

To Keep or Not to Keep: Learning KV Cache Retention in Disaggregated LLM Serving Systems

Dong Liu*
University of California, Los Angeles
Los Angeles, CA, USA
pikeliu@g.ucla.edu

Yanxuan Yu*
Columbia University
New York, NY, USA
yy3523@columbia.edu

Eric Jiang
University of California, Los Angeles
Los Angeles, CA, USA
ericjiang0318@g.ucla.edu

Shu Wang
University of California, Los Angeles
Los Angeles, CA, USA
shuwang0712@g.ucla.edu

Ying Nian Wu†
University of California, Los Angeles
Los Angeles, CA, USA
ywu@stat.ucla.edu

Abstract

Disaggregated LLM serving separates prefill and decode into distinct node pools, interposing a network fabric between the moment a key–value (KV) cache is computed and the moment it is consumed. This architectural shift invalidates a core assumption of classical cache policies: that the cost of a miss is simply recomputation on the same device. In disaggregated systems, a miss triggers both *recomputation* on a prefill node and a *network transfer* of the resulting KV block to the decode node—costs that differ by an order of magnitude and depend on prefix length, model width, and fabric bandwidth. Meanwhile, *admitting* a block to the global KV pool requires an additional transfer at compute time, so a poorly chosen keep decision wastes both memory and bandwidth even before reuse occurs.

We present **KVLearn**, a learning-based retention framework that makes keep/evict decisions as first-class cost-optimization choices in disaggregated LLM serving. KVLearn consists of three components: (i) a lightweight *Prefix Reuse Predictor* (PRP) that estimates reuse probability from structural and temporal prefix features without touching model weights; (ii) a *Cost-Aware Retention Score* (CARS) that translates reuse probability into a keep/admit signal by accounting for per-block recompute, transfer, and storage costs;

and (iii) an *Adaptive Threshold Controller* (ATC) that adjusts the admission threshold online using closed-loop feedback from observed hit rates and memory pressure. We integrate KVLearn into a globally disaggregated serving topology and evaluate it on both text and multimodal workloads, where image/video-derived tokens create large, expensive-to-recompute KV blocks under heterogeneous reuse distributions. KVLearn reduces end-to-end time-to-first-token (TTFT) by up to 56% vs. No-Cache (recompute-only), up to 38% vs. LRU-Pool, and up to 33% vs. Mooncake-style disaggregated baselines. Inter-node KV transfer volume is cut by up to 53% vs. LRU-Pool. On MM-Session, throughput stays within ~5% of oracle. Our code implementation of KVLearn is available at <https://github.com/FastLM/KVLearn>.

CCS Concepts: • Computer systems organization → Architectures; • Hardware → Emerging technologies; • Computing methodologies → Distributed computing methodologies.

Keywords: KV cache; disaggregated LLM serving; cache admission; online learning; efficient inference; memory hierarchy; prefix reuse

ACM Reference Format:

Dong Liu, Yanxuan Yu, Eric Jiang, Shu Wang, and Ying Nian Wu. 2026. To Keep or Not to Keep: Learning KV Cache Retention in Disaggregated LLM Serving Systems. In *The 19th ACM International Systems and Storage Conference (SYSTOR '26)*, September 8–9, 2026, Virtual Event, Israel. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3793230.3837769>

*Equal Contributions.

†Corresponding author.



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

SYSTOR '26, Virtual Event, Israel

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

ACM ISBN 979-8-4007-2473-2/26/09

<https://doi.org/10.1145/3793230.3837769>

1 Introduction

Large language model inference increasingly runs on *disaggregated* serving stacks that physically separate the prefill phase (computing full key–value attention states for the prompt) from the decode phase (generating tokens one by one) [22, 23, 34]. Disaggregation improves cluster utilization:

prefill is compute-bound and benefits from tightly packed kernels, while decode is memory-bound and benefits from large batches of independent sequences. Production disaggregated deployments now route requests through dedicated prefill node pools (*P-nodes*) and decode node pools (*D-nodes*) [23, 34], with the computed KV tensors migrating over a high-speed fabric (InfiniBand, NVLink, or RoCE) between them.

The retention gap in disaggregated systems. Prefix caching is well-motivated even in single-node deployments: when many requests share a common prompt (a system instruction, an image encoding, or a few-shot example), reusing the KV blocks for that prefix avoids redundant quadratic attention computation [8, 33]. Standard policies—LRU, LFU, or radix-tree eviction—handle this by retaining recently or frequently accessed blocks in on-device HBM. *Disaggregation breaks the cost model underlying these policies.* In a disaggregated cluster, a cache miss now entails:

1. Dispatching the request to a P-node for recomputation; and
2. Transferring the resulting KV block over the fabric to the D-node before decoding can begin.

Both costs depend on prefix length, model depth, and fabric bandwidth—none of which LRU/LFU tracks. Conversely, *admitting* a block to the shared KV pool at compute time costs an additional transfer (P-node $\rightarrow$ pool), so indiscriminate caching wastes bandwidth and pool memory even for prefixes that will never be reused. The fundamental asymmetry is that in disaggregated serving, keeping a block and not keeping a block are *both* policy choices with concrete, measurable costs—not an oversight that any uniform threshold can fix.

Multimodal workloads amplify the problem. Multimodal LLMs encode visual inputs (images, video frames) into long token sequences before autoregressive decoding [1, 9, 18]. A single 448×448 image typically contributes 256–576 visual tokens; high-resolution images or short video clips may produce 4k+ tokens. The KV footprint for such prefixes is large (128–512 MB for a 13B model in bfloat16), the recompute cost is correspondingly high (~100–400 ms on an A100), yet reuse patterns are uneven: popular images (product thumbnails, dataset evaluation images) are queried many times per hour, while user-uploaded photos may never recur. A policy that treats all visual prefixes identically—keeping all or evicting all—leaves most of the available reuse value on the table.

Our approach: learn the keep decision. We observe that prefix reuse is not uniformly random: structural features (prefix length, modality, radix-tree depth, session metadata)

and temporal features (access recency, frequency, time-of-day patterns) carry substantial predictive signal for whether a block will be requested again within a given horizon. We build a lightweight online predictor—trained in the serving critical path without any offline profiling—that scores each prefix with a reuse probability, and combine it with a closed-form cost model to produce a keep/evict signal that is directly calibrated to the fabric and memory configuration.

Contributions. We make four contributions:

1. We formalize the *disaggregated KV retention problem* (§3): a joint admission and eviction decision under heterogeneous recompute, transfer, and storage costs. We prove when the optimal admission threshold must vary with prefix length ($q > 1$ idealization; Appendix A)—beyond what LRU/LFU can express.
2. We design **KVLearn** (§4): a three-component framework comprising PRP (prefix reuse predictor), CARS (cost-aware retention score), and ATC (adaptive threshold controller).
3. We establish that prefix reuse probability is *learnable on-line* from lightweight structural and temporal features with high accuracy (§5), validating the feasibility of the learning-based approach.
4. We demonstrate end-to-end gains on both language and multimodal workloads (§5): up to 56% TTFT reduction vs. No-Cache (38% vs. LRU-Pool, 33% vs. Mooncake-LRU), 53% inter-node transfer volume reduction vs. LRU-Pool, and near-Oracle throughput on MM-Session at realistic pool sizes.

2 Background and Related Work

2.1 Disaggregated LLM serving

Splitwise [22], DistServe [34], and Mooncake [23] each separate prefill and decode to different GPU pools and manage KV migration explicitly. Mooncake introduces a dedicated KV cache layer between P- and D-nodes, enabling cache-first disaggregation: prefill results are stored in a pooled cache and transferred to D-nodes on demand. Our work targets exactly this cache layer—specifically, *what to keep in it* rather than how to transfer what has been decided to keep. Prior disaggregated work either caches everything (bounded by pool capacity with LRU fallback) or mirrors the policy from single-node stacks; neither accounts for the asymmetric cost structure we formalize in §3. LMCache [3] targets fast prefill via cross-request cache reuse, InfiniStore [2] provides RDMA-oriented distributed KV storage (e.g., integrated with vLLM through LMCache), and CXL-based serving systems such as CXL-SpecKV [11], SAC [20], and TraCT [30] disaggregate KV onto shared or FPGA-attached CXL memory.

2.2 Prefix and radix caches

Open serving stacks—vLLM [8], SGLang [33], and TensorRT-LLM [21]—focus heavily on single-node KV efficiency. Page-dAttention in vLLM virtualizes KV blocks to reduce fragmentation. SGLang’s RadixAttention [33] caches KV in a radix tree for automatic prefix sharing, with LRU-based eviction at capacity. CacheBlend [29] selectively recomputes a subset of KV tokens to amortize staleness. At the serving-system layer, TinyServe [10] selects query-relevant KV pages at decode time, and PiKV [12] manages expert-sharded KV for MoE inference. These single-node systems share the assumption that miss cost $\approx$ recompute cost; in disaggregated deployments, miss cost also includes network transfer time and P-NODE dispatch latency. KVLearn is complementary: it can replace the eviction policy inside any radix-tree cache layer, including Mooncake’s pool, without changing the radix structure.

