PROTECT BEFORE EVICTING: LAGRANGIAN-GUIDED ENCODER CACHING FOR DISAGGREGATED MULTIMODAL LLM SERVING

Anonymous authors
Paper under double-blind review

ABSTRACT

For multimodal models with separate encoders, repeated inputs allow encoder outputs to be reused across requests, even when the questions differ. A bounded encoder cache must decide which outputs to retain because their sizes, recomputation costs, and reuse frequencies differ. We present Protect Before Evicting, a Lagrangian-guided cache policy for disaggregated multimodal LLM serving. Offline calibration solves a scalar problem using an object profile and a cache budget. Online decisions use its solution to grant temporary protection, while allocations may override that protection when space is needed. The implementation preserves active-reference restrictions and hard capacity checks, without running an optimization solver on each request. We integrate the policy into vLLM and evaluate controlled image workloads with fixed profiles using Qwen3.5-9B on four H800 GPUs. At 16 requests/s, mean and p99 time to first token decrease by 10.03% and 13.04%, respectively, relative to vLLM's native encoder-cache replacement in the same serving stack. These results show a practical benefit from managing encoder-output retention separately from language generation.

1 INTRODUCTION

Multimodal LLMs process images, video, and audio as well as text (Xu et al., 2025). Serving these models requires more than efficient text generation: modality encoders add work whose size and cost depend on the input (Qiu et al., 2025). Disaggregated systems separate stages so their resources can be allocated independently (Zhong et al., 2024; Qiu et al., 2025). This helps manage where computation runs, but does not by itself decide which previously computed outputs should remain available for reuse.

Such reuse does not require identical user requests. Product queries may refer to the same catalog image, and document workflows may revisit a shared diagram with different questions. These are among the use cases identified for encoder-output caching in NVIDIA Dynamo (NVIDIA, 2026). The reusable object is an intermediate representation, not a previously generated answer. This makes repeated media a useful target for caching even when each request still needs its own language-model response. The question is how to spend limited cache space on these representations.

A retention decision must account for both future use and the work that a hit avoids. Keeping a large output can displace several smaller ones, while a cheap-to-recompute output may be less valuable than an equally popular expensive one. Cost-aware replacement and optimization-based cache design have long studied such tradeoffs (Cao & Irani, 1997; Dehghan et al., 2016). In an encoder-serving system, decisions must also fit its execution rules. vLLM keeps request references to encoder outputs and reclaims eligible entries when allocation needs space (vLLM Project, 2026b). A retention preference cannot make an output used by active requests safe to evict or prevent all future allocations.

We propose *Protect Before Evicting*, a Lagrangian-guided encoder-cache policy. The key idea is to assign temporary protection before deciding which output to remove. An offline calibration step uses reuse statistics, output sizes, and recomputation-cost estimates to select one shared memory coefficient. This coefficient balances holding an output against recomputing it later. At each access, the policy uses it to determine a protection window and the probability of granting that window.

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

When space is needed, the allocator first considers unprotected entries. If all eligible entries are protected, it overrides protection using a value-based fallback. Protection therefore guides retention without becoming an unconditional memory reservation. Calibration is outside the request path; the online rules do not require future accesses or repeated optimization solves.

The policy uses object identities, memory requirements, costs, and reuse statistics rather than the contents of a modality. Its formulation therefore also applies to video or audio outputs when a separate encoder produces a reusable representation independent of the text prompt, as in architectures with visual and audio encoders (Xu et al., 2025). Reuse requires a fixed model and preprocessing configuration, including any temporal context needed by that output. We use image encoding for the formulation and evaluation below; we do not report video or audio serving results.

Our work is related to caching at other levels of LLM inference. Zhu et al. (2023) study query-result caching and model selection with offline and online learning. SGLang and Prompt Cache reuse language-model attention states across prompts (Zheng et al., 2024; Gim et al., 2024). Learned prefix caching uses conversational content to guide their retention (Yang et al., 2025). We instead manage encoder outputs within one serving stack, without selecting a different model or replacing prompt-specific answers.

Next, we list the main contributions of this work:

- **Lagrangian formulation and calibration.** We formulate encoder-output retention as expected recomputation-cost minimization under a cache budget. Lagrangian memory accounting puts retention and recomputation on a common scale. A single-round calibration model selects a shared memory coefficient through a concave scalar optimization, combining reuse information, output size, and recomputation cost.
- **Optimization-guided online protection.** We translate the calibrated coefficient into object-specific protection windows and grant probabilities. This separates which outputs to protect from which outputs to evict. Protection can expire without eviction and can be overridden when an allocation needs space. Online decisions require neither future accesses nor another optimization solve.
- **Integration with vLLM.** We connect the policy to vLLM's scheduler, reference tracking, and worker-side tensor storage. The implementation preserves active-reference restrictions and hard allocation limits while changing retention preferences.
- **Evaluation on real GPUs.** We evaluate controlled image workloads with fixed profiles using Qwen3.5-9B on four H800 GPUs. At 16 requests/s, mean and p99 TTFT decrease by 10.03% and 13.04%, respectively, relative to native vLLM retention in the same E/PD stack (Section 6).

1.1 Related Work

Disaggregated and multimodal serving. DistServe separates prefill and decoding to reduce interference and allocate resources by stage (Zhong et al., 2024). ModServe studies the additional demands of multimodal inference and separates modality processing from language-model execution, with stage-aware scaling and scheduling (Qiu et al., 2025). Encoder-output reuse is also an existing system feature. NVIDIA Dynamo documents an embedding cache for repeated images, including catalog queries, shared figures, and multi-turn conversations (NVIDIA, 2026). Our focus is the retention decision within a bounded encoder-output cache, rather than a new disaggregation architecture or resource-assignment scheme. We leave routing and the model computation unchanged.

Query-result caching and online learning. Zhu et al. (2023) study the joint choice of cached query results and inference models. They analyze offline and online learning of query frequencies and processing costs in a tabular setting, using least-expected-cost and related caching rules. Their cache stores a query's result, not its decoder KV prefix; their Appendix B also discusses variable object sizes. Our retained object is instead an encoder representation that can be used by different text prompts. We keep the language model fixed and study allocation-safe retention in an E/PD implementation. The overlap is the use of reuse frequency and recomputation cost to value cache space. Unlike their online-learning setting, our experiments use fixed calibration profiles.

KV and prompt-state reuse. SGLang's RadixAttention manages KV reuse across shared prompt prefixes (Zheng et al., 2024). Prompt Cache introduces explicitly defined prompt modules whose attention states can be reused across prompts (Gim et al., 2024). Learned Prefix Caching predicts whether a conversation will continue from its contents and combines that prediction with recency to guide prefix eviction (Yang et al., 2025). vLLM's hash-based prefix cache identifies blocks using token content, preceding block hashes, and extra information such as multimodal hashes (vLLM Project, 2026c). These methods cache language-model states rather than raw encoder outputs. An unchanged visual representation can be reusable even when the corresponding language-model prefix does not match. Conversely, requests with a compatible shared prefix can benefit from KV reuse as well. Appendix A explains the distinction in detail. Our implementation does not change prefix matching, and the experiments disable prefix caching rather than measure combined gains from both mechanisms.

Replacement, admission, and memory accounting. GreedyDual-Size combines retrieval cost, object size, and an aging mechanism in replacement decisions (Cao & Irani, 1997). TinyLFU uses recent-frequency estimates to decide whether an incoming object should replace a resident candidate (Einziger et al., 2017). S3-FIFO uses three FIFO queues, including a small queue that filters many objects with little reuse (Yang et al., 2023). These policies address reuse and admission in different ways. Our comparison with native vLLM isolates a change within the evaluated serving stack, rather than a benchmark of these alternative policies.

Dehghan et al. (2016) formulate utility-based cache design and derive timer-based policies through dual and primal-dual optimization. Thus Lagrangian reasoning and per-object timers are not new to caching. Our method calibrates one memory coefficient before a serving interval, then uses it for probabilistic soft protection of encoder outputs. Expiry removes protection but does not itself evict an output. Allocation can override protection, and active request references remain ineligible for eviction. The value score is used only when all eligible outputs are protected, rather than as a global ordering for every replacement. These are the policy and execution choices studied in this paper.

2 Background and Motivation

2.1 Disaggregated multimodal serving

We consider image-to-text models with a separate visual encoder (Qiu et al., 2025). After preprocessing, encoding (E) maps an image to visual embeddings. Prefill (P) processes these embeddings and the text prompt, builds the language model's key-value (KV) state, and produces the first output token. Decoding (D) generates subsequent tokens (Zhong et al., 2024).

