INLAY: ACCELERATING POSITION-INDEPENDENT KV CACHING WITH SPARSE REPAIRS AND SHARED DECODE KERNELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Position-independent KV caching (PIC) reduces prefill computation by reusing cached text chunks across different contexts. However, this reuse does not automatically yield compact KV storage or efficient shared execution during decoding. Request-specific repairs can lead to duplicated KV states, while request-oriented attention may repeatedly load the same shared data. We present Inlay, a serving runtime that combines compact KV storage with shared attention execution for PIC decoding. Inlay separates immutable shared Base KV, token-granular request- and layer-specific Repair KV, and private append-only Decode KV, while preserving the PIC provider's repair policy. It reuses loaded Base KV across compatible queries, applies request-specific masks and position adjustments, and merges partial attention outputs. A CPU-side planner estimates the execution costs of Base-Repair and Private modes for each chunk and selects between them under the KV memory budget. Valid plans, metadata, and packed prompt KV are reused across decode steps. On A100 attention microbenchmarks using Qwen3-8B and Llama-3.1-8B configurations, Inlay achieves median kernel speedups of $1.24\times$ over FlashInfer and $1.32\times$ over PAT across the tested workloads and attention geometries. Under a 32 GiB KV budget with a 5% repair ratio, its compact layout increases estimated resident-request capacity by 48.5% over private KV storage. Runtime ablations further show that reusing metadata and plans reduces repeated preparation overhead.

1 INTRODUCTION

Large language model (LLM) applications increasingly assemble prompts from shared content sources, resulting in overlapping text chunks across requests from different users. Retrieval-augmented generation (RAG) draws query-relevant passages from shared corpora, allowing requests with overlapping retrieval results to include the same text chunks (Lewis et al., 2020; Jin et al., 2025; Yao et al., 2025a; Agarwal et al., 2025). Tool-using agents select tools from shared catalogs, causing requests that use overlapping tool sets to include the same tool schemas and descriptions (Schick et al., 2023; Qin et al., 2024; Liu et al., 2024; Yao et al., 2024). In enterprise data platforms, internal copilots and agents can similarly reuse descriptions of database connectors, analytics APIs, governed datasets, and workflow actions across requests (Snowflake, n.d.; Microsoft, 2026). Different requests select and order these text chunks according to retrieval results and task requirements, combining them with request-specific instructions and dialogue histories. Consequently, identical chunks can appear in different orders, at different positions, and after different preceding contexts.

Modern LLM serving systems exploit common prefixes across requests to reduce redundancy in both *prefill* and *decoding*. During prefill, prefix caches reuse previously computed KV cache for matching prefixes, avoiding repeated computation (Kwon et al., 2023; Zheng et al., 2024; Jin et al., 2025). During decoding, prefix-aware attention mechanisms store a single copy of common-prefix KV across requests within a batch and reuse loaded KV across their attention computations, reducing both HBM capacity usage and memory traffic (Juravsky et al., 2024; Ye et al., 2024; Yi et al., 2026). However, KV Cache depends on preceding context because causal self-attention incorporates information from earlier tokens into token representations. Identical text chunks following different prefixes therefore