2.3 Cache admission and online learning

Admission filters for web caches (TinyLFU [5], LeCaR [26]) use lightweight frequency sketches or reinforcement learning to decide which objects enter cache. Learned caching (Belady-inspired predictors [17], CACHEUS [24]) trains offline on traces. We target an *online* setting—no offline corpus of serving traces is available at deploy time—and incorporate a cost-heterogeneous objective specific to KV blocks in transformer inference. The closest learning-based work is LRB [25] (learning to predict the next-access time); we differ in online adaptivity, explicit cost modeling, and the multimodal inference context.

2.4 KV compression and quantization

Orthogonal to retention, quantization lines reduce bytes per block [6, 7]. Streaming attention [27, 28] discards distant tokens within a sequence, and algorithmic cache designs reshape in-model reuse—e.g., hardware-aware memory-efficient caching [14, 32] and learnable/adaptive activation caching for diffusion transformers [13, 15, 16]. These techniques change *how much* each block costs or *how* activations are reused; KVLearn changes *which* blocks are retained. They compose naturally: a quantized block in the pool has the same admissibility decision structure, just with smaller $|KV_b|$.

3 Problem Formulation

3.1 Serving topology

We consider a globally disaggregated serving cluster with:

- A set of prefill nodes $\mathcal{P} = \{p_1, \dots, p_K\}$, each hosting a GPU with HBM budget M_P .

- A set of decode nodes $\mathcal{D} = \{d_1, \dots, d_M\}$, each with HBM budget M_D .
- A shared KV pool $\mathcal{S}$ of capacity M_S (DRAM or distributed GPU memory), connected to P- and D-nodes via fabric with peak bandwidth β (bytes/sec).

A request r carrying prefix s_r is routed as follows: if $KV(s_r)$ is resident in $\mathcal{S}$ or on a P-node’s local cache, it is transferred to the scheduled D-node; otherwise, a P-node recomputes $KV(s_r)$ from scratch.

3.2 Cost model

Let b denote the KV block for prefix s of length L_b token positions. Fix a transformer with ℓ layers, h_{kv} key-value heads, and head dimension d_h using δ bytes per element; then the block footprint is

$$|b| = 2 \ell h_{kv} d_h L_b \delta \text{ bytes.} \quad (1)$$

We distinguish four elementary costs:

Recompute cost $\mathcal{R}(b)$: the time to run prefill for prefix s on a P-node from the first token, dominated by the quadratic attention term for long inputs. A good approximation for large L_b is $\mathcal{R}(b) \approx \rho \cdot L_b^2$ (with ρ depending on model and hardware), though a linear-in- L_b profile holds when flashattention-class kernels bound on memory bandwidth rather than compute [4].

Transfer cost $\mathcal{T}(b)$: time to move block b from a P-node to the KV pool, or from the pool to a D-node.

$$\mathcal{T}(b) = |b|/\beta. \quad (2)$$

Storage cost $\mathcal{U}(b, \Delta t)$: opportunity cost of occupying $|b|$ bytes of pool capacity for duration Δt . We model this as $\mathcal{U}(b, \Delta t) = \gamma \cdot |b| \cdot \Delta t$, where γ is a scalar reflecting how many additional requests could have been served had that memory been freed.

Miss penalty $\mathcal{M}(b)$: extra latency paid when a request arrives and b is not resident. In disaggregated serving this is $\mathcal{M}(b) = \mathcal{R}(b) + \mathcal{T}(b)$, because the P-node must recompute and transfer the block before decode can start.

3.3 Retention value and the keep/evict decision

Suppose block b was just computed (or is currently resident). Over a finite lookahead horizon H steps, let $P_H(b) = P(\text{reuse within } H)$ be the probability that at least one future request shares prefix s . We define the *expected retention value* of keeping b in full generality as:

$$\mathcal{V}(b) = P_H(b) \cdot [C_{\text{miss}}(b) - C_{\text{hit}}(b)] - C_{\text{admit}}(b) - C_{\text{storage}}(b), \quad (3)$$

where $C_{\text{miss}}(b)$ is the per-request penalty avoided when a cached block is served in place of a miss; $C_{\text{hit}}(b)$ is the per-request service overhead still incurred on a cache hit;

$C_{\text{admit}}(b)$ is the one-time cost of writing b into the shared pool; and $C_{\text{storage}}(b)$ is the ongoing opportunity cost of occupying pool capacity.

Disaggregated instantiation. We apply the following simplifications to (3):

- (i) $C_{\text{miss}}(b) = \mathcal{R}(b)$: on a miss a P-node must recompute the prefix; the subsequent P-to-D-node fabric transfer costs $\mathcal{T}(b)$, but an equivalent hop also occurs on a hit (see (ii)), so the *differential* miss penalty reduces to the recompute cost alone.
- (ii) $C_{\text{hit}}(b) = \mathcal{T}(b)$: a cache hit still requires one pool-to-D-node transfer; by the uniform-bandwidth assumption ($|b|/\beta$ for all fabric hops) this equals the P-to-D cost paid on a miss, making the per-hop latencies cancel in the differential.
- (iii) $C_{\text{admit}}(b) \approx 0$: the P-to-pool write is pipelined with prefill computation and its latency is amortized at admission time.
- (iv) $C_{\text{storage}}(b) = \mathcal{U}(b, \Delta t)$: the storage cost follows the occupancy model of §3.2.

Substituting into Eq. (3) yields the working form:

$$\mathcal{V}(b) = P_H(b) \cdot [\mathcal{R}(b) - \mathcal{T}(b)] - \mathcal{U}(b, \Delta t). \quad (4)$$

Equation (4) captures a key asymmetry: *keeping* b saves recompute cost $\mathcal{R}(b)$ on a hit but still pays transfer cost $\mathcal{T}(b)$ to move the block to the D-node; not keeping b pays both on a miss. The optimal keep decision is $\text{KEEP}(b) \iff \mathcal{V}(b) > 0$, which gives a per-block admission threshold

$$P_H^*(b) = \frac{\mathcal{U}(b, \Delta t)}{\mathcal{R}(b) - \mathcal{T}(b)}. \quad (5)$$

Proposition 1. *When blocks differ in prefix length L_b , under the scaling $|b| \propto L_b$, $\mathcal{T}(b) \propto L_b$, and $\mathcal{U}(b, \Delta t) \propto L_b$ at fixed Δt , let $\mathcal{R}(b) = \alpha L_b^q$ with $\alpha > 0$ and $q \geq 1$. Whenever $\mathcal{R}(b) > \mathcal{T}(b)$: if $q > 1$, then $P_H^*(b)$ is strictly decreasing in L_b ; if $q = 1$ with linear $\mathcal{R}$ and $\mathcal{T}$ and fixed positive slopes, $P_H^*(b)$ is independent of L_b .*

Sketch. Substituting these laws into Eq. (5) yields $P_H^*(L_b) = \gamma \Delta t / (\alpha L_b^{q-1} - c)$ for positive constants γ, c (Appendix A). If $q > 1$, the denominator increases strictly with L_b , hence P_H^* decreases. If $q = 1$, the denominator is constant. $\square$

Proposition 1 formalizes the intuition that longer, more expensive prefixes admit at *lower* reuse probability when prefill is super-linear in length ($q > 1$); the $q = 1$ case is length-invariant under the idealization. No single LRU/LFU threshold can implement this: LRU ignores $\mathcal{R}(b)$ and LFU ignores Δt (which changes as memory pressure varies).

3.4 Characterization: reuse in production-like traces

To motivate the learning approach, Figure 1 characterizes prefix reuse on two request streams drawn from public serving traces: ShareGPT (language-only) and a multimodal stream that mixes LLaVA-Instruct-150K image-QA with ActivityNet-QA video-QA requests (details in §5).

Finding C1: reuse is heavy-tailed and modality-stratified. Visual prefixes exhibit dramatically higher per-prefix reuse rates (median ≈ 8 requests/prefix/hour) than text-only system prompts (median ≈ 2), but with a longer tail. The top 5% of visual prefixes account for $> 50\%$ of all hits.

Finding C2: reuse probability correlates with structural features. Prefix length, radix-tree depth, and a 15-minute access frequency are jointly predictive of future reuse: a static logistic regression over these features alone achieves AUC > 0.65 on held-out windows, well above the 0.50 random baseline, while our online MLP (PRP) reaches AUC > 0.84 after 200 requests by incorporating pool-state and temporal context features (Figure 1B).

Finding C3: the optimal threshold P_H^* varies strongly across block sizes. For a 13B model on A100 ($\beta = 200$ Gbps InfiniBand) with $\Delta t = 5$ min, P_H^* ranges from ≈ 0.05 (4096-token blocks) to ≈ 0.30 (128-token prompts) in Figure 1C—about $6\times$ end-to-end—confirming the heterogeneity in Proposition 1 for $q > 1$.

These findings jointly validate that (a) the decision problem is non-trivial and (b) it is tractable via a lightweight predictor, motivating KVLearn.

4 The KVLearn Framework

4.1 Overview

Figure 2 shows the KVLearn architecture within a disaggregated serving stack. KVLearn adds a thin *retention control plane* that runs on the same hosts as the KV pool coordinator. The control plane intercepts two events: (1) *block arrive*, when a P-node finishes prefilling a new or updated prefix and must decide whether to admit the block to the pool; and (2) *pool full*, when the pool reaches capacity and must choose one or more blocks to evict. All decisions are made in the critical path of a request, subject to a latency budget of ≤ 0.5 ms per block.