Disaggregation places stages in separate worker instances. P/D disaggregation separates prefill from decoding to allow different resource allocations (Zhong et al., 2024). We focus on the E/PD layout in Figure 1 (left): E runs separately, while P and D remain in the same worker. This allows the encoder and language model to scale independently (Qiu et al., 2025). vLLM supports both E/PD and fully separated E/P/D deployments (vLLM Project, 2026a).

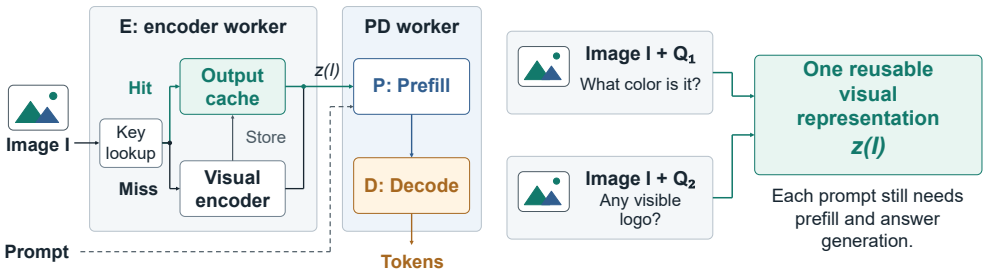

Figure 1: **Left:** Encoder-output caching in E/PD serving. A hit reuses the visual representation $z(I)$; a miss computes it. Both paths pass the representation to a PD worker. Preprocessing, routing, and worker multiplicity are omitted. **Right:** Different prompts can reuse the same visual representation. The illustration shows encoder-output reuse without additional KV prefix reuse. It does not imply that every request must recompute its entire prefill when prefix caching is available.

2.2 ENCODER-OUTPUT REUSE AND CACHE CAPACITY

An encoder cache stores visual representations, not prompts or generated answers. In Figure 1 (left), lookup happens before visual encoding. A hit retrieves an existing output; a miss runs the encoder and stores its output. The representation is then passed to the PD worker through an encoder-transfer interface (vLLM Project, 2026a). A hit avoids encoding, but does not by itself remove transfer or language-model work.

Reuse across prompts. Figure 1 (right) distinguishes repeated visual content from repeated requests. With a fixed encoder and preprocessing configuration, requests (I, q_1) and (I, q_2) can share $z(I)$ even when $q_1 \neq q_2$ (NVIDIA, 2026). For example, one prompt asks about an image's color and another about a visible logo. The language model still answers each prompt. We consider exact reuse, not matching visually similar images. Appendix A explains the distinction from KV prefix caching.

Eviction. Image resolution can change visual token counts and encoding work (Qiu et al., 2025). In vLLM, outputs with active request references cannot be evicted. Once the reference set is empty, an output may remain cached until an allocation needs its space (vLLM Project, 2026b). Our policy changes retention preferences among these eligible outputs, not this safety condition. Keeping an output saves recomputation only if it is reused before eviction, while a large output leaves less room for other images. We study this choice at a fixed cache budget.

3 PROBLEM FORMULATION AND DESIGN RATIONALE

Cache objects and accesses. We model one encoder cache in the E/PD layout of Figure 1. Its budget B is measured in visual embedding slots. Object i is the visual representation $z(I_i)$ of image I_i. We consider outputs that fit in the cache, with size $0 < m_i \leq B$ and profiled recomputation cost $c_i > 0$. We index image accesses by $t = 1, \ldots, T$ and write ω_t for the accessed object. A multi-image request contributes several accesses; t counts accesses, not seconds. For known objects, calibration also provides a reuse parameter p_i. These parameters are policy inputs, not observations of future requests.

Cache policy. Let π denote an online cache policy, and let $\mathcal{C}_t^\pi$ contain the cached object identifiers just before access t. The initial cache state is given. On a hit, where $\omega_t \in \mathcal{C}_t^\pi$, the manager reuses the stored output without allocating space for another copy. On a miss, it allocates space, computes the requested output, and stores it. If free space is insufficient, π chooses which unreferenced entries to evict before the allocation proceeds. If even removing all unreferenced entries would not provide enough space, the manager defers allocation. Actively referenced entries cannot be evicted (vLLM Project, 2026b). The policy may update retention priorities on both hits and insertions; entries otherwise remain cached until their space is needed. Thus, π controls retention preferences and eviction choices, while insertion is driven by observed accesses. Its decisions may be randomized and use the current access, cache state, past information, and available calibration data, but not future accesses.

Objective and constraints. We write $J(\pi)$ for the expected total recomputation cost incurred by cache misses over the T accesses. We seek a policy following the rules above that minimizes

$$\begin{aligned} \min_{\pi} \quad & J(\pi) := \mathbb{E}\left[\sum_{t=1}^{T} c_{\omega_t}\, \mathbf{1}\{\omega_t \notin \mathcal{C}_t^\pi\}\right] \\ \text{subject to} \quad & \sum_{i \in \mathcal{C}_t^\pi} m_i \leq B, \qquad t = 1, \ldots, T. \end{aligned} \tag{1}$$

The indicator equals one on a cache miss and zero on a hit. The expectation is over the workload and any policy randomization. The budget must also hold after every allocation. We consider no prefetching and leave routing and scheduling unchanged. The costs c_i are proxies for recomputation work, not a complete model of batching, transfer, or queueing latency.

A common scale for memory and recomputation. Keeping an output can save an encoding, but leaves less space for other outputs. Introduce $\lambda \geq 0$ to account for this use of memory. It converts one slot held for one access step into the units of c_i. It is neither a monetary price nor extra execution

time. The design calculation accounts for capacity using the shared-coefficient Lagrangian

$$\mathcal{L}(\pi, \lambda) = J(\pi) + \lambda \, \mathbb{E}\left[\sum_{t=1}^{T} \left(\sum_{i \in \mathcal{C}_t^\pi} m_i - B \right)\right]. \tag{2}$$

For fixed λ, the term $-\lambda BT$ is constant; the added policy-dependent term counts occupied slots over access steps. Holding object i for $d \geq 0$ steps costs $\lambda m_i d$ in this accounting, whereas avoiding one recomputation saves c_i. The comparison $\lambda m_i d < c_i$ motivates protection. Section 4 selects the coefficient through a single-round calibration model and uses it for online decisions. Physical capacity and reference checks remain mandatory.

4 Lagrangian-Guided Cache Protection

Our method first calibrates a shared memory coefficient, then uses it to protect selected outputs during serving. Protection is a retention preference, not a memory reservation. It can be overridden when an allocation needs space, unlike the safety restriction imposed by active references.

Preparing the profile. Let $\mathcal{K}$ be the finite set of known image identities with reuse parameters $p_i \in (0, 1]$, normalized so $\sum_{i \in \mathcal{K}} p_i = 1$. The size and cost inputs m_i, c_i are recorded or profiled estimates for the fixed encoder configuration. For reuse, a possible historical profile is $p_i = N_i / \sum_{j \in \mathcal{K}} N_j$, where $N_i > 0$ counts accesses to identity i in a completed observation window. Counts include hits as well as misses. These p_i describe the known pool, not unconditional frequencies when unknown objects also arrive. Actual allocator slot counts, rather than profile estimates, enforce capacity.

Offline calibration. The reference workload draws known identities independently with probabilities p_i. Its gap D_i counts intervening accesses before identity i returns: $\Pr(D_i = d) = p_i(1 - p_i)^d$ for integers $d \geq 0$; when $p_i = 1$, $D_i = 0$. The actual serving trace may differ from this reference workload. Let $M = \sum_{i \in \mathcal{K}} m_i$ and let $B_{\text{cal}} \leq B$ be the calibration budget. We use $B_{\text{cal}} > \sum_{i \in \mathcal{K}} p_i m_i$, which ensures a finite maximizer. To select the coefficient, we use an auxiliary model that applies the memory accounting in equation (2) to complete gaps between accesses. It replaces per-allocation capacity constraints by a long-run average budget and omits active-reference lifetimes. For calibration only, a decision may depend on the complete modeled gap. Minimizing over these gap decisions gives the dual function below; derivation details can be found in Appendix B. The single-round dual score $\overline{D}$ selects the coefficient:

$$\begin{aligned} \overline{D}(\lambda) &= (M - B_{\text{cal}})\lambda - \sum_{i \in \mathcal{K}} p_i \, \mathbb{E}\big[(\lambda m_i D_i - c_i)^+\big], \\ \lambda^\star &\in \arg\max_{\lambda \geq 0} \overline{D}(\lambda), \qquad x^+ = \max\{x, 0\}. \end{aligned} \tag{3}$$

The first term charges for keeping the full pool relative to B_{cal} slots. Over a modeled gap, keeping object i costs $\lambda m_i D_i$, while recomputing it costs c_i. When recomputation is cheaper, the positive part subtracts this saving, weighted by p_i. Expectations are over the geometric gaps. Maximizing this dual score selects the memory coefficient, not an eviction victim or an exact solution of equation (1). The problem is one-dimensional and concave; Appendix B gives a closed form for each expectation. Thus calibration needs no simulated requests or new encoder executions once the inputs are available. We solve it before serving and keep $\lambda^\star$ fixed within the interval. No optimization or multiplier update runs on an access.

