PRISM: REQUEST-LEVEL PROFITABILITY CONTROL FOR KV CACHE REUSE

Anonymous authors
Paper under double-blind review

ABSTRACT

Reusing key–value (KV) states is a central approach to reducing redundant computation in long-context language model serving. Existing serving systems cache shared prefixes to avoid repeated prefill, while modular caching and cache fusion extend reuse to structured context components. Cache storage, transfer, and sharing mechanisms further expand reuse across serving engines and compatible model variants. These mechanisms make candidate states available, but cache availability alone does not determine whether reuse is profitable for a request. Cache construction, transfer, and reuse preparation can consume the saved target computation, making compatibility, residency, and workload characteristics part of the latency decision. We introduce PRISM, which adds a request-level profitability admission layer after a candidate cache becomes available. The admission layer jointly checks cache compatibility, physical target residency, and a profiled cost margin, and routes rejected or failed requests to Native execution. Admitted requests execute a contiguous history suffix and the query against a read-only cache view, avoiding sparse cache assembly. On ARC and OpenBookQA, the resident executor reduces target-side latency by 37.6–53.9% in controlled long-history experiments, with the same observed accuracy as Native. On the paired evaluation, Tail is faster than the sparse-assembly proxy on 127/128 questions, with positive reductions persisting across fixed history bins, timing repeats, and tested tail ratios.

1 INTRODUCTION

Large language models (LLMs) are increasingly deployed for long-context question answering and extended reasoning, where both input histories and generated continuations can be lengthy. During autoregressive inference, models cache the key–value (KV) states of processed tokens to avoid recomputing these states at each decoding step. Prefix caching extends this benefit across requests: when requests share the same input prefix under the same model, the corresponding KV states can be reused, avoiding repeated prefix computation and reducing inference latency (Kwon et al., 2023; Zheng et al., 2024).

Recent systems extend this reuse beyond a shared prefix. Modular and position-aware methods reuse structured context components (Gim et al., 2024; Hu et al., 2025), while cache fusion combines cached states with selective recomputation to recover interactions across context chunks (Yao et al., 2025). Cache storage and transfer systems make states available across storage tiers and serving engines, and specialized sharing methods enable reuse across compatible model variants (Liu et al., 2025; 2026; Woo et al., 2026). These mechanisms broaden how candidate states can be obtained and executed, but they leave a request-level decision after cache availability: whether the complete reuse path is compatible, resident, and profitable for the current request.

The request-level admission problem. Once a candidate state is available, the runtime must decide whether to use it for the current request. Native execution processes the full history and query directly. Reuse avoids part of that target computation, but the candidate may fail compatibility checks, may not reside on the target device, or may require construction and preparation before execution. PRISM treats this as a request-level profitability admission problem: compatibility determines whether reuse is legal, residency determines whether the saved computation is immediately

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.

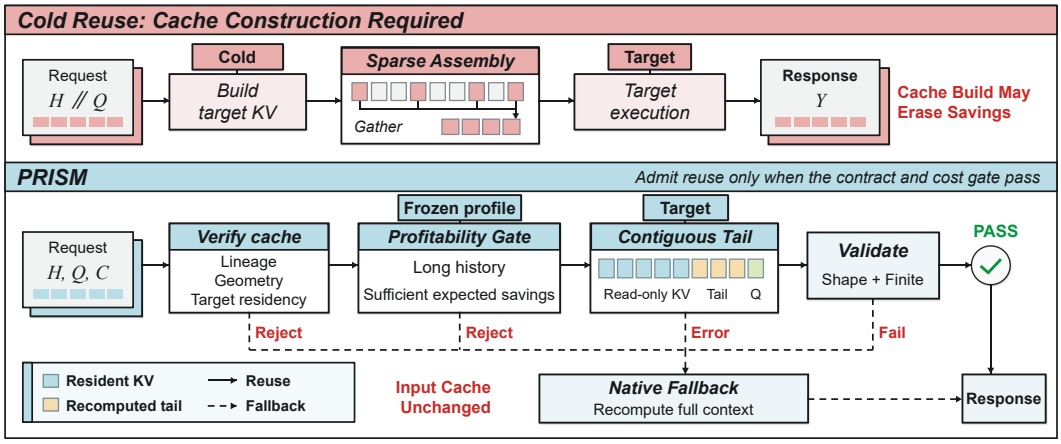


Figure 1: Motivation for profitability-gated KV reuse. Top: cache construction and sparse assembly add work before reuse can save target computation. Bottom: PRISM checks compatibility and target residency, then admits reuse when history length and profiled savings meet the admission conditions. Rejection or failed execution validation triggers Native fallback.

available, and profiled cost determines whether the complete path is worthwhile. Admission rejection or failed execution validation follows the same Native fallback path on the full input.

The profitability boundary. Our controlled analysis of target-native KV reuse identifies two cost regimes. With a compatible cache already resident on the target device, contiguous-tail execution avoids repeated target computation and yields larger savings as the controlled history grows, reaching 37.6–53.9% lower target-side latency than Native in the evaluated long-history screens. When construction is charged to the same request, the added work can consume that resident saving. Cache residency and history length therefore define a practical profitability boundary, which motivates an admission layer that evaluates the complete request path rather than cache availability alone.