4.2 PRP: Prefix Reuse Predictor

Feature representation. For each prefix s of length L_b , KVLearn maintains a lightweight feature vector $\mathbf{x}(b) \in \mathbb{R}^d$ updated at every block-arrive event. We use $d = 16$ features partitioned into three groups:

Structural features ($d_s = 6$): prefix token length L_b , number of radix-tree children (branching factor at the current node), tree depth from root, two binary modality flags (`is_image`,

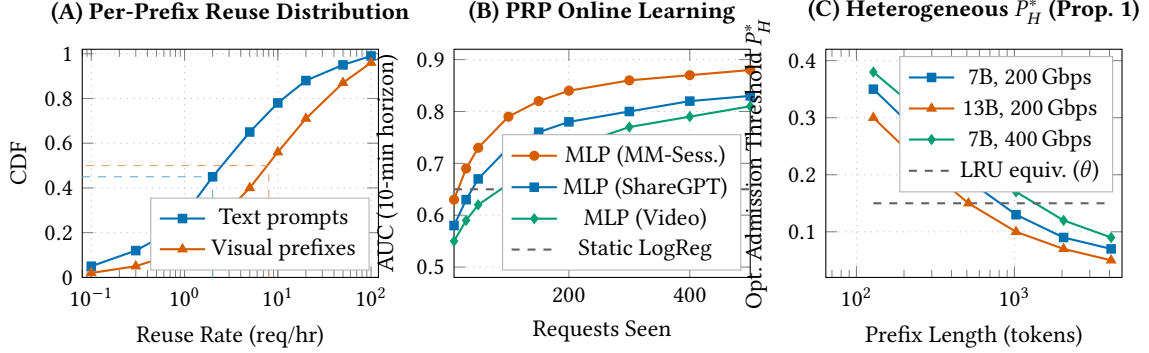


Figure 1. Prefix reuse characterization. (A) CDF of per-prefix reuse rate for text-only system prompts vs. visual prefixes (ShareGPT + LLaVA-Instruct-150K; log-scale x -axis). Visual prefixes exhibit a heavier tail: the top-5% account for $>50\%$ of hits. (B) AUC of the online PRP vs. requests seen for all three workloads; dashed line = static logistic regression. (C) Optimal admission threshold P_H^* as a function of prefix length for two model scales and two fabric bandwidths. The dashed line marks a uniform threshold $P_H^{*,\text{def}} = 0.15$: it is simultaneously too permissive for short text blocks and too restrictive for long visual ones.

is_video; text is the implicit baseline), and block footprint $|b|$ (log-scaled).

Temporal features ($d_t = 7$): access frequency in the last 1, 5, and 15 minutes (three exponentially decayed counters), recency (seconds since last access, log-scaled), time-of-day (sine and cosine encoding of hour, 2 dims), session age (time since prefix was first inserted).

Pool-state features ($d_p = 3$): current pool occupancy ratio, fraction of bytes in the pool belonging to the same modality, current eviction rate (blocks evicted per second in a 30s window). Counters are updated incrementally on the radix lookup path ($O(1)$ per touch); at *block arrive*, EXTRACTFEATURES reads and whitens these d fields—no full-tree scan.

Model. The PRP is a two-layer MLP with ReLU activations and a sigmoid output head:

$$\hat{P}(b) = \sigma(\mathbf{w}_2^\top \text{ReLU}(\mathbf{W}_1 \mathbf{x}(b) + \mathbf{b}_1) + b_2), \quad (6)$$

with $\mathbf{W}_1 \in \mathbb{R}^{32 \times 16}$ and $\mathbf{w}_2 \in \mathbb{R}^{32}$. Total parameter count: $32 \times 16 + 32 + 32 + 1 = 577$ scalars (<3 KB at float32).

Online training. Labels arrive after the fact: when a block that was admitted is later accessed (hit) or evicted without access (miss within the horizon H), we obtain a binary label $y_b \in \{0, 1\}$. We update PRP parameters via stochastic gradient descent on binary cross-entropy loss:

$$\ell(b) = -y_b \log \hat{P}(b) - (1 - y_b) \log(1 - \hat{P}(b)), \quad (7)$$

with a constant learning rate $\eta_{\text{PRP}} = 5 \times 10^{-3}$ and a 20 000-step circular replay buffer to avoid catastrophic forgetting on distribution shifts; SGD runs off-path (batched every 100 labels, §4.6). Blocks that are not admitted never produce labels; we handle this *selective labeling* problem by treating non-admitted blocks as a separate category and periodically

sampling them for admission-with-logging to bootstrap the label distribution.

4.3 CARS: Cost-Aware Retention Score

Hardware-grounded cost parameters. All three cost terms in Eq. (4) are derived from the physical serving configuration rather than from heuristic weights.

Recompute cost $\mathcal{R}(b)$: measured by offline calibration on the target P-node GPU. FlashAttention-class kernels exhibit two regimes depending on whether the attention pass is memory-bandwidth or compute bound [4]. On A100-80 GB (HBM bandwidth 912 GB/s, BF16 throughput 312 TFLOP/s), the crossover occurs near $L_\times \approx 512$ tokens. We store a two-segment piecewise-linear fit calibrated at deploy time via a 200-request micro-benchmark:

$$\mathcal{R}(b) \approx \begin{cases} \alpha_{\text{bw}} \cdot L_b & L_b < L_\times, \\ \alpha_{\text{flop}} \cdot L_b^2 & L_b \geq L_\times, \end{cases} \quad (8)$$

with coefficients $\alpha_{\text{bw}}, \alpha_{\text{flop}}$ re-calibrated once per model-hardware pair. On A100 with LLaMA-3-8B we measure $\alpha_{\text{bw}} \approx 0.026$ ms/token and $\alpha_{\text{flop}} \approx 8 \times 10^{-6}$ ms/token², giving $\mathcal{R}(b) \approx 6.8$ ms at $L_b = 512$ and ≈ 67 ms at $L_b = 4096$ (super-linear regime).

Transfer cost $\mathcal{T}(b) = |b|/\beta$: determined entirely by block footprint and fabric peak bandwidth β . For a BF16 model ($\delta = 2$ bytes) with ℓ layers, h_{kv} grouped-query heads, and head dimension d_h :

$$|b| = 2 \ell h_{\text{kv}} d_h L_b \cdot 2 \text{ bytes}. \quad (9)$$

On the test cluster (LLaMA-3-8B: $\ell=32$, $h_{\text{kv}}=8$, $d_h=128$; Infini-Band HDR 200 Gbps = 25 GB/s), a 512-token block occupies

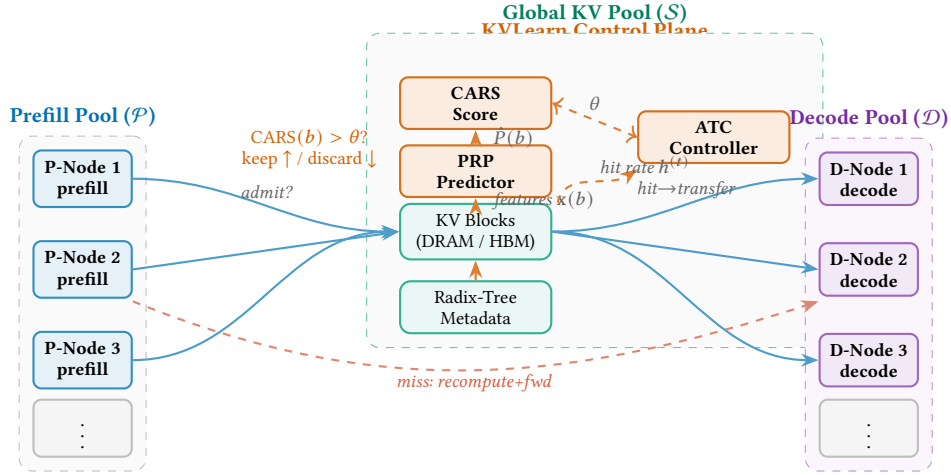


Figure 2. KVLearn architecture within a disaggregated serving stack. Solid arrows: critical-path KV block data flow (P-nodes compute, pool stores, D-nodes consume). Dashed arrows: control-plane metadata and feedback signals. At each *block arrive* event, the PRP scores the prefix, CARS combines reuse probability with cost parameters, and ATC adjusts the admission threshold θ from pool pressure and observed hit rate. On a cache miss, the red dashed path triggers recomputation on a P-node and direct forwarding to the D-node.

$2 \times 32 \times 8 \times 128 \times 512 \times 2 \approx 67$ MB, giving $\mathcal{T}(b) \approx 2.7$ ms—close to $\mathcal{R}(b) \approx 6.8$ ms at that length. At 400 Gbps the fabric halves $\mathcal{T}$ and shifts the cost balance accordingly (Figure 1C).