Online protection. For a known object and $\lambda^\star > 0$, the nominal protection window is

$$h_i = \left\lceil \frac{c_i}{\lambda^\star m_i} \right\rceil. \tag{4}$$

The ceiling rounds up to an integer number of access steps. More expensive outputs receive longer windows, while larger outputs use the memory allowance faster. At each access, draw a fresh auxiliary D_i from the geometric model and grant the window only if $\lambda^\star m_i D_i < c_i$. The sample randomizes the decision; it does not reveal the actual next access. The probability of granting protection is

$$\rho_i = \Pr(D_i < h_i) = 1 - (1 - p_i)^{h_i}. \tag{5}$$

Algorithm 1 Offline calibration and online cache protection

Require: Known-object profile $\{p_i, m_i, c_i\}_{i\in\mathcal{K}}$ and B_{cal}

Offline: solve equation (3) for $\lambda^\star$.
Compute v_i by equation (6); if $\lambda^\star > 0$, compute h_i by equation (4).
τ_i is an entry's protection deadline; unset deadlines are zero.
procedure ONACCESS(i, t)
    $\tau_i \leftarrow 0$
    if $i \in \mathcal{K}$ and $\lambda^\star > 0$ **then**
        Draw D_i from the geometric model
        if $\lambda^\star m_i D_i < c_i$ **then** $\tau_i \leftarrow t + h_i$
    end if
end procedure
procedure MAKEROOM(s, t)
    $\mathcal{F} \leftarrow$ unreferenced entries in queue order; $b \leftarrow$ free slots
    if $b + \text{SLOTS}(\mathcal{F}) < s$ **then return** DEFER
    while $b < s$ **do**
        $j \leftarrow$ first entry in $\mathcal{F}$ with $\tau_j \le t$
        if none exists **then** $j \leftarrow \arg\min_{j\in\mathcal{F}} v_j$, with FIFO ties
        $b \leftarrow b + \text{SLOTS}(j)$; evict j; remove it from $\mathcal{F}$
    end while
    return READY
end procedure
s is required space, including any outstanding scheduled reservations.
SLOTS gives actual slot counts, summed for a set of entries.
ONACCESS uses the updated tick after a hit or successful allocation.
MAKEROOM uses the current tick before the pending allocation.

Thus h_i sets the duration, while ρ_i determines how often that duration is granted. For example, $c_i = 4$ ms and $\lambda^\star m_i = 1$ ms per step give $h_i = 4$. With $p_i = 0.1$, the window is granted with probability $1 - 0.9^4 \simeq 0.34$.

Let τ_i be the protection deadline, with zero meaning no protection. At access tick t, a successful draw sets $\tau_i = t + h_i$; otherwise $\tau_i = 0$. Hits and successful miss allocations both replace the old decision, and unknown accesses also advance the clock. The window starts at access, not at reference release. An unreferenced entry is protected while the current tick is below τ_i. Expiration does not evict the output, and declining protection does not prevent insertion. Unknown objects and the $\lambda^\star = 0$ case receive no protection.

Making room. Eligible entries form a queue ordered by when their last reference was released. On memory pressure, evict the first unprotected entry. Only if all eligible entries are protected, select the smallest retention value

$$v_i = \frac{p_i c_i}{m_i} \quad (i \in \mathcal{K}), \qquad v_i = 0 \quad (i \notin \mathcal{K}), \tag{6}$$

with queue order breaking ties. This fallback favors outputs with more profiled reuse savings per slot; it does not rank every eviction by v_i. Repeat until sufficient space is free, without evicting active entries. Algorithm 1 includes both calibration and the runtime hooks.

Later intervals. The method can be recalibrated for a later interval using an updated object profile and a new $\lambda^\star$. Interval length is a deployment choice. Our experiments use prescribed popularity profiles held fixed during each run, rather than a tested estimation or automatic refresh mechanism.

5 SYSTEM IMPLEMENTATION

We implement the policy of Section 4 in vLLM V1, using the E/PD layout in Figure 1 and the reference-counted encoder cache (vLLM Project, 2026a;b). Our changes add a policy interface,

protection metadata, and configuration and logging support. The implementation and evaluation scripts are public.[1]

Request path and cache ownership. The proxy first sends the visual inputs to an E worker, without the text prompt. It waits for that request to finish, then forwards the original request to a PD worker. Multiple images can share one encoder request while retaining separate cache identities. The policy is attached to the shared multimodal `EncoderCacheManager`, rather than to E-specific execution code. It can therefore be configured on an E worker or a PD worker using this cache manager. Each worker maintains its own cache state, access clock, and policy configuration; the implementation does not coordinate eviction across workers. In the reported experiments, we change the E worker's retention policy and leave the PD configuration unchanged to isolate E-side retention.

Connecting policy decisions to execution. The scheduler checks the cache before scheduling an image for encoding. A hit or successful allocation invokes the policy hook and advances the access counter. The returned deadline is recorded while the output is referenced and carried over unchanged when it becomes eligible for eviction. The hook only updates metadata; it does not move or free tensors. For a miss, allocation checks the per-step encoder compute budget, actual embedding counts, and space already reserved by pending inputs. Selected eviction identifiers are included in the scheduler output. The model runner removes those tensors before executing the next scheduled work and stores newly computed outputs under their identifiers. This keeps the policy's preferences separate from reference safety and physical storage.

Separating retention from transfer. The implementation uses `ECExampleConnector`, a shared-directory connector that writes encoder outputs as SafeTensors files and loads them into a PD worker's device cache. The scheduler permits external-store lookup only on consumers. An E-side miss must therefore run the encoder even if a transfer file remains. This prevents the transfer store from hiding the effect of encoder-cache eviction. The scheduler also leaves transfer files intact when evicting local outputs, since a PD worker may still need to load them. Their storage and cleanup are separate from the bounded encoder cache, as detailed in Appendix C.

Profile loading and controls. The calibration tool writes a JSON mapping from multimodal hashes to p_i, m_i, c_i, together with $\lambda^\star$. At startup, the policy loads this table, precomputes protection windows, and initializes a seeded random generator. For generated profiles, the tool uses vLLM's image-byte hashing path with the model identifier and processor settings. Matching this path is necessary for a known image to find its profile entry. Policy selection, profile location, and random seed are process settings. A separate encoder-cache capacity setting allows the retention budget to change without changing the per-step compute budget; vLLM's minimum per-item capacity checks still apply.

Runtime work and diagnostics. An access updates metadata and, when protection is enabled for a known object, draws one auxiliary gap. No solver runs on the request path. Eviction scans the eligible queue and may repeat for a large allocation, so it is not a constant-time operation. Logging exposes hit and miss counts, forced protection overrides, known and unknown accesses, and the split between referenced, protected, unprotected, and free slots. Appendix C lists the code paths and configuration details.

6 Experimental Evaluation

We evaluate whether the policy reduces the time to first token (TTFT) when different requests reuse visual inputs. The main comparison changes encoder-cache retention within the same E/PD serving stack. Appendix D gives the recorded configurations, concurrency sweeps, and a separate four-image experiment with a larger cache.

Model and system. We use Qwen3.5-9B with BF16 weights on four H800 GPUs. One GPU runs E and the other three run independent PD workers, giving a 1E/3PD deployment. All workers use vLLM V1 with tensor parallel size one and eager execution. KV prefix caching is disabled on both E and PD. The configuration uses a 16,384-slot encoder-output cache, which equals to the default value in vLLM. The load and concurrency studies use this retention budget; Appendix D.5

[1] `https://anonymous.4open.science/r/vllm-8624`

reports the larger-cache configuration separately for ablation study. The batching limits are listed in Appendix D.1. Outputs pass through the shared-directory connector described in Section 5.

Baseline. We label the baseline *vLLM*: it uses vLLM's current encoder-cache replacement in the common serving stack. This replacement takes the oldest entry in the freeable queue when space is needed, a FIFO ordering among outputs with no active references (vLLM Project, 2026b). *Ours* replaces this E-side retention rule with Section 4; the model, PD configuration, and transfer path are the same. Thus the baseline is a running inference system, not a separate cache simulation.

