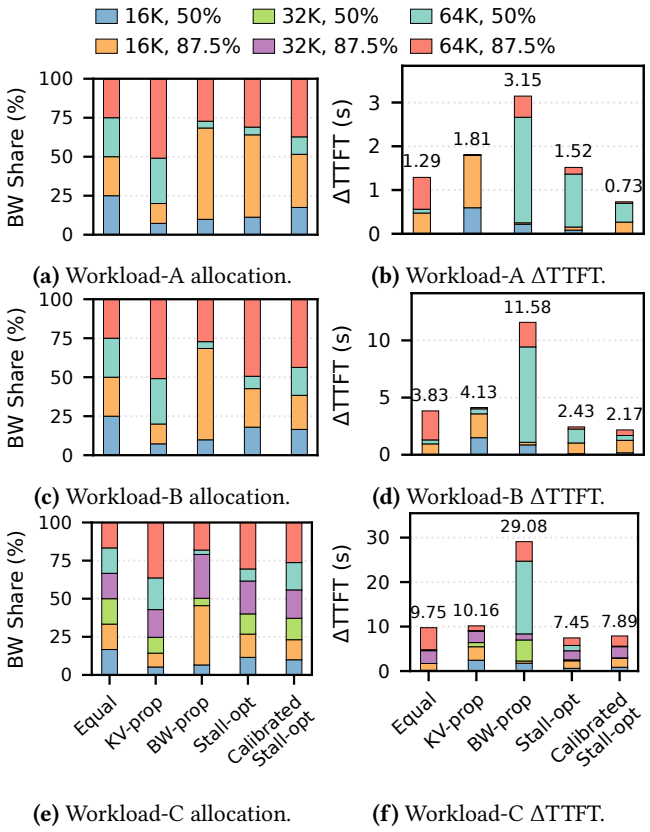
and startup costs, so ObjectCache should not assume that aggregation is always the better delivery mode.

5.6 Sensitivity to Bandwidth Changes

We evaluate bandwidth sensitivity by capping the S3-backed paths at 10 Gbps and reporting the TTFT increase relative to the corresponding 100 Gbps run. Figure 14 shows that S3Agg-LW is less sensitive to this bandwidth reduction than S3Batch-CW except in transfer-bound configurations.

This behavior follows from layerwise overlap. In S3Batch-CW, computation cannot start until the chunkwise KV load completes, so a bandwidth reduction largely appears as additional TTFT. In S3Agg-LW, only the non-overlapped portion of each layer transfer contributes to TTFT. Therefore, bandwidth changes affect TTFT mainly when the required overlap throughput exceeds the available bandwidth. Appendix Table A8 explains where this happens. At 64K and 50% hit rate, the required throughput is only 0.50 GB/s, well below 10 Gbps, so S3Agg-LW shows almost no sensitivity. At 4K and 50% hit rate, the predicted requirement is 1.45 GB/s, slightly above 10 Gbps, but the 100 Gbps run is already dominated by fixed costs and pipeline startup overheads; hence the bandwidth cap does not visibly dominate end-to-end TTFT.

The large increases appear at 87.5% hit rate, where the required throughput exceeds the capped bandwidth: 7.41 GB/s for 4K and 3.10 GB/s for 64K. These configurations become transfer-bound, as indicated by the markers in Figure 14, and therefore show the largest TTFT increase for S3Agg-LW. Overall, the results validate that layerwise loading is intrinsically less sensitive to bandwidth changes, but only while per-layer transfer can be hidden behind compute.



(a) Workload-A allocation. **(b)** Workload-A ΔTTFT.

(c) Workload-B allocation. **(d)** Workload-B ΔTTFT.

(e) Workload-C allocation. **(f)** Workload-C ΔTTFT.

Figure 16. Bandwidth scheduling under shared transfer caps. For each workload, the left panel shows the bandwidth allocated by each policy and the right panel shows the resulting added TTFT. Workload-A uses an 80 Gbps cap; Workload-B and Workload-C use 50 Gbps caps.

5.7 Bandwidth Allocation for Multi Tenants

We next evaluate how the bandwidth allocator behaves when multiple S3Agg-LW retrievals share a fixed transfer budget. Section 3.6 defines Stall-opt and Calibrated Stall-opt; here we compare them against three heuristic bandwidth-allocation baselines that do not model the layerwise overlap condition. *Equal* assigns the same bandwidth to every active request, which is attractive for fairness but ignores both KV cache size and compute slack. *KV-prop* allocates bandwidth in proportion to the retrieved KV cache size, favoring larger or higher-hit-rate requests. *BW-prop* allocates bandwidth in proportion to the zero-stall bandwidth estimate from B_{req}, so requests that require higher bandwidth to hide layerwise transfer receive a larger share. For Calibrated Stall-opt, we use a 5 Gbps margin, chosen from the S3Agg-LW rate sweep in Figure 15: the dash-dot calibrated targets shift the analytic perfect-overlap estimates to the measured knees of the TTFT curves.

Figure 16 compares the scheduling policies on three mixed workloads, and Appendix Table A9 reports the exact per-request bandwidth allocations. We construct the workloads

from the per-request bandwidth requirements in Appendix Table A8. Workloads A and B contain the same four requests: 16K and 64K contexts at 50% and 87.5% hit rates. Together, these requests require 91 Gbps in aggregate, or 111 Gbps after adding the 5 Gbps calibration margin to each request. The two workloads differ only in their shared bandwidth cap. Workload A uses an 80 Gbps cap to model moderate contention, where several requests can still remain near their calibrated overlap points. Workload B uses a 50 Gbps cap to stress the allocator under a tighter budget. Workload C extends this setting by adding 32K-context requests, resulting in six tenants with small, medium, and large KV cache footprints under the same 50 Gbps cap. Its aggregate requirement is 137 Gbps, or 167 Gbps after calibration. This workload tests whether the scheduler remains effective when the available bandwidth must be distributed across a denser set of transfer sensitivities. The measured per-configuration TTFTs are listed in Appendix Tables A10 and A11.

Across the three workloads, Calibrated Stall-opt outperforms the heuristic baselines by allocating bandwidth according to measured overlap requirements rather than per-request parity, KV size, or analytic demand alone. Measured as additional TTFT over the effectively unthrottled baseline, it reduces Equal's penalty by 1.2–1.8× (Figure 16); Appendix Table A12 gives the full breakdown. This improvement comes from assigning bandwidth near each request's useful overlap region. Equal ignores heterogeneous transfer sensitivity, KV-prop can starve small but latency-sensitive requests, and BW-prop can over-allocate beyond the point where extra bandwidth reduces TTFT. Calibrated Stall-opt avoids these extremes and reduces added TTFT under both moderate and tight caps. The only exception is Workload C, where Stall-opt is slightly better because the calibration margin can mildly over-provision some requests under a dense, heterogeneous 50 Gbps cap. Even there, Calibrated Stall-opt remains better than Equal, KV-prop, and BW-prop, preserving most of Stall-opt's benefit while improving robustness to measurement-level deviations from the analytic model.

6 Discussion

6.1 Position in the KV cache Hierarchy

ObjectCache is not a replacement for hot KV-cache tier, where GPU memory or DRAM-backed KV cache pools serve those hottest prefixes. ObjectCache instead provides a persistent, shared, S3-compatible capacity tier for large prefix pools, where placement flexibility and low storage cost matter. The key question is whether it can be made fast enough for serving-path prefix reuse. This role complements disaggregated inference systems that separate prefill, decode, and KV cache storage [27, 52, 55, 79]. By storing long-lived prefixes in object storage, ObjectCache lets requests run on any available GPU serving node rather than only where the prefix is cached.