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

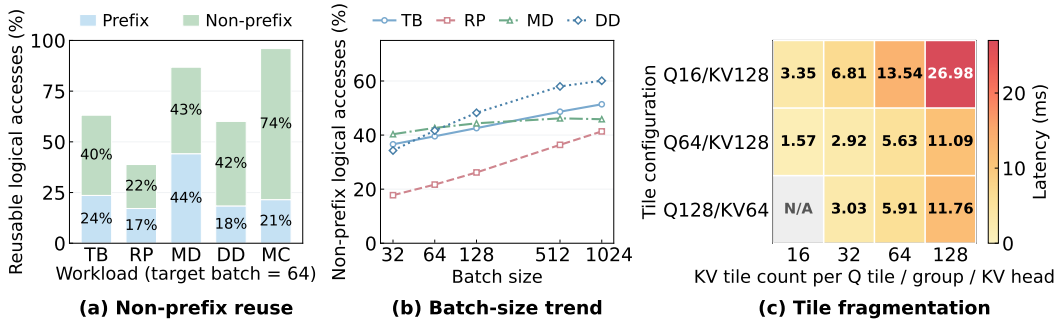

Figure 1: (a) Prefix and non-prefix logical KV access reuse at a target batch size of 64 (TB: ToolBench (Qin et al., 2024); RP: RAGPulse (Wang et al., 2025b); MD: MultiDoc2Dial (Feng et al., 2021); DD: Doc2Dial (Feng et al., 2020); MC: MCP-Bench (Wang et al., 2026a)). (b) Non-prefix logical KV access reuse versus batch size for TB, RP, MD, and DD. The vertical dotted line marks the target batch size in (a). (c) Partial-attention latency (ms) on an A100 GPU with query and KV lengths fixed at 8 and 2,000. KV tiles per query tile, group, and KV head vary across three tile configurations. Tile counts vary with valid KV data per tile; unused positions are masked. Latency excludes packing and result merging.

produce different KV Cache, preventing these prefix-based optimizations from directly exploiting non-prefix overlap in dynamically assembled prompts.

Position-independent caching (PIC) extends KV Cache reuse beyond common prefixes (Yao et al., 2025a; Hu et al., 2025; Agarwal et al., 2025; Yang et al., 2025c; Unnikrishnan, 2026; Ma et al., 2026; Liu et al., 2026a;b;c). The key insight is that updating the KV Cache for a small subset of tokens can recover important cross-chunk dependencies without recomputing the entire chunk. In a representative workflow, the KV cache of each reusable text chunk is independently computed offline and stored for reuse. During online prefill, the system retrieves the KV Cache of the chunks selected by a request, assembles them in prompt order, and adjusts positional encodings as needed. To account for dependencies on the new preceding context, it selectively recomputes the KV Cache for a subset of tokens while reusing the remaining entries. This enables chunk-level KV Cache reuse across different prefixes and chunk orderings with limited recomputation, reducing prefill computation and time to first token (TTFT).

Unfortunately, such prefill reuse does not automatically translate into shared storage and execution during decoding. Despite these limited updates, PIC runtimes that use request-private KV storage materialize a complete private KV Cache copy for each request after selective recomputation (Wang et al., 2025a), duplicating largely unchanged KV entries in HBM alongside request-specific repairs. At each decode step, attention processes these copies independently for each request, incurring redundant HBM reads and missing opportunities to batch query vectors for more efficient matrix computation over shared KV Cache. To quantify the opportunities for shared KV Cache storage and attention execution during decoding, we measure prefix and non-prefix reuse in terms of reusable logical KV accesses across workloads. Figure 1(a) reveals substantial reuse beyond common prefixes: at a target batch size of 64, non-prefix reuse accounts for approximately 22-74% of logical KV accesses across the five workloads. On ToolBench (Qin et al., 2024), non-prefix reuse reaches approximately 40%, compared with 24% for prefix reuse. Figure 1(b) further shows that non-prefix reuse generally increases with batch size, offering greater potential for shared KV storage and batched attention beyond common prefixes.

To exploit non-prefix KV Cache sharing enabled by PIC, we present Inlay, a PIC-native LLM serving runtime for efficient decoding. Its central idea is to coordinate KV Cache storage with attention execution to preserve and exploit KV sharing after selective recomputation. Inlay introduces two key designs. *Base-Repair-Decode storage* retains reusable KV Cache in shared storage while keeping repairs and generated tokens separate. *Cost-aware decode execution* chooses between shared Base reads and private KV packing for each chunk according to the active batch. Together, these designs reduce HBM capacity usage and memory traffic while improving computational efficiency and preserving request-specific repair semantics.

To reduce KV Cache duplication and allocation waste, Inlay separates each request's KV Cache into shared *Base KV*, sparse request- and layer-specific *Repair KV*, and private append-only *Decode KV* for newly generated tokens. Request-specific masks exclude Base entries replaced by Repair, so each logical position contributes once. Each request tracks the logical positions of its cached chunks and the Base entries replaced by its own Repair KV, allowing shared and private entries to be allocated independently of their positions in the prompt. Inlay allocates Repair KV in token slots across requests and layers according to their actual token counts. Multiple requests can use different slots in the same page, avoiding a separate page allocation for each sparse repair set.