Workload and protocol. Each request contains one image, a text-generator target of 256 tokens, and an output target of 128 tokens with early EOS stopping disabled. The known pool contains 20 images with nonuniform popularity. When drawing from this pool, the image with popularity rank $r \in \{1, \ldots, 20\}$ has probability $p_r = r^{-0.5} / \sum_{j=1}^{20} j^{-0.5}$. This finite Zipf distribution makes higher-ranked images more likely to recur. An access uses a newly generated image with probability 0.3; otherwise it samples the known pool. The pool contains four resolutions from 360×640 to 1440×2560. Images contain synthetic pixels, so the experiment controls size and reuse rather than natural-image semantics. Ours uses the prescribed pool probabilities and measured profile, fixed during the run, with $B_{\text{cal}} = B$ and no calibration margin. Both variants use the same generator settings and seed.

The load sweep submits 256 requests per operating point with Poisson arrivals. We report client TTFT for the paired runs with complete request and output accounting at 8, 12, and 16 requests/s. Each included run reports 256 completions, no failures, and 32,768 output tokens. The plots show individual run aggregates, not averages over independent repetitions.

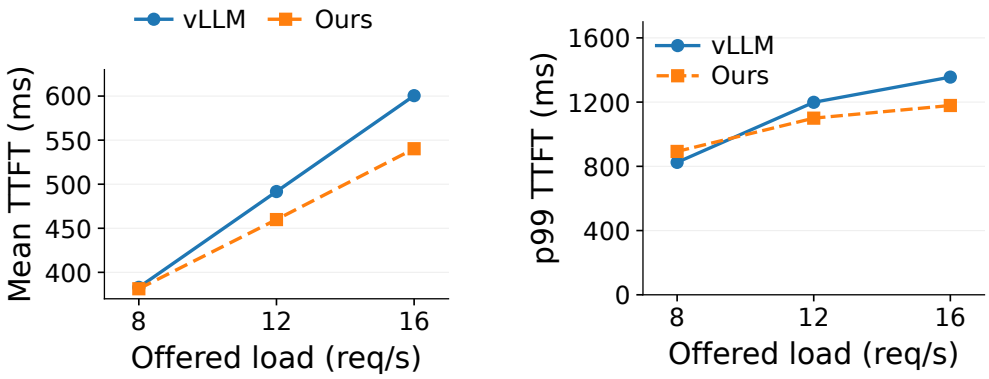

Figure 2: TTFT under increasing offered load on four H800 GPUs with Qwen3.5-9B and 1E/3PD serving. **Left:** Mean TTFT. **Right:** p99 TTFT. All shown pairs have complete aggregate output accounting. Both vertical axes are linear and start above zero. The horizontal axis is offered load, not achieved throughput.

TTFT under load. Figure 2 shows lower mean TTFT at all three reported loads. At 12 requests/s, mean TTFT falls from 491.67 to 459.88 ms, a 6.47% reduction. At 16 requests/s, it falls from 600.47 to 540.26 ms, a 10.03% reduction, while p99 falls from 1,355.66 to 1,178.90 ms, or 13.04%. At 8 requests/s, mean TTFT changes by only 1.61 ms, or 0.42%. One interpretation is that lighter traffic leaves less encoder-related waiting to remove, while routing, transfer, and PD prefill remain on the request path. At higher load, avoided encoding may also reduce contention. This explains a possible mechanism, not a measured utilization or queueing threshold. The higher p99 at 8 requests/s also shows that the tail benefit is not uniform. The main gain is in response start time; Table 5 gives the smaller changes in full-request latency and throughput.

Cache reuse and retention value. Figure 3 (left) shows logged hit fractions of 17.1% to 19.5% for vLLM and 34.4% to 35.3% for Ours across the reported loads. These counters support increased reuse, but are not exact request-boundary hit rates. For example, the final counter intervals cover 226 and 230 accesses, respectively, rather than all 256 client requests. Table 6 gives the underlying counts.

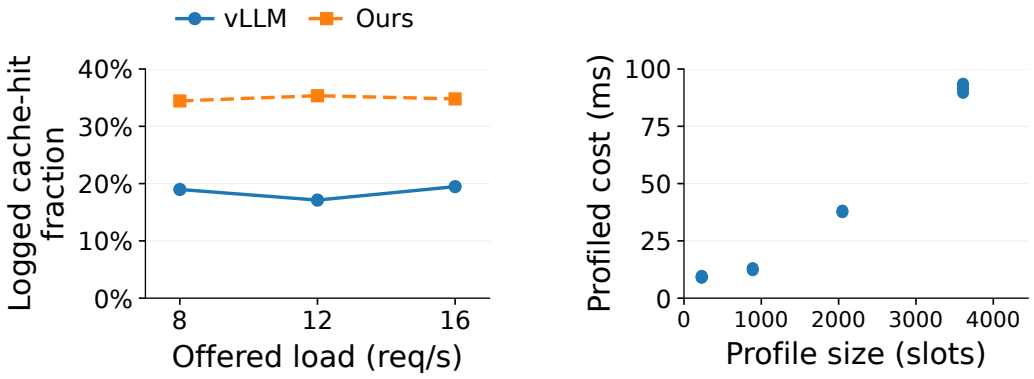

Figure 3: Cache behavior and calibration inputs for the load sweep. **Left:** Hit fractions reconstructed from differences of periodic cache-counter snapshots. Their windows are not synchronized to the client benchmark boundaries. **Right:** Recorded size and cost estimates for the 20 known images. The cost is a profiling proxy, not an isolated GPU-kernel time; overlapping points represent distinct objects.

The profile in Figure 3 (right) also shows why identical retention preferences need not suit all outputs. Recorded sizes range from 229 to 3,609 slots and costs from 8.90 to 93.64 ms. The cost per slot varies by 3.11 times. Using these inputs, calibration sets different protection windows rather than treating every cache hit as equally valuable. Appendix D.3 reports those windows and their protection probabilities. This characterization explains the policy inputs; it is not a component ablation or a measured decomposition of the TTFT improvement.

7 Conclusion and Limitation

We presented Lagrangian-guided encoder caching for disaggregated multimodal LLM serving. Offline calibration sets a shared memory coefficient, while online decisions combine temporary protection with eviction under memory pressure. The implementation preserves vLLM's reference checks and hard allocation limits. At 16 requests/s in the primary Qwen3.5-9B experiment on four H800 GPUs, the policy lowers mean TTFT by 10.03% and p99 TTFT by 13.04% relative to native vLLM retention (Figure 2). These results show that encoder-cache management can improve response-start latency without changing the model or its language-generation procedure.

However, there are also limitations. Our evaluation uses controlled synthetic workloads, fixed prescribed popularity profiles, and the reported Qwen3.5-9B deployments on H800 GPUs. It does not test popularity estimation and automatic profile refresh. The measurements are individual run aggregates, so tail-latency differences require caution. Benefits depend on load and configuration. The larger-cache study also changes the worker layout and workload; it is not a capacity-only ablation. The results therefore support the evaluated settings, not a general claim across serving workloads.

AI Use Statement

We used generative AI tools to improve the clarity and organization of the manuscript, and help draft and debug auxiliary code for system implementation. The research problem, core algorithmic ideas, mathematical formulation, system design, and experimental methodology were developed 100% by the human authors. All AI-assisted text and code were reviewed and verified by the authors, who take full responsibility for the content of this work.

References

Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *USENIX Symposium on Internet Technologies and Systems (USITS 97)*. USENIX Associ-

ation, 1997. URL https://www.usenix.org/conference/usits-97/cost-aware-www-proxy-caching-algorithms.

Mostafa Dehghan, Laurent Massoulié, Don Towsley, Daniel Menasché, and Y. C. Tay. A utility optimization approach to network cache design. In *IEEE INFOCOM 2016*, pp. 1–9, 2016. doi: 10.1109/INFOCOM.2016.7524445. URL https://arxiv.org/abs/1601.06838.

Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A highly efficient cache admission policy. *ACM Transactions on Storage*, 13(4), 2017. doi: 10.1145/3149371. URL https://doi.org/10.1145/3149371. Article 35.

In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. In *Proceedings of Machine Learning and Systems*, volume 6, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/hash/a66caa1703fe34705a4368c3014c1966-Abstract-Conference.html.

NVIDIA. Embedding cache. NVIDIA Dynamo documentation, 2026. URL https://docs.nvidia.com/dynamo/v1.0.0/user-guides/multimodal/embedding-cache. Version 1.0.0. Accessed September 13, 2026.

