Efficient Serving for Dynamic Agent Workflows with Prediction-based KV-Cache Management

Haoyu Zheng[1*] **Jiawei Jiang**[1,3*] **Fangcheng Fu**[4] **Binhang Yuan**[5]
Yongqiang Zhang[6] **Hao Wang**[1] **Jia Wu**[7] **Yuanyuan Zhu**[1†] **Xiao Yan**[2†]

[1] School of Computer Science, Wuhan University
[2] Institute for Math and AI, Wuhan, Wuhan University
[3] School of Computer Science, Central China Normal University
[4] School of Artificial Intelligence, Shanghai Jiao Tong University
[5] Department of Computer Science and Engineering, HKUST
[6] Dameng Database [7] School of Computing, Macquarie University

Abstract

LLM-based workflows compose specialized agents to execute complex tasks, and these agents usually share substantial context, allowing KV-Cache reuse to save computation. Existing approaches either manage KV-Cache at agent level and fail to exploit the reuse opportunities within workflows, or manage cache at the workflow level but assume that each workflow calls a static sequence of agents. However, practical workflows are typically dynamic, where the sequence of invoked agents and thus induced cache reuse opportunities depend on the context of each task. To serve such dynamic workflows efficiently, we build a system dubbed PBKV (**P**rediction-**B**ased **KV**-Cache Management). For each workflow, PBKV predicts the agent invocations in several future steps by fusing the guidance from historical workflows and context of the target workflow. Based on the predictions, PBKV estimates the reuse potential of cache entries and keeps the high-potential entries in GPU memory. To be robust to prediction errors, PBKV utilizes the predictions conservatively during both cache eviction and prefetching. Experiments on three workflow benchmarks show that PBKV achieves up to $1.85\times$ speedup over LRU on dynamic workflows, and up to $1.26\times$ speedup over the SOTA baseline KVFlow on the static workflow. Code at: `https://github.com/Hyzheng-code/PBKV`.

1 Introduction

Large Language Model (LLM)-based multi-agent systems have become a popular paradigm for complex reasoning and automation. Frameworks such as LangChain [1], AutoGen [2], and MetaGPT [3] decompose a complex task into a group of collaborating *agents* (e.g., a Planner, Coder, and Tester in code-generation pipelines), where each agent is invoked as a single LLM request. Such multi-agent collaboration on a single task is known as an *agentic workflow*.

Modern inference engines (e.g., vLLM [4] and SGLang [5]) share Key/Value tensors (i.e., KV-Cache) across requests, turning expensive prefill into cheap cache lookups. In an agentic workflow, agents naturally share substantial context, including the system prompt, tool/agent descriptions, and the history accumulated by upstream agents [6, 1, 7]. We term this reuse pattern *workflow-level cache reuse*. Ideally, the KV-Cache produced by one agent should be retained for the downstream agents,

*Equal contribution.
†Corresponding authors: Yuanyuan Zhu (`yyzhu@whu.edu.cn`) and Xiao Yan (`yanxiaosunny@whu.edu.cn`).

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

where the cache hit rate can exceed 90% [6]. However, in practice, this potential is bounded by *limited* GPU memory, making effective KV-Cache management crucial in multi-agent serving.

KV-Cache management is fundamentally a problem of *predicting* future access patterns. Existing inference engines commonly adopt the Least Recently Used (LRU) policy, which rests on a simple heuristic that the longest-idle cache is unlikely to be reused. However, in multi-agent settings, cache reuse is determined by the *workflow's structure* rather than temporal locality. A cache entry may remain idle across multiple agent invocations (e.g., during tool calls or transitions between agents) before being reused, and LRU may evict it prematurely during this idle period. As a result, LRU's temporal-locality assumption is misaligned with the structural nature of workflow-level cache reuse.

An idealized alternative is the classic Belady's algorithm [8], which achieves *offline*-optimal cache eviction by evicting the cache whose next access is farthest in the future. The recent work KVFlow [9] adapts this idea to multi-agent settings by assuming a *predefined* and *static* agent step graph (i.e., a global DAG that specifies the invocation orders of all agents), and evicting the KV-Cache of the agent whose next invocation is farthest away (i.e., with the largest 'steps-to-execution'). This design rests on a strong assumption: the future invocation order of agents is *known a priori*. This assumption fails on realistic dynamic workloads, where workflows are *runtime-dependent*. For example, a Tester may send the code back for rewriting; and a Retriever may spawn new sub-queries based on intermediate results. The realistic workflow structure cannot be determined in advance, thus effective KV-Cache management must **shift from assuming the future to predicting it**.

Challenge. However, realizing this shift faces two conflicting challenges: *(i)* agentic workloads are inherently hard to predict [10–14], because the stochastic decoding of LLMs injects uncertainty at each step, which propagates along the workflow path and yields high variability in the resulting agent invocation trace. Industrial studies from Microsoft [15] and IBM [16, 17] report that such uncertainty is common and consequential in agentic systems. *(ii)* Prediction errors are disproportionately costly. The cost of mistakes is amplified by the workflow: a cache miss in multi-agent serving may force re-prefill of tens of thousands of tokens *accumulated* along the entire workflow context.

Taken together, the structure of agentic workloads exposes substantial reuse potential, yet inevitably entails prediction errors and the steep cost of erroneous evictions. This motivates the central question of our work: *in a dynamic multi-agent system, how can we design a KV-Cache management framework that (i) is robust to predictor quality and (ii) benefits continuously even from imperfect predictors?*

Our Solution. Our key insights: (i) *multi-step* prediction with *complementary* signals can effectively replace the static assumption; and (ii) *conservative* designs can yield *stable* and *robust* gains.

Specifically, **for the predictor**, we propose two design principles: (i) *Complementary signal fusion*, which combines *cross-workflow* agent transition patterns with *per-request* semantics; (ii) *Multi-step horizon*, which is motivated by the fact that cache reuse distances in agentic workflows can span several invocations (e.g., across a retry loop), so a single-step predictor cannot distinguish cache that will be reused later from useless cache. Multi-step prediction can prevent this myopia *without* assuming a static workflow. **For the cache manager**, we likewise propose two designs: (i) *Hierarchical eviction.* Guided by the observation that the private cache of a terminated workflow has negligible reuse potential regardless of any prediction, we reclaim such *retired* cache first and then rank the remaining active cache by a lookahead reuse score from the predictor, yielding *stable* gains in the common case and *graceful* degradation under poor predictions. (ii) *Conservative prefetching*, which carefully *trades off* the cost and benefit of prefetching and consumes *only* otherwise-idle GPU space and PCIe bandwidth, so that it does not backfire even under poor predictions.

Together, these designs form a **P**rediction-**B**ased **KV**-Cache management framework, which we call PBKV. We evaluate PBKV on realistic dynamic workflows, where it reduces the average workflow latency by up to $1.85\times$ and improves the KV-Cache hit rate by up to $2.55\times$ over LRU. On static workflows, PBKV outperforms KVFlow on these metrics by up to $1.26\times$ and $1.39\times$, respectively. Furthermore, PBKV's predictor is *pluggable*; to ensure *robustness* across predictors, we prove that the performance is *Lipschitz-continuous* in prediction error, establishing *graceful degradation*.

Contributions. In summary, we make the following contributions:

- We formalize KV-Cache management in multi-agent serving as a problem of *predicting* future access patterns, and provide design guidelines for predictors in this setting.

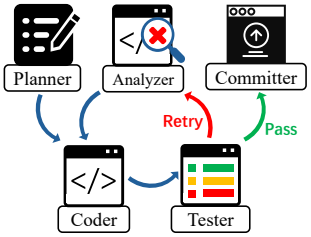


Figure 1: A call graph for the code-generation task. The Tester conditionally triggers a retry path through Analyzer and Coder, i.e., a retry loop.

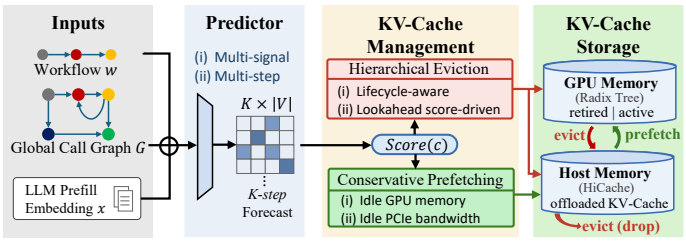


Figure 2: **System overview of PBKV.** For each active workflow w, the predictor produces a K-step forecast over upcoming agent invocations. The forecast drives a shared scoring function $Score(c)$, which feeds both a hierarchical eviction policy and a conservative prefetching policy on the two-tier KV-Cache storage.

- We propose PBKV, guided by two design principles, i.e., *hierarchical eviction* and *conservative prefetching*, that together identify cache with low reuse value for reclamation while preserving high-value cache. We further provide a *Lipschitz guarantee* that PBKV's performance degrades *gracefully* even under prediction error.
- We implement and evaluate PBKV across diverse workloads and LLMs. PBKV consistently outperforms LRU and the SOTA baseline in both cache hit rate and end-to-end performance.

2 Preliminaries

LLM-based Multi-Agent Serving. A multi-agent application can be abstracted as a directed graph $G = (V, E)$, where each node in V corresponds to an agent and each edge in E denotes an admissible transition between agents. Since G enumerates *every possible* transition pattern, we refer to it as the **global call graph**. Figure 1 shows a representative example for a code-generation task. Notably, G can contain loops, capturing retry loops common in realistic agentic workflows. A **workflow** is a single execution instance on G, consisting of an ordered sequence of agent invocations $a_1, a_2, \ldots$, where each $a_i \in V$ and each consecutive pair $(a_i, a_{i+1}) \in E$. Each **agent invocation** is an LLM request with a specific prompt, and thus benefits from KV-Cache reuse to reduce latency.

Agents typically share a large amount of KV-Cache, which we classify into two categories: (i) *global cache*, shared across agent instances from *different* workflows (e.g., the system prompt, tool/agent descriptions, and knowledge documents); and (ii) *private cache*, produced by upstream agents of a particular workflow and reusable only by downstream agents of the *same* workflow. Cross-workflow reuse of private cache is generically infeasible, because even under identical user prompts, the stochastic decoding of LLMs makes the workflows diverge within a few tokens.

Radix Tree. PBKV is built on *SGLang* [5], which organizes prefix-shared cache as a *Radix Tree*, with each node holding a contiguous token segment reusable by all requests sharing that prefix. *HiCache* [18] extends the cache into a two-tier hierarchy where cache evicted from GPU is retained in host memory and swapped back on a hit, avoiding full re-prefill at the cost of a PCIe transfer.

3 System Overview

The underlying design principle of PBKV is: predicting upcoming agent invocations to evaluate the *reuse value* of existing KV-Cache, which then drives both KV-Cache eviction and prefetching. As shown in Figure 2, PBKV consists of three components: (i) a *predictor* that produces a multi-step forecast for each active workflow, (ii) a set of *KV-Cache management policies* that translate the forecasts into eviction and prefetching decisions, and (iii) a *two-tier KV-Cache storage* organized as a Radix Tree on GPU memory and HiCache on host memory, on which the policies operate.

The predictor serves as the foundation of PBKV. **First**, it fuses two complementary signals: (i) *graph-level* agent transition patterns *shared* across requests, encoded in the global call graph G; and (ii) *workflow-level specifics* of the current request, reused from the LLM prefill embedding x. **Second**, it emits K probability distributions over upcoming agent invocations, rather than only the

next one, preventing *myopic eviction*. For instance, in Figure 1, when the current agent is Analyzer, a single-step predictor would reveal only the next Coder invocation and miss the Tester re-invocation two steps later, risking a wrongful eviction of the Tester's cache and a costly re-prefill.

The KV-Cache management policies consist of an eviction policy and a prefetching policy. They share a common reuse scoring mechanism (i.e., multi-step lookahead and cross-workflow aggregation) and a common design philosophy (i.e., embedding deterministic guardrails within a probabilistic system). Specifically, *(i) hierarchical eviction* reclaims retired cache from terminated workflows first as it carries no reuse potential, and only after it is exhausted does score-driven eviction take over the active cache. Thus, performance degrades gracefully when the predictor is unreliable, while the upside is preserved when it is accurate. *(ii) Conservative prefetching* is motivated by the asymmetry that a prefetch always pays its cost while the benefit materializes only when the prediction is correct. PBKV therefore restricts prefetching to otherwise-idle GPU space and PCIe bandwidth, so that even under poor predictions, prefetching neither displaces valuable cache nor competes for on-path bandwidth.

In the following section, we elaborate on the design of each component and their coordination.

4 Design of PBKV

In this part, we design a predictor as the foundation of subsequent KV-Cache Management.

4.1 Workflow Prediction

Predictor Design. Building on the principles above (i.e., multi-signal fusion and multi-step horizon), we instantiate the predictor with two desiderata: (i) leveraging structural priors in the global call graph G, and (ii) generalizing to unseen workflow prefixes at runtime to accommodate the dynamic nature of agentic workflows. We adopt **GraphSAGE** [19] as the backbone, which satisfies both: (i) it operates directly on the *graph* through neighborhood sampling and aggregation, preserving the structural priors, and (ii) it is *inductive*, performing forward inference on unseen prefixes.