6.2 When Layerwise Object Storage Helps

Layerwise ObjectCache is most useful when recomputation is costly, partial prefill computation provides enough per-layer overlap window, and the storage tier can deliver near the required overlap bandwidth. When the compute window is too small, bandwidth is below the overlap requirement, or the prefix hit is too small to amortize S3 overheads, the system should instead fall back to chunkwise transfer, a DRAM hot cache, or recomputation.

6.3 Limitations and Future Work

In this paper we optimize the serving-path data movement mechanism for S3-compatible object storage. Our prototype evaluates the critical data path and scheduling policy on a small cluster, which is sufficient to isolate the effects of aggregation, RDMA transfer, layerwise delivery, and bandwidth scheduling, but does not capture all production effects. Future work should validate ObjectCache with concurrent clients, long-running prefix churn, failures, and integration with batching [78], chunked prefill [2], routing [16], and eviction policies [19, 68].

ObjectCache also assumes that cached KV can be addressed by prefix chunk and layer range. Irregular layouts, packed objects, or compressed KV representations [6, 37] are compatible only if the storage server has sufficient metadata to translate layerwise requests. Finally, production deployments need scalable connection management, per-tenant admission control, and stronger isolation around RDMA credentials [41, 56, 59, 63, 77].

7 Related Work

KV cache management and prefix reuse. Paged KV allocation, radix-tree prefix matching, and KV offloading systems provide the software substrate that ObjectCache builds on [7, 32, 33, 53, 58, 72–74, 81]. These systems typically store KV in fixed-size chunks. ObjectCache preserves this chunk-wise, hash-addressed layout, but changes how remote storage delivers matched chunks to the serving node. CacheFlow [46] accelerates KV cache restoration by scheduling recomputation and I/O across token, layer, and GPU dimensions inside the serving system. ObjectCache addresses a different bottleneck: how an S3-compatible object tier should fetch, aggregate, and deliver many matched KV chunks in layer order to the inference engine.

Disaggregated LLM serving. Prior work on prefill/decode separation and remote KV pools shows that LLM serving can benefit from decoupling compute roles and moving KV cache data over the network [8, 9, 20, 27–29, 31, 42, 52, 54, 55, 60, 79]. ObjectCache follows this architectural direction, but targets a persistent S3-compatible object storage which targets for a cloud native solution.

S3-over-RDMA and object storage. Recent S3-over-RDMA systems split the HTTP control path from an RDMA data

path, improving large-object transfer over TCP-based S3 [10, 15, 25, 45, 47–49, 62, 65]. Low-latency object stores such as S3 Express reduce access latency but still expose single-object request semantics [80]. ObjectCache builds on this trend by adding KV-aware multi-object aggregation and layerwise delivery to S3-compatible storage.

KV compression and small-state attention. KV quantization, compression, MLA-style architectures, recent-window methods, and state-retention techniques reduce the bytes retained per token [3, 17, 19, 22, 26, 34, 35, 38, 39, 57, 61, 71, 75, 76]. These techniques are complementary to ObjectCache. They can reduce transfer volume, but they also shrink each useful object payload, making S3 overhead more dominant.

8 Conclusion

S3-compatible object storage is a natural storage tier for large prefix KV cache pools, but standard S3 exposes the mismatched serving-path semantics. ObjectCache adds the missing serving-path semantics via protocol-scheduling co-design. Our prototype shows that these pieces must work together. RDMA improves raw S3 transport, aggregation amortizes network overhead, layerwise delivery overlaps transfer with GPU compute, and bandwidth-aware scheduling reduces added TTFT under shared bandwidth. On a 100 Gbps RoCE prototype with Llama 3.1 8B, ObjectCache reaches within 5.6% of the optimal local DRAM solution at 64K context length. Our results suggest that object storage can serve as a runtime KV cache backend rather than only a cold archive, while retaining the persistence, capacity, placement flexibility, and low storage cost that make object storage attractive.

References

[1] Mohamad Abou Ali, Fadi Dornaika, and Jinan Charafeddine. 2025. Agentic AI: a comprehensive survey of architectures, applications, and future directions. *Artificial Intelligence Review* 59, 1 (2025), 11.

[2] Arney Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2025. Efficient llm inference via chunked prefills. *ACM SIGOPS Operating Systems Review* 59, 1 (2025), 9–16.

[3] Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 4895–4901.

[4] Anthropic. 2026. Models & Pricing. https://platform.claude.com/docs/en/about-claude/pricing.

[5] Ceph. 2026. Ceph - a scalable distributed storage system. https://github.com/ceph/ceph.

[6] Chi-Chih Chang, Wei-Cheng Lin, Chien-Yu Lin, Chong-Yan Chen, Yu-Fang Hu, Pei-Shuo Wang, Ning-Chi Huang, Luis Ceze, Mohamed Abdelfattah, and Kai-Chiang Wu. 2025. Palu: KV-cache compression with low-rank projection. In *International Conference on Learning Representations*, Vol. 2025. 50222–50249.

[7] 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.

[8] Yukang Chen, Weihao Cui, Han Zhao, Ziyi Xu, Xiaoze Fan, Xusheng Chen, Yangjie Zhou, Shixuan Sun, Bingsheng He, and Quan Chen. 2026. Towards High-Goodput LLM Serving with Prefill-decode Multiplexing. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2030–2047.

[9] Rongxin Cheng, Yuxin Lai, Xingda Wei, Rong Chen, and Haibo Chen. 2026. KUNSERVE: Parameter-centric Memory Management for Efficient Memory Overloading Handling in LLM Serving. In *Proceedings of the 21st European Conference on Computer Systems (EuroSys '26)*. Association for Computing Machinery, 1244–1260. doi:10.1145/3767295.3769348

[10] Cloudian. 2025. Supercharging Vector Database Indexing: 8x Faster with Cloudian S3 RDMA, Milvus and NVIDIA. https://cloudian.com/blog/supercharging-vector-database-indexing-8x-faster-with-cloudian-s3-rdma-and-nvidia/.

[11] 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.

[12] DAOS. 2026. DAOS. https://docs.daos.io/v2.6/.

[13] DeepSeek. 2026. DeepSeek R1 Distill Qwen 7B. https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-7B.

[14] DeepSeek. 2026. Models & Pricing. https://api-docs.deepseek.com/quick_start/pricing.

[15] Dell Technologies. 2026. Accelerating AI Workloads with RDMA for S3-compatible storage: A Game-Changer with Dell ObjectScale. https://infohub.delltechnologies.com/en-uk/p/accelerating-ai-workloads-with-rdma-for-s3-compatible-storage-a-game-changer-with-dell-objectscale/.

[16] Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks Lakshmanan, and Ahmed H Awadallah. 2024. Hybrid llm: Cost-efficient and quality-aware query routing. In *International Conference on Learning Representations*, Vol. 2024. 41348–41366.

[17] Dayou Du, Shijie Cao, Jianyi Cheng, Luo Mai, Ting Cao, and Mao Yang. 2026. Bitdecoding: Unlocking tensor cores for long-context llms with low-bit kv cache. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1–13.