PRISM. We introduce PRISM, a request-level profitability admission layer that operates after a candidate cache becomes available. Direct full reuse is the query-only reference when a complete target-resident state is directly consumable; the evaluated PRISM candidate is the contiguous-tail path. The controller verifies cache lineage, tensor geometry, position span, and physical target residency, then checks minimum history length and a profiled savings margin. Figure 1 (bottom) illustrates this admission principle. Admitted requests execute a contiguous history suffix and the query against a read-only cache view, avoiding sparse cache assembly. The candidate is returned after finite-value and output-shape checks; admission rejection, execution errors, or failed validation route the request to Native execution on the full input. PRISM therefore connects cache compatibility and residency to an explicit cost decision and a fail-closed execution choice.

We evaluate the resident Tail executor and the admission layer on ARC and OpenBookQA. Controlled long-history experiments show 37.6–53.9% target-side latency reduction relative to Native with the same observed accuracy. In the paired comparison with the sparse-assembly proxy, Tail is faster on 127/128 questions, and the positive reduction remains stable across fixed history bins, timing repeats, and tested tail ratios. Construction-inclusive measurements locate the cost boundary for the complete request path, while offline policy replay evaluates when the admission margin supports reuse. Together, these evaluations connect resident execution gains to a request-level profitability decision.

Our contributions are as follows:

1. **A request-level profitability boundary.** We characterize how cache residency and history length shape the latency benefit of contiguous-tail reuse, and how construction cost changes the complete request-level comparison.
2. **A profitability-gated admission layer.** We combine cache compatibility and residency checks with history and cost gates, read-only contiguous-tail execution, and Native fallback.

3. **Paired experimental evidence.** We compare the resident Tail path with Native and a sparse-assembly proxy, and show stable positive gains across paired questions, history bins, timing repeats, and tail ratios.

2 Related Work

Existing KV systems improve different parts of the reuse pipeline: they retain state, locate reusable blocks, assemble cached and fresh context, or transfer state across model instances (Kwon et al., 2023; Yao et al., 2025; Liu et al., 2026). Our focus is the decision that follows these mechanisms. Once a candidate state exists, a serving system still has to decide whether it is compatible and cost-effective for the current request, and when a failed check should trigger Native recomputation. PRISM treats this runtime boundary as a first-class systems problem rather than as an implicit consequence of cache availability.

Serving substrates and structured reuse. vLLM manages and shares KV blocks through PagedAttention (Kwon et al., 2023), while SGLang uses RadixAttention to index and reuse shared prefixes (Zheng et al., 2024). LMCache supports cache offloading and transfer across storage tiers and serving engines, and studies when cache loading is faster than prefill (Liu et al., 2025). Prompt Cache and EPIC enable modular or position-independent context reuse (Gim et al., 2024; Hu et al., 2025). These storage and assembly mechanisms provide candidate states for a request-level choice between reuse and Native recomputation. PRISM complements this line by validating model and tensor geometry, checking physical target residency, and applying the same decision rule to warm and cold requests.

Fusion, partial recomputation, and model-variant sharing. CacheBlend combines cached and recomputed context for retrieval-augmented generation, using selective recomputation to recover cross-chunk attention while reusing cached states (Yao et al., 2025). This design motivates the sparse-assembly proxy in our evaluation. A related line studies reuse across model variants. DroidSpeak selectively recomputes layers when sharing KV state across fine-tuned variants with the same architecture (Liu et al., 2026). ICaRus freezes a shared logical encoder and fine-tunes logical decoders to preserve identical KV states across specialized models (Woo et al., 2026). Activated LoRA applies adapter weights only from an invocation point onward, allowing the preceding base-model cache to be reused (Greenewald et al., 2025). PRISM addresses the complementary question of when a candidate reuse path should be admitted: an unknown lineage, incompatible shape or position span, cold construction, or insufficient history is routed to Native through the same admission rule.

Cache lifecycle and quality control. KV reuse also interacts with edits, model updates, compression, and eviction. KVEraser replaces a selected interval with learned KV states for localized context removal (Li et al., 2026). KVShareArena benchmarks reuse across contexts and model versions, reports cache construction separately, and derives a cache-build amortization condition (Shi & Lou, 2026). KVDiagnosis examines failures under cache compression (Qiu et al., 2026), while Trust the Mass studies eviction quality and the memory actually retained by implementations (Shi & Gu, 2026). PRISM turns compatibility, physical target residency, and a request-level latency margin into an executable admission rule with Native fallback.

3 Preliminaries

Request and target-native state. We represent a request as $x = (m, H, Q)$, where $m = (M, \mathcal{T})$ identifies the target model and tokenizer, $H = (h_1, \ldots, h_\ell)$ is the tokenized history, and Q is the query. For target model f_θ, Native evaluates the complete sequence,

$$Y_N(x) = f_\theta(H \,\|\, Q), \qquad T_N(x) = T_\theta^{\text{target}}(H, Q).$$

A target-native cache is $C_\theta(H) = \mathrm{KV}_\theta(H)$. When this complete state is directly consumable, full reuse provides the query-only reference path. PRISM instantiates the reusable candidate as a contiguous-tail path and verifies cache metadata before skipping target computation; task accuracy is evaluated separately.