As shown in Figure 3, the predictor fuses three complementary streams. *(i) Topology-aware agent embedding* $\mathbf{h}_{\text{cur}}$: each agent's learnable embedding is refined by two GraphSAGE layers over a transition matrix estimated from offline traces, so that the resulting $\mathbf{H}^{(2)}$ encodes both the agent's identity and its neighborhood in G. *(ii) Attention-based history aggregation* $\mathbf{h}_{\text{path}}$: since two workflows at the same current agent may have arrived through very different prefixes, we summarize prior agent representations via scaled dot-product attention with $\mathbf{h}_{\text{cur}}$ as the query, up-weighting the most informative prefix agents over naive mean pooling. *(iii) Semantic signal from prefill* $\mathbf{h}_{\text{txt}}$: to distinguish requests with the same prefix but diverging intent, we project the post-norm hidden state of the last prefill token, obtaining the signal essentially for free. The three streams are concatenated and passed through a two-layer MLP that jointly emits logits over agents for each of the next K steps in a single forward pass, avoiding autoregressive error accumulation while keeping per-invocation latency to that of a single inference. Full architecture is detailed in Appendix D.

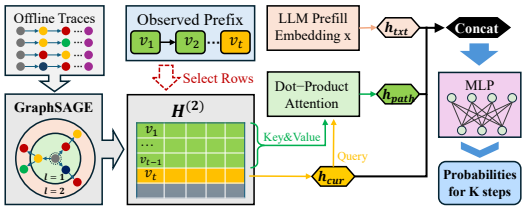


Figure 3: **Architecture of the predictor.** It fuses a topology-aware agent embedding from GraphSAGE (h_{cur}), an attention-based workflow prefix summary (h_{path}), and a semantic signal reused from prefill (h_{txt}), then jointly predicts the next K agent probability distributions via an MLP.

Training and Performance. The predictor is trained on offline invocation traces with cross-entropy loss, pairing each prefix and its prefill embedding with the next K agents as labels (padding positions after $\langle\text{END}\rangle$ are masked). The resulting predictor has roughly 350K parameters and processes a batch of 1,024 requests in 1.56 ms, rendering its runtime overhead negligible. On the HoVer [20] dataset with the LangChain [1] framework, training on $1K$ traces achieves an accuracy of 0.94 at 1-step and 0.77 at 3-step (500-trace test set). Scaling curves and comparisons against alternative predictors are provided in Appendix E, with full hyperparameters in the supplementary code.

4.2 Lookahead KV-Cache Eviction

Building on the multi-step predictor, we design a *lookahead* KV-Cache eviction policy.

4.2.1 Base: Lifecycle-Aware KV-Cache Eviction

Our observation and design. Once a workflow terminates, its residual private cache (which we call *retired cache*) has negligible reuse potential and should be reclaimed first. Under LRU, such cache can only age out passively, and LRU may evict still-valuable cache of other workflows during this aging window. Additionally, retired cache is not all equivalent, we further rank them by the number of workflows that have accessed them. Thus, the popular shared prefix can be preserved longer.

Implementation. We tag each cache node with the workflows that have accessed it. The server continuously listens for and records workflow termination messages from clients. Any cache node whose associated workflows have *all* terminated is tagged as retired and prioritized for eviction. Despite its simplicity, this change alone improves the average hit rate by up to $1.66\times$ in our experiments.

4.2.2 Lookahead Score-Driven KV-Cache Eviction

Retired cache is not always abundant. Under high load, once the retired cache is drained, lifecycle-aware eviction degenerates back to LRU. We notice that the reuse likelihood also *varies* across the active cache. For example, (i) among the *global cache*, descriptions of popular agents are reused more frequently than those of rarely used ones; and (ii) among the *private cache*, retention priority should scale with the predicted probability that the owning agent is invoked in the future. We therefore extend eviction from a binary lifecycle label to a continuous score that reflects predicted reuse value.

Cross-Workflow Value Aggregation. KV-Cache is shared across workflows, thus its reuse value should be aggregated *globally*. Therefore, the score of a cache node should grow with *(i)* the number of workflows likely to reuse it, and *(ii)* each workflow's reuse probability. We further annotate each cache node c with a per-workflow access indicator vector $A_w(c) \in \{0,1\}^{|V|}$. For example, if c has been accessed by Agents 1 and 3 from workflow w, then $A_w(c) = [1, 0, 1, \dots]$. For each workflow w, the predictor emits a next-step distribution, which we decompose into an agent-access probability vector P_w and a termination probability $p_{w,\langle\text{END}\rangle}$. The single-step reuse value of c is then defined as

$$\text{Value}(c) \;=\; \sum_{w \,\in\, \mathcal{W}^{\text{act}}(c)} A_w(c) \cdot P_w \tag{1}$$

where $\mathcal{W}^{\text{act}}(c)$ denotes the set of active workflows associated with node c. Intuitively, $\text{Value}(c)$ *aggregates* the probability that each workflow's next invocation will touch node c. Notably, this cross-workflow aggregation *naturally* protects the global cache and popular-prefix cache.

From One-Step to K-Step Lookahead. As described above, a single-step view remains myopic, so we extend the score to a K-step horizon by leveraging the predictor's multi-step outputs. Two practical effects must be accounted for: (i) predictions further into the future are objectively less reliable, regardless of the predictor's stated confidence; and (ii) steps following a predicted termination are meaningless and would pollute the score if left in. To address them, we introduce a confidence decay factor $\gamma < 1$ and a *cumulative survival probability* $s_w^{(k)} = \prod_{j=1}^{k-1}\big(1 - p_{w,\langle\text{END}\rangle}^{(j)}\big)$, i.e., the probability that workflow w remains active at step k. Figure 4 illustrates the computation, which we formalize as:

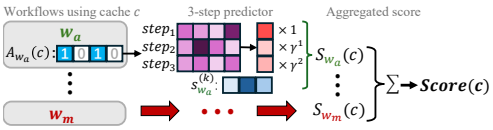


Figure 4: **Computing the cross-workflow reuse score.** For each active workflow w accessing cache node c, the K-step (K=3 here) predictor outputs per-step access probabilities, which are weighted by the survival probability $s^{(k)}$ and confidence factor γ^{k-1} and summed across m workflows as $Score(c)$.

$$\text{Score}(c) \;=\; \sum_{k=1}^{K} \gamma^{k-1} \sum_{w \,\in\, \mathcal{W}^{\text{act}}(c)} s_w^{(k)} \cdot A_w(c) \cdot P_w^{(k)} \tag{2}$$

Implementation. On every workflow state change (i.e., new agent invocation or termination), the predictor refreshes its forecast and updates the scores of the affected cache nodes. On eviction, nodes are sorted in ascending order of $\mathrm{Score}(c)$ and evicted until the requested space is freed.

4.2.3 Hierarchical Eviction: Unifying the Two Policies

The two policies can be naturally combined because retired cache nodes receive a score of 0 under Equation 2 (as they satisfy $\mathcal{W}^{\text{act}}(c) = \emptyset$). That said, the "no value" judgment of retired cache is *deterministic*, whereas the "zero score" of an active node is merely a *probabilistic* estimate bounded by both the horizon K and the predictor's accuracy. We therefore adopt a *hierarchical eviction* strategy, in which active cache is spared until all retired cache is drained. The underlying design philosophy is to *embed deterministic guardrails within a probabilistic system*, so that performance degrades gracefully under unreliable predictions while preserving the upside under good predictions.

4.3 Conservative KV-Cache Prefetching

Building on the predictor, a complementary optimization is to proactively load likely-to-be-reused cache nodes into GPU memory before they are hit, leveraging SGLang's HiCache infrastructure to hide transfer latency behind ongoing decode steps rather than exposing it on the request critical path.

Prefetching Principle. At its core, prefetching trades *known-valuable* GPU cache for *speculatively valuable* host cache, with the *fixed costs* in scheduling and PCIe bandwidth. Given the inevitable prediction errors in dynamic workflows, the asymmetry between deterministic cost and probabilistic benefit motivates a *conservative* principle that prefers risk avoidance over aggressive speculation.

Prefetching Design. Concretely, we do *not* evict any active cache to make room for prefetched data, for three considerations: (i) as argued above, evicting known-valuable cache is inherently risky under dynamic workflows; (ii) under high concurrency, such evictions can disrupt the radix tree's prefix structure; and (iii) if the evicted active cache is hit shortly after, the system pays an additional GPU-reload cost or, in the worst case, a full re-prefill. *Instead*, we restrict the prefetch region to the union of currently *free* space and *retired* cache (§4.2.1), whose total size we denote as S_a. This design guarantees that prefetching perturbs existing GPU-resident cache only minimally, if at all.

Beyond GPU space, prefetching also consumes PCIe bandwidth and therefore *competes* with prefill requests. To avoid interference, PBKV activates prefetching only on pure-decode batches (>90% of all batches in our measurements). Within such a batch, we bound the transferable volume as $S_{bw} = Bandwidth \cdot StepDuration$, where the PCIe bandwidth and decode step duration are dynamically updated by their runtime averages. The final budget is $S = \min\{S_a,\ S_{bw}\}$, which typically hides prefetching cost behind a single decode step and uses only otherwise-idle bandwidth.

To rank candidate nodes in the host memory, we reuse the one-step reuse value $Value(c)$ defined in Eq. 1. We deliberately restrict ranking to the *one-step* horizon because prefetching is naturally incremental: a node that will be reused two steps ahead can simply be fetched at the next step. In contrast, eviction must look ahead across K steps, since a mistaken eviction incurs a costly reload or even a full re-prefill. Selecting the optimal subset under budget S is formally a knapsack problem and hence *NP-hard*. Since prefetch decisions lie on the scheduler's critical path, we use a lightweight greedy heuristic that loads candidate nodes in descending order of value until the budget is exhausted.

An aggressive variant that permits evicting active cache for prefetch space is evaluated in Appendix G.

5 Discussion

Positioning. While our predictor builds on standard GraphSAGE, attention, and MLP components, our primary contribution lies in robust KV-Cache management despite inevitable prediction errors. We also provide design guidelines for predictors in this setting (Section 7 explains why existing predictors fall short) and instantiate one accordingly. PBKV deliberately keeps the predictor module *pluggable*, so that it can benefit from future predictors. We further prove in Appendix I that PBKV degrades *gracefully* even under poor predictions (i.e., the degradation is *Lipschitz-continuous*), that is:

Theorem 5.1. *Let $\widehat{E}_B$ and $E^\star_B$ be PBKV's eviction set and the cost-minimizing set under the ground truth, respectively, and ϵ^γ_n the per-node prediction error. The eviction cost regret of PBKV satisfies*

$$0 \;\le\; \mathcal{R}(B) \;:=\; \mathcal{L}\big(\widehat{E}_B\big) - \mathcal{L}\big(E_B^\star\big) \;\le\; \frac{1}{2(1-\gamma)} \sum_{c \,\in\, \widehat{E}_B \triangle E_B^\star} \epsilon_c^\gamma,$$

where $\widehat{E}_B \triangle E_B^\star$ denotes the symmetric difference between PBKV's eviction set and the optimum. The bound $\mathcal{R}(B) \to 0$ when the prediction is perfect and grows linearly in the prediction error.

Complexity. PBKV maintains the score of candidate nodes in a *heap* structure. Inspecting the top candidate takes $O(1)$, while popping, insertion, and score updates take $O(\log n)$ for n cache nodes, keeping the scheduler overhead negligible even under high concurrency (measured in Appendix H).

Limitations. (i) The predictor requires adaptation when the workload or agent set changes, while PBKV can fall back to lifecycle-aware eviction during this process. To assess robustness to workload shifts, we conduct out-of-distribution (OOD) evaluations in Appendix C. (ii) Cache-reuse patterns across agents are not universal but rather depend on the *client's specific* message-passing convention. The indicator $A_w(c)$ in Eq. 1 captures a *basic form*, namely the set of agents that have directly accessed c in workflow w. It admits straightforward *customization*, e.g., when a framework specifies that agent j inherits cache accessed by agent i, $A_w(c)[j]$ can be set to 1 whenever $A_w(c)[i] = 1$.

6 Experimental Evaluation

In this section, we evaluate PBKV on realistic multi-agent workloads and ablate each component.

6.1 Experimental Settings

Testbed and Models. Our experiments are conducted on a server with $8\times$ NVIDIA A6000 (48 GB) GPUs interconnected via NVLink, 128 virtual CPU cores, 512 GB of memory, and 20 GB/s PCIe bandwidth. We use Qwen3-14B and Qwen3-32B (two-way tensor parallelism) as the base LLMs.

Baselines. We compare PBKV against **(i) LRU** on SGLang+HiCache (`HICACHE_RATIO=1`, allocating equal capacity on GPU and host memory), the default policy of SGLang. **(ii) KVFlow** [9], the SOTA workflow-aware policy, whose eviction is *driven* by the 'steps-to-execution' distance on a static DAG. Since this distance is *undefined* under runtime-dependent *loops*, we construct a static workflow scenario for fair comparison. To isolate the contribution of each component, we further evaluate two ablation variants of PBKV: **(i) PBKV-LAE**, which enables only Lifecycle-Aware Eviction; and **(ii) PBKV-HE**, which adds Hierarchical Eviction on top of PBKV-LAE but disables prefetching.

Workloads. We evaluate PBKV on three representative multi-agent workloads, each pairing a public benchmark with a widely-adopted agent framework to reflect *real-world* deployment: *(i) fact verification*, using the HoVer [20] dataset with the LangChain [1] agent framework; *(ii) code generation*, using SWE-bench [21] with Microsoft AutoGen [2]; and *(iii) document analysis*, using FinanceBench [22] with CrewAI [23], which constitutes a *static* workflow for direct comparison against KVFlow. Detailed workload characterization is provided in Appendix A. Following prior work [9, 24], we set the number of concurrent workflows to induce memory pressure, thereby effectively evaluating KV-Cache management policies. For each workload, we train the predictor on offline traces from tasks disjoint from those used in serving evaluation.