Haoran Qiu, Anish Biswas, Zihan Zhao, Jayashree Mohan, Alind Khare, Esha Choukse, Íñigo Goiri, Zeyu Zhang, Haiying Shen, Chetan Bansal, Ramachandran Ramjee, and Rodrigo Fonseca. ModServe: Modality- and stage-aware resource disaggregation for scalable multimodal model serving. In *Proceedings of the 2025 ACM Symposium on Cloud Computing*, SoCC '25, pp. 817–830. Association for Computing Machinery, 2025. doi: 10.1145/3772052.3772254. URL https://doi.org/10.1145/3772052.3772254.

vLLM Project. Disaggregated encoder. vLLM documentation, 2026a. URL https://docs.vllm.ai/en/v0.22.1/features/disagg_encoder/. Version 0.22.1. Accessed September 13, 2026.

vLLM Project. EncoderCacheManager. vLLM source code, 2026b. URL https://github.com/vllm-project/vllm/blob/v0.22.1/vllm/v1/core/encoder_cache_manager.py. Version 0.22.1, module vllm.v1.core.encoder_cache_manager. Accessed September 13, 2026.

vLLM Project. Automatic prefix caching. vLLM design documentation, 2026c. URL https://docs.vllm.ai/en/v0.22.1/design/prefix_caching/. Version 0.22.1. Accessed September 13, 2026.

Jin Xu, Zhifang Guo, Jinzheng He, Hangrui Hu, Ting He, Shuai Bai, Keqin Chen, Jialin Wang, Yang Fan, Kai Dang, Bin Zhang, Xiong Wang, Yunfei Chu, and Junyang Lin. Qwen2.5-Omni technical report, 2025. URL https://arxiv.org/abs/2503.20215.

Dongsheng Yang, Austin Li, Kai Li, and Wyatt Lloyd. Learned prefix caching for efficient LLM inference. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-1526. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/414f642a1ea9350006669774cba9bcd4-Abstract-Conference.html.

Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and K. V. Rashmi. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023. URL https://github.com/Thesys-lab/sosp23-s3fifo.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210. USENIX Association, 2024. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

Banghua Zhu, Ying Sheng, Lianmin Zheng, Clark Barrett, Michael I. Jordan, and Jiantao Jiao. Towards optimal caching and model selection for large model inference. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://papers.neurips.cc/paper_files/paper/2023/hash/b914a8fcea5c176cf1ed75c762ce27fd-Abstract-Conference.html.

A Encoder-Output Caching and KV Prefix Caching

Different cached states. An encoder cache retains the visual representation $z(I)$ produced before the image is combined with the text prompt. For the separate visual encoder considered here, this representation can be reused across prompts when the image, encoder, and preprocessing configuration remain unchanged (NVIDIA, 2026). A KV prefix cache instead retains attention keys and values computed by the language model for a prompt prefix (vLLM Project, 2026c). Thus, an encoder hit and a KV prefix hit refer to different intermediate results.

Matching depends on the prefix. vLLM identifies a reusable KV block by its tokens, its parent block hash, and additional information such as multimodal input hashes. The parent hash carries the preceding context, while the multimodal hash distinguishes images that occupy the same placeholder positions. vLLM reuses complete matching blocks (vLLM Project, 2026c). This gives two cases. If two requests place different text before the same image, their image-related KV blocks need not match, even though $z(I)$ is unchanged. If the requests instead share a compatible prefix containing the image and differ only afterward, the complete matching prefix blocks may be reused. The new suffix still needs to be processed. Different questions therefore neither guarantee nor rule out KV prefix reuse; their position in the prompt matters.

Relation to Figure 1. The right panel isolates encoder-output reuse without additional KV prefix reuse. Its separate prompts still require answers, but the figure does not require full prefill recomputation when matching KV blocks are available. Encoder-output caching can avoid visual encoding without a KV prefix hit, while prefix caching can avoid language-model computation for matching blocks (NVIDIA, 2026; vLLM Project, 2026c). The two mechanisms can be used together. Our policy manages encoder outputs and leaves KV prefix matching unchanged.

B Calibration Details and Profile Scope

An auxiliary gap-retention model. We derive equation (3) for an auxiliary long-run model of the known calibration pool $\mathcal{K}$. This is not an exact reformulation of equation (1). Requests in this model are independent with probabilities p_i. For object i, the gap D_i counts the intervening accesses between two consecutive requests for that object, as defined in Section 4. We count memory just after each access: the requested object occupies its service step, and retention can occupy the D_i intervening steps. This convention is local to the auxiliary model and is distinct from the pre-access cache state used to identify misses in equation (1).

For every integer $d \geq 0$, let $x_i(d) \in [0, 1]$ be the probability of discarding object i immediately after its service step and recomputing it at the next access, conditional on a gap of length d. The alternative is to retain it throughout the gap. Write x for the collection of these decisions. Conditioning on the complete gap is allowed only in this auxiliary model; it is not information available to the online policy. We also omit active-reference lifetimes and impose a long-run average budget instead of a separate capacity constraint at every allocation.

Cost and memory accounting. Keeping every known object would use $M = \sum_{i \in \mathcal{K}} m_i$ slots. Discarding object i across a gap of length d saves $m_i d$ slot-steps and incurs recomputation cost

c_i on return. Such intervals occur at rate p_i per access in the stationary reference workload. The auxiliary problem is therefore

$$\begin{aligned} \min_{0\le x_i(d)\le 1} \quad & \sum_{i\in\mathcal{K}} p_i c_i \,\mathbb{E}[x_i(D_i)] \\ \text{subject to} \quad & M - \sum_{i\in\mathcal{K}} p_i m_i \,\mathbb{E}[D_i x_i(D_i)] \le B_{\text{cal}}. \end{aligned} \tag{7}$$

Both expectations use the geometric gap distribution. The constraint counts average occupied slots, not a bound on instantaneous occupancy. Indeed, discarding every object between accesses leaves average occupancy $\sum_i p_i m_i$, since $\mathbb{E}[D_i] = (1-p_i)/p_i$. This explains the budget condition in Section 4.

Deriving the scalar dual. For a multiplier $\lambda \ge 0$, the Lagrangian of equation (7) is

$$\mathcal{L}_{\text{gap}}(x,\lambda) = (M - B_{\text{cal}})\lambda + \sum_{i\in\mathcal{K}} p_i \,\mathbb{E}[(c_i - \lambda m_i D_i) x_i(D_i)]. \tag{8}$$

For fixed λ, each object and gap length can be considered separately. For any real coefficient a, $\min_{0\le x\le 1} ax = \min\{0, a\}$. Consequently,

$$\begin{aligned} \inf_x \mathcal{L}_{\text{gap}}(x,\lambda) &= (M - B_{\text{cal}})\lambda + \sum_{i\in\mathcal{K}} p_i \,\mathbb{E}[\min\{0, c_i - \lambda m_i D_i\}] \\ &= (M - B_{\text{cal}})\lambda - \sum_{i\in\mathcal{K}} p_i \,\mathbb{E}[(\lambda m_i D_i - c_i)^+] \\ &= \overline{D}(\lambda). \end{aligned} \tag{9}$$

Maximizing this dual function gives the calibration problem in equation (3). The positive-part subtraction is thus the result of minimizing over the auxiliary gap decisions, rather than an additional penalty chosen by hand.

Connection to the online rule. At a given multiplier, retention is preferred in the auxiliary calculation when $\lambda m_i d < c_i$; discarding is preferred when the inequality is reversed, and either choice minimizes the Lagrangian at equality. This yields the threshold used in Section 4. During serving, the actual future gap is unknown, so the implementation uses an independent auxiliary draw to grant a finite protection window. It does not implement x using a future gap, nor does the derivation uniquely determine its deadline and pressure-override rules. Those rules adapt the accounting comparison to an online, reference-constrained cache. The derivation establishes the stated calibration function, not optimality or an occupancy guarantee for the deployed policy.

Interpreting the single-round score. Using $\mathbb{E}[D_i] = (1-p_i)/p_i$ and the identity $a - (a-b)^+ = \min\{a, b\}$ for real a, b, equation (3) becomes

$$\overline{D}(\lambda) = \lambda \left(\sum_{i\in\mathcal{K}} p_i m_i - B_{\text{cal}} \right) + \sum_{i\in\mathcal{K}} p_i \,\mathbb{E}\big[\min\{\lambda m_i D_i, c_i\}\big]. \tag{10}$$

The minimum compares holding memory over a modeled gap with paying for recomputation on return. It motivates the runtime comparison but does not reveal which choice will be better for a realized future gap. This simplified dual model is not the exact dual of the causal, reference-constrained problem in equation (1). Its maximization alone gives neither optimality nor a hard-capacity guarantee for the deployed policy.