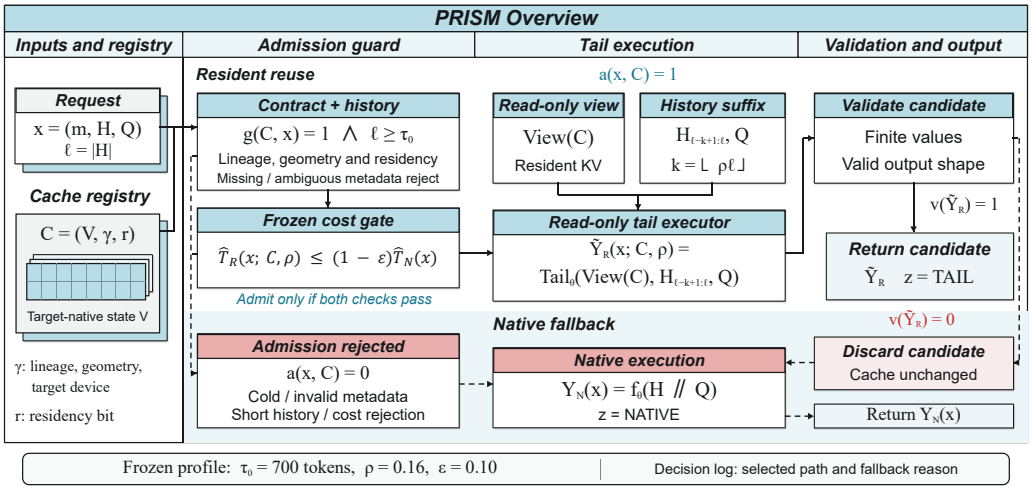


Figure 2: PRISM design and execution flow. The admission guard checks the cache contract, history length, and profiled savings before invoking the read-only tail executor. The upper path returns a candidate that passes finite-value and output-shape checks. Admission rejection or failed execution validation follows the lower Native fallback path, leaving the input cache unchanged. The bottom strip records the frozen operating profile and decision log.

Resident and cold cost. For history length ℓ and tail ratio ρ, let $k_\rho(\ell) = \lfloor \rho\ell \rfloor$ and let $H_{\ell-k+1:\ell}$ denote the contiguous suffix of $k = k_\rho(\ell)$ tokens. Given a resident cache handle C, the candidate output and its measured cost are

$$\begin{aligned} \widetilde{Y}_R(x; C, \rho) &= \mathrm{Tail}_\theta(\mathrm{View}(C), H_{\ell-k+1:\ell}, Q)\,, \\ T_R(x; C, \rho) &= T_{\mathrm{prep}}(C, \ell - k) + T_\theta^{\mathrm{target}}(H_{\ell-k+1:\ell}, Q), \\ T_C(x; C, \rho) &= T_{\mathrm{build}}(C_\theta(H)) + T_R(x; C, \rho). \end{aligned} \tag{1}$$

$\mathrm{View}(C)$ denotes read-only access to the input cache; the measured executor copies its prefix into a request-local cache before recomputing the suffix. T_{prep} includes this copy. T_R is the target-resident mechanism cost; T_C also charges cache construction and is the request-level cold cost. The timing breakdown calls construction the source stage and resident execution, including preparation, the target stage.

4 PRISM: Profitability-Gated KV Reuse

Overview. Figure 2 shows how PRISM turns the profitability boundary into an execution choice through four stages: inputs and registry, admission guard, tail execution, and validation and output. Given a request $x = (m, H, Q)$, the cache registry supplies a target-native handle C. The admission guard first checks cache compatibility, target residency, and minimum history length, then uses a frozen cost profile to test whether predicted reuse savings meet the required margin. Only requests passing both checks enter the resident reuse path.

For an admitted request, the tail executor copies the prefix from a read-only view of C into a request-local cache, then processes the contiguous history suffix of $k = \lfloor \rho\ell \rfloor$ tokens and the query Q. The candidate is returned with the Tail path label after finite-value and output-shape checks. Rejected requests follow the Native path along the bottom of the figure; an execution error or failed validation also triggers this fallback, discarding the candidate and preserving the input cache. Native processes the complete history and query, so each request returns either a validated tail candidate or the Native output.

Algorithm 1 Fail-closed PRISM admission and execution.

Require: request $x = (m, H, Q)$, cache handle C, profile $(\tau_0, \rho, \varepsilon)$
Ensure: output Y and path label z
if $g(C, x) = 0 \vee |H| < \tau_0$ **then** ▷ contract and minimum-history gate
  return $(\mathrm{Native}_\theta(x), \textsc{native})$
end if
if $\widehat{T}_R(x; C, \rho) > (1 - \varepsilon)\widehat{T}_N(x)$ **then** ▷ reject when profiled savings miss the margin
  return $(\mathrm{Native}_\theta(x), \textsc{native})$
end if
$k \leftarrow \lfloor \rho |H| \rfloor$ ▷ select a contiguous suffix
$\widetilde{Y} \leftarrow \mathrm{Tail}_\theta(\mathrm{View}(C), H_{|H|-k+1:|H|}, Q)$ ▷ $\perp$ on execution error
if $\neg v(\widetilde{Y})$ **then** ▷ finite-value and shape postcondition
  return $(\mathrm{Native}_\theta(x), \textsc{native})$
end if
return $(\widetilde{Y}, \textsc{tail})$

4.1 Cache contract and admission rule

Compatibility and residency. The registry looks up the exact tokenized history H before returning a cache handle $C = (V, \gamma, r)$, where V is the target-native KV state, $r \in \{0, 1\}$ is its residency bit, and

$$\gamma = (h_M, h_{\mathcal{T}}, s, \delta, p, d)$$

contains model and tokenizer lineage hashes, tensor shape s, dtype δ, sequence-position span p, and target device d. For request x, define

$$\gamma(x) = \big(h_M(x), h_{\mathcal{T}}(x), s(x), \delta(x), p(H), d(x)\big).$$

The structural compatibility predicate is

$$g(C, x) \triangleq r\, \mathbb{I}[\gamma = \gamma(x)]\,. \tag{2}$$

A missing or ambiguous field makes $g(C, x) = 0$, so the request follows Native execution.