Metrics. We evaluate each policy along three *complementary* dimensions: *(i) End-to-end workflow latency*, measured from workflow submission to final completion; *(ii) Per-agent TTFT*, defined as the mean time-to-first-token across every agent invocation along a workflow; *(iii) Average KV-Cache hit rate* (token-level) on GPU memory, which *directly* reflects the quality of cache management.

6.2 Main Results.

Observations. We take HoVer + LangChain workload as a representative case. Under K=3 and γ=0.7, Table 1 shows that PBKV reduces end-to-end latency and per-agent TTFT by up to $1.85\times$ and $2.03\times$ over LRU, respectively. This directly confirms that PBKV delivers substantial gains in both system efficiency and quality of service (i.e., QoS). The KV-Cache hit rate exposes the source of these gains: **(i) LRU** achieves a hit rate below 30% on both models, confirming that it fails to capture the workflow-level reuse. **(ii) PBKV-LAE**, which merely identifies and evicts *retired cache* first, already

Table 1: Performance of policies across workloads and LLMs. The metrics are described above. As different workloads have different GPU memory footprints, we set the concurrency limit to 72/24/48, respectively, and study its sensitivity in Section 6.3. The *static* workload is designed for comparison with KVFlow, which assumes a predefined static call graph. (Full results with std see Appendix B)

Workload	Policy	Qwen3-32B			Qwen3-14B		
		Lat. (s)↓	PA. TTFT (s)↓	Hit Rate (%)↑	Lat. (s)↓	PA. TTFT (s)↓	Hit Rate (%)↑
HoVer + LangChain (with iterative refinement *loops*)	LRU	189.66	16.65	27.09	139.59	10.90	28.79
	PBKV-LAE	146.67	11.83	44.91	119.13	9.57	41.18
	PBKV-HE	108.86	8.95	66.01	80.62	6.53	64.15
	Full PBKV	102.60	8.22	69.10	76.15	5.50	68.29
SWE-bench + AutoGen (with retry *loops*)	LRU	271.90	5.04	46.34	214.74	4.18	48.53
	PBKV-LAE	233.86	3.73	59.51	177.50	3.26	60.41
	PBKV-HE	177.22	2.46	75.31	141.53	2.19	73.36
	Full PBKV	160.18	2.27	79.94	118.61	2.05	77.07
FinanceBench + CrewAI (*static*)	LRU	130.34	12.93	27.95	96.11	9.02	26.58
	KVFlow	101.57	9.91	39.87	80.13	7.42	39.65
	Full PBKV	80.53	8.00	53.44	65.01	6.06	55.07

lifts the hit rate by up to $1.66\times$ over LRU, showing that even simple *lifecycle-awareness* is highly effective. **(iii) PBKV-HE** additionally incorporates the workflow predictor and reuse scoring, pushing the hit rate to 66.01%, because it assesses the reuse value of active cache through *cross-workflow aggregation* and *multi-step lookahead*. As a result, eviction decisions remain informed even after the retired cache is exhausted, and the policy never degenerates back to LRU. **(iv) Full PBKV** introduces *speculative prefetching* on top of PBKV-HE and attains a hit rate of 69.10% ($2.55\times$ over LRU).

We further note that the hit rates are higher near the start and the end of each run, when the pressure is low. Restricted to the steady-state interior, LRU's hit rate drops to $\sim$10% while full PBKV's sustains $\sim$50%. Their behavior is detailed in Section 6.4. Results of other tests exhibit similar trends.

Ablation Study. Comparing the three PBKV variants, the contributions of Lifecycle-Aware Eviction, Hierarchical Eviction, and speculative prefetching are *incremental*. The step from PBKV-LAE to PBKV-HE is pronounced, whereas the step from PBKV-HE to full PBKV is *modest*. This modest gap stems from two factors: (i) the prediction errors of the multi-step predictor on realistic dynamic workflows, and (ii) our deliberately *conservative* prefetching design (Section 4.3), which refuses to gamble known-valuable active cache for speculative gains. Appendix G evaluates an aggressive alternative that evicts active cache to free prefetch space, confirming that our conservative variant delivers stable benefits while avoiding the backfire of aggressive prefetching under poor predictions.

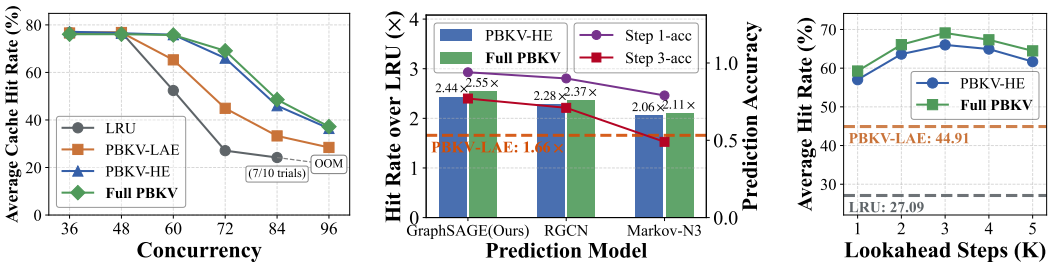
Performance on Static Workflow. Table 1 also reports the results under a static workflow (FinanceBench + CrewAI). KVFlow infers the steps-to-execution from the predefined call graph and prioritizes the cache accordingly, yielding a moderate improvement over LRU. PBKV achieves a *further* gain, which we attribute to two design choices. *(i) Lifecycle-aware eviction.* Like LRU, KVFlow can also mistakenly evict active cache that will still be reused, whereas PBKV's lifecycle-aware eviction prevents this misjudgment. *(ii) Cross-workflow reuse aggregation.* KVFlow prioritizes each cache node by the *minimum* steps-to-execution across all related workflows, which captures *when* it will be reused but not how *popular* it is. PBKV instead *aggregates* the predicted reuse contribution from every active workflow accessing the node, so the score of popular cache grows naturally.

Remark on Static Workflows. Given the deterministic agent invocation order in *static* workflows, PBKV could be further tuned by safely relaxing several mechanisms (e.g., adopting more aggressive prefetching and disabling the multi-step confidence decay γ). However, we retain these mechanisms in our evaluation for compatibility and consistency across both dynamic and static workloads.

6.3 Sensitivity Analysis

In this section, we vary several configurations to characterize their impact on PBKV and its variants. All experiments below are conducted on the HoVer + LangChain workload with Qwen3-32B.

Concurrency. It directly governs GPU memory pressure. As shown in Figure 5a, all hit rates degrade as concurrency rises, yet PBKV consistently and greatly outperforms LRU. Below 60 workflows,



(a) Average cache hit rate vs. concurrency. Excessive concurrency causes LRU to run out of memory (OOM).

(b) Prediction accuracy of different backbones (lines) and the corresponding hit-rate ratio of PBKV over LRU (bars).

(c) Average cache hit rate vs. lookahead horizon K (i.e., prediction step).

Figure 5: Sensitivity analysis of PBKV and its variants on HoVer + LangChain with Qwen3-32B.

evictions are too rare to differentiate any policy. At the other extreme, LRU exhausts memory and crashes in 3/10 trials at concurrency 84 and 10/10 at 96. We therefore adopt a concurrency of 72 for HoVer + LangChain in the main experiments to evaluate all policies under reasonable pressure.

Predictor Backbone. PBKV treats the predictor as a pluggable component. We replace our GraphSAGE backbone with R-GCN and a third-order Markov model (Markov-N3), each trained on the same $1K$ traces. As shown in Figure 5b, the variants exhibit lower prediction accuracy, which translates monotonically into lower cache hit rates. Even so, the weakest backbone still strictly outperforms the prediction-free PBKV-LAE, confirming that PBKV is robust to predictor quality and that a reasonable predictor typically delivers additional benefits on top of lifecycle awareness.

Lookahead Horizon. As shown in Figure 5c, the best hit rate occurs at K=3: a smaller horizon induces myopic eviction, whereas a larger one dilutes the score with inaccurate long-range predictions. The gap between full PBKV and PBKV-HE stays nearly constant across K by design, since prefetching ranks candidates by the one-step value alone (Section 4.3) and is decoupled from K.

6.4 Why PBKV Works: Analysis of Cache Behavior.

To understand the sources of PBKV's gains, we trace the KV-Cache hit rate of each policy throughout a full run in Figure 6. Key events (marked by dashed vertical lines) divide the run into several phases:
(i) Warm-up (0-30 s). With ample GPU memory, all policies rapidly exceed 80% without eviction, *confirming our motivation* in Section 1 that multi-agent workflows expose abundant reuse potential.
(ii) Onset of memory pressure (30-55 s). Around 30 s, GPU memory saturates and forces evictions, causing LRU's hit rate (red) to drop sharply. PBKV-LAE (orange) follows the same trajectory, as no workflow has terminated to release retired cache. In contrast, PBKV-HE (blue) and full PBKV (green) decline only *gradually*, since their scoring identifies low-value *active* cache for eviction.
(iii) Retired-cache window (55-80 s). From 55 s onward, the first-completed workflows release retired cache. PBKV-LAE immediately capitalizes on this signal and recovers visibly. PBKV-HE and PBKV enjoy the same benefit through their hierarchical eviction. Thereafter, LRU stays around a 10% hit rate and only recovers near the end when the draining request pool relieves pressure.
(iv) Retired-cache drained (80-115 s). Once the retired cache is drained, PBKV-LAE reverts to LRU. Thereafter, its hit rate recovers only *briefly* when batches of workflows release retired cache concurrently. PBKV-HE and PBKV also decline but remain well above LRU, thanks to their scoring.
(v) Steady-state serving (115-245 s). PBKV and PBKV-HE greatly outperform others, showing the lookahead score remains effective under prolonged pressure. PBKV further improves PBKV-HE by a *modest but consistent* margin, reflecting the bounded but stable benefit of *conservative* prefetching.
(vi) Tail phase (after 245 s). As the request pool drains, eviction pressure subsides and the hit rate climbs back to roughly 90% by completion (around 285 s). Full PBKV enters and exits this phase *first*, achieving the lowest average job completion time and the highest throughput among all policies.

Summary. This phase-by-phase analysis reveals the *complementary* roles of PBKV's three mechanisms. Lifecycle-aware eviction provides a deterministic improvement whenever retired cache is available; hierarchical eviction sustains improvements under prolonged pressure via multi-step lookahead and cross-workflow aggregation; and speculative prefetching contributes a bounded but

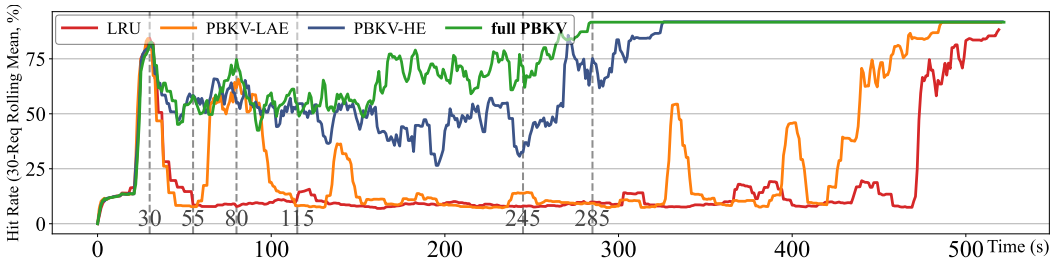


Figure 6: KV-Cache hit rate of each policy over time on the HoVer + LangChain workload, served by Qwen3-14B under a concurrency limit of 72. Dashed vertical lines mark key events.

stable improvement on top. Together, they enable PBKV to maintain a substantial and sustained lead over LRU, directly translating into lower latency (as shown in Table 1) and higher throughput.

7 Related Work

KV-Cache Management for Multi-Agent Serving. Modern inference engines (e.g., vLLM [4] and SGLang [5]) manage KV-Cache via paged attention and radix-tree prefix caching under a default LRU policy. Orthogonal efforts such as HiCache [18] and LMCache [25] extend this with hierarchical GPU-CPU-disk offloading, while KVCOMM [26] and CacheBlend [27] target non-prefix reuse. All of them are oblivious to workflows. RLT [28] adopts randomized eviction, while Wang et al. [29] exploit request-category reuse patterns. Neither captures workflow-level dependencies. Some recent systems try to close this gap. Continuum [30] targets ReAct-style single-agent tool calling, pinning KV entries with a dynamic TTL learned from per-tool duration statistics, whereas KVFlow [9] assumes a predefined Agent Step Graph and uses 'steps-to-execution' scores for eviction and prefetching. Helium [31] models workflows as DAG-based query plans for proactive caching and cache-aware scheduling. However, Continuum focuses on single-agent tool calling, while KVFlow and Helium rely on predefined workflow structures that struggle with runtime-dependent loops. PBKV fills this gap by driving *workflow-level* KV-Cache management with multi-step *prediction* for each workflow.