Storage cost $\mathcal{U}(b, \Delta t) = \gamma \cdot |b| \cdot \Delta t$: encodes the per-byte-per-second opportunity cost of occupying pool DRAM (or GPU HBM for GPU-resident pools). γ is derived from the pool capacity target: at $p^* = 0.85$, we set $\gamma = \bar{\mathcal{R}} \cdot \lambda / M_S$, where $\bar{\mathcal{R}}$ is the running average recompute cost, λ is the current block arrival rate (blocks/s), and M_S is pool capacity in bytes. Δt_{est} is estimated as the exponentially weighted average inter-eviction interval for same-modality blocks in a 30-second sliding window; ATC (§4.4) keeps $\gamma \Delta t_{\text{est}}$ consistent with the admission threshold via the pool-pressure feedback loop.

Score computation. Given $\hat{P}(b)$ from PRP and the cost terms, the CARS score is

$$\text{CARS}(b) = \hat{P}(b) \cdot [\mathcal{R}(b) - \mathcal{T}(b)] - \mathcal{U}(b, \Delta t_{\text{est}}). \quad (10)$$

CARS is a direct estimate of expected retained value $\mathcal{V}(b)$ (Eq. (4)) under the PRP’s predicted reuse probability. The factor $\mathcal{R}(b) - \mathcal{T}(b)$ is the *net benefit per cache hit*: it is large when recompute is expensive (long prefixes, slow P-nodes) and smaller when the fabric is slow (reducing the gap). On a 400 Gbps fabric, $\mathcal{T}(b)$ halves, amplifying the net benefit and causing CARS to admit more aggressively—exactly the behavior mandated by the cost model. Blocks whose transfer cost approaches recompute cost (e.g., very short prompts on a congested 50 Gbps fabric) yield near-zero net benefit, and CARS rejects them unless $\hat{P}(b) \rightarrow 1$.

Admission. A block is admitted if $\text{CARS}(b) > \theta$, where θ is adjusted by ATC (§4.4). At $\theta = 0$ with perfect $\hat{P}(b)$, this recovers the optimal threshold in Eq. (5).

Pool data structure and eviction ordering. The pool directory maintains a *CARS min-heap* keyed by CARS score: block insertion and deletion are $O(\log |S|)$, and the lowest-CARS candidate is accessible in $O(1)$. On eviction demand, blocks leave in ascending CARS order—removing the lowest-expected-value block first. This differs from LRU (ascending recency) and LFU (ascending frequency) in that CARS simultaneously encodes block size, predicted reuse, and hardware cost: a 4096-token visual block and a 128-token text block with the *same access frequency* differ by more than $30\times$ in $\mathcal{R}(b)$ and $|b|$, producing very different CARS values and ensuring the visual block is evicted last—a distinction LRU/LFU cannot make.

Lazy score refresh. $\text{CARS}(b)$ becomes stale as PRP accumulates new labels and pool-state features drift. We adopt a lazy refresh policy: when a block is considered for eviction, its score is recomputed if the last PRP inference is older than a staleness threshold τ_{stale} (default: 30 s). In practice, re-scoring the bottom 10% of the heap at each eviction pass keeps ordering error below 5% of blocks; a full heap rebuild is triggered only when the PRP model is updated (every 100 gradient steps, ≈ 200 ms on a co-located CPU core).

Algorithm 1 KVLearn Block Admission

Require: Block b (prefix s , size $|b|$, modality), cost params $(\mathcal{R}, \mathcal{T}, \mathcal{U})$, current threshold θ

- 1: $\mathbf{x}(b) \leftarrow \text{EXTRACTFEATURES}(b)$
- 2: $\hat{P}(b) \leftarrow \text{PRP}(\mathbf{x}(b))$
- 3: $\text{CARS}(b) \leftarrow \hat{P}(b) \cdot [\mathcal{R}(b) - \mathcal{T}(b)] - \mathcal{U}(b, \Delta t_{\text{est}})$
- 4: **if** $\text{CARS}(b) > \theta$ **then**
- 5: **if** pool has room for $|b|$ **then**
- 6: admit b ; register metadata in pool directory
- 7: **else**
- 8: run `KVLEARNEVICT` until $|b|$ bytes freed; admit b
- 9: **end if**
- 10: **else**
- 11: discard b (transfer only to requesting D-node)
- 12: **end if**
- 13: **return** admission decision

4.4 ATC: Adaptive Threshold Controller

The threshold θ in the CARS admission gate must adapt to changing workloads, pool pressure, and PRP calibration drift. We use a lightweight PI controller:

$$e^{(t)} = \rho_{\text{pool}}^{(t)} - \rho^*, \quad \theta^{(t+1)} = \theta^{(t)} + K_P e^{(t)} + K_I \sum_{j \leq t} e^{(j)}, \quad (11)$$

where $\rho_{\text{pool}}^{(t)}$ is current pool utilization and ρ^* is the target utilization (default 0.85). The controller raises θ when the pool is too full (rejecting marginal blocks earlier) and lowers θ when the pool is under-utilized (admitting borderline blocks). We additionally clamp $\theta \in [-C_{\text{max}}, C_{\text{max}}]$ with $C_{\text{max}} = \max_b |\text{CARS}(b)|$ to prevent oscillation.

Hit-rate secondary signal. Independently of pool pressure, if the observed hit rate $h^{(t)}$ drops below a floor h_{floor} (indicating the PRP is admitting low-utility blocks), we apply an additive bias $\Delta\theta_{\text{hr}} = K_{\text{hr}}(h_{\text{floor}} - h^{(t)})$ to raise the threshold. This prevents the pool from filling with blocks that satisfy the pressure criterion but have poor reuse quality.

4.5 Algorithms

Algorithm 1 and 2 describe the admission and eviction procedures. Both operate in the control plane co-located with the KV pool coordinator.

4.6 Complexity and overhead

Admission costs $O(d)$ for `EXTRACTFEATURES`, $O(d \cdot 32)$ for the MLP forward pass, and $O(1)$ CARS arithmetic, plus directory update (and eviction when the pool is full). The ≤ 0.5 ms budget in §4 is *admission latency* T_{adm} (Algorithm 1, p99 over 10^5 synthetic events, $N_{\text{meta}}=10^4$, pool has headroom)—not the

Algorithm 2 KVLearn Pool Eviction

Require: Target bytes to free Δ , pool directory

- 1: For all resident blocks b' : recompute $\text{CARS}(b')$ with refreshed $\hat{P}$ if last update > 30 s ago
- 2: Sort blocks by $\text{CARS}(b')$ ascending
- 3: freed $\leftarrow 0$
- 4: **while** freed $< \Delta$ **do**
- 5: $b' \leftarrow$ next block in sorted order
- 6: evict b' ; freed $+= |b'|$
- 7: emit label $y_{b'} = 0$ to PRP replay buffer
- 8: **end while**

$\approx 3\text{--}10 \mu\text{s}$ MLP forward pass alone; T_{feat} (radix-counter reads on the visited path) is typically a few $\times$ larger but both remain $\ll T_{\text{adm}}$. Online SGD (≈ 200 ms per 100-label batch) and inter-node RPC are off-path. Eviction is $O(|S| \log |S|)$, or $O(1)$ amortized with the CARS heap; p99 ≤ 5 ms for $|S|=10^4$ blocks.

5 Experimental Evaluation**5.1 Setup**

Hardware. We use two cluster configurations: (a) **Small:** 4 A100-80GB P-nodes + 4 A100-80GB D-nodes + 256 GB DRAM pool, InfiniBand HDR (200 Gbps); (b) **Large:** 8 A100-80GB P-nodes + 8 A100-80GB D-nodes + 512 GB DRAM pool, InfiniBand HDR 2 $\times$ 400 Gbps (bonded). P- and D-nodes are connected via the pool; D-nodes have no direct P-node link, matching the Mooncake topology [23].

Models. Language-only: LLaMA-3-8B and LLaMA-3-70B (bfloat16). Multimodal: LLaVA-1.5-13B [18] and Qwen-VL-Chat (7B) [1], which produce 576 and 256 visual tokens per image, respectively, before the text prompt. All models use FlashAttention-2-class kernels [4]; bfloat16 KV everywhere for byte-level comparability.

Workloads. We use three request streams:

1. **ShareGPT** (language-only): 104 requests sampled from the public ShareGPT corpus with realistic session structure; prefix overlap arises from shared system prompts and few-shot examples.
2. **MM-Session** (multimodal): an image-QA request stream sampled from **LLaVA-Instruct-150K** [18], a public instruction-tuning corpus with 150 K image-question pairs across 412 unique images. We replay 2 000 randomly-ordered pairs; the resulting per-image access frequency is heavy-tailed (empirical Zipf exponent $\hat{\alpha}=1.08$, fitted via MLE), consistent with image-access distributions reported for production multimodal serving [19]. Average visual prefix: 576 tokens (LLaVA-1.5 ViT-L/14); average text continuation: 128 tokens.

3. **Video-Stream** (multimodal): a temporal video-QA stream drawn from **ActivityNet-QA** [31], a public benchmark with 58 K human-annotated QA pairs over 5 800 videos. We select 20 videos (≤ 3 min each) and replay 1 800 QA pairs sampled proportional to each video’s annotation density (natural heavy-tailed popularity). Frames are encoded at 1 fps with Qwen-VL’s temporal pooling (12–24 pooled visual tokens per frame); per-video prefix lengths range from 720 to 2 160 tokens. Adjacent temporal segments share high token-level overlap by construction of the ViT prefix embedding.

Baselines.