Profitability. With profiled costs $\widehat{T}_R$ and $\widehat{T}_N$, the frozen admission decision is

$$a(x, C) = \mathbb{I}\Big[g(C, x) = 1 \ \wedge\ \ell \geq \tau_0 \ \wedge\ \widehat{T}_R(x; C, \rho) \leq (1 - \varepsilon)\widehat{T}_N(x)\Big]\,. \tag{3}$$

The resident operating profile uses $\tau_0 = 700$ history tokens, $\rho = 0.16$, and $\varepsilon = 0.10$. In the specified controller, admission uses cache metadata and profiled costs. Section 5.4 separately examines whether natural-length calibration data support a history threshold under the same 10% cost margin.

4.2 Fail-closed execution

The candidate is produced from a read-only cache view and is exposed only after finite-value and shape checks. Let

$$v(\widetilde{Y}) = \mathbb{I}[\mathrm{Finite}(\widetilde{Y}) \wedge \mathrm{ShapeValid}(\widetilde{Y})]$$

denote the executor postcondition. An execution error is represented by $\perp$, with $v(\perp) = 0$. The controller-level output is

$$Y(x, C) = \begin{cases} \widetilde{Y}_R(x; C, \rho), & a(x, C) = 1 \ \wedge\ v(\widetilde{Y}_R) = 1, \\ Y_N(x), & \text{otherwise}, \end{cases} \tag{4}$$

where $\widetilde{Y}_R$ is the candidate returned by the tail executor. Equation (4) specifies execution validity; benchmark correctness is measured separately on the sealed questions. The input cache remains read-only throughout candidate execution, so an execution error or failed check leaves it unchanged and triggers Native fallback.

The design assigns three explicit responsibilities: the registry supplies C and γ, the guard computes g and a, and the executor returns either a validated tail candidate or Native. Path labels distinguish Tail from Native. Appendix B analyzes the cost boundary, admission margin, and execution invariants; Appendix C discusses deployment-scale scheduling.

5 EXPERIMENTS

The experiments address three questions: **when resident reuse reduces latency**, **how the Tail path compares with a sparse-assembly proxy**, and **which requests the cost gate sends to Native**. Sections 5.2–5.4 follow this order, and Section 5.5 examines changes in target and execution mechanism.

5.1 EXPERIMENTAL SETUP

Datasets and Models. We evaluate ARC (Clark et al., 2018) and OpenBookQA (Mihaylov et al., 2018), two multiple-choice question-answering tasks with different content distributions. The formal package spans 128 sealed questions across both tasks; the controlled long-history screens retain matched questions while extending history to $8\times$ and $16\times$. The formal natural-length histories contain 96–231 tokens before the tail operation, whereas the controlled screens stretch the matched histories to 792–2,496 tokens. The ARC and OpenBookQA experiments use Qwen3-4B-Instruct-2507 (Qwen Team, 2025), with its tokenizer, prompt format, parser, and bfloat16 execution fixed across comparisons. The directional checks use Qwen3-4B-Thinking-2507 as the target: Tail and sparse assembly use target-native caches, while the DroidSpeak mechanism transfers a cache from the Instruct variant.

Baselines. We compare three execution paths under the primary target-native protocol and use direct full reuse as a reference in the separate runtime. *Native* runs the target model on the complete history and query without constructing a reusable source cache. *CacheBlend-style* (Yao et al., 2025) is a target-native sparse-assembly proxy that selects cached pieces and assembles them before target execution. *Tail* keeps the target-native dense history cache and recomputes one contiguous suffix together with the query. Direct full reuse is the query-only reference when the complete cache can be consumed without recomputation. Tail uses ratio $\rho = 0.16$ in the main operating point; the sensitivity check evaluates $\rho = 0.08$. We use Tail for the measured partial-recompute path and PRISM for the controller that admits it only after lineage, geometry, residency, and cost checks. The separate CacheBlend and DroidSpeak comparisons in Section 5.5 use mechanism implementations with their own matched Native references.

Environment. The target-native experiments use PyTorch 2.6.0 and Transformers 4.52.4 (Paszke et al., 2019; Wolf et al., 2020), seed 20260917, prefill chunks of up to 1024 tokens, and at most 64 generated tokens. All methods use the same timing code within a comparison cell. We report target-side warm time and cold total time, where total time includes cache construction plus target execution. Accuracy and parse failure are computed with the task answer extractor. Long-history screens use one timing repeat per method; the formal package has 128 unique questions across both tasks and three timing repeats per method, giving 1,152 raw rows. For the primary comparisons, we average timing repeats within each question, then average the per-question ratios $1-T_{\text{TAIL}}/T_{\text{reference}}$; the question is the unit of aggregation.

Evaluation protocol. The offline natural-length replay takes the median of the three timings per question and uses a deterministic calibration/holdout split per dataset with a fixed 10% cost margin. It searches history thresholds on calibration data, requiring admitted candidates, no observed Tail correctness harm, and aggregate policy cost below 90% of matched Native cost. The selected policy is evaluated once on holdout groups. This calibration analysis tests whether a profitable natural-length policy exists; the resident operating profile in Section 4 remains fixed. Controlled history extensions retain the identity of their matched questions.