To reduce redundant HBM reads and improve computational efficiency during decoding, Inlay groups compatible query vectors referencing the same Base KV, enabling shared data loads and batched attention while preserving request-specific positions and repair masks. Shared Base and private KV partitions produce partial outputs, which are merged using their softmax normalization factors. However, shared KV reads do not always reduce attention latency, because partitioning attention introduces additional execution overhead. Figure 1(c) illustrates how tile count and configuration affect partial-attention latency. Inlay therefore uses a CPU-side planner to balance shared-read savings against execution overhead under a KV memory budget. The planner can retain shared execution for some Base segments and pack others with private KV within the same batch. Valid plans, metadata, and packed prompt KV are reused across decode steps to reduce repeated preparation.

We evaluate Inlay's storage efficiency and attention latency on an NVIDIA A100 GPU using Qwen3-8B (Yang et al., 2025a) and Llama-3.1-8B (Grattafiori et al., 2024) configurations and workloads from ToolBench (Qin et al., 2024), RAGPulse (Wang et al., 2025b), and MultiDoc2Dial (Feng et al., 2021). We test four PIC methods: CacheBlend (Yao et al., 2025a), EPIC (Hu et al., 2025), QCFuse (Yan et al., 2026), and MEPIC (Wang et al., 2025a). Across tested workloads and attention geometries, Inlay achieves attention-kernel speedups of 1.24× over FlashInfer (Ye et al., 2025) and 1.32× over the prefix-aware attention baseline PAT (Yi et al., 2026), at matching batch sizes. Integrated with CacheBlend, EPIC, and QCFuse, Inlay achieves kernel speedups of 1.43–1.98× and resident KV memory reductions of 50.1–63.6% compared with their request-private FlashInfer baselines, preserving the same logical inputs and PIC repair results.

2 Related Work

KV Cache reuse and management. PagedAttention, SGLang, RAGCache, Preble, and PEEK support block-level KV Cache reuse across requests with common prefixes (Kwon et al., 2023; Zheng et al., 2024; Jin et al., 2025; Srivatsa et al., 2025; Xie et al., 2026). ContextPilot reorganizes inputs to enable prefix-based reuse of non-prefix overlap (Jiang et al., 2026). vAttention, MemServe, Mooncake, LMCache, Pensieve, CachedAttention, InferCept, and Continuum optimize KV cache allocation, placement, transfer, and retention (Prabhu et al., 2025; Hu et al., 2024; Qin et al., 2025; Liu et al., 2025; Yu et al., 2025; Gao et al., 2024; Abhyankar et al., 2024; Li et al., 2025). AsymCache coordinates cache residency with attention execution, while DiffKV manages irregular sparsity caused by cache compression (Shi et al., 2026; Zhang et al., 2025). Inlay supports non-prefix KV Cache reuse by retaining shared Base KV and packing sparse Repair KV into token slots, with memory allocated according to repair demand across requests and layers.

Position-independent reuse for prefill. Prompt Cache supports modular prompt reuse (Gim et al., 2024). CacheBlend, EPIC, Cache-Craft, KVShare, CacheClip, A^3, CacheTune, and SparseX adapt cached chunks to new contexts through repair selection and recomputation (Yao et al., 2025a; Hu et al., 2025; Agarwal et al., 2025; Yang et al., 2025c;b; Zhou et al., 2025; Li et al., 2026; Zhang et al., 2026). KVShare additionally updates repair selection during decoding. KVLink, C^2KV, and ReCache explore prompt or KV-state composition (Yang et al., 2025d; Du et al., 2026; Fang et al., 2026), while Irminsul, RedKnot, HYPIC, and LinearKV adapt reuse to different attention structures (Ma et al., 2026; Liu et al., 2026a;b;c). These reuse and repair mechanisms reduce repeated prefill work, but do not by themselves eliminate repeated KV reads across requests during decoding.