[18] Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. 2024. A survey on rag meeting llms: Towards retrieval-augmented large language models. In *Proceedings of the 30th ACM SIGKDD conference on knowledge discovery and data mining*. 6491–6501.

[19] Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S Kevin Zhou. 2026. Ada-kv: Optimizing kv cache eviction by adaptive budget allocation for efficient llm inference. *Advances in Neural Information Processing Systems* 38 (2026), 113152–113188.

[20] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. {ServerlessLLM}:{Low-Latency} serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 135–153.

[21] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. {Cost-Efficient} large language model serving for multi-turn conversations with {CachedAttention}. In *2024 USENIX annual technical conference (USENIX ATC 24)*. 111–126.

[22] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2024. Model tells you what to discard: Adaptive kv cache compression for llms. In *International Conference on Learning Representations*, Vol. 2024. 22975–22988.

[23] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt cache: Modular attention reuse for

low-latency inference. *Proceedings of Machine Learning and Systems* 6 (2024), 325–338.

[24] Google. 2026. Models & Pricing. https://cloud.google.com/gemini-enterprise-agent-platform/generative-ai/pricing.

[25] Yingyi Hao, Ting Yao, Xingda Wei, Dingyan Zhang, Tianle Sun, Yiwen Zhang, Zhiyong Fu, Huatao Wu, and Rong Chen. 2026. Fast Cloud Storage for {AI} Jobs via Grouped {I/O}{API} with Transparent {Read/Write} Optimizations. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*. 255–270.

[26] 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.

[27] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, et al. 2024. Memserve: Context caching for disaggregated llm serving with elastic memory pool. *arXiv preprint arXiv:2406.17565* (2024).

[28] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Chenxi Wang, Jiang Xu, Shuang Chen, Hao Feng, Sa Wang, Yungang Bao, et al. 2025. ShuffleInfer: Disaggregate LLM inference for mixed downstream workloads. *ACM Transactions on Architecture and Code Optimization* 22, 2 (2025), 1–24.

[29] Junhao Hu, Jiang Xu, Zhixia Liu, Yulong He, Yuetao Chen, Hao Xu, Jiang Liu, Jie Meng, Baoquan Zhang, Shining Wan, et al. 2025. {DEEPSERVE}: Serverless Large Language Model Serving at Scale. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 57–72.

[30] IBM. 2026. Granite 3.3 8B. https://huggingface.co/ibm-granite/granite-3.3-8b-instruct.

[31] Aditya K Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. 2025. Pod-attention: Unlocking full prefill-decode overlap for faster llm inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 897–912.

[32] 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.

[33] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 155–172.

[34] 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.

[35] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434* (2024).

[36] Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, et al. 2025. Lmcache: An efficient KV cache layer for enterprise-scale LLM inference. *arXiv preprint arXiv:2510.09665* (2025).

[37] 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.

[38] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2023. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems* 36 (2023), 52342–52364.

[39] 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. In *Proceedings of the 41st International Conference on Machine Learning*. 32332–32344.

[40] LMCache. 2026. LMCache. https://github.com/lmcache/lmcache.

[41] Jiaqi Lou, Xinhao Kong, Jinghan Huang, Wei Bai, Nam Sung Kim, and Danyang Zhuo. 2024. Harmonic: Hardware-assisted {RDMA} performance isolation for public clouds. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1479–1496.

[42] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. 2025. Helix: Serving large language models over heterogeneous gpus and network via max-flow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 586–602.

[43] Fanxu Meng, Pingzhi Tang, Zengwei Yao, Xing Sun, and Muhan Zhang. 2026. TransMLA: Migrating GQA models to MLA with full deepseek compatibility and speedup. *Advances in Neural Information Processing Systems* 38 (2026), 81977–82019.

[44] Meta. 2026. Llama 3.1 8B. https://huggingface.co/meta-llama/Llama-3.1-8B.

[45] MinIO. 2025. MinIO S3 over RDMA. https://blog.min.io/s3-over-rdma/.

[46] Sean Nian, Jiahao Fang, Qilong Feng, Zhiyu Wu, and Fan Lai. 2026. CacheFlow: Efficient LLM Serving with 3D-Parallel KV Cache Restoration. *arXiv preprint arXiv:2604.25080* (2026).

[47] NVIDIA. 2025. How to Unlock Accelerated AI Storage Performance With RDMA for S3-Compatible Storage. https://blogs.nvidia.com/blog/s3-compatible-ai-storage/.

[48] NVIDIA. 2026. NVIDIA cuObject: GPUDirect Storage for Objects. https://docs.nvidia.com/gpudirect-storage/cuobject/index.html.

[49] NVIDIA. 2026. NVIDIA cuObject server v1.0.0 Release Notes. https://docs.nvidia.com/gpudirect-storage/cuobject/cuobject-server-release-notes/index.html.

[50] Nvidia. 2026. NVIDIA Inference Xfer Library (NIXL). https://github.com/ai-dynamo/nixl.

[51] OpenAI. 2026. Models & Pricing. https://developers.openai.com/api/docs/pricing.

[52] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.

[53] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vattention: Dynamic memory management for serving llms without pagedattention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 1133–1150.

[54] Ruoyu Qin, Weiran He, Yaoyu Wang, Zheming Li, Xinran Xu, Yongwei Wu, Weimin Zheng, and Mingxing Zhang. 2026. Prefill-as-a-Service: KVCache of Next-Generation Models Could Go Cross-Datacenter. *arXiv preprint arXiv:2604.15039* (2026).

[55] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Heyi Tang, Feng Ren, Teng Ma, Shangming Cai, Yineng Zhang, Mingxing Zhang, et al. 2024. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *ACM Transactions on Storage* (2024).

[56] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. 2021. {ReDMArk}: Bypassing {RDMA} security mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)*. 4277–4292.

[57] Noam Shazeer. 2019. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150* (2019).

[58] 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.

[59] Anna Kornfeld Simpson, Adriana Szekeres, Jacob Nelson, and Irene Zhang. 2020. Securing {RDMA} for {High-Performance} datacenter storage systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[60] Zhaoyuan Su, Zeyu Zhang, Tingfeng Lan, Zirui Wang, Juncheng Yang, and Yue Cheng. 2026. MorphServe: Efficient and Workload-Aware LLM Serving via Runtime Quantized Layer Swapping and KV Cache Resizing. In *Proceedings of Machine Learning and Systems*.

[61] Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. [n. d.]. ShadowKV: KV Cache in Shadows for High-Throughput Long-Context LLM Inference. In *Forty-second International Conference on Machine Learning*.

[62] Shuwen Sun, Isaac Khor, Ji-Yong Shin, and Peter Desnoyers. 2025. A Fast, Efficient, and Strongly-Consistent Object Store. In *Proceedings of the 2025 ACM Symposium on Cloud Computing*. 708–721.

[63] Shin-Yeh Tsai, Mathias Payer, and Yiying Zhang. 2019. Pythia: remote oracles for the masses. In *28th USENIX Security Symposium (USENIX Security 19)*. 693–710.

[64] UCX. 2026. Unified Communication X. https://github.com/openucx/ucx.

[65] VAST Data. 2025. S3 over RDMA: Scaling the KV Cache Data Plane. https://www.vastdata.com/blog/s3-over-rdma-scaling-the-kv-cache-data-plane.

[66] 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).

[67] vLLM. 2026. vLLM. https://github.com/vllm-project/vllm.

[68] 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 *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 465–482.