5.2 MAIN EFFECTIVENESS RESULTS

Resident reuse scales with history length. The first experiment measures the resident execution regime. Table 1 reports the target-resident comparison at $8\times$ and $16\times$ history; Table 2 gives the per-task latencies. At $8\times$, the tail path reduces mean target-side latency by 37.6%; at $16\times$, the reduction is 53.9%. Every paired row is faster in the warm comparison, and Native and Tail have the same observed accuracy on these screening questions. The matched histories isolate the effect of history length.

Table 1: Tail versus Native at two controlled history lengths. Reduction is the mean paired ratio $1 - T_{\text{TAIL}}/T_{\text{NATIVE}}$, with descriptive paired bootstrap intervals; Native and Tail accuracy is 87.50%.

History	Resident warm			Cold total	
	Reduction (%)	95% CI (%)	Wins	Reduction (%)	Wins
8×	37.56	[34.94, 40.34]	ALL	−13.99	NONE
16×	53.90	[52.37, 55.60]	ALL	−15.37	NONE

Table 2: Resident target latency by task and controlled history. Reduction averages $1 - T_{\text{TAIL}}/T_{\text{NATIVE}}$ across paired timings; Native and Tail have identical correctness within each task.

History	Dataset	Mean tokens	Warm time (s)		Reduction	Wins
			Native	Tail	(%)	
8×	ARC	1042	0.3159	0.1938	38.37	ALL
	OpenBookQA	1000	0.3052	0.1922	36.76	ALL
16×	ARC	2084	0.4991	0.2252	54.60	ALL
	OpenBookQA	2000	0.4820	0.2248	53.21	ALL

Table 3: Tail versus the sparse proxy at two controlled history lengths. Reduction is the mean paired ratio $1 - T_{\text{TAIL}}/T_{\text{proxy}}$; the pooled row summarizes the paired comparisons.

History	Resident warm		Cold total	
	Reduction (%)	Wins	Reduction (%)	Wins
8×	12.75	ALL	11.96	ALL
16×	26.07	ALL	11.69	ALL
Pooled	19.41	ALL	11.83	ALL

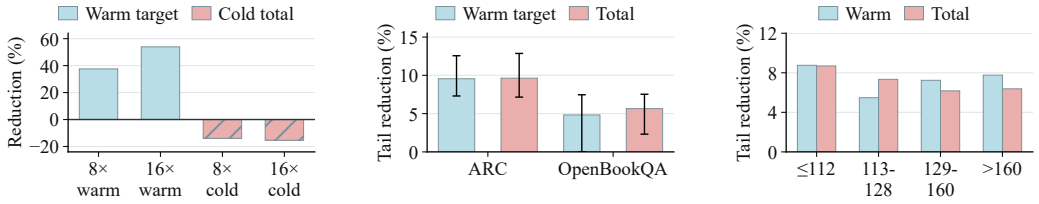
Tail execution improves on the sparse proxy. The tail path is faster than the CacheBlend-style sparse-assembly proxy across the controlled paired comparisons, by 19.41% in warm time and 11.83% in cold total time on average (Table 3). Figure 3 places the resident/cold boundary beside the task and history-bin comparisons. The implementations also differ in their source construction, so the comparison assesses the complete mechanisms. Native supplies the request-level cold reference, which includes no separate cache-construction stage.

5.3 Mechanism analysis

The mechanism persists on 128 unique questions. The second experiment compares Tail and sparse assembly on 128 natural-length questions with repeated timing. After averaging three repeats per question, Tail reduces the mean paired warm-time ratio by 7.19% relative to the CacheBlend-style proxy (paired bootstrap (Efron & Tibshirani, 1994) 95% CI [4.23, 9.49]%) and the total-time ratio by 7.64% (CI [5.43, 9.70]%), with 127/128 positive pairs in both metrics. ARC gives 9.55% warm and 9.64% total reduction; OpenBookQA gives 4.82% and 5.65%. Tail and Native produce identical correctness on all 128 questions (ARC 93.75%, OpenBookQA 78.13%). Table 4 reports the task-level results.

Source and target costs both contribute, and timing repeats agree. Tail reduces mean paired cache-construction time relative to the proxy by 7.60% on ARC and 8.44% on OpenBookQA, and mean paired target-side warm time by 9.55% and 4.82%, respectively. Per-repeat estimates are 7.70%, 7.59%, and 7.63% total reduction for repeats 0, 1, and 2, and each repeat retains 127/128 positive pairs. Figure 4 compares these contributions with repeat stability and tail-ratio sensitivity.

The effect survives fixed history stratification. Table 5 reports warm and total reductions within fixed token-count bins on the same 128 natural-length questions. Total-time reduction is positive in every bin, with 127/128 positive pairs overall.



(a) Resident versus cold cost. **(b)** Formal effect by task. **(c)** Effect across history bins.

Figure 3: Resident gains and execution efficiency. (a) Tail reduction relative to Native in resident warm and cold total time; (b) Tail reduction relative to the sparse proxy by task, with paired bootstrap intervals; (c) the same comparison across fixed history bins.

Table 4: Formal results, averaging three timing repeats per question. L: mean history tokens; CB: sparse proxy; N: Native. Native and Tail share accuracy on the pooled set. Wins report positive Tail–proxy comparisons.