Agentic Workflow Prediction. A growing body of work predicts the structure of agentic workflows. Parrot [32] introduces semantic variables to recover inter-request dependencies, Ayo [33] compiles LLM applications into primitive-level dataflow graphs for end-to-end optimization, and Autellix [10] elevates programs to first-class scheduling units. Such abstractions assume a *(near-)static* DAG or dataflow graph and struggle to faithfully express the conditional loops common in practical multi-agent workflows. Another thread of work instead reasons about workflow structure online through speculative execution; e.g., PASTE [11] exploits recurring tool-call patterns and predictable data dependencies to speculatively invoke tools, and Speculative Actions [34] generalizes speculation to agent actions. These methods accommodate dynamic workflows but predict only the next step, which can lead to *myopic eviction* when driving cache eviction; and autoregressive rollout also accumulates errors. Therefore, we develop a multi-step predictor customized for KV-Cache management.

8 Conclusion

We present PBKV for realistic dynamic multi-agent workflows. First, PBKV proposes guidelines for a dedicated *predictor*, i.e., (i) fusing complementary signals from the global call graph and per-request prefill embeddings, and (ii) forecasting over a *multi-step* horizon to avoid myopic decisions. Second, PBKV adopts a *conservative* policy, i.e., (i) hierarchical eviction that combines lifecycle-aware reclamation with lookahead, cross-workflow-aggregated scoring, and (ii) conservative prefetching that consumes only otherwise-idle GPU space and PCIe bandwidth. **Theoretically**, we prove that PBKV degrades gracefully under prediction error. **Empirically**, PBKV significantly outperforms LRU across models and workloads, and also outperforms the SOTA method on static workflows.

Acknowledgments and Disclosure of Funding

This work was sponsored by the CCF-DAMENG Research Fund (No. CCF-DAMENG202605), the National Natural Science Foundation of China (No. 62441229 and 62672371), the Innovative Research Group Project of Hubei Province (No. 2024AFA017), and the New Cornerstone Science Foundation through the XPLORER PRIZE. This work was also supported by the WHU-Kingsoft Joint Lab.

Competing Interests. The authors declare no competing financial interests relevant to this work.

References

[1] Harrison Chase. LangChain. `https://github.com/langchain-ai/langchain`, 2022. Software.

[2] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*, 2024. URL `https://openreview.net/forum?id=uAjxFFing2`.

[3] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=VtmBAGCN7o`.

[4] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph 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*, pages 611–626, 2023.

[5] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. SGLang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37: 62557–62583, 2024.

[6] Yongtong Wu, Shaoyuan Chen, Yinmin Zhong, Rilin Huang, Yixuan Tan, Wentao Zhang, Liyue Zhang, Shangyan Zhou, Yuxuan Liu, Shunfeng Zhou, Mingxing Zhang, Xin Jin, and Panpan Huang. DualPath: Breaking the storage bandwidth bottleneck in agentic LLM inference, 2026. URL `https://arxiv.org/abs/2602.21548`.

[7] LangChain AI. LangGraph. `https://github.com/langchain-ai/langgraph`, 2024. Software.

[8] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[9] Zaifeng Pan, Ajjkumar Patel, Yipeng Shen, Zhengding Hu, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. KVFlow: Efficient prefix caching for accelerating LLM-based multi-agent workflows. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL `https://openreview.net/forum?id=5Iw1nDtYmT`.

[10] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E Gonzalez, et al. Autellix: An efficient serving engine for LLM agents as general programs. *arXiv preprint arXiv:2502.13965*, 2025.

[11] Yifan Sui, Han Zhao, Rui Ma, Zhiyuan He, Hao Wang, Jianxun Li, and Yuqing Yang. Act while thinking: Accelerating LLM agents via pattern-aware speculative tool execution. *arXiv preprint arXiv:2603.18897*, 2026.

[12] Marcel Wagenländer, Otto White, Britannio Jarrett, Pedro Silvestre, Yanda Tao, Guo Li, Huanzhou Zhu, Llúis Vilanova, and Peter Pietzuch. Scepsy: Serving agentic workflows using aggregate LLM pipelines, 2026. URL https://arxiv.org/abs/2604.15186.

[13] Jiaxin Zhang, Prafulla Kumar Choubey, Kung-Hsiang Huang, Caiming Xiong, and Chien-Sheng Wu. Agentic uncertainty quantification. *arXiv preprint arXiv:2601.15703*, 2026.

[14] Jiaxin Zhang, Caiming Xiong, and Chien-Sheng Wu. Agentic confidence calibration. *arXiv preprint arXiv:2601.15778*, 2026.

[15] Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. AgentRx: Diagnosing ai agent failures from execution trajectories. *arXiv preprint arXiv:2602.02475*, 2026.

[16] Dany Moshkovich, Hadar Mulian, Sergey Zeltyn, Natti Eder, Inna Skarbovsky, and Roy Abitbol. Beyond black-box benchmarking: Observability, analytics, and optimization of agentic systems. *arXiv preprint arXiv:2503.06745*, 2025.

[17] Dany Moshkovich and Sergey Zeltyn. Taming uncertainty via automation: Observing, analyzing, and optimizing agentic AI systems. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 3840–3844. IEEE, 2025.

[18] LMSYS Org and SGLang Team. SGLang HiCache: Fast hierarchical KV caching with your favorite storage backends. https://lmsys.org/blog/2025-09-10-sglang-hicache/, 9 2025.

[19] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[20] Yichen Jiang, Shikha Bordia, Zheng Zhong, Charles Dognin, Maneesh Singh, and Mohit Bansal. HoVer: A dataset for many-hop fact extraction and claim verification. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 3441–3460, 2020.

[21] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

[22] Pranab Islam, Anand Kannappan, Douwe Kiela, Rebecca Qian, Nino Scherrer, and Bertie Vidgen. FinanceBench: A new benchmark for financial question answering. *arXiv preprint arXiv:2311.11944*, 2023.

[23] CrewAI. CrewAI. https://github.com/crewAIInc/crewAI, 2023. Software.

[24] qiaoling chen, Zhisheng Ye, Tian Tang, Peng Sun, Boyu Tian, Guoteng Wang, Shenggui Li, Yonggang Wen, Zhenhua Han, and Tianwei Zhang. CONCUR: High-throughput agentic batch inference of LLM via congestion-based concurrency control. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=U7c6glPdDW.

[25] 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, 2025. URL https://arxiv.org/abs/2510.09665.

[26] Hancheng Ye, Zhengqi Gao, Mingyuan Ma, Qinsi Wang, Yuzhe Fu, Ming-Yu Chung, Yueqian Lin, Zhijian Liu, Jianyi Zhang, Danyang Zhuo, and Yiran Chen. KVCOMM: Online cross-context KV-cache communication for efficient LLM-based multi-agent systems. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=yGOytgjurF.

[27] 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*, pages 94–109, 2025.

[28] Fangzhou Wu, Sandeep Silwal, and Qiuyi (Richard) Zhang. Randomization boosts KV caching, learning balances query load: A joint perspective. In *International Conference on Learning Representations*, volume 2026, pages 85754–85780, 2026.

[29] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. KVCache cache in the wild: Characterizing and optimizing KVCache cache at a large cloud provider. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 465–482, 2025.

[30] Hanchen Li, Qiuyang Mang, Runyuan He, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Alvin Cheung, Joseph E. Gonzalez, and Ion Stoica. Continuum: Efficient and robust multi-turn LLM agent scheduling with KV cache time-to-live. In *ICLR 2026 Workshop on Lifelong Agents: Learning, Aligning, Evolving*, 2026. URL https://openreview.net/forum?id=sVzK0LC9pn.

[31] Noppanat Wadlom, Junyi Shen, and Yao Lu. Efficient LLM serving for agentic workflows: A data systems perspective. *Proc. ACM Manag. Data*, 4(3), May 2026. doi: 10.1145/3802046. URL https://doi.org/10.1145/3802046.

[32] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of LLM-based applications with semantic variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 929–945, 2024.

[33] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. Towards end-to-end optimization of LLM-based applications with Ayo. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025. doi: 10.1145/3676641.3716278. URL https://doi.org/10.1145/3676641.3716278.

[34] Naimeng Ye, Arnav Ahuja, Georgios Liargkovas, Yunan Lu, Kostis Kaffes, and Tianyi Peng. Speculative actions: A lossless framework for faster AI agents. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=P0GOk5wslg.

[35] James Thorne, Andreas Vlachos, Christos Christodoulopoulos, and Arpit Mittal. FEVER: a large-scale dataset for fact extraction and VERification. In *NAACL-HLT*, 2018.

[36] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=SJU4ayYgl.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[38] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.

[39] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European semantic web conference*, pages 593–607. Springer, 2018.

[40] Rana Shahout, Eran Malach, Chunwei Liu, Weifan Jiang, Minlan Yu, and Michael Mitzenmacher. Don't stop me now: Embedding based scheduling for LLMs. In *International Conference on Learning Representations*, volume 2025, pages 63345–63368, 2025.

[41] Oscar Skean, Md Rifat Arefin, Dan Zhao, Niket Nikul Patel, Jalal Naghiyev, Yann Lecun, and Ravid Shwartz-Ziv. Layer by layer: Uncovering hidden representations in language models. In *International Conference on Machine Learning*, pages 55854–55875. PMLR, 2025.

[42] Tom Ulanovski, Eyal Blyachman, and Maya Bechler-Speicher. Improving LLM predictions via inter-layer structural encoders. In *ICLR Workshop on Geometry-grounded Representation Learning and Generative Modeling*, 2026.

[43] Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. *Journal of the ACM (JACM)*, 68(4):1–25, 2021.

[44] Dhruv Rohatgi. Near-optimal bounds for online caching with machine learned advice. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1834–1845. SIAM, 2020.

[45] Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. *Communications of the ACM*, 65(7):33–35, 2022.

A Workload Characterization

Table 2 summarizes the key characteristics of the three evaluated workloads, including context lengths, KV-cache footprints, prefix-sharing ratios, and reuse distances.

Table 2: Workload characterization of the evaluated multi-agent workflows. We report the mean, P50, and P90 of each metric.

Workload	Metric	Mean	P50	P90
HoVer	Input context (tokens)	1,780.14	1,527	2,467
	KV footprint (GiB)	0.546	0.506	0.627
	Prefix sharing (%)	78.57	80.03	83.57
	Reuse distance (tokens)	25,204.28	7,040	48,803
	Reuse distance (GiB)	6.15	1.72	11.91
SWE-bench	Input context (tokens)	3,966.08	3,519	6,840
	KV footprint (GiB)	1.186	1.103	1.997
	Prefix sharing (%)	86.56	83.79	91.85
	Reuse distance (tokens)	40,136.39	31,136	98,460
	Reuse distance (GiB)	9.80	7.60	24.04
FinanceBench	Input context (tokens)	2,167.12	1,986	3,641
	KV footprint (GiB)	0.687	0.627	1.044
	Prefix sharing (%)	65.82	65.60	69.69
	Reuse distance (tokens)	18,519.16	5,901	67,222
	Reuse distance (GiB)	4.52	1.44	16.41

Metric Definitions. Input context is measured per agent invocation, including the system prompt, task input, and workflow history. KV footprint denotes the sequential KV-cache footprint per workflow, estimated using approximately 0.25 MiB/token for Qwen3-32B, excluding padding and fragmentation. Prefix sharing measures the token-prefix overlap between consecutive agent invocations. Reuse distance measures the intervening token accesses between consecutive accesses to the same cache node in the global access stream, including concurrent workflows. Its memory equivalent is computed using the same per-token KV-cache footprint.

Observation. All three workloads exhibit substantial prefix sharing and long reuse distances, highlighting the opportunities for workflow-aware KV-cache management under memory pressure.

B Main Results with Standard Deviations.

In our main experiments, we evaluate LRU, KVFlow (only on the static workload due to its inherent limitation), and our PBKV along with its variants on two LLMs and three workloads. Each setting is repeated for 10 runs, with mean and standard deviation reported in Table 3.

C Generalization under Distribution Shifts

We evaluate PBKV's generalization under two types of distribution shifts: cross-benchmark transfer and cross-repository transfer. For each setting, we consider two variants: (i) *zero-shot*, which directly deploys the predictor trained on the source workload, and (ii) *few-shot*, which further fine-tunes

Table 3: Full results with standard deviations across multi-agent workloads. Each cell reports mean $\pm$ standard deviation over multiple runs. The *static* workload (FinanceBench+CrewAI) is included for comparison with KVFlow, which assumes a predefined static call graph. Concurrency limits are 72 for HoVer+LangChain, 24 for SWE-bench+AutoGen, and 48 for FinanceBench+CrewAI.