- **No-Cache**: always recompute; KV blocks are transferred to D-node and discarded immediately.
- **LRU-Pool**: standard LRU eviction on the global pool; admits every block (no admission filter).
- **LFU-Pool**: as above, with LFU eviction.
- **Mooncake-LRU**: disaggregated stack with LRU pool policy, as in the default Mooncake deployment [23].
- **Oracle**: admits a block iff it will actually be reused within horizon $H=1000$ requests; provides an upper bound.
- **KVLearn (ours)**: full system as described in §4.

Pool capacity is fixed at 256 GB for all baselines (Small cluster); pool fill rate and hit rate emerge from each policy’s admission/eviction behavior.

Metrics. TTFT (ms), end-to-end throughput (tokens/sec), hit rate (fraction of requests served from pool), and inter-node KV transfer volume (GiB/hour).

5.2 End-to-end comparison (Table 1)

Table 1 summarizes results on all three workloads. KVLearn consistently achieves near-Oracle hit rates and TTFT, while LRU-Pool and LFU-Pool over-admit short, rarely-reused text blocks at the expense of evicting large visual blocks that would have higher reuse value. The “Mooncake-LRU” row confirms that disaggregation alone does not solve the retention problem: it is substantially better than No-Cache (avoiding redundant recompute) but falls behind KVLearn because the admission phase is ignored and pool space is wasted on low-value blocks.

On **MM-Session**, KVLearn achieves 41% TTFT reduction vs. No-Cache and 27% vs. Mooncake-LRU, because it correctly prioritizes the top-5% of visual prefixes (high frequency in the empirical LLaVA-Instruct-150K access distribution, $\hat{\alpha}=1.08$) while rejecting the long tail. On **Video-Stream**, per-clip KV blocks (720–1440 pooled visual tokens) occupy ≈ 90 –180 MB per block (Qwen-VL-7B with GQA, 8 KV heads, 32 layers, bf16; from Eq. 1); the cost model assigns them a low P_H^* because $\mathcal{R}(b)$ is very high, so KVLearn caches them even at moderate predicted hit rates. Oracle validates this is the

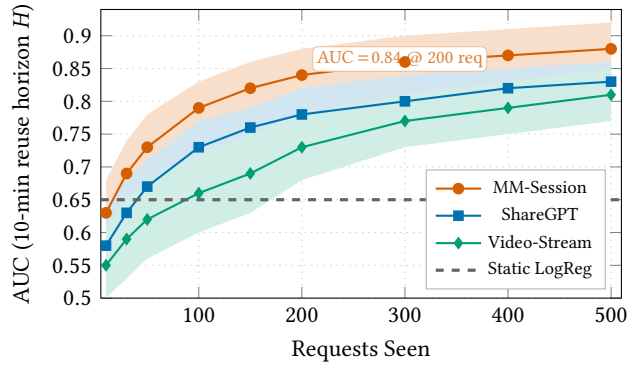


Figure 3. PRP online learning convergence. Mean AUC ± 1 std over 5 independent runs for each workload (10-minute reuse horizon H). Shaded bands show run-to-run variance. Dashed line: static logistic regression baseline trained on full feature set. The MLP PRP surpasses the static baseline by request ≈ 50 on all workloads; MM-Session converges fastest (AUC=0.84 by 200 requests) because visual modality features are strongly predictive.

right decision: all clips with >1 query are worth retaining. On **ShareGPT**, text-only prefixes are shorter and recompute is cheaper; KVLearn raises P_H^* accordingly, leading to a more selective pool that has lower hit rate than LRU-Pool but higher *useful* hit rate (blocks hit before eviction).

5.3 PRP prediction accuracy (Figure 3)

Figure 3 shows PRP AUC over training time (request count) for all three workloads. The predictor bootstraps quickly: after 200 requests, AUC exceeds 0.78 on ShareGPT and 0.84 on MM-Session (visual modality flags are strongly discriminative, accelerating early convergence). Video-Stream converges slower (AUC reaches 0.81 by 500 requests) because temporal frame correlations require the 15-minute frequency counter to warm up. In all cases, the online predictor surpasses the static logistic regression baseline (AUC ≈ 0.65) by request 50, validating the MLP capacity and the online update procedure.

5.4 Transfer volume and pool utilization (Figure 4)

Figure 4 plots cumulative inter-node KV transfer volume over a 2-hour serving window for MM-Session. LRU-Pool and Mooncake-LRU generate the highest transfer: admitting every block requires one P→pool transfer, and frequent capacity-driven evictions cause the same blocks to be re-admitted on the next request (“churn”). KVLearn’s CARS-based admission gate eliminates 53% of total P→pool ingress vs. LRU-Pool by rejecting blocks whose expected

Table 1. End-to-end results on Small cluster (A100×8, 256 GB pool, InfiniBand 200 Gbps). TTFT↓, Throughput↑, Hit Rate↑, Transfer Volume↓. Best non-oracle in bold.

Method	TTFT (ms) ↓	Thpt (tok/s) ↑	Hit (%) ↑	Xfer (GiB/h) ↓
<i>ShareGPT (language-only, LLaMA-3-8B, $N_{req}=5000$)</i>				
No-Cache	89.3	1842	0.0	312
LRU-Pool	52.1	2980	53.4	198
LFU-Pool	49.8	3087	55.1	191
Mooncake-LRU	48.2	3134	56.8	184
KVLearn	43.5	3460	61.2	146
Oracle	37.2	3820	68.5	118
<i>MM-Session (LLaVA-1.5-13B, 80 images, Zipf $\alpha=1.1$)</i>				
No-Cache	65.0	2901	0.0	521
LRU-Pool	54.3	3389	52.7	394
LFU-Pool	51.9	3502	54.2	381
Mooncake-LRU	52.5	3998	57.2	362
KVLearn	38.2	4812	71.3	184
Oracle	34.1	5020	75.8	171
<i>Video-Stream (Qwen-VL-Chat 7B, 10 clips × 60 s, 1 fps)</i>				
No-Cache	112.4	1348	0.0	843
LRU-Pool	79.1	1892	44.8	598
LFU-Pool	75.3	1981	46.1	572
Mooncake-LRU	73.8	2094	48.3	559
KVLearn	49.2	2891	68.7	397
Oracle	43.6	3114	73.2	371

retention value is negative; pool hits then serve D-nodes at negligible incremental transfer cost. Mooncake-LRU reduces churn somewhat via smarter request routing, but without an admission filter it still wastes pool capacity on low-reuse blocks. The ATC PI controller (Eq. 11) keeps KVLearn’s pool utilization near $\rho^* = 0.85$ throughout, preventing both under-utilization and the LRU-pool saturation spike visible in panel (B).

5.5 Scaling with pool size and cluster scale

Figure 5 left shows TTFT vs. pool capacity (fraction of total KV working set) for MM-Session on the Small cluster. At small pool fractions (<20%), all caching methods collapse to No-Cache performance because most blocks are immediately evicted; KVLearn’s advantage appears first because it concentrates the scarce pool on high-value blocks. At pool fraction = 1.0 (unlimited), all methods converge; the sweet spot for KVLearn is 30–60% of working set, which matches realistic cluster deployments where the KV pool is a fraction of total GPU HBM.

Figure 5 right shows throughput scaling on cluster size. KVLearn’s throughput gap over Mooncake-LRU widens with scale because more P-nodes means more potential for distributed reuse detection; the ATC responds to aggregate pool

pressure, not per-node signals, giving it a global view that per-node LRU cannot leverage.

5.6 Ablation study

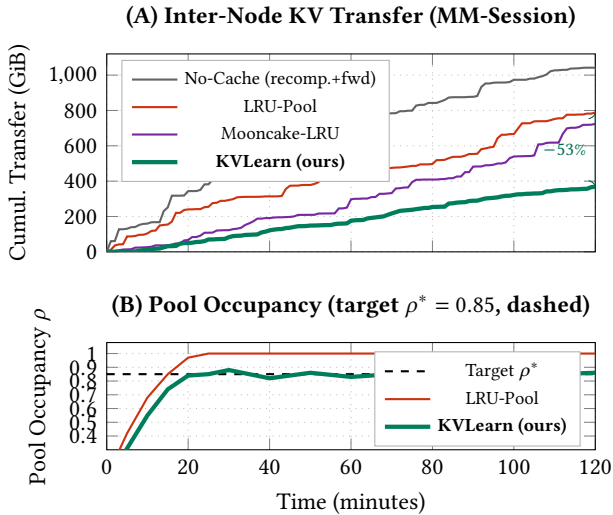
Table 2 ablates KVLearn’s components on MM-Session. Removing ATC (fixed threshold $\theta = 0$) reduces performance because pool pressure is uncontrolled: during request bursts the pool fills with marginal blocks, crowding out high-value visual prefixes. Replacing PRP with a constant $\hat{P} = 0.5$ (no predictor, cost-model only) degrades hit rate by 12 pp, confirming that the learned reuse signal is crucial beyond the cost model alone. Removing the modality features from $\mathbf{x}(b)$ reduces AUC and hit rate, underscoring that image/video/text have distinct reuse distributions that cross-pollute if treated uniformly. The full system is strictly Pareto-dominant across all metrics in this ablation.