[69] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, et al. 2025. Burstgpt: A real-world workload dataset to optimize llm serving systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*. 5831–5841.

[70] Yuxing Xiang, Xue Li, Kun Qian, Yan Zhang, Wenyuan Yu, Ennan Zhai, Xin Jin, and Jingren Zhou. 2026. {ServeGen}: Workload Characterization and Generation of Large Language Model Serving in Production. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 1845–1859.

[71] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, Vol. 2024. 21875–21895.

[72] 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.

[73] Lu Ye, Ze Tao, Yong Huang, and Yang Li. 2024. Chunkattention: Efficient self-attention with prefix-aware kv cache and two-phase partition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 11608–11620.

[74] Weiping Yu, Ye Jiarui, He Mengke, Junfeng Liu, and Siqiang Luo. 2025. On 10x Better Scalability: KV Stores Scale Up KV Cache. *arXiv preprint arXiv:2511.16138* (2025).

[75] Amir Zandieh, Majid Daliri, Majid Hadian, and Vahab Mirrokni. 2026. TurboQuant: Online Vector Quantization with Near-optimal Distortion Rate. In *Proceedings of the International Conference on Learning Representations*.

[76] 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.

[77] Chenxingyu Zhao, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. 2025. {White-Boxing}{RDMA} with {Packet-Granular} Software Control. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 427–449.

[78] Zhen Zheng, Xin Ji, Taosong Fang, Fanghao Zhou, Chuanjie Liu, and Gang Peng. 2026. BatchLLM: Optimizing Large Batched LLM Inference with Global Prefix Sharing and Throughput-oriented Token Batching. In *Proceedings of Machine Learning and Systems (MLSys)*.

[79] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 193–210.

[80] Jiatang Zhou, Kaisong Huang, and Tianzheng Wang. 2026. Milliscale: Fast Commit on Low-Latency Object Storage. *arXiv preprint arXiv:2603.02108* (2026).

[81] Jing Zou, Shangyu Wu, Hancong Duan, Qiao Li, and Chun Jason Xue. 2026. ContiguousKV: Accelerating LLM Prefill with Granularity-Aligned KV Cache Management. *arXiv preprint arXiv:2601.13631* (2026).

Table A1. Steady-state boundary-granularity recompute cost for Llama 3.1 8B.

GPU	Ctx.	Hit (%)	$G = 16$	$G = 512$	Δ
A100	4K	12.5	322.6 ± 8.6	344.2 ± 5.9	21.6
		25.0	270.5 ± 0.4	310.3 ± 1.8	39.7
		37.5	236.3 ± 1.4	264.5 ± 0.6	28.2
		50.0	192.6 ± 0.4	222.7 ± 0.2	30.2
		62.5	149.3 ± 2.0	182.1 ± 0.0	32.9
		75.0	109.0 ± 0.1	141.8 ± 1.5	32.7
		87.5	70.2 ± 0.9	101.9 ± 0.3	31.7
	64K	12.5	11643.8 ± 9.3	11686.9 ± 12.7	43.1
		25.0	10768.3 ± 13.2	10810.8 ± 1.1	42.5
		37.5	9616.7 ± 2.3	9671.8 ± 6.2	55.2
		50.0	8236.6 ± 6.8	8283.8 ± 6.5	47.2
		62.5	6577.9 ± 9.0	6645.4 ± 9.5	67.6
		75.0	4676.2 ± 1.0	4763.2 ± 0.1	87.0
		87.5	2538.5 ± 3.1	2642.0 ± 0.5	103.6
H100	4K	12.5	143.5 ± 0.2	162.3 ± 0.5	18.8
		25.0	122.8 ± 0.0	140.7 ± 0.3	17.9
		37.5	107.1 ± 1.5	125.9 ± 0.5	18.8
		50.0	87.5 ± 0.7	106.2 ± 1.5	18.6
		62.5	69.0 ± 0.1	87.9 ± 0.9	18.9
		75.0	50.0 ± 0.1	70.8 ± 1.0	20.8
		87.5	33.7 ± 0.1	51.7 ± 0.8	17.9
	64K	12.5	4948.6 ± 14.4	5009.5 ± 2.9	60.9
		25.0	4547.8 ± 0.4	4605.5 ± 5.5	57.7
		37.5	4015.1 ± 1.9	4067.1 ± 2.9	52.0
		50.0	3410.1 ± 3.6	3482.9 ± 1.2	72.7
		62.5	2699.0 ± 0.6	2757.0 ± 0.7	58.0
		75.0	1911.2 ± 3.4	1985.0 ± 3.1	73.7
		87.5	1022.1 ± 1.1	1090.5 ± 1.8	68.5

Table A2. Steady-state boundary-granularity recompute cost for DeepSeek-R1-Distill-Qwen-7B.

GPU	Ctx.	Hit (%)	$G = 16$	$G = 512$	Δ
A100	4K	12.5	283.1 ± 1.2	317.0 ± 2.8	34.0
		25.0	251.2 ± 2.9	278.5 ± 0.3	27.3
		37.5	212.0 ± 2.9	243.7 ± 2.9	31.7
		50.0	177.7 ± 1.6	206.3 ± 0.4	28.6
		62.5	140.6 ± 0.7	168.5 ± 0.9	27.8
		75.0	102.6 ± 0.1	132.2 ± 0.3	29.6
		87.5	60.2 ± 0.0	93.5 ± 0.3	33.3
	64K	12.5	9633.1 ± 19.9	9708.1 ± 17.7	75.0
		25.0	8893.6 ± 26.1	8966.9 ± 0.1	73.3
		37.5	7932.3 ± 16.4	7994.1 ± 8.8	61.8
		50.0	6769.8 ± 16.1	6836.8 ± 8.2	66.9
		62.5	5414.7 ± 2.2	5479.2 ± 1.3	64.5
		75.0	3847.8 ± 3.2	3914.7 ± 19.3	66.9
		87.5	2098.0 ± 3.2	2179.5 ± 9.9	81.5
H100	4K	12.5	133.7 ± 0.8	149.8 ± 0.5	16.1
		25.0	117.1 ± 0.2	134.3 ± 1.0	17.3
		37.5	98.4 ± 0.3	115.8 ± 0.6	17.4
		50.0	83.9 ± 2.0	99.5 ± 0.9	15.6
		62.5	68.1 ± 1.8	81.1 ± 0.1	12.9
		75.0	45.6 ± 0.5	64.2 ± 0.7	18.6
		87.5	31.7 ± 0.3	46.2 ± 0.2	14.5
	64K	12.5	4259.2 ± 8.6	4311.1 ± 0.8	51.9
		25.0	3898.4 ± 1.2	3934.0 ± 3.1	35.6
		37.5	3425.6 ± 0.4	3454.9 ± 2.3	29.3
		50.0	2895.8 ± 0.0	2922.4 ± 2.2	26.6
		62.5	2280.2 ± 2.8	2329.3 ± 4.9	49.1
		75.0	1616.2 ± 0.5	1647.2 ± 3.0	31.0
		87.5	847.2 ± 1.1	904.5 ± 0.2	57.2

A Additional Motivation Data

Tables A1 and A2 report the full steady-state measurements behind the cache-boundary granularity study. For each semantic hit boundary, the prompt reuses $M-G$ tokens, where $M = C \cdot r$ is the base hit boundary and $G \in \{16, 512\}$. Thus $G = 512$ recomputes 496 more tokens than $G = 16$ in every row. Entries are TTFT in milliseconds over steady-state trials 1–2, reported as mean ± one standard deviation. $\Delta = T_{G=512} - T_{G=16}$ uses the same steady-state trials and excludes trial 0 to avoid shape-warmup effects.