Model	Workload	Policy	Latency (s) ↓	Per-Agent TTFT (s) ↓	Cache Hit Rate (%) ↑
Qwen3-32B	HoVer + LangChain	LRU	189.66 ± 4.70	16.65 ± 0.78	27.09 ± 1.16
		PBKV-LAE	146.67 ± 6.73	11.83 ± 0.74	44.91 ± 2.20
		PBKV-HE	108.86 ± 6.93	8.95 ± 0.74	66.01 ± 2.79
		Full PBKV	102.60 ± 7.28	8.22 ± 0.71	69.10 ± 2.23
	SWE-bench + AutoGen	LRU	271.90 ± 9.14	5.04 ± 0.26	46.34 ± 2.01
		PBKV-LAE	233.86 ± 12.70	3.73 ± 0.44	59.51 ± 2.92
		PBKV-HE	177.22 ± 10.36	2.46 ± 0.37	75.31 ± 2.57
		Full PBKV	160.18 ± 10.89	2.27 ± 0.35	79.94 ± 3.26
	FinanceBench + CrewAI (*static*)	LRU	130.34 ± 5.37	12.93 ± 0.51	27.95 ± 0.82
		KVFlow	101.57 ± 5.27	9.91 ± 0.57	39.87 ± 1.43
		Full PBKV	80.53 ± 5.06	8.00 ± 0.48	53.44 ± 2.07
Qwen3-14B	HoVer + LangChain	LRU	139.59 ± 3.13	10.90 ± 0.35	28.79 ± 1.04
		PBKV-LAE	119.13 ± 6.04	9.57 ± 0.58	41.18 ± 3.17
		PBKV-HE	80.62 ± 5.13	6.53 ± 0.34	64.15 ± 1.99
		Full PBKV	76.15 ± 5.95	5.50 ± 0.42	68.29 ± 2.32
	SWE-bench + AutoGen	LRU	214.74 ± 6.98	4.18 ± 0.28	48.53 ± 3.02
		PBKV-LAE	177.50 ± 9.05	3.26 ± 0.51	60.41 ± 4.62
		PBKV-HE	141.53 ± 7.57	2.19 ± 0.43	73.36 ± 3.95
		Full PBKV	118.61 ± 8.06	2.05 ± 0.47	77.07 ± 3.71
	FinanceBench + CrewAI (*static*)	LRU	96.11 ± 4.88	9.02 ± 0.44	26.58 ± 0.86
		KVFlow	80.13 ± 4.16	7.42 ± 0.41	39.65 ± 1.84
		Full PBKV	65.01 ± 4.12	6.06 ± 0.40	55.07 ± 2.39

the predictor on 10 traces from the target workload. We compare both variants against LRU and PBKV-LAE. All experiments use Qwen3-32B, with other settings following Section 6.1.

Cross-Benchmark Transfer. We replace HoVer with FEVER [35] in the LangChain-based fact-verification workload, using the predictor originally trained on HoVer. Table 4 reports the results.

Table 4: Cross-benchmark generalization from HoVer to FEVER with Qwen3-32B.

Policy	1-step Acc. (%) ↑	3-step Acc. (%) ↑	Lat. (s) ↓	PA-TTFT (s) ↓	Hit Rate (%) ↑
LRU	–	–	131.71	15.90	30.12
PBKV-LAE	–	–	81.36	10.26	58.63
PBKV (Zero-shot)	88.63	64.24	65.97	8.41	69.32
PBKV (Few-shot)	91.47	70.16	61.02	7.74	71.86

Cross-Repository Transfer. We train the predictor on a subset of code repositories and evaluate it on tasks from entirely unseen repositories in SWE-bench+AutoGen. Table 5 reports the results.

Table 5: Cross-repository generalization to unseen code repositories with Qwen3-32B.

Policy	1-step Acc. (%) ↑	3-step Acc. (%) ↑	Lat. (s) ↓	PA-TTFT (s) ↓	Hit Rate (%) ↑
LRU	–	–	256.35	4.88	49.24
PBKV-LAE	–	–	218.97	3.34	61.27
PBKV (Zero-shot)	90.05	84.81	188.95	2.96	71.41
PBKV (Few-shot)	92.77	90.40	171.02	2.13	76.26

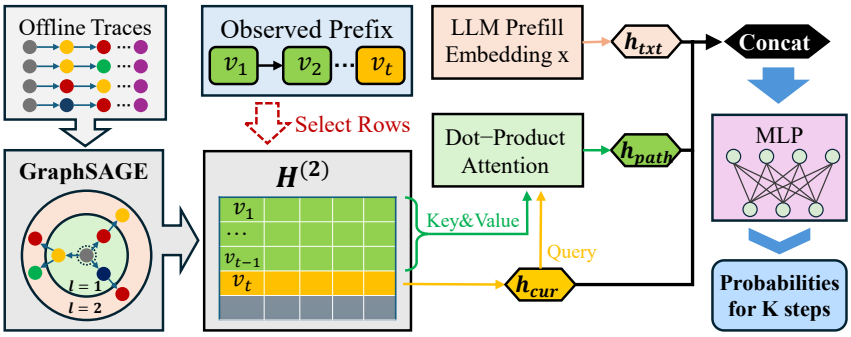


Figure 7: **Architecture of the predictor.** It fuses a topology-aware agent embedding from GraphSAGE (h_{cur}), an attention-based workflow prefix summary (h_{path}), and a semantic signal reused from prefill (h_{txt}), then jointly predicts the next K agent probability distributions via an MLP head.

Results. As shown in Tables 4 and 5, PBKV consistently outperforms LRU and PBKV-LAE under both distribution shifts, even without target-domain fine-tuning. Few-shot adaptation further improves prediction accuracy, cache hit rate, and serving latency.

D Predictor Details: Architecture and Training

This appendix complements Section 4.1 with the full design of the predictor, including the rationale behind the GraphSAGE backbone, the detailed architecture, and the training procedure.

Overview. The predictor adopts *GraphSAGE* [19] as its backbone, fuses *multiple complementary* signals, and jointly forecasts the next *few* invoked agents (i.e., Multi-Step Prediction).

Why GraphSAGE? Our predictor design follows two desiderata. *First*, the predictor must serve *all* possible workflows across every admissible agent transition, so it is desirable to leverage the structural priors encoded in the global call graph G. *Second*, the dynamic nature of agentic workflows requires the predictor to generalize to unseen subgraphs encountered at runtime.

Guided by these desiderata, we adopt **GraphSAGE** as the backbone of predictor. *First*, GraphSAGE learns node representations directly on the graph through neighborhood sampling and aggregation, thereby preserving the structural priors encoded in agent transitions, which an MLP would otherwise discard. *Second*, GraphSAGE is an *inductive* framework, which performs forward inference on unseen workflow prefix without retraining on the entire graph. This property aligns naturally with the dynamic nature of agentic workflows, and is preferable to transductive alternatives (e.g., GCN [36]) that rely on a fixed training graph. For completeness, we also implement and evaluate several baseline predictors (including MLPs, GCNs, and Markov models), detailed results are in Appendix E.

Architecture. Figure 7 illustrates the architecture of predictor. To forecast the future invocations of an active workflow, the predictor fuses three complementary streams as described below:

(i) Topology-aware agent representation. We assign each agent $v \in V$ a learnable embedding $\mathbf{e}_v \in \mathbb{R}^d$, stacked into $\mathbf{H}^{(0)} \in \mathbb{R}^{|V|\times d}$. $A \in \mathbb{R}^{|V|\times|V|}$ denotes the row-normalized forward *transition matrix* estimated from the offline traces. We then apply two GraphSAGE propagation layers:

$$\mathbf{H}^{(\ell)} = \mathrm{ReLU}\Big(\big[\mathbf{H}^{(\ell-1)} \,\|\, A\mathbf{H}^{(\ell-1)}\big](\mathbf{W}^{(\ell)})^\top\Big), \quad \mathbf{W}^{(\ell)} \in \mathbb{R}^{d\times 2d}, \quad \ell = 1, 2 \tag{3}$$

where $\|$ denotes feature-wise concatenation. Thus, for each agent, the resulting $\mathbf{H}^{(2)}$ encodes both its own identity (through $\mathbf{H}$) and the structural priors induced by its neighborhood in G (through $A\mathbf{H}$).

(ii) Attention-based history aggregation. While $\mathbf{H}^{(2)}$ equips each agent with a topology-aware representation, it does not yet capture the state of a *workflow*. In fact, two workflows residing at the same agent v_t may have arrived through very different prefixes. To leverage such prefix information, we summarize the prefix via scaled *dot-product attention*, using the current agent representation

$\mathbf{h}_{\text{cur}} = \mathbf{H}^{(2)}_{v_t}$ as the query and the prefix representations as both keys and values:

$$\alpha_i = \frac{\exp\big((\mathbf{W}_q \mathbf{h}_{\text{cur}})^\top \mathbf{H}^{(2)}_{v_i} / \sqrt{d}\big)}{\sum_{j<t} \exp\big((\mathbf{W}_q \mathbf{h}_{\text{cur}})^\top \mathbf{H}^{(2)}_{v_j} / \sqrt{d}\big)}, \qquad \mathbf{h}_{\text{path}} = \sum_{i<t} \alpha_i \, \mathbf{H}^{(2)}_{v_i} \tag{4}$$

Compared with naive mean pooling, attention allows the predictor to upweight those prefix agents that are most informative of the current state.

(iii) Semantic signal from prefill. Graph topology and workflow history alone do not distinguish requests that share the same prefix but diverge in intent. To capture such *request-specific* semantics, we reuse the post-norm hidden state $\mathbf{x}$ of the last prefill token and project it through a single linear layer, i.e., $\mathbf{h}_{\text{txt}} = \text{ReLU}(\mathbf{W}_t \mathbf{x})$. As $\mathbf{x}$ is a by-product of prefill, this signal is obtained essentially for free.

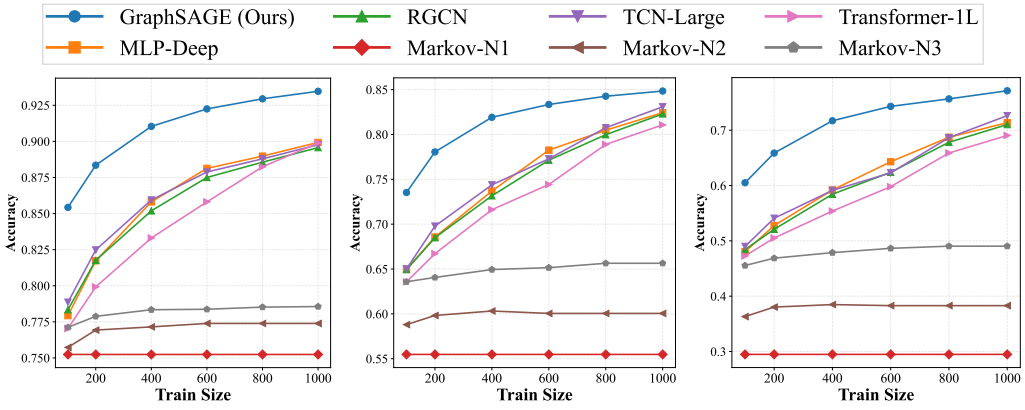
Multi-step prediction head. The above streams are concatenated and fed into a two-layer MLP with dropout, which jointly emits logits over agents for each of the next K steps. Emitting all K logits from a shared backbone in one forward pass further avoids autoregressive error accumulation and bounds per-invocation latency to that of a single inference.

Training Strategy. We train the predictor on offline invocation traces. For each position t within a workflow, we pair the prefix $(v_1, \ldots, v_t)$ and its prefill embedding with the future agents $(v_{t+1}, \ldots, v_{t+K})$ as labels. The token $\langle\text{END}\rangle$ is treated as a regular target so that the termination probability $p_{w,\langle\text{END}\rangle}$ is properly learned, and only the padding positions after $\langle\text{END}\rangle$ in a target window are masked from the cross-entropy loss. The resulting predictor has roughly 350K parameters (orders of magnitude smaller than the served LLM) and processes a batch of 1,024 requests in 1.56 ms, rendering its runtime overhead negligible. Detailed hyperparameter settings and training configurations are provided in the supplementary code.

E Workflow Prediction

This appendix supplements Section 4.1 with three additional results: (i) the scaling behavior of our predictor across training-set sizes (Section E.1), (ii) a comparison against alternative predictor families (Section E.2), and (iii) the sensitivity of prediction accuracy to which transformer layer the prefill semantic signal is extracted from (Section E.3).

Setup. All experiments use the HoVer [20] dataset with the LangChain [1] agent framework. We report top-1 accuracy at horizons k=1, 2, 3 (denoted s_1, s_2, s_3), averaged over 10 random seeds.



(a) Prediction Accuracy of Step 1. (b) Prediction Accuracy of Step 2. (c) Prediction Accuracy of Step 3.

Figure 8: Top-1 prediction accuracy as a function of training-set size at horizons k=1, 2, 3. Our predictor leads every baseline at every train size shown.

E.1 Scaling Behavior

Figure 8 plots top-1 accuracy against the number of training traces. Two observations emerge. First, learning-based predictors scale with data while the Markov baseline saturates within 100 traces,

justifying predicting from a learned representation rather than from tabulated transition probabilities. Second, our predictor leads every learning-based baseline at every horizon and the lead widens as the horizon grows.

E.2 Comparison Across Predictor Families

Table 6: Predictor comparison at 1000 training traces, grouped by architectural family (mean $\pm$ std over 10 seeds).

Family	Predictor	s_1	s_2	s_3
GraphSAGE (Ours)		$\mathbf{0.935} \pm 0.007$	$\mathbf{0.848} \pm 0.012$	$\mathbf{0.771} \pm 0.019$
R-GCN	R-GCN	0.896 ± 0.008	0.823 ± 0.011	0.710 ± 0.018
Transformer	1L	0.899 ± 0.008	0.811 ± 0.023	0.690 ± 0.034
	2L	0.896 ± 0.011	0.810 ± 0.022	0.688 ± 0.040
TCN	SMALL	0.901 ± 0.006	0.824 ± 0.015	0.717 ± 0.019
	LARGE	0.898 ± 0.006	0.831 ± 0.007	0.726 ± 0.019
MLP	TINY	0.786 ± 0.003	0.656 ± 0.004	0.494 ± 0.003
	SMALL	0.889 ± 0.004	0.801 ± 0.012	0.684 ± 0.021
	DEEP	0.899 ± 0.006	0.824 ± 0.016	0.714 ± 0.024
Linear Probe	POST-NORM	0.840 ± 0.026	0.732 ± 0.026	0.576 ± 0.036
kNN	k=5	0.839 ± 0.008	0.699 ± 0.007	0.564 ± 0.008
	k=20	0.817 ± 0.005	0.707 ± 0.007	0.576 ± 0.006
Markov	n=1	0.752 ± 0.000	0.555 ± 0.000	0.295 ± 0.000
	n=2	0.774 ± 0.000	0.601 ± 0.000	0.383 ± 0.000
	n=3	0.786 ± 0.000	0.656 ± 0.000	0.490 ± 0.000

Table 6 compares our predictor against every baseline family we implemented: parameter-free baselines (Markov, kNN), a linear probe on prefill states, MLPs, sequence models (Transformer [37], TCN [38]), and a relational graph encoder (R-GCN [39]). Our predictor outperforms the strongest variant of every other family on all three horizons. The lead widens further at longer horizons, consistent with the multi-step argument that long-horizon prediction benefits most from a rich summary of the workflow prefix.