Computing the score. For $\lambda > 0$ and $0 < p_i < 1$, define $h_i(\lambda) = \lceil c_i/(\lambda m_i) \rceil$. Using the geometric tail $\Pr(D_i \ge h) = (1-p_i)^h$ for integers $h \ge 0$ gives

$$\begin{aligned} \mathbb{E}\big[(\lambda m_i D_i - c_i)^+\big] &= \sum_{d=h_i(\lambda)}^{\infty} (\lambda m_i d - c_i) p_i (1-p_i)^d \\ &= (1-p_i)^{h_i(\lambda)} \left[\lambda m_i \left(h_i(\lambda) + \frac{1-p_i}{p_i} \right) - c_i \right]. \end{aligned} \tag{11}$$

At an integer threshold, the first summand is zero and the formula is unchanged. The expectation is zero when $\lambda = 0$ or $p_i = 1$. One score evaluation uses one expression per known object. The score

is concave and piecewise linear on $\lambda > 0$; a maximizer may be a corner, without a zero ordinary derivative. This numerical calculation uses the stored profile, not GPU execution.

Budget and boundary cases. Since $\overline{D}(\lambda) \le \lambda(\sum_{i\in\mathcal{K}} p_i m_i - B_{\mathrm{cal}}) + \sum_{i\in\mathcal{K}} p_i c_i$, our condition $B_{\mathrm{cal}} > \sum_i p_i m_i$ makes the score tend to $-\infty$ as λ grows and ensures a finite maximizer. When $B_{\mathrm{cal}} \ge M$, zero is a maximizer. The implementation then disables protection rather than creating infinite windows; unknown traffic can still cause pressure. A configured margin $\epsilon \in [0, 1)$ can set $B_{\mathrm{cal}} = (1-\epsilon)B$, provided the condition above is preserved. This changes the calibration target, not physical capacity.

Inputs and probability normalization. The supplied profiling tool estimates c_i from a difference between median miss and hit timings, not an isolated kernel measurement, and m_i may be estimated. Actual slot counts enforce allocation safety. For the possible historical-count construction, use completed observations, count both hits and misses by image identity, and omit zero-count objects from $\mathcal{K}$. Normalized pool parameters are not whole-stream frequencies. If a fraction $\nu \in [0, 1]$ of independently mixed accesses has a new identity, the unconditional probability of known object i is $(1-\nu)p_i$. Calibration uses p_i, while runtime ticks include unknown accesses. The geometric gap is therefore a reference model, not an exact full-stream gap law.

Later profiles. Reusing size and cost estimates may be reasonable while the encoder and execution configuration remain unchanged, even when popularity needs updating. Refreshing only $\lambda^\star$ leaves stale object-level probabilities and fallback values, so the whole profile must be considered. The experiments do not evaluate historical estimation, change detection, or a refresh period. A live profile switch would also need to specify how existing deadlines are handled; no such mechanism is assumed here.

C Implementation Details

This appendix describes the public branch linked in Section 5. Table 1 maps the main components to their source files. The policy code does not replace vLLM's model kernels, request-reference tracking, or scheduler-to-worker messaging. It changes retention decisions through those existing interfaces.

Table 1: Source files and their roles. Paths are relative to the repository root.

Source file	Role
`vllm/v1/core/encoder_cache_policy.py`	Policy interface, profile loading, protection decisions, and forced-eviction values.
`vllm/v1/core/encoder_cache_manager.py`	Reference and slot accounting, access hooks, eligible queue, and eviction notifications.
`vllm/v1/core/sched/scheduler.py`	Cache checks, allocation decisions, encoder scheduling, and consumer load metadata.
`vllm/v1/worker/gpu_model_runner.py`	Encoding, local tensor storage, removal of evicted tensors, and embedding gathering.
`vllm/distributed/ec_transfer/ec_connector/example_connector.py`	Shared-file output transfer between E and PD workers.
`examples/disaggregated/disaggregated_encoder/disagg_epd_proxy.py`	Visual-input dispatch to E, followed by forwarding of the original request to PD.
`tools/precompute_mm_pool.py`	Pool generation, profile measurement, and invocation of the offline calibration solve.

Policy interface and local state. The manager calls `on_arrival` with an image identifier, its actual embedding count, and the updated access tick. The policy returns an absolute protection deadline. It uses the stored profile size for its protection calculation, rather than replacing that estimate with the supplied allocator count. The separate `eviction_value` method is called only for a forced override. Unknown identifiers return no protection and a zero override value. The interface also supports FIFO and a no-retention mode. The latter removes an output when its last reference is released; it can still share the output while requests reference it.

The manager stores request-ID sets and an ordered queue of unreferenced entries. Its pending deadline is transferred to the eligible entry without restarting the window. The worker's `encoder_cache` is a separate hash-to-tensor dictionary. Eviction updates the manager first; the scheduler sends `free_encoder_mm_hashes`, and the worker removes these tensors when updating its state. This is different from freeing a tensor inside the policy callback.

Request and transfer sequence. The proxy's default merged-encoding mode puts all images from a request into one E request and omits its text. It can instead issue separate per-image E requests. After E completes, the proxy passes the original request to PD. The E producer executes the visual encoder for scheduled misses and returns without running the language-model generation path. On PD, the connector loads scheduled external outputs before the runner gathers visual embeddings for the language model. A local PD cache hit needs no file load. The proxy does not perform cache-aware worker selection, and the policy does not maintain a shared eviction order across E workers.

The connector copies newly computed outputs to host memory and writes a SafeTensors file under an image-hash directory. Consumers load it onto their device. This is file-based transfer, not direct device-to-device transport. Producer scheduling does not consult this external store to bypass an E cache miss. Also, local cache eviction is not connected to deletion of a transfer file. The connector contains a deferred-deletion helper, but this scheduler does not call it for locally evicted outputs.

By default, loading does not consume the transfer file. This avoids deleting a file that another PD worker may still read. The comparison launcher clears the transfer directory before starting each policy deployment. The files are not covered by the encoder-cache budget B and can accumulate during a run as new identities arrive. The branch therefore does not provide a bounded persistent-store policy through encoder eviction alone. It also does not establish an end-to-end recovery guarantee for missing transfer files.

Profile construction and identity. The tool exposes pool generation, measurement, and coefficient-solving stages. The generated profile is keyed by the same normal image-byte hash path used by the server, including the model identifier and processor arguments. The text prompt is not part of this image identity. Different byte encodings of a visually identical image need not have the same identifier. User-supplied identifiers follow a different path and must remain consistent across profiling and serving.

For cost measurement, the tool compares median response timings from repeated hit and miss calls, with encoder-cache resets before miss measurements. Its timer uses the first eligible streaming response event, which need not contain a generated token. Thus the recorded c_i is a timing proxy, not an isolated encoder-kernel duration or a measured per-request saving. Its token-query size estimate subtracts a fixed text allowance from reported prompt usage; a pixel-based estimate is available as a fallback. Neither estimate replaces actual slot accounting. The solve stage writes $\lambda^\star$ and the per-hash p_i, m_i, c_i table to JSON. Serving loads this file at initialization. The branch does not implement the historical estimator or live profile-switching mechanism discussed as possible deployments in Section 4.

Configuration. The environment variables `VLLM_ENCODER_CACHE_POLICY`, `VLLM_ENCODER_CACHE_POLICY_CONFIG`, and `VLLM_ENCODER_CACHE_POLICY_SEED` select the policy, profile file, and random seed. The policy names are `fifo`, `nocache`, and `offline`; the last means offline-calibrated parameters, not future-access information. The seed defaults to zero. A positive `VLLM_ENCODER_CACHE_SIZE` overrides the configured cache size independently of the encoder compute budget. The final budgets are raised, if necessary, to fit the largest supported multimodal item. Reported experiments should therefore identify the effective capacity, not only the requested setting. `EC_MERGE_ENCODE` controls merged image dispatch and defaults to one. `EC_STORE_CONSUME_ON_LOAD` defaults to zero; enabling it deletes files after consumer loads and requires care when consumers share a directory.

Instrumentation and work. A positive `VLLM_ENCODER_CACHE_STATS_INTERVAL_SEC` enables periodic cumulative cache statistics; zero disables these logs. Known and unknown counters classify profile membership, not cache hits and misses. Occupancy counters distinguish referenced entries, protected unreferenced entries, unprotected unreferenced entries, and empty slots. Their sum is the cache capacity. Computing a full occupancy snapshot scans eligible entries, as can victim selection. Consequently, metadata storage and selection work grow with the pool and cache populations; the code alone does not establish negligible overhead at large scale. Producer and consumer

timing logs also include synchronization, save, or load work. They should not be labeled as isolated GPU encoding time, and periodic snapshots need not align with individual benchmark boundaries.

D Experimental Details and Additional Results

D.1 Configurations and reporting protocol

Serving configuration. Table 2 records the settings of the main load sweep. The encoder retains outputs across requests within its reported 16,384-slot budget; the shared transfer directory is outside that budget. The baseline and Ours use the same recorded worker settings, except for the E-side policy selection and its profile. The logs label these modes `fifo` and `offline`, respectively. Here `offline` refers to precomputed parameters, not knowledge of future realized requests. The label vLLM denotes native replacement in this common E/PD implementation, not an unmodified release binary.