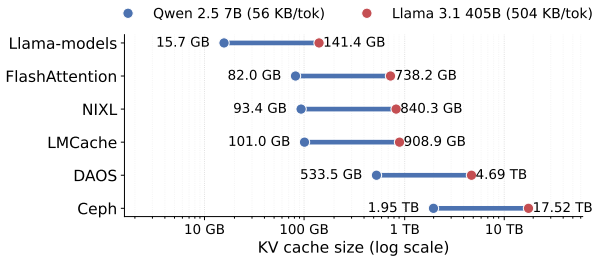


Figure A1. Repository-scale prefixes can translate into large KV cache footprints. Ranges are computed from repository token counts and two representative KV cache sizes per token.

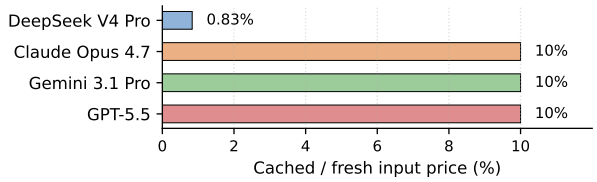


Figure A2. Major APIs price cached input far below fresh input, reflecting the economic value of prefix reuse in deployed LLM services [4, 14, 24, 51].

B Measured Cache-Boundary Recompute

C Pinned CPU-GPU H2D Microbenchmarks

D Per-model, per-GPU recompute and bandwidth heatmaps

Figures A4–A7 extend the main-paper recompute and required-throughput surfaces across the four representative 7–8B models and both GPU platforms. Total cached KV size is GPU-agnostic (Figure A4); per-layer compute time (Figure A5), MFU (Figure A6), and required bandwidth (Figure A7) each stack A100 (top row) and H100 (bottom row) to expose the GPU-platform effect at constant model and context.

E Optimal Aggregation Size Benchmark

Here we test what is the optimal aggregation size for different chunk granularities.

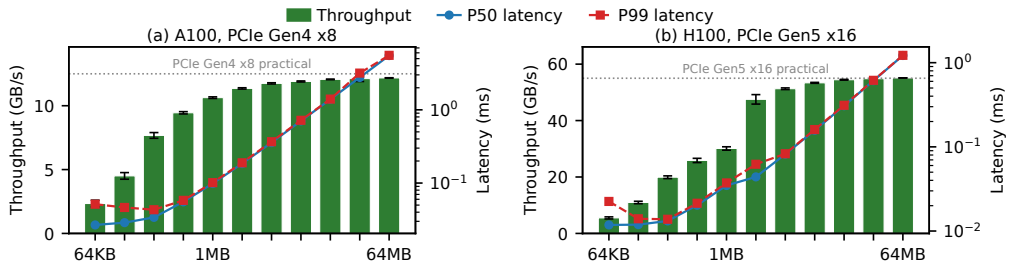
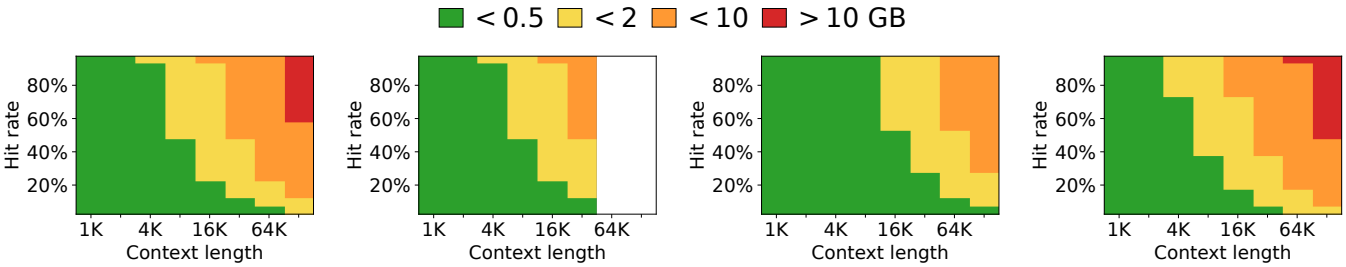
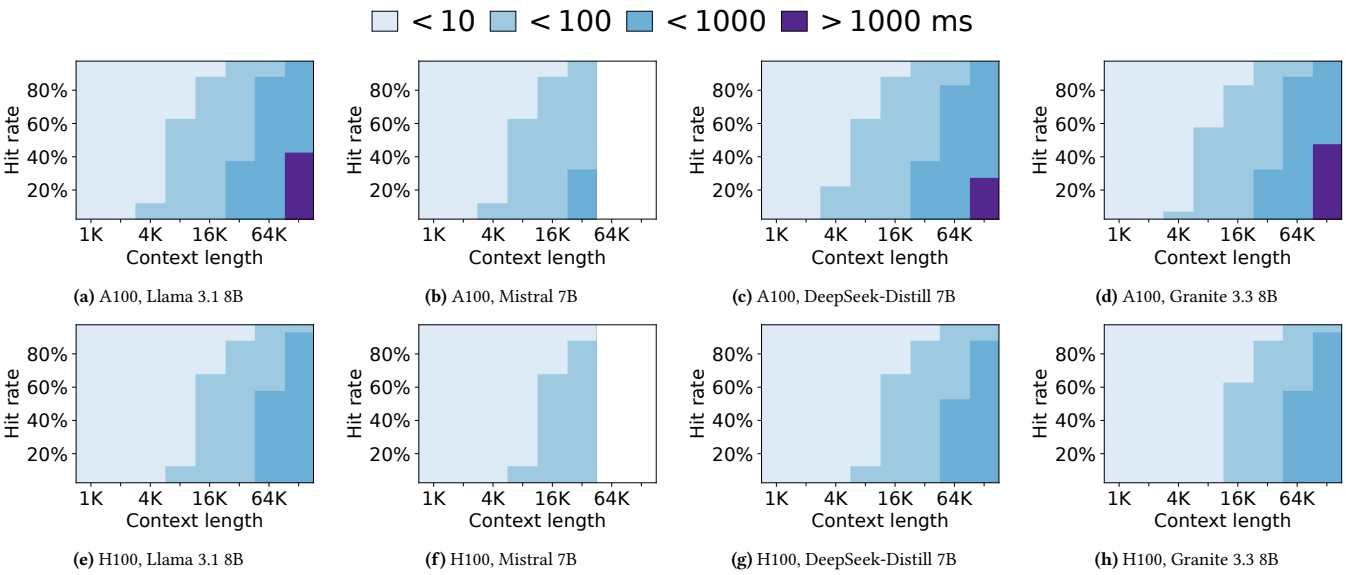


Figure A3. Pinned CPU→GPU H2D throughput and latency vs. block size for A100 (PCIe Gen4 x8) and H100 (PCIe Gen5 x16). Bars report mean throughput ($\pm 1\sigma$ over 200 per-size samples); right-axis lines report P50/P99 latency across 64 KB–64 MB blocks. The source buffer is placed on NUMA 0 in both setups. A100 saturates near 12 GB/s; H100 saturates near 55 GB/s for blocks ≥8 MB.