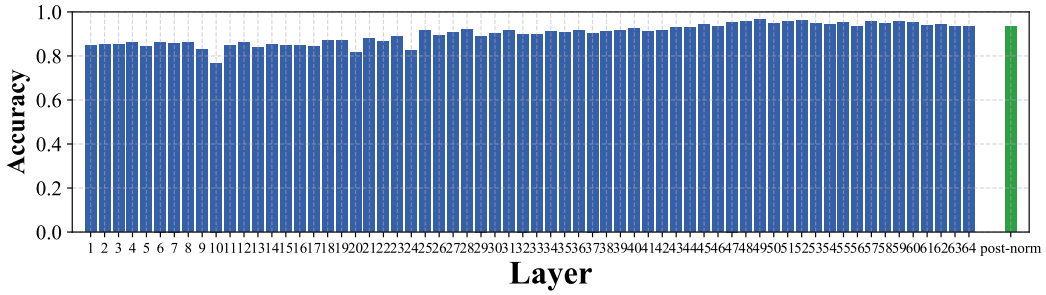
E.3 Choice of the Layer for Semantic Extraction

The prefill semantic signal h_{txt} described in Section 4.1 is extracted from a single layer of the served LLM, and a natural question is which layer to choose. Prior studies [40–42] report that the optimal layer for hidden state extraction is uncertain and depends jointly on the served model and the downstream task. We therefore study this choice empirically on Qwen3-32B by training the predictor with h_{txt} taken from each transformer block output (ℓ=1, . . . , 64) and from the post-norm hidden state *post-norm*, holding the rest of the architecture and the training protocol fixed.

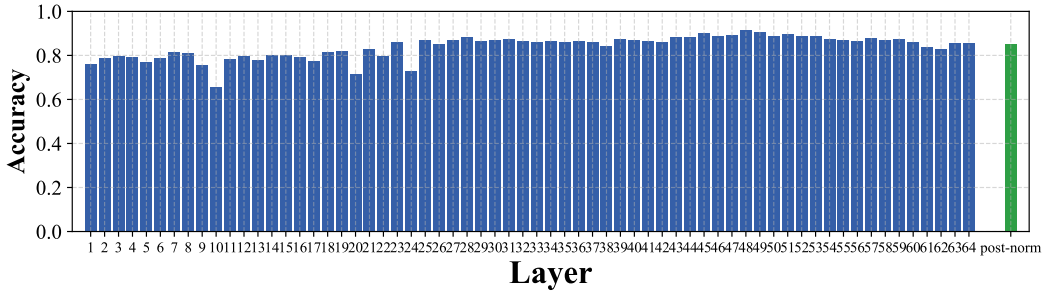
Figure 9 reports the resulting s_1, s_2, s_3 accuracies. Consistent with prior observations, the optimal layer varies across horizons and is highly specific to the served LLM and the workload at hand: deploying our system on a different LLM or workload would require re-running the layer scan to identify a new optimum. To preserve compatibility with arbitrary served LLMs and workloads, and to keep the predictor self-contained, we extract h_{txt} from *post-norm* in all other experiments. The post-norm hidden state is the standard input to the LLM's output head, and is therefore universally accessible across LLMs without extra instrumentation, regardless of how many transformer blocks the model contains.

E.4 Serving Performance with Limited Training Data

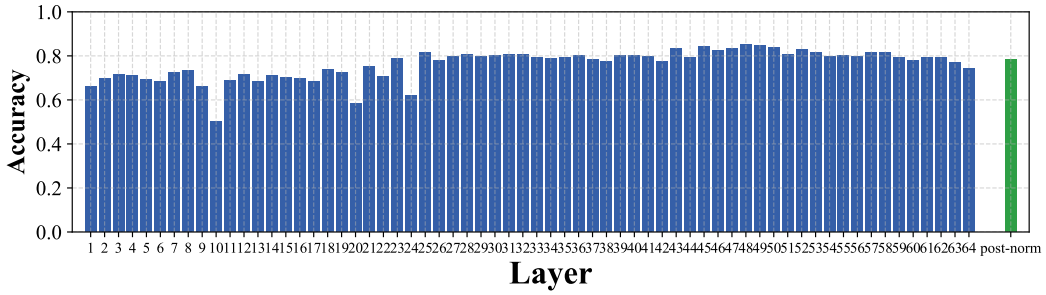
We evaluate PBKV's serving performance with limited predictor training data on HoVer+LangChain with Qwen3-32B. We consider two additional variants: (i) PBKV-U, which replaces predictor outputs with uniform distributions and requires no training traces, and (ii) PBKV trained on only one trace. We also include SGLang's built-in LFU policy as a popularity-based baseline.



(a) Prediction Accuracy of Step 1.



(b) Prediction Accuracy of Step 2.



(c) Prediction Accuracy of Step 3.

Figure 9: Top-1 prediction accuracy as a function of the layer from which the prefill semantic signal h_{txt} is extracted. The served LLM is Qwen3-32B, which exposes 64 transformer block outputs (ℓ=1, . . . , 64) followed by the post-norm hidden state *post-norm* that is fed into the output head.

As shown in Table 7, PBKV-U outperforms LRU, LFU, and PBKV-LAE even without a trained predictor, demonstrating the benefits of PBKV's cache-management design. Training with just one trace further reduces latency from 135.72 s to 126.50 s, indicating that PBKV can benefit from even limited training data.

F Sensitivity to the Confidence Decay Coefficient

The score computation uses a coefficient γ to discount the contribution of farther-step predictions, reflecting their decreasing reliability. We study its sensitivity on Qwen3-32B under the HoVer+LangChain workload. As shown in Figure 10, $\gamma = 0.7$ achieves the best performance. A smaller γ can cause the score to decay too rapidly, suppressing lookahead and degrading toward myopic behavior; while a larger γ will over-weight unreliable predictions at distant steps, leading to misguided eviction decisions.

Table 7: Serving performance with limited predictor training data on HoVer+LangChain with Qwen3-32B.

Policy	Lat. (s) ↓	PA-TTFT (s) ↓	Hit Rate (%) ↑
LRU	189.66	16.65	27.09
LFU	207.33	18.87	22.40
PBKV-LAE	146.67	11.83	44.91
PBKV-U	135.72	10.79	49.03
PBKV (1 trace)	126.50	9.81	52.89
Full PBKV	102.60	8.22	69.10

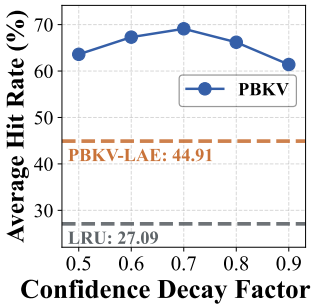


Figure 10: Average cache hit rate vs. confidence decay coefficient γ.

G Aggressive vs. Conservative Prefetching under Varying Prediction Accuracy

G.1 Motivation and Setup

Section 4.3 argues that PBKV adopts a conservative prefetching principle: under dynamic workflows, the uncertainty of prediction makes "displacing known-valuable active cache in exchange for speculatively valuable cache" an asymmetric trade-off between deterministic cost and probabilistic benefit. This appendix empirically validates this design choice.

Aggressive variant. We construct an aggressive prefetching variant that relaxes only the space constraint of Section 4.3, while keeping the rest of the design unchanged (i.e., the predictor, the scoring function, the activation only on pure-decode batches, and the PCIe bandwidth budget S_{bw}). Specifically, beyond the conservative budget S_a, the aggressive variant is additionally permitted to displace at most $\rho \cdot S_{\text{total}}$ of active cache to make room for prefetching, where S_{total} denotes the total GPU cache capacity. Active nodes are evicted in ascending order of their lookahead score, ensuring that the lowest-scored nodes are displaced first.

Prediction noise injection. To isolate the effect of prediction accuracy, we apply a controlled perturbation to the predictor's output. Let $P_w^{(k)}$ denote the GraphSAGE next-step distribution for workflow w at horizon k, and let U denote the uniform distribution over agents. The perturbed distribution is:

$$\tilde{P}_w^{(k)} = (1-\lambda)\, P_w^{(k)} + \lambda\, U, \tag{5}$$

where $\lambda \in [0,1]$ denotes the noise level. $\lambda = 0$ corresponds to the original predictor, while $\lambda = 1$ degenerates to a uniform (i.e., random) prediction. The perturbation is applied to the predictor outputs consumed by both eviction and prefetching, simulating a system-wide degradation of the prediction signal.

Workload and metrics. We follow the main experimental setup, i.e., the HoVer + LangChain workload with Qwen3-32B at a concurrency of 72. Each configuration is repeated 10 times, and we report the mean and standard deviation of the average cache hit rate.

G.2 Experiment 1: Sensitivity to the Prefetching Space Budget

We first sweep ρ under the noise-free setting $\lambda = 0$ to identify the optimal aggressiveness of the variant.

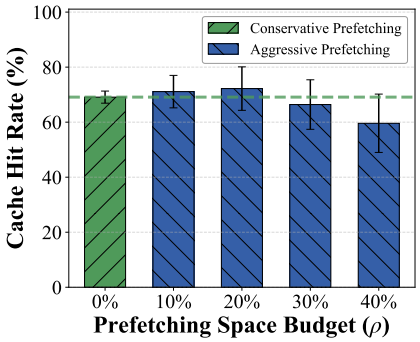


Figure 11: Cache hit rate of conservative and aggressive prefetching across different prefetching space budgets ρ, under the noise-free setting ($\lambda = 0$). Error bars denote standard deviation over 10 trials.

As shown in Figure 11, the hit rate of aggressive prefetching first increases and then decreases with ρ, peaking at $\rho = 20\%$ with roughly a 3-percentage-point margin over conservative prefetching. As ρ further grows, the hit rate falls below the conservative baseline, indicating that the cumulative cost of mistakenly displacing active cache outweighs the gain from prefetching.

More notably, the standard deviation grows monotonically with ρ even under the noise-free setting. At the optimal point $\rho^* = 20\%$, the standard deviation is already $\sim 3.5\times$ that of conservative prefetching. This indicates that, although aggressive prefetching can deliver expected gains, it harms the stability of the system.

In the following analysis, we use $\rho^* = 20\%$, at which aggressive prefetching achieves its best performance.

G.3 Experiment 2: Sweeping the Prediction Noise

Under $\rho = \rho^*$, we sweep the noise level and compare PBKV-HE (no-prefetching reference), conservative prefetching (the default in PBKV), and aggressive prefetching.

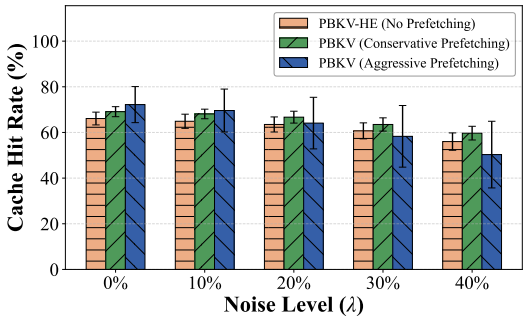


Figure 12: Cache hit rate of PBKV-HE, conservative prefetching, and aggressive prefetching across different noise levels λ. Error bars denote standard deviation over 10 trials.

As shown in Figure 12, two key observations emerge:

(i) The advantageous regime of aggressive prefetching is narrow. Only at $\lambda = 0$ and $\lambda = 10\%$ does aggressive prefetching outperform conservative prefetching, and the margin shrinks rapidly as the noise increases. At $\lambda = 20\%$, aggressive prefetching is already overtaken by conservative prefetching. In practice, the realistic operating regime of aggressive prefetching is even narrower.

(ii) Aggressive prefetching falls below PBKV-HE under moderate-to-high noise. At $\lambda = 30\%$, aggressive prefetching already underperforms PBKV-HE, which performs no prefetching at all; at $\lambda = 40\%$ the gap further widens. This shows that aggressive prefetching becomes counterproductive under inaccurate predictions, with its cost outweighing its benefit. In contrast, conservative prefetching consistently outperforms PBKV-HE across all noise levels, reflecting its lower-bound guarantee.

G.4 Justification of the Design Choice

In summary, aggressive prefetching only achieves a marginal and unstable advantage over conservative prefetching when the prediction accuracy is sufficiently high (i.e., the noise level is below 20%). However, dynamic multi-agent workflows are inherently hard to predict, especially under complex agent frameworks. Therefore, PBKV adopts conservative prefetching to obtain stable gains.