Dataset	L	**Tail reduction (%)**			**Accuracy**	**Wins**
		Warm/CB	Total/CB	Total/N	(%)	
ARC	128.3	9.55	9.64	−18.61	93.75	ALL
OpenBookQA	110.9	4.82	5.65	−19.68	78.13	63/64
Pooled	119.6	7.19	7.64	−19.14	85.94	127/128

Table 5: Tail reduction relative to the sparse proxy across fixed history-length bins of the 128-question formal set. Reductions average $1 - T_{\text{TAIL}}/T_{\text{proxy}}$ per question after three timing repeats.

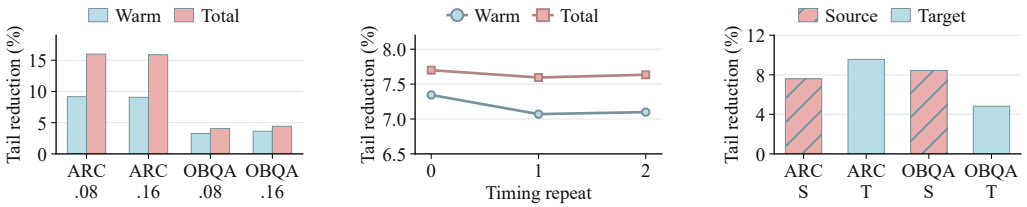
History tokens	**Warm reduction (%)**	**Total reduction (%)**
≤ 112	8.76	8.69
113–128	5.48	7.34
129–160	7.24	6.17
> 160	7.76	6.38

The execution gain is stable across tail ratios. We vary the tail ratio around the operating point, reporting reductions as ratios of task-level mean times. On the completed screening rows, Native and Tail have identical accuracy at $\rho = 0.08$ and $\rho = 0.16$, while Tail remains faster than the sparse proxy in every reported cell. ARC warm/total reductions are 9.16%/15.99% at $\rho = 0.08$ and 9.06%/15.88% at $\rho = 0.16$; OpenBookQA gives 3.26%/4.06% and 3.63%/4.43%, respectively. The similar values across ratios support using $\rho = 0.16$ as a fixed operating point for the evaluated workload.

5.4 Admission Boundary

The cost gate locates the request-level boundary. Construction-inclusive replay evaluates the complete reuse path after cache availability, using separate calibration and holdout groups with a fixed 10% cost margin. The resident/cold measurements establish the corresponding boundary: Tail reduces resident warm time by 37.6% and 53.9% at 8× and 16× history, respectively, with 8/8 paired wins in both regimes, whereas charging cache construction makes the corresponding cold totals 14.0% and 15.4% slower than Native. On the natural-length replay, the warm diagnostic ratios are 0.9712/0.9946 for ARC and 1.0011/0.9990 for OpenBookQA on calibration/holdout groups, all above the fixed 0.90 admission cutoff. The replay therefore identifies the operating boundary: resident long-history requests clear the measured margin, while natural-length requests are routed to Native.

Resident execution and admission form one request-level decision. The paired evaluation shows that the Tail path is consistently less costly than sparse assembly, with 127/128 positive pairs and stable reductions across the tested history bins, timing repeats, and tail ratios. The admission layer then compares this complete reuse path with Native, including cache preparation and



(a) Tail-ratio sensitivity. **(b)** Timing-repeat stability. **(c)** Source/target contribution.

Figure 4: Mechanism robustness. Tail reductions relative to the sparse proxy: (a) tail-ratio sensitivity on the screening set; (b) three timing repeats and (c) cache-construction and target-execution contributions on the 128-question formal set.

Table 6: Natural-length policy replay.

Dataset	Accuracy	Warm diagnostic ratio	
	(%)	Calibration	Holdout
ARC	93.75	0.9712	0.9946
OpenBookQA	78.13	1.0011	0.9990

Table 7: Thinking-target Tail versus matched Native on the paired screening set. Total time includes cache construction.

Dataset	Accuracy (%)		Total time (s)		Slowdown
	Native	Tail	Native	Tail	(%)
ARC	25.00	25.00	1.827	1.917	4.92
OpenBookQA	12.50	6.25	1.821	1.855	1.87

construction. PRISM thus preserves resident gains when the cache contract and cost margin hold, while routing requests outside that profitability region to Native. Compatibility, residency, cost, and fallback are evaluated as one request-level control decision.

5.5 Transfer and baseline boundary

Construction-inclusive checks. With the Thinking target, cold Tail incurs 4.92% and 1.87% overhead relative to matched Native on ARC and OpenBookQA, with no paired timing wins in either task (Table 7). CacheBlend (Yao et al., 2025) and DroidSpeak (Liu et al., 2026) mechanism controls incur 2.55–6.46% overhead. In a separate runtime, minimal full reuse is the query-only reference and Tail is within 0.41%/0.23% of it on ARC/OpenBookQA; the CacheBlend and DroidSpeak mechanisms incur 39.52–47.32% overhead against their matched Native baselines. These checks use different references and question sets; Appendix A reports the remaining paired timing and accuracy details.

6 Conclusion

PRISM adds a request-level profitability admission layer after a candidate KV cache becomes available, complementing existing storage, fusion, and sharing mechanisms. The layer unifies cache compatibility, physical target residency, profiled cost, and fail-closed Native fallback, while the read-only contiguous-tail executor realizes admitted reuse. Controlled long-history experiments show 37.6–53.9% lower target-side latency than Native with the same observed accuracy. On the paired evaluation, Tail is faster than the sparse-assembly proxy on 127/128 questions, with positive reductions across fixed history bins, timing repeats, and tested tail ratios. Construction-inclusive measurements and offline replay locate the profitability boundary for the complete request path. Together, these results position PRISM as a request-level controller that turns cache availability into an explicit decision about when reuse is worth executing.