Shared-KV storage and decode execution. Hydragen, ChunkAttention, DeFT, CoDec, and PAT exploit shared prefixes or tree-structured KV through query aggregation, tree-aware scheduling, partial attention, or pack–forward–merge (Juravsky et al., 2024; Ye et al., 2024; Yao et al., 2025b; Wang et al., 2026b; Yi et al., 2026). MoSKA combines shared-context attention with sparse routing

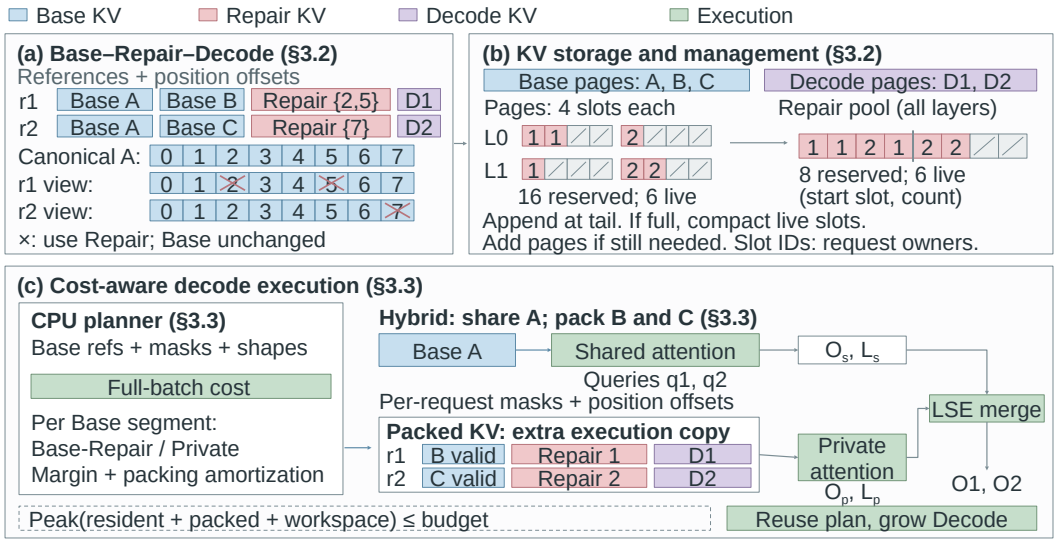

Figure 2: Inlay storage and execution overview. (a) Shared canonical Base KV with request-specific repairs replacing the crossed entries. (b) Compact Repair allocation reduces page waste across requests and layers. (c) The planner selects Base-Repair or Private execution per segment under a KV budget that includes additional packed copies. The hybrid example shares Base A, packs valid B/C entries with each request's Repair and Decode KV, and merges partial outputs using log-sum-exp (LSE) normalization.

and disaggregated execution (Rhee et al., 2025). CoDec also uses cost estimates for task partitioning and scheduling. For position-independent sharing, MEPIC recomputes each chunk's first block and shares the rest, while MiniPIC supports multiple PIC methods through user-controlled block-level reuse primitives (Wang et al., 2025a; Ordonez & Parnell, 2026). TokenDance stores a shared master with block-sparse differences applied during cache transfer (Bian et al., 2026). Inlay preserves the provider's token sequence and repair policy, storing request- and layer-specific repairs at token granularity without restricting them to fixed blocks. It executes decode attention over resident shared Base, sparse Repair, and private append-only Decode KV, reusing loaded Base KV across compatible queries within a batch while applying per-occurrence position offsets and repair masks. Its planner selects shared execution or private packing per chunk, balancing saved reads against execution and amortized preparation costs under a KV memory budget that includes packed copies.

3 The Design of Inlay

3.1 System Overview

Inlay is a runtime system that accelerates decoding attention by exploiting non-prefix KV Cache reuse enabled by position-independent caching (PIC). As shown in Figure 2, Inlay combines *Base-Repair-Decode storage* with *cost-aware decode execution*. The storage layout uses shared Base KV, sparse request- and layer-specific Repair KV, and private append-only Decode KV, with compact Repair slots within physical pages (Section 3.2). A CPU-side planner estimates data-access and task-execution costs to select an execution plan under a KV memory budget (Section 3.3).

We assume a prefill–decode (PD) disaggregated architecture, with Inlay deployed on the decode side and prefill computation unchanged. A unified decode-side interface receives reusable chunk KV, request- and layer-specific repair positions, and repaired KV from different PIC algorithms. Inlay stores chunk KV as Base and repaired entries as Repair, preserving each algorithm's repair decisions.

3.2 KV Storage and Memory Management

Storage layout and semantics. PIC recomputes the KV Cache for selected tokens within cached chunks while reusing the remaining entries. Copying an entire chunk into a private cache would

duplicate its unchanged KV. Inlay instead maintains immutable shared *Base KV* and stores replacements as request- and layer-specific *Repair KV*. Repair storage also holds private prompt KV for positions without a reusable Base. Newly generated KV is appended to private *Decode KV*.