(a) Llama 3.1 8B **(b)** Mistral 7B **(c)** DeepSeek-Distill 7B **(d)** Granite 3.3 8B

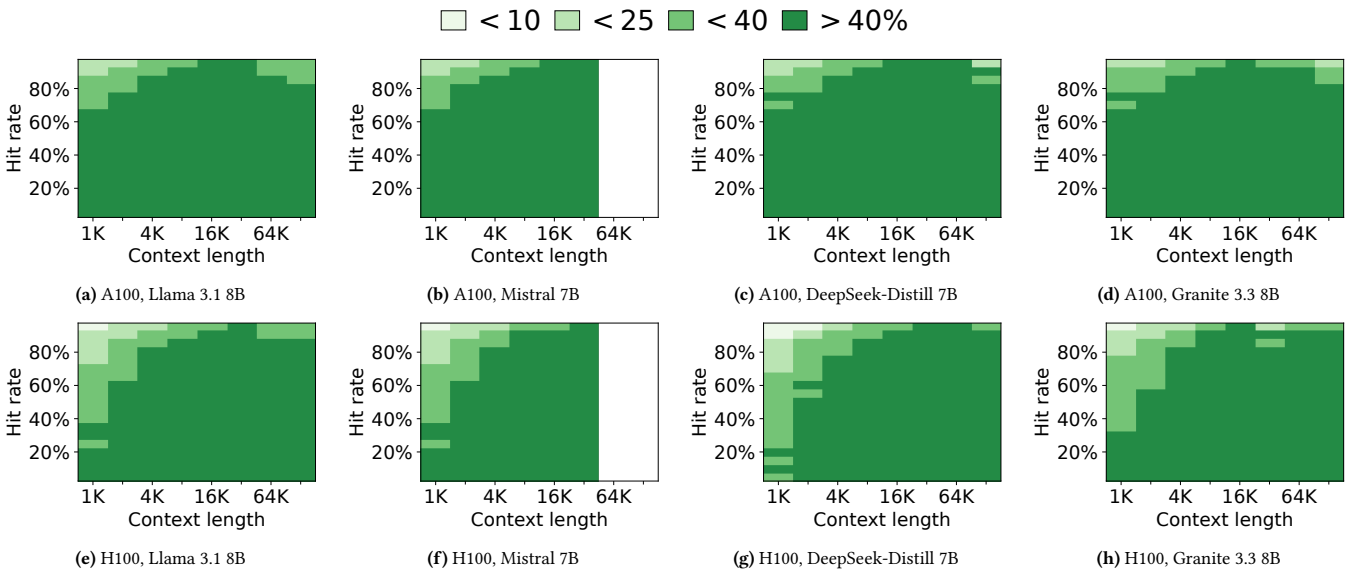
Figure A4. Total cached KV size per (context length, hit rate) configuration for each model. KV size is independent of GPU platform, so each value is reported once.



(a) A100, Llama 3.1 8B **(b)** A100, Mistral 7B **(c)** A100, DeepSeek-Distill 7B **(d)** A100, Granite 3.3 8B

(e) H100, Llama 3.1 8B **(f)** H100, Mistral 7B **(g)** H100, DeepSeek-Distill 7B **(h)** H100, Granite 3.3 8B

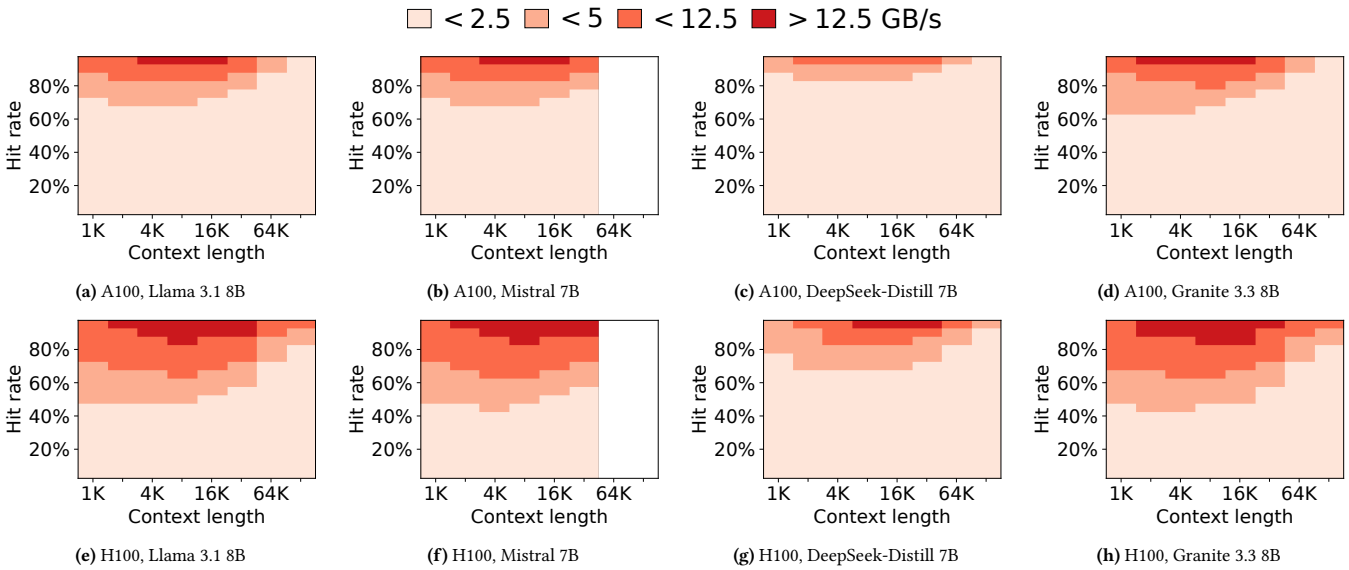
Figure A5. Per-layer compute time across four representative models on A100 (top row) and H100 (bottom row). Blank configurations indicate unsupported or unmeasured context lengths, such as Mistral 7B beyond its 32K window.



(a) A100, Llama 3.1 8B **(b)** A100, Mistral 7B **(c)** A100, DeepSeek-Distill 7B **(d)** A100, Granite 3.3 8B

(e) H100, Llama 3.1 8B **(f)** H100, Mistral 7B **(g)** H100, DeepSeek-Distill 7B **(h)** H100, Granite 3.3 8B

Figure A6. Model FLOP utilization (MFU) across four representative models on A100 (top row) and H100 (bottom row). Low-MFU cells indicate configurations that do not saturate GPU compute; high-MFU cells identify compute-dense configurations.



(a) A100, Llama 3.1 8B **(b)** A100, Mistral 7B **(c)** A100, DeepSeek-Distill 7B **(d)** A100, Granite 3.3 8B

(e) H100, Llama 3.1 8B **(f)** H100, Mistral 7B **(g)** H100, DeepSeek-Distill 7B **(h)** H100, Granite 3.3 8B

Figure A7. Required per-layer transfer throughput to fully overlap cached KV loading with compute across four models on A100 (top row) and H100 (bottom row). Faster compute on H100 raises the bandwidth needed for perfect overlap.