AI USE STATEMENT

During manuscript preparation, the authors used generative artificial intelligence (AI) tools for language editing, manuscript organization, figure and table refinement, interpretation of experimental results, reference checking, and LaTeX debugging. AI-assisted suggestions for presentation and analysis were checked against the source data and implementation by the authors. The authors take responsibility for the final text, figures, claims, and experimental results.

REFERENCES

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try ARC, the AI2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018. URL https://arxiv.org/abs/1803.05457.

Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. Chapman and Hall/CRC, 1994. ISBN 9780412042317. URL https://www.routledge.com/An-Introduction-to-the-Bootstrap/Efron-Tibshirani/p/book/9780412042317.

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, pp. 325–338, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/hash/a66caa1703fe34705a4368c3014c1966-Abstract-Conference.html.

Kristjan Greenewald, Luis A. Lastras, Thomas Parnell, Vraj Shah, Lucian Popa, Giulio Zizzo, Chulaka Gunasekara, Ambrish Rawat, and David Daniel Cox. Activated LoRA: Fine-tuned LLMs for intrinsics. In *Advances in Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=S6SKa97Gm0.

Junhao Hu, Wenrui Huang, Weidong Wang, Haoyi Wang, Tiancheng Hu, Qin Zhang, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. EPIC: Efficient position-independent caching for serving large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 24391–24402. PMLR, 2025. URL https://proceedings.mlr.press/v267/hu25j.html.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Mufei Li, Shikun Liu, Dongqi Fu, Haoyu Wang, Yinglong Xia, Hong Li, Hong Yan, and Pan Li. KVEraser: Learning to steer KV cache for efficient localized context erasing. *arXiv preprint arXiv:2606.17034*, 2026. URL https://arxiv.org/abs/2606.17034.

Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, and Junchen Jiang. LMCache: An efficient KV cache layer for enterprise-scale LLM inference. *arXiv preprint arXiv:2510.09665*, 2025. URL https://arxiv.org/abs/2510.09665.

Yuhan Liu, Yuyang Huang, Jiayi Yao, Shaoting Feng, Zhuohan Gu, Kuntai Du, Hanchen Li, Yihua Cheng, Junchen Jiang, Shan Lu, Madan Musuvathi, and Esha Choukse. DroidSpeak: KV cache sharing across fine-tuned model variants. In *Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 319–338, 2026. URL https://www.usenix.org/conference/nsdi26/presentation/liu-yuhan.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2381–2391, 2018. doi: 10.18653/v1/D18-1260. URL https://aclanthology.org/D18-1260/.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Chen Qiu, Ziwu Liu, Chao Fei, Guozhong Li, and Panos Kalnis. KVDiagnosis: A diagnostic benchmark for KV-cache compression in long-context language models. *arXiv preprint arXiv:2608.09412*, 2026. URL https://arxiv.org/abs/2608.09412.

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

Jack Shi and Jerry Gu. Trust the mass: Forced weights in KV-cache eviction. *arXiv preprint arXiv:2608.25230*, 2026. URL https://arxiv.org/abs/2608.25230.

Xi Shi and Qian Lou. KVShareArena: KV-cache reuse across contexts and model checkpoints. *arXiv preprint arXiv:2609.10266*, 2026. URL https://arxiv.org/abs/2609.10266.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, 2020. doi: 10.18653/v1/2020.emnlp-demos.6. URL https://aclanthology.org/2020.emnlp-demos.6/.

Sunghyeon Woo, Jaeeun Kil, Hoseung Kim, Minsub Kim, Joonghoon Kim, Ahreum Seo, Sungjae Lee, Minjung Jo, Jiwon Ryu, Baeseong Park, Se Jung Kwon, and Dongsoo Lee. ICaRus: Identical cache reuse for efficient multi-model inference. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/37f6be32f832caf0f7980469fb06165b-Abstract-Conference.html.

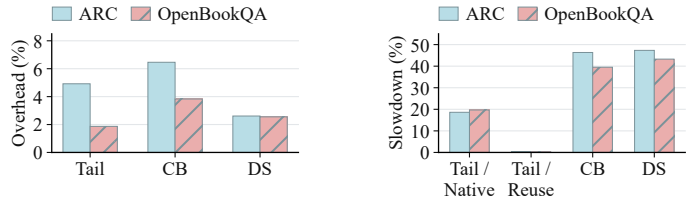
Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 94–109, 2025. doi: 10.1145/3689031.3696098. URL https://doi.org/10.1145/3689031.3696098.

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, pp. 62557–62583, 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.

A TRANSFER AND BASELINE AUDITS

Figure 5 summarizes the Thinking-target screen and cold-cost comparisons. The Tail/Native bars use the natural-length target-native package in Table 4; the remaining bars use the separate runtime.

Full-reuse reference. In the separate runtime, direct full reuse is the query-only reference for an already materialized target-native history cache. Tail is within 0.41% on ARC and 0.23% on OpenBookQA of this reference. We use this comparison to calibrate the partial-recompute executor; the request-level contribution is the cost gate that determines whether the selected reuse path should run or return to Native.



(a) Thinking-target overhead. **(b)** Cold-path baseline ladder.