Each Base reference records a canonical range, its logical position within the request, and the position offset used to interpret the cached keys. This mapping allows requests to share the same Base at different logical positions without copying it. Each Repair entry records its layer and logical token position, identifying the Base entry it replaces when one exists.

For request r and layer l, let $\mathcal{P}_{r,l}$ denote the prompt positions supplied by private KV. The effective KV at prompt position i is

$$\mathrm{KV}_{r,l}(i) = \begin{cases} \mathrm{Repair}_{r,l}(i), & i \in \mathcal{P}_{r,l}, \\ \mathrm{Base}_{r,l}(i), & i \notin \mathcal{P}_{r,l}, \end{cases} \tag{1}$$

where $\mathrm{Base}_{r,l}(i)$ denotes the shared entry selected by the request's position mapping. Request-specific masks exclude Base entries replaced by Repair, so each logical position contributes exactly once.

Page allocation and compact Repair slots. Like PagedAttention (Kwon et al., 2023), Inlay allocates physical KV pages on demand for Base, Repair, and Decode under a common KV budget. Base pages are shared through request references, while Decode pages are private and grow with each request. All layers share one page pool, avoiding uniform per-layer reservations when repair ratios differ. Repair slot counts match each request's repair volume at each layer, including Layer 0.

With B slots per page, a separate k-token Repair allocation reserves $B\lceil k/B \rceil$ slots. Repair allocations instead share pages across requests and layers and need not start at page boundaries. Each slot range maps to physical pages; request and layer metadata identify its logical positions.

Repair allocation and reclamation. New Repair allocations append to the tail of the slot space. When a request finishes, the runtime removes it from execution and waits for its last GPU use before reclaiming Repair slots, releasing Decode pages, and decrementing Base reference counts. If tail space is insufficient, the allocator compacts live Repair entries and acquires additional pages within the KV budget only if still needed. Compaction moves live entries only after affected GPU operations complete and updates their address mappings and execution plans before reuse. Referenced Base pages remain allocated; otherwise, the cache policy may retain or reclaim them.

3.3 Cost-aware Decode Execution

The CPU-side planner uses Base references, Repair masks, and attention shapes to select execution layouts. The GPU-side executor performs KV packing and attention according to the plan.

Execution over sparse KV overlays. The planner selects Base-Repair or Private execution per chunk, allowing both modes to coexist in a hybrid layout.

- ***Base-Repair execution:*** The planner groups Base references in a hash table using their physical page-ID sequences. Each request retains its own position offset and Repair mask. Queries from requests referencing the same Base share its KV reads, while each request also accesses its private Repair and Decode KV. The executor applies the visibility rule in Equation 1. Shared reads reduce KV I/O, but partitioning attention can leave tiles partly filled and produce partial outputs that require merging.
- ***Private and hybrid execution:*** In Private mode, each request's visible entries from the selected Base segments are packed with its Repair and Decode KV into a private execution buffer. It can reduce the number of attention tasks, at the cost of copying and repeated Base reads across requests. A hybrid layout keeps some chunks in Base-Repair mode and packs others for Private execution. Both modes apply the request's position adjustments.

Additional private buffers are allocated only for requests using Private or hybrid execution. Requests using only Base-Repair execution allocate no private copy. The resident representation is retained, and the planner includes actual packed capacity in its KV memory budget.

When a request's KV cache is divided into multiple partitions, each partition produces an attention output O_s and a log normalizer L_s. The outputs are merged using a softmax over the partition normalizers:

$$O = \sum_s w_s O_s, \qquad w_s = \mathrm{softmax}(\{L_t\})_s = \frac{\exp(L_s)}{\sum_t \exp(L_t)}. \tag{2}$$

Each visible KV position belongs to exactly one partition. The merged output is therefore equivalent to attention over the effective KV Cache, up to floating-point and positional-transformation error.