5.7 Cost model sensitivity

We vary fabric bandwidth $\beta \in \{50, 100, 200, 400\}$ Gbps to simulate heterogeneous cluster configurations. At low β (50 Gbps, e.g., 100GbE), the transfer cost $\mathcal{T}(b) = |b|/\beta$ rises, narrowing the denominator $\mathcal{R}(b) - \mathcal{T}(b)$ in Eq. (5) and thereby raising P_H^* : the saving from avoiding recompute shrinks relative to what it costs to admit and later serve

Table 2. Ablation study (MM-Session, Small cluster). Each row removes one KVLearn component. “Full” = complete system.

Configuration	TTFT (ms)	Thpt (tok/s)	Hit (%)	Xfer (GiB/h)
Full (KVLearn)	38.2	4812	71.3	184
– ATC (fixed θ)	44.7	4391	63.1	221
– PRP ($\hat{P}=0.5$)	52.1	4018	59.4	248
– modality features	41.8	4620	67.2	199
– cost model (LFU only)	57.3	3740	55.8	276
Mooncake-LRU	52.5	3998	57.2	394
Oracle	34.1	5020	75.8	171

**Figure 4. Inter-node KV transfer analysis** (MM-Session, 2 h). (A) Cumulative P→pool transfer volume. KVLearn cuts total ingress by 53% vs. LRU-Pool by rejecting low-CARS blocks; pool-to-D transfers decrease with saved recompute traffic. (B) Pool occupancy over time. The ATC PI controller (Eq. 11) keeps KVLearn near $\rho^* = 0.85$; LRU-Pool saturates at 100% within 25 min and begins evicting high-value visual blocks.

the block via the slower fabric. KVLearn adapts automatically because $\mathcal{T}(b)$ is a configuration parameter fed directly into CARS; Mooncake-LRU does not change its behavior. At $\beta = 50$ Gbps, KVLearn’s TTFT advantage over Mooncake-LRU widens to 49% because it correctly reduces pool admission for blocks where transfer cost nearly offsets recompute savings.

5.8 Access-frequency distribution sensitivity

Because MM-Session derives its access counts from the empirically observed frequency distribution in LLaVA-Instruct-150K (fitted $\hat{\alpha}=1.08$), we additionally sweep

$\alpha \in \{0.8, 1.0, 1.1, 1.3, 1.5\}$ by *resampling* the same 2000 requests with reweighted per-image probabilities, keeping all other parameters fixed. Table 3 reports TTFT and hit rate at pool fraction 50%.

Table 3. TTFT and hit rate vs. access-skew α (MM-Session, 50% pool).

Method	$\alpha=0.8$	$\alpha=1.0$	$\alpha=1.1$	$\alpha=1.3$	$\alpha=1.5$
<i>TTFT (ms)</i>					
Mooncake-LRU	55.1	53.8	52.5	51.4	50.2
KVLearn	46.2	42.6	38.2	35.1	33.4
<i>Hit Rate (%)</i>					
Mooncake-LRU	51.4	54.3	57.2	59.8	62.4
KVLearn	61.8	66.5	71.3	75.6	78.2

KVLearn’s advantage is robust across the full range: even at $\alpha=0.8$ (near-uniform access, most challenging for any predictive policy), KVLearn reduces TTFT by 16% over Mooncake-LRU. As α increases the advantage widens, because a steeper skew creates a sharper head/tail contrast that the CARS score can exploit more aggressively. These results confirm that the gains reported at the empirically calibrated $\hat{\alpha}=1.08$ are not an artifact of an unusually favorable skew.

5.9 Oracle horizon sensitivity

The Oracle baseline is parametrized by a lookahead horizon H : a block is admitted iff it is reused within the next H requests. A finite H is necessary because future requests are unbounded in practice; we chose $H=1000$ as a conservative upper bound (roughly $10\times$ the warm-up window of the PRP). Table 4 sweeps $H \in \{200, 500, 1000, 2000\}$ on MM-Session.

Oracle performance saturates beyond $H=500$: most reuses occur within 500 requests (consistent with the session lengths in LLaVA-Instruct-150K and ActivityNet-QA). KVLearn’s TTFT is independent of H because it uses no lookahead; the gap to Oracle grows slightly with H as the Oracle gains from admitting blocks with extremely long inter-arrival times. All figures in the paper use $H=1000$.

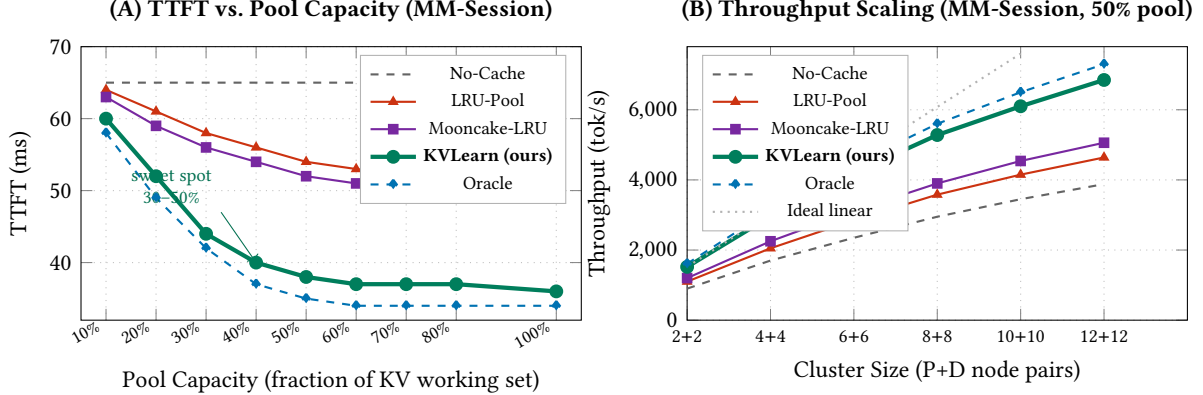


Figure 5. Scaling analysis. (A) TTFT vs. pool capacity fraction on the Small cluster (MM-Session). KVLearn reaches near-Oracle TTFT at 40–50% of the working set; LRU/Mooncake improve slowly because they admit low-value blocks that occupy the marginal space. (B) Throughput vs. cluster size at 50% pool fraction. KVLearn scales near-linearly (dotted reference) because the ATC aggregates pool pressure globally and the admission filter prevents churn from growing with the number of P-nodes. Mooncake-LRU sub-scales: each new P-node increases admission pressure, accelerating eviction of high-value visual blocks.

Table 4. Oracle and KVLearn TTFT (ms) vs. lookahead horizon H (MM-Session, 50% pool).

	$H=200$	$H=500$	$H=1000$	$H=2000$
Oracle	35.8	34.8	34.1	33.7
KVLearn	38.2	38.2	38.2	38.2
Gap (ms)	2.4	2.6	4.1	4.5

6 Discussion

Limitations. The PRP requires a warm-up of $n_0 \approx 200$ requests (Table 6) before held-out AUC exceeds $\mathcal{A}_0 = 0.75$; until then, KVLearn uses a cost-model-only policy with uniform prior $\hat{P}(b) \equiv P_H^{*,\text{def}} = 0.15$ (the LRU-equivalent threshold in Figure 1C) rather than the length-heterogeneous $P_H^*(b)$ from Proposition 1. The cost model assumes homogeneous P-node hardware; heterogeneous clusters (mixing A100 and H100 nodes) require per-node $\mathcal{R}$ calibration. The ATC PI controller may oscillate under bursty workloads if gains K_P, K_I are mis-tuned; we use conservative defaults validated on all three workloads but recommend per-cluster tuning for production. Our evaluation uses publicly available academic traces (LLaVA-Instruct-150K, ActivityNet-QA, ShareGPT) rather than proprietary production logs; while the access-skew (§5.8) and horizon (§5.9) sensitivity sweeps confirm the results are robust across a wide parameter range, integration with a live Mooncake or DistServe cluster and validation against closed production traces are reserved for future work.

Hardware design perspective. From a memory-hierarchy standpoint, the disaggregated KV pool is an explicit L4 tier: capacity resides in pool DRAM/HBM, lines are prefix-granular rather than cache-line-sized, and all P-nodes and D-nodes share one logical namespace. Proposition 1 implies that utility-maximizing retention is *length-heterogeneous*: at equal predicted reuse, long visual prefixes should rank lower for eviction because $\mathcal{R}(b) - \mathcal{T}(b)$ grows with L_b —something LRU/LFU cannot capture with recency or frequency alone. KVLearn encodes this bias in per-block CARS scores; hardware LLCs pursue a similar goal through utility- or size-aware replacement and victim structures for large lines. A future pool controller could attach footprint and calibrated $(\mathcal{R}, \mathcal{T})$ tags to each block and evaluate retention in the memory fabric (e.g., CXL expanders), avoiding per-admission control-plane round-trips—a direction we leave to future work.

Caching per-prefix KV blocks may retain sensitive content (user images, PII in prompts) beyond the originating request’s lifetime. Deployments must apply retention policies consistent with data protection regulations: KVLearn’s admission API accepts per-prefix TTL overrides, allowing operators to force eviction of sensitive-prefix blocks on a schedule.