Table 2: Hardware and recorded serving settings for the main load sweep. Software settings come from startup logs, benchmark arguments, and the saved profile.

Setting	Value
Model and dtype	Qwen3.5-9B; BF16
Hardware and layout	Four H800 GPUs; 1E/3PD
Parallelism per worker	Tensor, pipeline, and data parallel size 1
Execution	Eager; asynchronous scheduling disabled
KV prefix caching	Disabled on E and PD
E retention budget	16,384 embedding slots
E per-step batched-token limit	114,688
E / PD maximum sequences	16 / 128 per worker
E / PD maximum model length	262,144 / 16,384
E / PD GPU-utilization setting	0.01 / 0.70
Transfer	Shared-directory `ECExampleConnector`
Text / output generator targets	256 / 128 tokens
Requests and images	256 requests per point; one image each
Known pool	20 identities; Zipf exponent 0.5
Novel-image probability	0.3 per image access
Arrival process	Poisson; 4, 8, 12, 16 requests/s attempted
Benchmark and engine seed	0
Additional benchmark warmups	0; initial test request still runs
Calibration budget and margin	$B_{\mathrm{cal}} = 16{,}384$; margin 0
Stored coefficient	$\lambda^\star = 8.1125024408 \times 10^{-7}$

The startup logs explicitly set `encoder_cache_size=16384` through an override while `max_num_batched_tokens=114688` remains unchanged. These are separate limits in the recorded main run, not two cache capacities used at different load points. We call 16,384 the reference budget of these experiments rather than a version-independent default for every vLLM deployment. The four-image configuration in Appendix D.5 instead reports 114,688 slots in its cache-manager counters.

The logs identify the engine build as `v0.1.dev16997+g22165ca24`. The measurements use this development build rather than an unmodified v0.22.1 release. The saved model path identifies Qwen3.5-9B; an immutable weight revision is not recorded.

Workload construction. The benchmark selects `mm-fixed-pool`. Its known images are deterministic synthetic PNGs with dimensions 360×640, 720×1280, 1080×1920, and 1440×2560. The probabilities in the profile sum to one over these 20 identities. Since novel images use 30% of the mixed stream, these are conditional pool probabilities, as explained in Appendix B. The profile is supplied rather than estimated from a preceding production trace. The output request disables early EOS stopping, while other generation settings follow the recorded server configuration.

Ordering and cache state. Each policy deployment processes its load points in sequence. The benchmark performs an initial test request before the measured requests, despite having zero additional warmups. There is no cache reset between every point of a sweep, so later points inherit the state created by earlier traffic. The common seed and generator settings support paired comparisons, but the runs are not independent cold-start repetitions or a steady-state saturation test.

Completion checks. A latency comparison is included when both runs report the requested 256 completions, no failed requests, and the expected aggregate 32,768 output tokens, with no matched proxy error for those benchmark request identifiers. Table 3 lists the attempted pairs that fail these checks in the displayed suites. Excluded pairs are left out of latency curves for both methods. All remaining operating points of each selected sweep are retained, including those where vLLM is faster. These aggregate checks do not establish semantic equivalence of generated answers.

Table 3: Attempted pairs excluded from the displayed latency comparisons. Each run submits 256 requests and targets 32,768 output tokens. C1 and C16 are the concurrency suites defined in Appendix D.4. "Complete / failed" follows the benchmark's counters.

Suite	Load or concurrency	Complete / failed		Output tokens	
		vLLM	Ours	vLLM	Ours
Load	4	256 / 0	256 / 0	32,640	32,768
C1	32	256 / 0	256 / 0	32,768	32,640
C16	8	255 / 1	256 / 0	32,512	32,768
C16	64	254 / 2	256 / 0	32,512	32,768

Measurements. TTFT is the client benchmark's time to first output token, and request latency ends when its response completes. Achieved request throughput is completed requests divided by the recorded benchmark duration, including draining the finite batch. It is not the offered request rate or an SLO-qualified capacity estimate. Each shown point is one run per policy. Latency plots use linear axes with nonzero lower limits chosen to contain both policies with padding. Markers are included measurements; straight segments only guide the eye between them. No latency value or marker is inserted at an excluded operating point. Parameter bar charts retain a zero baseline. The retained records contain aggregate quantiles rather than full request-level samples, so we do not construct empirical CDFs or confidence intervals from them. In a 256-request run, only a few observations determine p99.

D.2 Full metrics for the load sweep

Table 4 reports the exact TTFT comparisons underlying Figure 2. A percentage reduction is $100(1 - L_{\text{ours}}/L_{\text{vLLM}})$, where L is the stated latency metric. Positive values indicate a reduction; negative values indicate an increase. At 8 requests/s, the 0.42% change in mean TTFT is small, and p99 increases by 8.26%. At 12 and 16 requests/s, both reported TTFT metrics improve.

Table 4: Load-sweep TTFT, in milliseconds. Percentage reductions are computed from the unrounded benchmark records.

Offered	Mean TTFT			p99 TTFT		
req/s	vLLM	Ours	Reduction (%)	vLLM	Ours	Reduction (%)
8	382.98	381.37	0.42	824.66	892.77	-8.26
12	491.67	459.88	6.47	1198.73	1100.00	8.24
16	600.47	540.26	10.03	1355.66	1178.90	13.04

These metrics change little in the main sweep, so Table 5 replaces two nearly overlapping curves. At 16 requests/s, the mean-request-latency reduction is 1.73% and the throughput increase is 0.45%. Encoder reuse does not remove the downstream generation of the 128-token response. The archive does not provide synchronized queue or utilization measurements that establish why the TTFT gain is small at 8 requests/s. Less encoder-related waiting is a plausible interpretation, not evidence that all four GPUs are idle or that a stability boundary has been measured.

Table 5: Full-request metrics for the same load-sweep pairs as Figure 2. Latency ends at response completion; throughput divides completed requests by the finite run duration.

Offered	Mean request latency (s)			Throughput (requests/s)		
req/s	vLLM	Ours	Reduction (%)	vLLM	Ours	Gain (%)
8	3.660	3.703	-1.18	7.289	7.283	-0.09
12	4.077	4.086	-0.23	10.462	10.436	-0.25
16	4.583	4.504	1.73	13.266	13.326	0.45

D.3 Cache-counter windows and calibration inputs

Cache snapshots. The reported hit fraction uses the increment in hits divided by the increment in hits plus misses between successive saved snapshots. These counters are periodically logged; the snapshots do not occur exactly at client run boundaries. Table 6 includes all four intervals so the differencing can be reconstructed, including the first interval from the run with incomplete client output. Its presence does not validate that first run as a latency comparison. The 12 requests/s interval, for example, contains 257 and 266 counted accesses. This confirms that the intervals should not be treated as exactly 256-request hit-rate measurements.

Table 6: Differences of the main sweep's saved cache snapshots. Hit percentages use the displayed counts. Overrides are forced protection overrides in Ours over the same snapshot interval; vLLM has none. The 4 requests/s client pair is excluded from latency comparisons.

Offered	vLLM			Ours			Ours
req/s	Hits	Misses	Hit (%)	Hits	Misses	Hit (%)	Overrides
4	47	206	18.58	84	169	33.20	36
8	48	205	18.97	84	160	34.43	31
12	44	213	17.12	94	172	35.34	35
16	44	182	19.47	80	150	34.78	32

A hit counter records cache-manager reuse and does not by itself distinguish reuse of an in-flight output from later reuse after its references are released. We do not multiply aggregate hits by the average pool cost to claim exact saved GPU time. The primary suite's queue CSVs contain no samples; they are missing measurements, not measured zero queues.

Profile characterization. The saved main profile has total estimated size $M = 39{,}060$ slots and $\sum_{i \in \mathcal{K}} p_i m_i = 1{,}809.41$ slots. These are profile quantities, not direct measurements of instantaneous cache occupancy. The stored coefficient and equations (4) and (5) yield windows from 17 to 53 access steps and protection probabilities from 0.433 to 0.994. Figure 4 shows all 20 entries, ordered by decreasing known-pool probability. The values describe how the method uses its inputs; they are not observed residency durations or an ablation of individual components. The cost estimates are the hit/miss timing proxies described in Appendix C.

D.4 Concurrency-controlled serving

The archive also contains two single-image concurrency sweeps with the same Qwen3.5-9B, 1E/3PD layout, 16,384-slot E cache, 20-image pool, and 0.3 novelty setting. Here the offered rate is unbounded and the client caps outstanding requests at 8, 16, 24, 32, 48, or 64. The horizontal axis is therefore concurrency, not requests per second. Each point submits 256 requests with the same text and output targets as the load sweep.