Figure 5: Transfer and baseline boundaries. (a) Thinking-target total-time overhead; (b) cold-path slowdown against matched references. Tail/Native uses the formal target-native set; the other bars use the separate runtime. CB/DS: CacheBlend/DroidSpeak mechanism implementations; Reuse: minimal full reuse.

Table 8: Thinking-target screening: Tail and CacheBlend use target-native caches; DroidSpeak uses Instruct→Thinking transfer. Slowdown is the ratio of method and matched Native mean times minus one; no comparison has a timing win.

Dataset	Method	Accuracy (%)		Total time (s)		Slowdown
		Native	Method	Native	Method	(%)
ARC	Tail	25.00	25.00	1.827	1.917	4.92
	CacheBlend	25.00	25.00	1.807	1.924	6.46
	DroidSpeak	25.00	18.75	1.843	1.891	2.61
OpenBookQA	Tail	12.50	6.25	1.821	1.855	1.87
	CacheBlend	12.50	6.25	1.803	1.872	3.83
	DroidSpeak	12.50	12.50	1.840	1.887	2.55

B THEORETICAL ANALYSIS

We derive three properties of the controller in Eqs. (1)–(4). All costs refer to the same request and execution environment; request arguments are suppressed for clarity.

B.1 RESIDENCY AND THE PROFITABILITY BOUNDARY

Let $B = T_{\text{build}}(C_\theta(H)) \geq 0$ and $G = T_N - T_R$ be the construction cost and resident savings, with $T_N > 0$. For a desired margin $0 \leq \varepsilon < 1$, Eq. (1) gives

$$\begin{aligned} T_N - T_C &= G - B, \\ T_C \leq (1-\varepsilon)T_N \quad &\Longleftrightarrow \quad B \leq G - \varepsilon T_N. \end{aligned} \tag{B.1}$$

Derivation. Substitute $T_C = B + T_R$ and rearrange. Thus, cold reuse strictly reduces latency exactly when $B < G$, while a resident request meets the desired margin when $G \geq \varepsilon T_N$. Residency removes B from the current request's cost, explaining why the same executor can be profitable in the resident regime and favor Native when construction is charged. The history-length check restricts the profiled operating region; the cost gate tests the request's savings within that region.

B.2 Admission margin under cost uncertainty

For an admitted request whose candidate passes validation, suppose the profiled costs satisfy the one-sided error bounds

$$T_R \le \widehat{T}_R + \eta_R, \qquad T_N \ge \widehat{T}_N - \eta_N,$$

where $\eta_R, \eta_N \ge 0$. Let $D \ge 0$ bound the admission and validation overhead beyond T_R, so that the complete successful path costs $T_P \le T_R + D$.

Savings bound. Any request admitted by Eq. (3) satisfies

$$T_N - T_P \ge \varepsilon \widehat{T}_N - (\eta_N + \eta_R + D). \tag{B.2}$$

Proof. Combining the two error bounds with the admission inequality yields

$$T_N - T_P \ge \widehat{T}_N - \eta_N - \widehat{T}_R - \eta_R - D \ge \varepsilon \widehat{T}_N - \eta_N - \eta_R - D.$$

Consequently, $\eta_N + \eta_R + D < \varepsilon \widehat{T}_N$ is sufficient for a strict latency reduction on the successful path. The margin therefore provides a budget for cost uncertainty and controller overhead. With $\varepsilon = 0.10$, that budget is $0.10\widehat{T}_N$; applying the bound requires error and overhead bounds for the operating environment.

B.3 Fail-closed execution and cache preservation

Execution invariant. Assume the registry metadata faithfully identifies the supplied state, the cache view is read-only, and Native execution completes. Algorithm 1 returns either a tail candidate satisfying $g(C, x) = 1$, $a(x, C) = 1$, and $v(\widetilde{Y}_R) = 1$, or the result of Native execution on the original request. Candidate execution leaves the input cache unchanged.

Proof. Rejection at either guard returns Native before the tail executor is called. After admission, an execution error is represented by $\perp$ and fails validation, as does any candidate violating the finite-value or shape checks. Both cases return Native. The remaining branch returns a validated candidate. Since the executor accesses the input cache through a read-only view, none of these branches modifies its contents. These are execution and state-preservation properties; task correctness is evaluated by the benchmark protocol in Section 5.1.

For latency accounting, let D bound controller overhead on the chosen branch and let F denote time spent in a discarded tail attempt. A guard rejection costs at most $T_N + D$, while fallback after an attempted execution costs at most $T_N + F + D$, using the actual Native cost of that branch. These expressions identify the costs to include when profiling the complete fallback path.

C Deployment and future directions

PRISM identifies a request-level profitability boundary and turns that boundary into an auditable admission rule. The current evidence also points to a direct systems agenda for extending the controller from controlled measurements to multi-request serving.

Serving integration. The next implementation step is to connect the registry and cost profile to a production cache manager. Versioned lineage and ownership metadata would cover invalidation, model updates, and tenant isolation. A scheduler can use the same contract to account for queueing, batching, HBM pressure, and asynchronous cache movement before admitting a request. This integration would enable throughput and P95/P99 studies under measured request arrivals, while preserving the warm/cold decomposition in Eq. (1).

Model and workload coverage. Evaluating additional architectures, retrieval-heavy prompts, code reasoning, and longer independent histories would characterize how residency thresholds vary across models and task distributions. The existing lineage and geometry contract provides the interface for these evaluations. A deployment controller could refresh the profiled admission parameters from measured latency and residency while retaining fail-closed fallback.