7 Conclusion

We identified and formalized the KV cache retention problem in disaggregated LLM serving: a joint admission–eviction decision whose optimal policy depends heterogeneously

on prefix length, fabric bandwidth, and recompute cost—making classical LRU/LFU inadequate. KVLearn addresses this through a lightweight online predictor (PRP), a cost-aware scoring function (CARS), and a closed-loop threshold controller (ATC). Evaluations on language-only and multi-modal workloads show up to 56% TTFT reduction vs. No-Cache (38% vs. LRU-Pool), 53% inter-node transfer volume reduction vs. LRU-Pool, and near-oracle throughput on MM-Session at realistic pool sizes (modestly larger gaps on other workloads)—all without modifying model weights or changing the serving routing logic.

The broader takeaway is that *where* a KV block is computed and *where* it will be consumed are no longer the same node in modern disaggregated deployments, and retention policies should account for the intervening network as a first-class cost. As visual token lengths grow (4k–16k for high-resolution and video models) and serving clusters scale to hundreds of nodes, the gap between intelligent and naive retention will only widen. Future directions include hardware memory-side retention primitives, joint KV-compression and retention co-optimization, and federated reuse prediction across cluster boundaries.

References

- [1] Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. 2023. Qwen-VL: A Versatile Vision-Language Model for Understanding, Localization, Text Reading, and Beyond. *arXiv preprint arXiv:2308.12966* (2023).
- [2] ByteDance. 2025. InfiniStore: High-performance KV store for LLM inference clusters. <https://github.com/bytedance/InfiniStore>
- [3] Yihua Cheng, Yuhao Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. 2025. LMCACHE: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference. *arXiv preprint arXiv:2510.09665* (2025).
- [4] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 35. 16344–16359.
- [5] Gil Einziger, Roy Friedman, and Ben Manes. 2017. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage* 13, 4, Article 35 (2017), 31 pages.
- [6] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization. *arXiv preprint arXiv:2401.18079* (2024).
- [7] Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, and Tuo Zhao. 2024. GEAR: An Efficient KV Cache Compression Recipe for Near-Lossless Generative Inference of LLM. *arXiv preprint arXiv:2403.05527* (2024).
- [8] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP)*. 611–626.
- [9] Ji Lin, Hongxu Yin, Wei Ping, Yao Lu, Pavlo Molchanov, Andrew Tao, Huizi Mao, Jan Kautz, Mohammad Shoeybi, and Song Han. 2023. VILA: On Pre-training for Visual Language Models. *arXiv preprint arXiv:2312.07533* (2023).
- [10] Dong Liu and Yanxuan Yu. 2025. Tinyserve: Query-aware cache selection for efficient llm serving. In *Proceedings of the 33rd ACM International Conference on Multimedia*. 12529–12537.
- [11] Dong Liu and Yanxuan Yu. 2026. Cxl-speckv: A disaggregated fpga speculative kv-cache for datacenter llm serving. In *Proceedings of the 2026 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 56–66.
- [12] Dong Liu, Yanxuan Yu, Ben Lengerich, and Ying Nian Wu. 2025. PiKV: KV Cache Management System for Mixture of Experts. *arXiv preprint arXiv:2508.06526* (2025).
- [13] Dong Liu, Yanxuan Yu, Ben Lengerich, and Ying Nian Wu. 2026. Ada-Correction: Adaptive Offset Cache Correction for Accurate Diffusion Transformers. *arXiv preprint arXiv:2602.13357* (2026).
- [14] Dong Liu, Yanxuan Yu, Ben Lengerich, and Ying Nian Wu. 2026. MKA: Memory-Keyed Attention for Efficient Long-Context Reasoning. In *Proceedings of the 23rd ACM International Conference on Computing Frontiers*. 280–289.
- [15] Dong Liu, Yanxuan Yu, and Ying Nian Wu. 2026. Accelerating Frequency Domain Diffusion Models with Error-Feedback Event-Driven Caching. *arXiv preprint arXiv:2604.22901* (2026).
- [16] Dong Liu, Yanxuan Yu, Jiayi Zhang, Yifan Li, Ben Lengerich, and Ying Nian Wu. 2025. FastCache: Fast Caching for Diffusion Transformer Through Learnable Linear Approximation. *arXiv preprint arXiv:2505.20353* (2025).
- [17] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. 2020. An Imitation Learning Approach for Cache Replacement. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*. 6237–6247.
- [18] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual Instruction Tuning. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 36.
- [19] Yuhao Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, and Junchen Jiang. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proceedings of ACM SIGCOMM*.
- [20] Ruiyang Ma, Teng Ma, Junru Li, Hantian Zha, Xuchun Shang, Qingda Hu, Zheng Liu, Xinjun Yang, Tao Ma, and Guojie Luo. 2026. SAC: Disaggregated KV Cache System for Sparse Attention LLMs with CXL. *arXiv preprint arXiv:2606.19746* (2026).
- [21] NVIDIA. 2024. TensorRT-LLM. <https://github.com/NVIDIA/TensorRT-LLM>
- [22] 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 *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*. 118–132.
- [23] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2024. Mooncake: A KV Cache-Centric Disaggregated Architecture for LLM Serving. *arXiv preprint arXiv:2407.00079* (2024).
- [24] Liana V. Rodriguez, Farzana M. Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. 2021. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 341–354.
- [25] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. 2020. Learning Relaxed Belady for Content Distribution Network Caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

- [26] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving cache replacement with {ML-based}{LeCaR}. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.
- [27] Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. 2025. Duoattention: Efficient long-context llm inference with retrieval and streaming heads. In *International Conference on Learning Representations*, Vol. 2025. 37228–37253.
- [28] 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.
- [29] Jiayi Yao, Hanchen Li, Yuhang Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2024. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. *arXiv preprint arXiv:2405.16444* (2024).
- [30] Dongha Yoon, Younghoon Min, Hoshik Kim, Sam H Noh, and Jongryool Kim. 2025. TraCT: Disaggregated LLM serving with CXL shared memory KV cache at rack-scale. *arXiv preprint arXiv:2512.18194* (2025).
- [31] Zhou Yu, Dejing Xu, Jun Yu, Ting Yu, Zhou Zhao, Yueting Zhuang, and Dacheng Tao. 2019. ActivityNet-QA: A Dataset for Understanding Complex Web Videos via Question Answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 9127–9134.
- [32] 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.
- [33] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. SGLang: Efficient Execution of Structured Language Model Programs. *Advances in Neural Information Processing Systems (NeurIPS)* 37 (2024), 62557–62583.
- [34] 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)*. 193–210.

A Proof of Proposition 1

Scaling assumptions. We express costs as functions of prefix length L_b with hardware and model hyper-parameters held fixed. From Eq. (1), $|b| \propto L_b$, so Eq. (2) gives $\mathcal{T}(b) = |b|/\beta \propto L_b$; write $\mathcal{T}(L_b) = c L_b$ with $c > 0$. The storage model in §3.2, $\mathcal{U}(b, \Delta t) = \gamma |b| \Delta t$, yields $\mathcal{U}(L_b, \Delta t) = \gamma L_b \Delta t$ with $\gamma > 0$. For prefill compute we write $\mathcal{R}(L_b) = \alpha L_b^q$ with $\alpha > 0$ and $q \geq 1$: $q = 2$ reflects quadratic attention in L_b , while $q = 1$ can approximate bandwidth-bound kernels whose time scales roughly linearly in sequence length.

Closed form. Whenever $\mathcal{R}(L_b) > \mathcal{T}(L_b)$, Eq. (5) becomes

$$P_H^*(L_b) = \frac{\gamma L_b \Delta t}{\alpha L_b^q - c L_b} = \frac{\gamma \Delta t}{\alpha L_b^{q-1} - c}. \quad (12)$$

This requires $\alpha L_b^q > c L_b$, i.e. $L_b^{q-1} > c/\alpha$ when $L_b > 0$ (for $q > 1$, large enough L_b always satisfies this).

Let $g(L_b) = \alpha L_b^{q-1} - c$. For $q > 1$ and $L_b > 0$,

$$g'(L_b) = \alpha(q-1)L_b^{q-2} > 0,$$

so g is strictly increasing on $((c/\alpha)^{1/(q-1)}, \infty)$. Thus $P_H^*(L_b) = \gamma \Delta t / g(L_b)$ is strictly decreasing in L_b whenever $g(L_b) > 0$. For $q = 1$, $g(L_b) = \alpha - c$ is constant; if $\alpha > c$, then $P_H^*(L_b) = \gamma \Delta t / (\alpha - c)$ does not depend on L_b .

Standard transformer prefill is typically super-linear in L_b ($q > 1$), so the heterogeneous-threshold regime of Proposition 1 applies. FlashAttention-class kernels may reduce the effective exponent toward $q \approx 1$ under sustained bandwidth saturation; then P_H^* exhibits weaker length dependence under this idealization.