We call the two configurations C1 and C16. C1 sets the E worker's maximum sequence count to one and uses $\lambda^\star = 1.2575612501 \times 10^{-6}$. C16 allows 16 E sequences and uses a separate profile with $\lambda^\star = 8.1539395765 \times 10^{-7}$. These are separate operating configurations, not a one-factor ablation: their profiled costs differ as well as the sequence limit. Table 7 reports all their valid pairs.

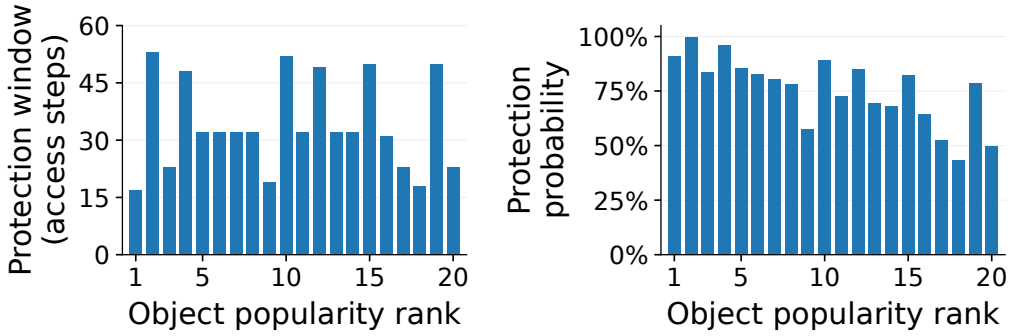

Figure 4: Protection parameters calculated from the recorded main profile. **Left:** Nominal protection window h_i. **Right:** Probability ρ_i of receiving that window on an access. Rank 1 has the largest known-pool probability. These are formula evaluations on stored inputs, not new serving measurements.

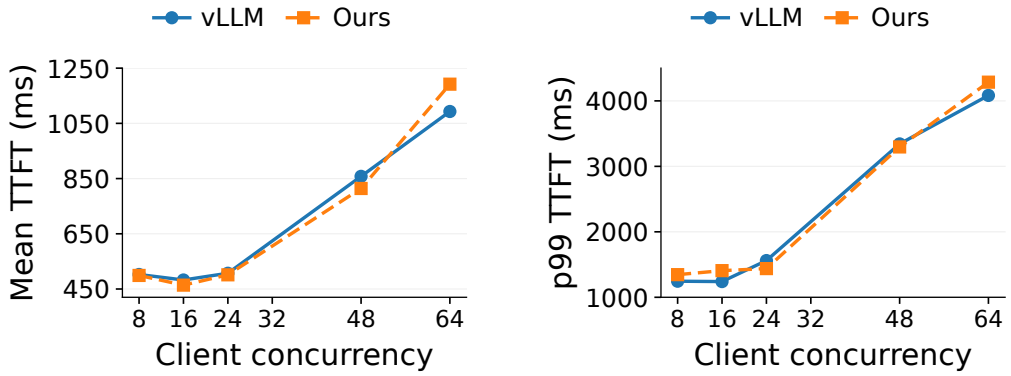

Figure 5: Concurrency sweep C1, with one allowed E sequence. **Left:** Mean TTFT. **Right:** p99 TTFT. Markers denote all valid measured pairs, joined by straight guide lines. Concurrency 32 has no marker because its output accounting is incomplete (Table 3); the connecting segment is not a measurement at 32. Both vertical axes start above zero.

C1 reduces mean TTFT at concurrency 48 from 858.07 to 814.10 ms, or 5.12%. At concurrency 64, mean TTFT instead changes from 1,093.01 to 1,191.93 ms. The lower-concurrency tail results are also mixed. The benefit therefore depends on concurrency and the serving configuration.

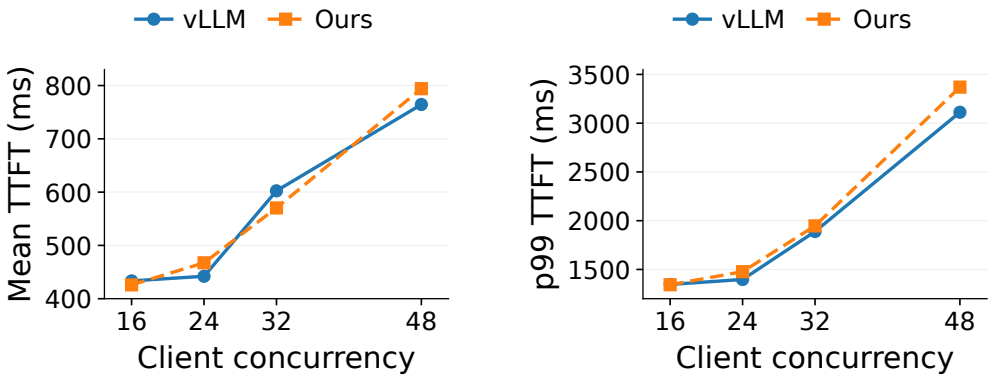

Figure 6: Concurrency sweep C16, with up to 16 E sequences and a separate calibration profile. **Left:** Mean TTFT. **Right:** p99 TTFT. Markers show all valid pairs at concurrency 16, 24, 32, and 48, joined by straight guide lines. The attempted 8 and 64 pairs fail completion or output checks and are omitted. Both vertical axes start above zero.

C16 reduces mean TTFT at concurrency 32 from 602.43 to 570.33 ms, or 5.33%, but does not improve mean or tail TTFT at every concurrency. The two sweeps should not be pooled as repeated estimates of one performance curve.

Table 7: All valid concurrency comparisons. TTFT is in milliseconds and achieved throughput in requests/s. Configurations C1 and C16 have different sequence limits and profiles.

		Mean TTFT		p99 TTFT		Throughput	
Suite	Concurrency	vLLM	Ours	vLLM	Ours	vLLM	Ours
C1	8	502.58	499.12	1245.45	1345.71	2.195	2.150
C1	16	482.67	464.01	1239.94	1407.04	4.237	4.199
C1	24	507.08	501.36	1560.87	1438.30	5.929	5.871
C1	48	858.07	814.10	3342.84	3296.44	9.458	9.463
C1	64	1093.01	1191.93	4083.05	4285.78	11.041	11.049
C16	16	433.40	425.94	1346.19	1343.14	4.486	4.399
C16	24	442.09	467.32	1397.06	1477.14	6.366	6.131
C16	32	602.43	570.33	1888.89	1947.06	7.834	7.700
C16	48	764.44	794.13	3112.10	3369.58	9.829	10.081

D.5 Ablation: Four-image serving with a larger cache

We also report a separate archived configuration with a 114,688-slot encoder cache and four images per request. It uses Qwen3.5-9B in BF16, but has one E worker and one PD worker instead of three PD workers. Each point submits 200 requests with a 256-token text target and a fixed 128-token output target. The 20-image pool uses Zipf exponent 0.5 and novelty probability 0.1. Both policies complete all 200 requests, report no failures, and produce 25,600 output tokens at each of the three offered loads. Their aggregate input-token counts also match, and the proxy logs contain no errors.

The runtime capacity and the stored calibration target differ in this configuration. The profile retains $B_{\text{cal}} = 16{,}384$ and $\lambda^\star = 8.0894258951 \times 10^{-7}$, rather than recalibrating to all 114,688 slots. Table 8 also shows changes in topology, image count, and novelty. We therefore report it as a separate larger-cache operating configuration, not an isolated estimate of the effect of increasing capacity.

Table 8: Configuration differences for the reference and larger-cache experiments. The larger-cache experiment is not a one-factor capacity ablation.

Setting	Reference load sweep	Larger-cache four-image sweep
Encoder / PD workers	1 / 3	1 / 1
Image count per request	1	4
Requests per point	256	200
Novel-image probability	0.3	0.1
Encoder retention capacity	16,384	114,688
E batched-token limit	114,688	114,688
E / PD maximum sequences	16 / 128	16 / 128
E / PD GPU-utilization setting	0.01 / 0.70	0.30 / 0.70
KV prefix caching	Disabled	Disabled
Calibration budget	16,384	16,384

Table 9: Complete larger-cache results. All latency values are in seconds. Throughput is completed requests per second. No points are excluded in this sweep.

Mean TTFT		p99 TTFT		Mean request		Throughput	
vLLM	Ours	vLLM	Ours	vLLM	Ours	vLLM	Ours
5.199	3.890	9.084	7.620	28.158	23.619	3.107	3.325

The mean request latency falls from 28.158 to 23.619 s, a 16.12% reduction, and throughput increases from 3.107 to 3.325 requests/s, or 7.03%. Mean TTFT falls from 5.199 to 3.890 s (25.19%) and p99 TTFT from 9.084 to 7.620 s (16.12%).