H Scheduler Overhead Analysis

A natural concern with prediction-driven cache management is whether the additional scheduler logic erodes the cache-hit-rate gains. We instrument every PBKV scheduler hook in our SGLang implementation and measure per-call wall-clock latency over a full run of the HoVer+LangChain workload on Qwen3-32B at a concurrency of 72, identical to the operating point used throughout Section 6.2.

Table 8: Per-call latency of PBKV's scheduler components on Qwen3-32B with the HoVer+LangChain workload at concurrency 72. EvictDecision selects victim nodes on memory pressure; PredictInfer runs one forward pass of the GraphSAGE predictor; ScoreUpdate refreshes a cache node's heap entry after a new prediction; PrefetchDecision ranks host-resident candidates under the (S_a, S_{bw}) budget; DecodeStep is one batched decode iteration, included as a reference. **Note the unit difference**: ScoreUpdate is in microseconds (µs); all other components in milliseconds (ms).

Component	Mean	P99
EvictDecision	0.44 ms	0.81 ms
PredictInfer	1.18 ms	1.68 ms
ScoreUpdate	1.53 **µs**	2.69 **µs**
PrefetchDecision	0.64 ms	1.60 ms
DecodeStep	12.34 ms	43.14 ms

Table 8 reports the mean and P99 latency of EvictDecision, PredictInfer, ScoreUpdate, and PrefetchDecision, with DecodeStep included as a reference for relative cost.

Among them, EvictDecision, ScoreUpdate, and PrefetchDecision run on the CPU scheduler thread and operate purely on PBKV's heap-based bookkeeping structures, concurrent with the GPU decode kernel. PredictInfer executes on the GPU but is issued on a separate CUDA stream that overlaps with the ongoing decode kernel. None of the four therefore sit on the GPU critical path. ScoreUpdate in particular is essentially free at 1.53 μs on average, precisely what allows per-node scores to be refreshed at high frequency without serializing the scheduler. Eviction and prefetch decisions both average well under 1 ms (0.44 ms and 0.64 ms), and even predictor inference – the dominant component – averages just 1.18 ms, less than 10% of a mean decode step (12.34 ms), and the other components fall well below this. These overheads do not show up in end-to-end performance: PBKV achieves a $1.85\times$ speedup over LRU on this workload (Section 6.2).

I Smoothness Analysis

Overview. This appendix establishes the smoothness pillar of PBKV's algorithms-with-predictions guarantee. We organize the analysis into four layers. Lemma I.1 identifies the per-node score $\mathrm{SCORE}(c)$ as the expected discounted miss count attributable to evicting c, thereby grounding the proxy cost in a system-relevant quantity. Lemma I.2 shows that this score is Lipschitz-continuous in

the predicted distributions. Corollary I.3 translates this score-level continuity into pairwise ranking stability. Theorem I.4 further lifts it to a cost-level regret bound. All Lipschitz-type results share a single multiplier with a K-independent upper bound $1/\big(2(1-\gamma)\big)$ that is also independent of $|\mathcal{W}_{\mathrm{act}}|$ and the cache size, which justifies treating the predictor as a pluggable module: any future improvement in predictor accuracy directly tightens all bounds.

I.1 Setup

Probabilistic model. We posit a ground-truth conditional distribution under which each active workflow w produces, at every step k, a true distribution $P_w^{(k)} \in \Delta^{|V|}$ over $V \cup \{\langle \mathrm{END} \rangle\}$, conditioned on the workflow having not terminated before step k. Here $\Delta^{|V|}$ denotes the $|V|$-dimensional probability simplex over the $|V|+1$ outcomes in $V \cup \{\langle \mathrm{END} \rangle\}$, following the standard topological convention. The predictor outputs an estimate $\widehat{P}_w^{(k)}$ on the same simplex. Consistent with the sequential agent execution adopted by mainstream multi-agent frameworks (e.g., LangChain [1], Microsoft AutoGen [2], MetaGPT [3]), we model each active workflow as invoking exactly one agent per step (or terminating), so that $P_w^{(k)}$ is the marginal distribution of the step-k invocation conditioned on survival to step k. Let $p_{w,\langle \mathrm{END} \rangle}^{(j)}$ denote the conditional termination probability at step j. The cumulative survival factor follows the standard hazard-product decomposition $s_w^{(k)} = \prod_{j=1}^{k-1}\big(1 - p_{w,\langle \mathrm{END} \rangle}^{(j)}\big)$. Throughout, $\mathcal{W}_{\mathrm{act}}(c)$ denotes the set of active workflows associated with node c at the moment the eviction decision is made.

Eviction-only abstraction. Following the classical formulation of caching analysis pioneered by Belady [8] and adopted by recent algorithms-with-predictions work [43, 44], we evaluate eviction policies under the *eviction-only abstraction*, i.e., a node, once evicted, remains out of the cache for the entire K-step horizon over which the cost is measured. This abstraction isolates the quality of the eviction decision from orthogonal storage-hierarchy mechanisms (e.g., HiCache re-admission on miss), which are properties of the underlying system rather than of the eviction policy itself. PBKV's eviction policy operates on the same abstraction and is agnostic to whether a second-tier storage is present. The composition of our analysis with HiCache is discussed at the end of Section I.4.

Access-indicator semantics. For each active workflow $w \in \mathcal{W}_{\mathrm{act}}(c)$, the access indicator $A_w(c) \in \{0,1\}^{|V|}$ marks the set of agents within w that access c, denoted $\mathcal{O}_w(c) := \{a \in V : A_w(c)_a = 1\}$. Two cases arise. (i) For private cache, $|\mathcal{W}_{\mathrm{act}}(c)|$ is small and $A_w(c)$ is typically one-hot, since the owning agent of c is determined by the upstream agent that produced it within w. (ii) For global cache (e.g., system prompts or tool/agent descriptions), $|\mathcal{W}_{\mathrm{act}}(c)|$ may be large and each $A_w(c)$ may have multiple ones, reflecting that distinct agents within the same workflow all access the shared prefix. The score formula in Eq. (2) treats both cases under a single mathematical form, with the cross-workflow outer sum protecting popular-prefix nodes by design (Section 4.2). For notational convenience, we adopt the convention that $A_w(c)$ is implicitly zero-padded on the $\langle \mathrm{END} \rangle$ coordinate, so that the inner product

$$A_w(c)^\top P_w^{(k)} \;=\; \sum_{a \in V} A_w(c)_a \, P_w^{(k)}(a) \;=\; P_w^{(k)}\big(\mathcal{O}_w(c)\big) \tag{6}$$

is dimensionally consistent and equals the probability that workflow w invokes some agent in $\mathcal{O}_w(c)$ at step k.

Past access equals future hit. The radix-tree prefix structure adopted by SGLang [5] ensures that $A_w(c)$, defined from the observed access history, also characterizes future hits within the same workflow. Specifically, a cache node c corresponds to a fixed token prefix; if agent a has invoked the radix path through c in w, then any subsequent invocation of a within w (e.g., under a retry loop) prepends the same system prompt and upstream context, and therefore traverses the same prefix and hits c provided c remains cached. Conversely, an agent that has never accessed c does not have c on its execution path and will not hit c in future invocations within w. Hence, throughout the analysis, "w would hit c at step k if c remained cached" is equivalent to "w invokes some $a \in \mathcal{O}_w(c)$ at step k".

Unit-size abstraction and the role of node size. A natural question is why $\mathrm{SCORE}(c)$ in Eq. (2) does not contain an explicit $|c|$ factor. This is a deliberate design choice: for a cache node c of size $|c|$, both the space cost of retaining it and the wall-clock cost of a miss on it scale linearly with $|c|$, since a miss triggers the re-prefill (or PCIe transfer under HiCache) of $|c|$ tokens. The two factors

of $|c|$ therefore cancel out when the quantity of interest is value per unit space. By construction, $\mathrm{SCORE}(c)$ measures exactly this per-unit-space value, which is also the canonical ranking criterion of fractional-knapsack-style cache replacement: ranking by per-unit-space value, rather than by absolute value, is what minimizes total miss cost under a space budget. Multiplying $\mathrm{SCORE}(c)$ by $|c|$ would conflate "valuable" with "large" and bias eviction against small-but-hot nodes, which is the opposite of the intended behavior.

For the regret analysis, we adopt the standard unit-size abstraction [8, 43, 44], under which each node contributes an equal volume to the eviction budget and the budget-B eviction reduces to selecting B lowest-scoring nodes. This abstraction is consistent with the size-heterogeneous setting via a standard reduction in which each node of size $|c|$ is treated as $|c|$ unit-size virtual nodes sharing the same per-unit-space score $\mathrm{SCORE}(c)$, whereby cardinality-constrained selection over virtual nodes recovers size-constrained selection over original nodes. Since radix-tree nodes are page-sized in practice and $|c| \ll B$, the boundary effect of integer-budget rounding is at most one node and does not affect the asymptotic behavior of the bound. The Lipschitz and regret bounds below therefore transfer to the size-heterogeneous setting without modification of the multiplier $1/\big(2(1-\gamma)\big)$.

Perturbation model. Following standard practice in algorithms-with-predictions analyses for caching [43, 44], we measure prediction error via the per-step ℓ_1 deviation and its node-local discount-weighted aggregate:

$$\epsilon_w^{(k)} := \big\| P_w^{(k)} - \widehat{P}_w^{(k)} \big\|_1, \qquad \epsilon_c^{\gamma} := \sum_{w \,\in\, \mathcal{W}_{\mathrm{act}}(c)} \sum_{k=1}^{K} \gamma^{k-1}\, \epsilon_w^{(k)}. \tag{7}$$

Throughout, we let $\mathrm{SCORE}(c)$ and $\widehat{\mathrm{SCORE}}(c)$ denote the per-node scores computed from $\{P_w^{(k)}\}$ and $\{\widehat{P}_w^{(k)}\}$ respectively, both via Eq. (2).

I.2 From Score to Expected Miss Count

We first establish the system-level meaning of $\mathrm{SCORE}(c)$, which grounds the subsequent regret analysis in a quantity of direct interest to caching theory.

Lemma I.1 (Score equals expected discounted miss count attributable to eviction)**.** *Under the eviction-only abstraction (Section I.1), for any cache node c that is evicted at the decision point and not re-admitted within the horizon,*

$$\mathrm{EMC}(c) \;:=\; \sum_{k=1}^{K} \gamma^{k-1}\, \mathbb{E}\big[\textit{misses on } c \textit{ at step } k\big] \;=\; \mathrm{SCORE}(c), \tag{8}$$

where the expectation is taken under the ground-truth distribution and the misses are aggregated across all $w \in \mathcal{W}_{\mathrm{act}}(c)$.

Proof. Fix a cache node c and a workflow $w \in \mathcal{W}_{\mathrm{act}}(c)$. Let $\#\mathrm{miss}_w^{(k)}(c) \in \{0,1\}$ denote the indicator that w incurs a miss on c at step k. Since w invokes exactly one agent per step (or terminates), and any invocation of an agent $a \in \mathcal{O}_w(c)$ would have traversed c by the radix-tree prefix structure (Section I.1), we have

$$\#\mathrm{miss}_w^{(k)}(c) \;=\; \mathbf{1}\big[w \text{ survives to step } k \text{ and invokes some } a \in \mathcal{O}_w(c)\big], \tag{9}$$

where the equality uses that, under the eviction-only abstraction, every such invocation is a miss. The events $\{w \text{ invokes } a \text{ at step } k\}_{a \in V}$ are mutually exclusive conditional on survival to step k, so

$$\Pr\big[w \text{ invokes some } a \in \mathcal{O}_w(c) \,\big|\, w \text{ survives to } k\big] \;=\; \sum_{a \in \mathcal{O}_w(c)} P_w^{(k)}(a) \;=\; A_w(c)^{\top} P_w^{(k)}. \tag{10}$$

Multiplying by $\Pr[w \text{ survives to } k] = s_w^{(k)}$ gives $\mathbb{E}\big[\#\mathrm{miss}_w^{(k)}(c)\big] = s_w^{(k)} \cdot A_w(c)^{\top} P_w^{(k)}$. Summing across $w \in \mathcal{W}_{\mathrm{act}}(c)$ by linearity of expectation and discounting by γ^{k-1} over $k = 1, \ldots, K$, comparing with Eq. (2) yields $\mathrm{EMC}(c) = \mathrm{SCORE}(c)$. □

Implications. Lemma I.1 provides three benefits. (i) It removes any circularity between the scoring rule and the cost it is evaluated against, since $\mathrm{SCORE}(c)$ is now identified with a quantity defined

purely from the ground-truth distribution and the cache state. (ii) It places PBKV's analysis on the same footing as classical caching results [8], where eviction policies are evaluated through their effect on miss counts, independent of any specific reload mechanism. (iii) It motivates the proxy cost adopted in Section I.4 as a system-relevant rather than ad-hoc surrogate.

I.3 Lipschitz Continuity of the Score

We next show that $\mathrm{SCORE}(c)$ depends continuously on the predicted distributions, with a Lipschitz constant whose dependence on the prediction horizon K admits a uniform K-independent upper bound.

Lemma I.2 (Lipschitz continuity of the score)**.** *For any cache node c and any pair of distribution sets $\{P_w^{(k)}\}$ and $\{\widehat{P}_w^{(k)}\}$ in $\Delta^{|V|}$,*

$$\left|\mathrm{SCORE}(c) - \widehat{\mathrm{SCORE}}(c)\right| \;\le\; \frac{1-\gamma^K}{2(1-\gamma)}\,\epsilon_c^{\gamma} \;\le\; \frac{\epsilon_c^{\gamma}}{2(1-\gamma)}. \tag{11}$$

The Lipschitz multiplier admits a K-independent upper bound $1/\big(2(1-\gamma)\big)$ that is also independent of $|\mathcal{W}_{\mathrm{act}}(c)|$ and the cache size; the dependence on these quantities is fully isolated in ϵ_c^{γ}.