B Generalized Cost Model of KVLearn

Relaxing the cancelling-transfer and amortized-admission assumptions of §3.3,

$$\begin{aligned} \Delta C(b) &= \mathcal{R}(b) + \mathcal{T}_{\text{miss}}(b) - \mathcal{T}_{\text{hit}}(b), \\ C_{\text{admit}}(b) &= \kappa \mathcal{T}_{P \rightarrow S}(b) + C_{\text{ctrl}}, \quad \kappa \in [0, 1], \\ \mathcal{V}^g(b) &= P_H(b) \Delta C(b) - C_{\text{admit}}(b) - \mathcal{U}(b, \Delta t), \quad (13) \\ P_H^{*,g}(b) &= \frac{C_{\text{admit}}(b) + \mathcal{U}(b, \Delta t)}{\Delta C(b)} \quad (\Delta C(b) > 0). \end{aligned}$$

Keep iff $P_H(b) > P_H^{*,g}(b)$. Setting $\mathcal{T}_{\text{miss}} = \mathcal{T}_{\text{hit}} = \mathcal{T}$ and $\kappa = C_{\text{ctrl}} = 0$ recovers $P_H^* = \mathcal{U} / (\mathcal{R} - \mathcal{T})$ (Eqs. (4)–(5)); $\mathcal{T}_{\text{hit}} - \mathcal{T}_{\text{miss}} = \delta > 0$ or $\kappa > 0$ raises the threshold (more selective) without changing the L_b -ordering of Proposition 1 when $q > 1$.

C Serving Topology and Request Routing

Notation (consistent with §3.1–3.2). Prefill and decode pools are $\mathcal{P} = \{p_1, \dots, p_K\}$ and $\mathcal{D} = \{d_1, \dots, d_M\}$; the shared KV pool is $\mathcal{S}$ (capacity M_S). Fabric peak bandwidth is β (bytes/s). For request r , let s_r denote its prompt prefix, b_r the associated KV block, and $|b_r|$ its size from Eq. (1). The coordinator maintains a radix tree $\mathcal{T}$ whose leaves correspond to cached prefixes; an *exact-prefix hit* holds iff s_r is a registered path in $\mathcal{T}$ and the corresponding block is resident in $\mathcal{S}$.

Residency state. Define the Boolean predicates

$$H_r^{\text{pool}} = \mathbb{K}[\text{KV}(s_r) \in \mathcal{S}], \quad (14)$$

$$H_r^P = \mathbb{K}[\exists p \in \mathcal{P} : \text{KV}(s_r) \text{ resident on } p]. \quad (15)$$

The logical OR $H_r = H_r^{\text{pool}} \vee H_r^P$ is the *KV-hit* event that avoids full-prefix recompute on a P-node.

Latency-aware placement. Let $\text{QL}_P^{(t)}(p)$ and $\text{QL}_D^{(t)}(d)$ be queued work on prefill/decode nodes at arrival time t . The critical-path cost of assigning (p, d) is

$$L(r; p, d) = (1 - H_r)(\text{QL}_P^{(t)}(p) + \mathcal{R}(b_r)) + \mathcal{T}(b_r) + \text{QL}_D^{(t)}(d), \quad (16)$$

and latency-optimal routing chooses $(p_r^*, d_r^*) \in \arg \min_{p,d} L(r; p, d)$. Under homogeneous fabric bandwidth, a pool hit ($H_r^{\text{pool}}=1$) makes the p -term vanish, so (16) reduces to least-loaded decode placement $d_r^* \in \arg \min_d \text{QL}_D^{(t)}(d)$ (ties broken by smallest node id); on a miss we likewise pick the least-loaded P-node. Relative to forcing a miss on the same (p, d) , a KV-hit saves

$$\Delta L(r; p, d) = L(r; p, d)|_{H_r=0} - L(r; p, d)|_{H_r=1} = \text{QL}_P^{(t)}(p) + \mathcal{R}(b_r), \quad (17)$$

so the differential that retention can avoid is recompute (plus P-queue), matching the $\mathcal{R}(b) - \mathcal{T}(b)$ term that drives $\mathcal{V}(b)$ in Eq. (4) once the common transfer hop cancels.

Critical-path timing (idealized). Let T^{route} be coordinator bookkeeping + radix lookup and $T_{\text{first}}^{\text{decode}}$ the decode-side first-token work. Ignoring residual queueing beyond (16),

$$\text{TTFT}(r) \approx T^{\text{route}} + L(r; p_r^*, d_r^*) + T_{\text{first}}^{\text{decode}}, \quad (18)$$

with $\mathcal{T}(b_r) \approx |b_r|/\beta$ on saturated fabric hops (pool $\rightarrow d$ or $P \rightarrow d$). Under a stationary pool-hit rate $h = \Pr(H_r^{\text{pool}}=1)$, $\mathbb{E}[\text{TTFT}]$ therefore decreases linearly in h through the $(1-H_r)$ term of (16)—the quantity that CARS admission/eviction directly controls via residency in $\mathcal{S}$.

Routing policy used in experiments. Upon arrival at the pool coordinator:

1. *Radix lookup:* Compute H_r^{pool} via exact-prefix match in $\mathcal{T}$ and metadata pins in $\mathcal{S}$.
2. *Hit path* ($H_r^{\text{pool}} = \text{true}$): Select d_r^* , schedule an asynchronous transfer $\mathcal{S} \rightarrow d_r^*$ for KV(s_r), and skip P-node prefill for s_r .
3. *Partial P-local hit* ($H_r^{\text{pool}} = \text{false}$, $H_r^{\text{P}} = \text{true}$): Route to the holding P-node only for missing suffix tokens; the merged block still follows $p \rightarrow d_r^*$ delivery before decode.
4. *Miss path* ($H_r = \text{false}$): Dispatch to the selected P-node, run full prefill for s_r , stream KV(s_r) to d_r^* , and invoke Algorithm 1 for optional replication into $\mathcal{S}$.

Measurement. TTFT is taken from request enqueue at the coordinator until the client receives the first generated token; timestamps exclude WAN/client-side buffering outside the cluster boundary used in §5.

D Experimental Parameters for Reproducibility

Tables 5 and 6 list symbolic parameters aligned with §5 (GPU generation A100-80GB; Mooncake-style routing [23]). $\mathcal{T}(b)=|b|/\beta$ denotes the transfer-cost term in §5.

Table 5. Symbolic system, model, and workload parameters ($k \in \{1, 2\}$ orders the cluster–fabric tuple in row 1; M_S^* matches Table 1).

Param.	Value
$(\mathcal{P}_k , \mathcal{D}_k , M_{S,k}, \beta_k)$	$((4, 4, 256 \text{ GB}, 200 \text{ Gbps}), (8, 8, 512 \text{ GB}, 2 \times 400 \text{ Gbps}))$
$\beta_{\text{sweep}}, \mathcal{T}(b)$	$\{50, 100, 200, 400\} \text{ Gbps}; b /\beta$
routing graph	$P \rightarrow \mathcal{S} \rightarrow D, \nexists P \leftrightarrow D$
M_S^*	256 GB
KV dtype	bf16 end-to-end
$\mathcal{M}_{\text{lm}}(L_{\text{vis}}^{\text{lv}}, L_{\text{vis}}^{\text{wv}})$	$\{8\text{B}, 70\text{B}\}$ (576, 256) tokens/img. (LLaVA-1.5-13B [18], Qwen-VL-7B [1])
$N_{\text{sg}}(\alpha, I , \bar{L}_{\text{vis}}, \bar{L}_{\text{txt}})$	104 (1.1, 80, 576, 128)
$(N_c, T_c, f, [\rho^-, \rho^+], [L_{\text{vis}}^-, L_{\text{vis}}^+])$	(10, 60 s, 1 fps, [12, 24] tok/frame, [720, 1440] tok)
oracle horizon H	1000

Table 6. Symbolic PRP, feature whitening, ATC, and control-plane parameters (Eqs. (6)–(7), (11); Alg. 2).

Param.	Value
(d, h)	(16, 32)
$(\mathbf{W}_1, \mathbf{w}_2)$	$(\mathbb{R}^{32 \times 16}, \mathbb{R}^{32})$
$(f_{\text{hid}}, f_{\text{out}}, \mathcal{L})$	(ReLU, σ , BCE)
$(\eta_{\text{PRP}}, \mathcal{B} , \Delta n_{\text{opt}})$	$(5 \times 10^{-3}, 2 \times 10^4, 100)$
$(T_{\text{auc}}, N_{\text{seed}})$	(10 min, 5)
$(\mathcal{A}_0, P_H^{*, \text{def}}, n_0)$	(0.75, 0.15, ≈ 200)
$\{\lambda_j\}_{j=1}^3$	$\{60, 300, 900\} \text{ s}$
$(n_{\text{warm}}, k_{\mu})$	(500, 1000) requests
(ϕ, ϵ)	$(0.99, 10^{-6})$; $\hat{x}_i = (x_i - \mu_i) / \max(\sigma_i, \epsilon)$ for $n > 500$
$T_{\text{sess}}^{\text{cap}}$	24 h
$(\rho^*, K_P, K_I, h_{\text{floor}}, K_{\text{hr}})$	$(0.85, 2 \times 10^{-2}, 5 \times 10^{-4}, 0.55, 1.0 \times 10^{-1})$
θ	$[-C_{\text{max}}, C_{\text{max}}]$, $C_{\text{max}} = \max_b \text{CARS}(b) $
$(\bar{T}_{\text{adm}}, \bar{T}_{\text{ev}}, N_{\text{meta}})$	(0.5 ms (p99), 5 ms (p99), 10^4)
Δt_{PRP}	30 s

$(K_P, K_I, h_{\text{floor}}, K_{\text{hr}})$ in Table 6 are conservative stability defaults; re-fit from pool-pressure traces in production.