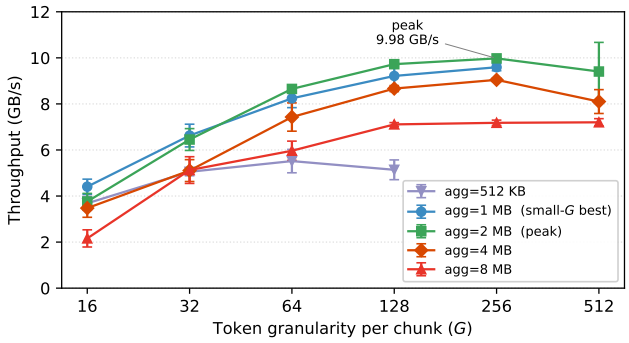


Figure A8. S3RDMA Agg throughput across aggregation sizes and chunk granularities. The sweep covers `agg_size` $\in$ {512KB,1MB,2MB,4MB,8MB} and $G \in \{16, 32, 64, 128, 256, 512\}$. Throughput peaks at G=256 with `agg_size`=2 MB (9.98 GB/s); for G=16, `agg_size`=1 MB is best.

F Reference Algorithms and Cost Table

This appendix collects the full pseudocode for the ObjectCache layerwise GET and the Calibrated Stall-opt scheduler, together with the storage-tier cost breakdown referenced in the main discussion.

Table A3. ObjectCache layerwise GET (full pseudocode).

for $\ell = 0, \ldots, L-1$ **do**
  $B_\ell \leftarrow \emptyset$
  for each key H_j in `chunk_keys` **do**
    $o \leftarrow \ell \cdot S$
    append RangeGet(H_j, o, S) to B_ℓ
  RDMAWrite(`client_buffer`$[\ell], B_\ell$)
  NotifyLayerReady(ℓ)

Table A4. Calibrated Stall-opt layerwise bandwidth allocation (full pseudocode).

Input: epoch requests $\mathcal{R}$, bandwidth cap B, scheduler margin Δ; per-request bytes-per-layer b_i and per-layer compute time t_i.
for each request $i \in \mathcal{R}$ **do**
  $n_i \leftarrow b_i / t_i + \Delta$
Allocate up to n_i while capacity remains.
Redistribute unassigned budget to requests with remaining per-layer stall.
Hold per-request rates stable for this epoch.
Dispatch layer payloads with weighted deficit round robin.

Table A5. Approximate cost comparison across storage tiers (2025 cloud pricing).

Tier	Capacity	$/GB/month	Latency
GPU VRAM (H100 80GB)	80 GB	$25+	<1 μs
CPU DRAM	512 GB	$3–5	100 ns
Remote DRAM (RDMA)	TB-scale	$1–2	10–100 μs
NVMe SSD (local)	TB-scale	$0.10	100 μs
S3/Object Storage	PB-scale	$0.02	10–50 ms

Table A6. Extra GPU prefill latency per cache-hit boundary when chunk granularity increases from G=16 (vLLM [32] default) to G=512 (Mooncake [55]), measured on A100/H100. Each boundary recomputes up to 496 redundant tokens. At 1K QPS, the Llama 8B-64K penalty wastes ~1,500 GPU hours per day for both A100 and H100.

	Llama 3.1 8B		**DeepSeek R1 7B**	
Metric	**4K**	**64K**	**4K**	**64K**
Δ FLOPs	8.0 T	24.0 T	7.3 T	19.5 T
A100 Δt	31.0±5.5 ms	63.7±23.7 ms	30.3±2.7 ms	70.0±6.9 ms
H100 Δt	18.8±1.0 ms	63.4±8.3 ms	16.1±1.9 ms	40.1±12.4 ms

Δ FLOPs: extra recompute work per request from the coarser chunk tail (496 additional tokens), constant across hit rates $r \in \{12.5\%, 25\%, \ldots, 87.5\%\}$. Latency rows: mean ± std of $\Delta t = T_{G=512} - T_{G=16}$ measured at each r with one chunk miss. Steady-state after warm-up; full per-hit breakdowns in Appendix Tables 1–2.

Table A7. Client-visible element count for S3RDMA Agg bounded layerwise aggregation.

	Context = 4K			Context = 64K		
chunk (tokens)	16	64	256	16	64	256
number (hit=87.5%)	224	56	14	3,584	896	224
original elements	7,168	1,792	448	114,688	28,672	7,168
agg payload size	1 MB	2 MB	2 MB	1 MB	2 MB	2 MB
elements per agg	16	8	2	16	8	2
elements after agg	448	224	224	7,168	3,584	3,584
reduction factor	16×	8×	2×	16×	8×	2×

G Compute time and required throughput at the canonical configurations

Table A8 reports total prefill compute time, per-layer compute time, and the per-layer transfer throughput needed for full overlap, for the (C, r) configurations used throughout the streaming-overlap analysis and scheduler workloads.

Table A8. Per-layer compute time and required transfer throughput to fully overlap one layer of cached KV loading with compute, for Llama 3.1 8B on A100 80GB. Total cached KV bytes = $r \cdot C \cdot L \cdot$ b with L=32 layers and b=4096 bytes per token per layer. $T^{(\ell)}_{\text{compute}} = T_{\text{total}}/L$. Required transfer throughput = KV/T_{total} (equivalently per-layer-KV / per-layer-compute).

Context Length C	Hit Rate r	Cached Tokens	T_{total} (ms)	$T^{(\ell)}_{\text{compute}}$ (ms / layer)	Req. BW (GB/s)
4K	0.500	2,048	185.31	5.79	1.45
4K	0.875	3,584	63.47	1.98	7.41
16K	0.500	8,192	955.89	29.87	1.12
16K	0.875	14,336	281.76	8.80	6.67
32K	0.500	16,384	2,589.25	80.91	0.83
32K	0.875	28,672	763.19	23.85	4.92
64K	0.500	32,768	8,672.79	271.02	0.50
64K	0.875	57,344	2,423.90	75.75	3.10

H Per-request allocations behind Figure 16

Table A9 reports the per-request transfer throughput each scheduling policy assigns under shared 80 Gbps and 50 Gbps caps to show moderate and stronger contention for the scheduler workloads.

Table A9. Per-request allocated transfer throughput (Gbps) for the parallel scheduler workloads, using an 80 Gbps cap for Workload A and a 50 Gbps cap for Workloads B and C.

Request	Equal	KV-prop	BW-prop	Stall-opt	Cal. Stall-opt
Workload A: 16K/64K, 50%/87.5%, 80 Gbps cap					
16K, 50%	20.00	5.82	7.89	8.99	13.99
16K, 87.5%	20.00	10.18	46.85	42.25	27.25
64K, 50%	20.00	23.27	3.48	3.96	8.96
64K, 87.5%	20.00	40.73	21.78	24.81	29.81
Workload B: 16K/64K, 50%/87.5%, 50 Gbps cap					
16K, 50%	12.50	3.64	4.93	8.99	8.26
16K, 87.5%	12.50	6.36	29.28	12.35	10.93
64K, 50%	12.50	14.55	2.17	3.96	8.96
64K, 87.5%	12.50	25.45	13.61	24.70	21.85
Workload C: 16K/32K/64K, 50%/87.5%, 50 Gbps cap					
16K, 50%	8.33	2.60	3.28	5.76	4.97
16K, 87.5%	8.33	4.55	19.45	7.62	6.58
32K, 50%	8.33	5.19	2.42	6.64	7.03
32K, 87.5%	8.33	9.09	14.36	10.78	9.30
64K, 50%	8.33	10.39	1.44	3.96	8.96
64K, 87.5%	8.33	18.18	9.04	15.24	13.15

Table A10 reports the measured S3Agg-LW-cap TTFT for the exact allocation configurations in Table A9. Table A12 sums those per-request measurements into the total TTFT used by the three scheduling workloads.