Proof. We bound the per-term deviation $\big|s_w^{(k)} A_w(c)^\top P_w^{(k)} - \widehat{s}_w^{(k)} A_w(c)^\top \widehat{P}_w^{(k)}\big|$ in two steps and then aggregate over k and w.

Step 1 (per-term bound). The survival factor $s_w^{(k)}$ takes values in $[0,1]$ as a product of $[0,1]$ terms, and so does $\widehat{s}_w^{(k)}$. The standard telescoping identity for products of $[0,1]$-valued sequences gives

$$\big|s_w^{(k)} - \widehat{s}_w^{(k)}\big| \;\le\; \sum_{j=1}^{k-1} \big|p_{w,\langle\mathrm{END}\rangle}^{(j)} - \widehat{p}_{w,\langle\mathrm{END}\rangle}^{(j)}\big|. \tag{12}$$

For any pair of probability distributions P, Q on a finite outcome space and any subset S of outcomes, the difference of subset-event probabilities is bounded by the total variation distance, $|P(S)-Q(S)| \le \mathrm{TV}(P,Q) = \frac{1}{2}\|P-Q\|_1$. Applied to the singleton $S = \{\langle\mathrm{END}\rangle\}$, this yields $\big|p_{w,\langle\mathrm{END}\rangle}^{(j)} - \widehat{p}_{w,\langle\mathrm{END}\rangle}^{(j)}\big| \le \frac{1}{2}\epsilon_w^{(j)}$, and substituting into (12) gives

$$\big|s_w^{(k)} - \widehat{s}_w^{(k)}\big| \;\le\; \frac{1}{2}\sum_{j=1}^{k-1} \epsilon_w^{(j)}. \tag{13}$$

Similarly, by the zero-padding convention (6), $A_w(c)^\top P_w^{(k)} = P_w^{(k)}\big(\mathcal{O}_w(c)\big) \in [0,1]$ is a subset-event probability under $P_w^{(k)}$, and the same holds for $A_w(c)^\top \widehat{P}_w^{(k)}$. Applying the same total-variation bound to the subset $S = \mathcal{O}_w(c)$,

$$\big|A_w(c)^\top P_w^{(k)} - A_w(c)^\top \widehat{P}_w^{(k)}\big| \;\le\; \frac{1}{2}\,\epsilon_w^{(k)}. \tag{14}$$

Combining (13) and (14) via the elementary identity $|uv - \widehat{u}\widehat{v}| \le |v-\widehat{v}| + |u-\widehat{u}|$ for $u, \widehat{u}, v, \widehat{v} \in [0,1]$ yields

$$\left|s_w^{(k)} A_w(c)^\top P_w^{(k)} - \widehat{s}_w^{(k)} A_w(c)^\top \widehat{P}_w^{(k)}\right| \;\le\; \frac{1}{2}\sum_{j=1}^{k} \epsilon_w^{(j)}. \tag{15}$$

Step 2 (aggregation). Multiplying (15) by γ^{k-1}, summing over $k = 1, \dots, K$, and exchanging the order of summation between k and j,

$$\sum_{k=1}^{K} \gamma^{k-1} \cdot \frac{1}{2}\sum_{j=1}^{k} \epsilon_w^{(j)} \;=\; \frac{1}{2}\sum_{j=1}^{K} \gamma^{j-1}\epsilon_w^{(j)} \cdot \frac{1-\gamma^{K-j+1}}{1-\gamma} \;\le\; \frac{1-\gamma^K}{2(1-\gamma)} \sum_{j=1}^{K} \gamma^{j-1}\epsilon_w^{(j)}, \tag{16}$$

where the inner sum is a finite geometric series and the inequality uses $1-\gamma^{K-j+1} \le 1-\gamma^K$ for $j \ge 1$ and $\gamma \in (0,1)$. Summing across $w \in \mathcal{W}_{\mathrm{act}}(c)$ yields the tighter form $(1-\gamma^K)\epsilon_c^{\gamma}/\big(2(1-\gamma)\big)$ in Eq. (11), and the looser corollary $\epsilon_c^{\gamma}/\big(2(1-\gamma)\big)$ follows from $1-\gamma^K \le 1$. □

Locality remark. Only workflows in $\mathcal{W}_{\mathrm{act}}(c)$ contribute to ϵ_c^γ, so prediction errors on workflows w with $A_w(c) = 0$ leave $\mathrm{SCORE}(c)$ unchanged. This locality reflects the radix-tree prefix structure respected by the scoring rule and matches the design intent of Section 4.2.

Corollary I.3 (Ranking stability)**.** *Let c_1, c_2 be two cache nodes with $\mathrm{SCORE}(c_1) > \mathrm{SCORE}(c_2)$, and let $\Delta := \mathrm{SCORE}(c_1) - \mathrm{SCORE}(c_2)$. If*

$$\frac{1-\gamma^K}{2(1-\gamma)}\big(\epsilon_{c_1}^\gamma + \epsilon_{c_2}^\gamma\big) \;<\; \Delta, \tag{17}$$

then $\widehat{\mathrm{SCORE}}(c_1) > \widehat{\mathrm{SCORE}}(c_2)$, and the eviction preference between c_1 and c_2 is preserved.

Proof. By Lemma I.2 applied to c_1 and c_2 separately, together with the triangle inequality. □

I.4 Eviction Cost Regret

We now lift the score-level continuity to a cost-level regret bound, which is the form prescribed by the smoothness desideratum of the algorithms-with-predictions framework [45].

Proxy cost. We adopt

$$\mathcal{L}(E) \;:=\; \sum_{c\in E} \mathrm{SCORE}(c), \tag{18}$$

which, by Lemma I.1, equals the expected discounted miss count incurred by evicting the set E under the eviction-only abstraction. This grounds the proxy cost in the canonical quantity studied since Belady [8] and isolates the eviction-policy quality from orthogonal storage-hierarchy concerns. The same surrogate has also been used in algorithms-with-predictions analyses for caching [43, 44], where costs are likewise defined through predicted access patterns rather than realized latencies.

Regret definition. For the regret analysis, we specialize $\{P_w^{(k)}\}$ to the ground-truth distributions and $\{\widehat{P}_w^{(k)}\}$ to the predictor outputs. Freeing a budget of B nodes reduces to a cardinality-constrained selection (Section I.1). PBKV ranks candidate nodes by $\widehat{\mathrm{SCORE}}$ and evicts

$$\widehat{E}_B \;:=\; \arg\min_{|E|=B} \sum_{c\in E} \widehat{\mathrm{SCORE}}(c), \tag{19}$$

whereas the cost-minimizing eviction set under the ground truth is $E_B^\star := \arg\min_{|E|=B} \mathcal{L}(E)$. The eviction cost regret of PBKV is

$$\mathcal{R}(B) \;:=\; \mathcal{L}\big(\widehat{E}_B\big) - \mathcal{L}\big(E_B^\star\big) \;\geq\; 0, \tag{20}$$

where non-negativity follows from the optimality of $E_B^\star$ for $\mathcal{L}$.

Theorem I.4 (Eviction cost regret bound)**.** *Under the eviction-only abstraction and the unit-size abstraction (Section I.1), for any predictor outputs $\{\widehat{P}_w^{(k)}\}$, the eviction cost regret of PBKV satisfies*

$$\mathcal{R}(B) \;\leq\; \sum_{c\,\in\,\widehat{E}_B \triangle E_B^\star} \delta_c \;\leq\; \frac{1-\gamma^K}{2(1-\gamma)} \sum_{c\,\in\,\widehat{E}_B \triangle E_B^\star} \epsilon_c^\gamma, \tag{21}$$

where $\triangle$ denotes the symmetric set difference and $\delta_c := \big|\mathrm{SCORE}(c) - \widehat{\mathrm{SCORE}}(c)\big|$. As the prediction error vanishes, $\mathcal{R}(B) \to 0$.

Proof. Partition $\widehat{E}_B \cup E_B^\star$ into $\mathcal{A} := \widehat{E}_B \setminus E_B^\star$, $\mathcal{B} := E_B^\star \setminus \widehat{E}_B$, and $\mathcal{C} := \widehat{E}_B \cap E_B^\star$. Both eviction sets have cardinality B, hence $|\mathcal{A}| = |\mathcal{B}|$. By construction of $\widehat{E}_B$ as the minimizer of $\sum_{c\in E} \widehat{\mathrm{SCORE}}(c)$ over cardinality-B sets, and after cancelling the common $\mathcal{C}$-terms,

$$\sum_{c\in\mathcal{A}} \widehat{\mathrm{SCORE}}(c) \;\leq\; \sum_{c\in\mathcal{B}} \widehat{\mathrm{SCORE}}(c). \tag{22}$$

Therefore

$$\mathcal{R}(B) = \sum_{c\in\mathcal{A}} \mathrm{SCORE}(c) - \sum_{c\in\mathcal{B}} \mathrm{SCORE}(c) \tag{23}$$

$$\leq \sum_{c\in\mathcal{A}} \big(\mathrm{SCORE}(c) - \widehat{\mathrm{SCORE}}(c)\big) + \sum_{c\in\mathcal{B}} \big(\widehat{\mathrm{SCORE}}(c) - \mathrm{SCORE}(c)\big) \tag{24}$$

$$\leq \sum_{c\in\mathcal{A}\cup\mathcal{B}} \delta_c \;=\; \sum_{c\,\in\,\widehat{E}_B \triangle E_B^\star} \delta_c, \tag{25}$$

where the first inequality follows from (22) and the second from $|\mathrm{SCORE}(c) - \widehat{\mathrm{SCORE}}(c)| \leq \delta_c$ on each side. The remaining inequality of Eq. (21) follows by applying Lemma I.2 to each δ_c. □

Interpretation. Theorem I.4 bounds the regret of PBKV's eviction decisions on the expected discounted miss count, the canonical cost in caching analysis dating back to Belady [8]. Two observations are worth emphasizing. First, the multiplier $(1-\gamma^K)/\big(2(1-\gamma)\big)$ matches that of Lemma I.2, so the score-level and cost-level bounds share a single K-independent upper bound $1/\big(2(1-\gamma)\big)$. Second, only nodes in the symmetric difference $\widehat{E}_B \triangle E_B^\star$ contribute to $\mathcal{R}(B)$. A correctly ranked node, relative to the eviction boundary, contributes zero regret regardless of the magnitude of its individual score error. This boundary-localized regret structure indicates that the bound is governed by ranking errors near the eviction frontier rather than by aggregate score errors, consistent with the ranking-stability view of Corollary I.3.

Scope of the Guarantee. Following prior theoretical studies on caching [8, 43, 44], we analyze the expected discounted cache miss count and bound the associated eviction regret, as end-to-end serving latency is influenced by complex factors such as batching, queueing, and GPU scheduling.

I.5 Summary

This appendix establishes the smoothness pillar of PBKV's algorithms-with-predictions guarantee along four layers: identification of $\mathrm{SCORE}(c)$ as the expected discounted miss count under the eviction-only abstraction (Lemma I.1), per-node score Lipschitz continuity (Lemma I.2), pairwise ranking stability (Corollary I.3), and cost-level regret on the resulting proxy cost (Theorem I.4). All Lipschitz-type results share a single K-independent multiplier $1/\big(2(1-\gamma)\big)$, independent of $|\mathcal{W}_{\mathrm{act}}|$ and the cache size. By analyzing eviction quality under the same eviction-only abstraction adopted in classical caching theory, the analysis treats PBKV's eviction policy as a self-contained component decoupled from any specific storage-hierarchy mechanism. This justifies treating the predictor as a pluggable module in PBKV's design (Section 4.2): any future improvement in predictor accuracy directly tightens all bounds, and the bounds themselves remain meaningful in deployments without second-tier storage.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and Section 1 accurately reflect the design principles, theoretical guarantee, and empirical gains over LRU/KVFlow reported in Sections 4, 5, 6, and Appendix I.

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We mention our limitations in Section 5.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

Justification: We prove in Appendix I that PBKV degrades gracefully even under poor predictions (i.e., the degradation is Lipschitz-continuous).

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We disclose these information needed for reproduction in Section 6, and the code is available at `https://github.com/Hyzheng-code/PBKV`.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: Code is available at `https://github.com/Hyzheng-code/PBKV`; all datasets used (HoVer, FEVER, SWE-bench, FinanceBench) are publicly available.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: The settings are provided in Section 6.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: We provide error bars in Table 3 and Table 6.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)

- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The testbed is detailed in Section 6.1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We have read and understood the code of ethics and have made every effort to adhere to it.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: The work is a systems-level efficiency optimization for LLM serving and has no direct foreseeable negative societal impact beyond those of the underlying LLMs.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release any pretrained model or scraped dataset that poses a risk of misuse.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All used assets (Qwen3, SGLang, LangChain, AutoGen, CrewAI, HoVer, SWE-bench, FinanceBench) are cited with their original publications/repositories and used under their respective open-source licenses.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The PBKV implementation is publicly available at `https://github.com/Hyzheng-code/PBKV`, with instructions for reproducibility.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or any research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: LLMs are the inference target of our system. We used an LLM solely for editing assistance.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.