Table A10. Measured S3Agg-LW-cap TTFT for the exact allocation configurations in Table A9.

Workload	Policy	Request	Rate (Gbps)	Avg. TTFT (ms)	Stdev (ms)
Workload A: 16K + 64K at 50% and 87.5%, 80 Gbps cap					
A	Equal	16K, 50%	20.00	1,002.9	3.9
A	Equal	16K, 87.5%	20.00	851.0	4.3
A	Equal	64K, 50%	20.00	8,188.5	29.2
A	Equal	64K, 87.5%	20.00	3,330.7	16.3
A	KV-prop	16K, 50%	5.82	1,592.2	1.1
A	KV-prop	16K, 87.5%	10.18	1,577.9	6.4
A	KV-prop	64K, 50%	23.27	8,104.5	25.7
A	KV-prop	64K, 87.5%	40.73	2,616.0	34.1
A	BW-prop	16K, 50%	7.89	1,213.8	12.9
A	BW-prop	16K, 87.5%	46.85	416.7	0.9
A	BW-prop	64K, 50%	3.48	10,513.9	17.2
A	BW-prop	64K, 87.5%	21.78	3,086.3	3.8
A	Stall-opt	16K, 50%	8.99	1,080.4	2.1
A	Stall-opt	16K, 87.5%	42.25	453.5	2.0
A	Stall-opt	64K, 50%	3.96	9,312.4	4.3
A	Stall-opt	64K, 87.5%	24.81	2,754.5	14.3
A	Cal. Stall-opt	16K, 50%	13.99	1,001.8	4.4
A	Cal. Stall-opt	16K, 87.5%	27.25	644.7	0.3
A	Cal. Stall-opt	64K, 50%	8.96	8,531.7	3.5
A	Cal. Stall-opt	64K, 87.5%	29.81	2,636.3	5.2
Workload B: 16K + 64K at 50% and 87.5%					
B	Equal	16K, 50%	12.50	1,021.7	5.1
B	Equal	16K, 87.5%	12.50	1,308.8	5.8
B	Equal	64K, 50%	12.50	8,448.4	8.1
B	Equal	64K, 87.5%	12.50	5,139.7	14.3
B	KV-prop	16K, 50%	3.64	2,489.4	5.2
B	KV-prop	16K, 87.5%	6.36	2,467.9	1.5
B	KV-prop	64K, 50%	14.55	8,541.0	189.2
B	KV-prop	64K, 87.5%	25.45	2,714.4	19.4
B	BW-prop	16K, 50%	4.93	1,861.4	1.9
B	BW-prop	16K, 87.5%	29.28	608.8	3.0
B	BW-prop	64K, 50%	2.17	16,443.2	7.9
B	BW-prop	64K, 87.5%	13.61	4,751.4	12.2
B	Stall-opt	16K, 50%	8.99	1,082.5	2.7
B	Stall-opt	16K, 87.5%	12.35	1,327.8	24.1
B	Stall-opt	64K, 50%	3.96	9,319.5	30.2
B	Stall-opt	64K, 87.5%	24.70	2,788.1	31.5
B	Cal. Stall-opt	16K, 50%	8.26	1,166.8	5.9
B	Cal. Stall-opt	16K, 87.5%	10.93	1,473.3	3.2
B	Cal. Stall-opt	64K, 50%	8.96	8,524.8	10.8
B	Cal. Stall-opt	64K, 87.5%	21.85	3,090.4	24.8

Table A11. Measured S3Agg-LW-cap TTFT for the exact allocation configurations in Table A9.

Workload	Policy	Request	Rate (Gbps)	Avg. TTFT (ms)	Stdev (ms)
Workload C: 16K + 32K + 64K at 50% and 87.5%, 50 Gbps cap					
C	Equal	16K, 50%	8.33	1,167.9	17.2
C	Equal	16K, 87.5%	8.33	1,914.6	12.3
C	Equal	32K, 50%	8.33	2,626.0	24.4
C	Equal	32K, 87.5%	8.33	3,785.7	23.6
C	Equal	64K, 50%	8.33	8,302.2	34.6
C	Equal	64K, 87.5%	8.33	7,565.0	51.8
C	KV-prop	16K, 50%	2.60	3,434.8	2.5
C	KV-prop	16K, 87.5%	4.55	3,406.4	0.1
C	KV-prop	32K, 50%	5.19	3,563.8	6.5
C	KV-prop	32K, 87.5%	9.09	3,475.9	10.2
C	KV-prop	64K, 50%	10.39	8,216.7	455.0
C	KV-prop	64K, 87.5%	18.18	3,678.9	53.5
C	BW-prop	16K, 50%	3.28	2,757.7	11.8
C	BW-prop	16K, 87.5%	19.45	878.9	0.9
C	BW-prop	32K, 50%	2.42	7,357.9	5.1
C	BW-prop	32K, 87.5%	14.36	2,274.2	20.8
C	BW-prop	64K, 50%	1.44	24,447.9	54.1
C	BW-prop	64K, 87.5%	9.04	6,981.1	25.6
C	Stall-opt	16K, 50%	5.76	1,609.5	3.5
C	Stall-opt	16K, 87.5%	7.62	2,067.5	4.1
C	Stall-opt	32K, 50%	6.64	2,841.8	12.6
C	Stall-opt	32K, 87.5%	10.78	2,949.3	9.7
C	Stall-opt	64K, 50%	3.96	9,301.6	28.3
C	Stall-opt	64K, 87.5%	15.24	4,299.3	52.1
C	Cal. Stall-opt	16K, 50%	4.97	1,850.0	4.9
C	Cal. Stall-opt	16K, 87.5%	6.58	2,381.4	2.0
C	Cal. Stall-opt	32K, 50%	7.03	2,746.2	1.2
C	Cal. Stall-opt	32K, 87.5%	9.30	3,403.5	25.8
C	Cal. Stall-opt	64K, 50%	8.96	8,231.0	468.1
C	Cal. Stall-opt	64K, 87.5%	13.15	4,892.7	1.5

Table A12. Total TTFT for the scheduling workloads using the measured S3Agg-LW-cap configurations in Tables A10 and A11. The baseline is the sum of the same request configurations without an effective rate limit.

Workload	Policy	Total TTFT (ms)	No-limit base (ms)	ΔTTFT (ms)
A	Equal	13,373.0	12,084.2	+1,288.8
A	KV-prop	13,890.5	12,084.2	+1,806.3
A	BW-prop	15,230.7	12,084.2	+3,146.5
A	Stall-opt	13,600.9	12,084.2	+1,516.7
A	Cal. Stall-opt	12,814.5	12,084.2	+730.3
B	Equal	15,918.6	12,084.2	+3,834.4
B	KV-prop	16,212.6	12,084.2	+4,128.5
B	BW-prop	23,664.8	12,084.2	+11,580.6
B	Stall-opt	14,517.9	12,084.2	+2,433.7
B	Cal. Stall-opt	14,255.3	12,084.2	+2,171.1
C	Equal	25,361.3	15,616.1	+9,745.3
C	KV-prop	25,776.4	15,616.1	+10,160.3
C	BW-prop	44,697.7	15,616.1	+29,081.6
C	Stall-opt	23,069.1	15,616.1	+7,453.0
C	Cal. Stall-opt	23,505.0	15,616.1	+7,888.9