Layout cost estimation. The planner evaluates each candidate layout p for the full batch because changing one chunk's execution mode can affect task grouping, packing, and merging elsewhere in the plan. For a candidate layout p, let $\mathcal{K}(p)$ denote its attention kernel launches, and let $t = t(k)$ denote the compiled configuration of launch k. The estimated execution time of that launch is

$$\widehat{T}_{p,k} = \ell_t + \max\left(\frac{D_{p,k}}{B_t},\ Q_{p,k}\right), \tag{3}$$

where ℓ_t is the fixed launch overhead, $D_{p,k}$ is the estimated access volume, B_t is the calibrated effective access rate, and $Q_{p,k}$ is the estimated task completion time. The maximum approximates overlap between data access and task execution. The planner derives $D_{p,k}$ from task loads and stores, including repeated KV loads and partial-output writes. Shared KV is counted once per load, and masked positions are excluded only when their loads are skipped. The effective access rate B_t is calibrated using the same access-counting convention.

To estimate task completion time, let S be the number of available SMs and r_t the maximum resident CTAs per SM for configuration t. The latter is determined by shared-memory, thread, register, and hardware limits. Task j has estimated cost $s_j = u_t + v_t n_j$, where n_j is its number of executed KV-tile iterations, u_t is the fixed task cost, and v_t is the cost per iteration. The estimator preserves the task list within each CTA, assigns CTA work to at most Sr_t concurrent slots, and sets $Q_{p,k}$ to the latest predicted completion time. The parameters B_t, u_t, v_t, and ℓ_t are calibrated with microbenchmarks for the target configuration and concurrency conditions.

Execution mode selection. For sequential attention, merge, and update stages, the estimated per-step cost is

$$\widehat{C}_p = \sum_{k \in \mathcal{K}(p)} \widehat{T}_{p,k} + \widehat{T}_{\mathrm{merge},p} + \widehat{T}_{\mathrm{update},p}. \tag{4}$$

Merge and per-step data movement are estimated separately from their work sizes using calibrated costs; updates include incremental KV packing and separate query-gather and output-scatter operations. Operations fused into attention or merging are counted in those stages. Partial-output writes are charged to attention, and their reads are charged to merging.

For a candidate chunk, $\widehat{C}_{\mathrm{shared}}$ and $\widehat{C}_{\mathrm{private}}$ refer to full-batch plans that differ in the execution mode of that chunk. Let I_p be the new preparation cost that delays GPU execution, and let $H \geq 1$ be a conservative reuse horizon supplied by the runtime. The planner selects Private execution when

$$\widehat{C}_{\mathrm{private}} + \frac{I_{\mathrm{private}}}{H} < (1-\epsilon)\left(\widehat{C}_{\mathrm{shared}} + \frac{I_{\mathrm{shared}}}{H}\right), \tag{5}$$

where ϵ is a switching margin. Private execution is eligible only if resident KV pages, packed buffers, and peak temporary workspace fit the KV memory budget, including buffers that coexist during a switch. Otherwise, the chunk remains in Base-Repair mode. A chunk with multiple users can still use Private execution when lower task and merge costs outweigh the additional reads and amortized preparation cost.

Initial packing and plan preparation that delays GPU execution are one-time costs included in I_p, rather than the per-step estimate in Equation 4. Reusing valid packed prompt KV incurs no new prompt-packing cost. Appendix A gives the full cost model, amortization policy, and peak-memory checks.

Plan reuse and incremental updates. The host runtime caches the selected layout, address mappings, attention plans, and packed prompt KV. While the plan remains valid, the runtime appends new Decode KV and updates Decode lengths and visibility information. New requests, completed

requests, or changes to the page or Repair-slot layout can invalidate the plan, triggering replanning or reconstruction of the affected state. A background CPU thread searches for improved plans on a fixed copy of batch metadata, while the foreground thread dispatches GPU execution using a valid plan. If no compatible plan is available after a batch change, the foreground thread prepares a memory-feasible Base-Repair fallback before execution.

Before adopting a completed plan, the foreground thread checks that it still matches the current batch, configuration, and memory budget. Valid plans take effect at a decode-step boundary after any required GPU preparation; outdated plans are discarded. The foreground thread attaches the latest Decode metadata before GPU execution. Appendices C and D describe the complete plan-update and cache-management procedures.

4 Evaluation

4.1 Experimental setup

Models and testbed. We evaluate Inlay on NVIDIA A100-SXM4-40GB GPUs using BF16 KV, with Qwen3-8B and Llama-3.1-8B model and tokenizer configurations. We use PyTorch 2.7.0+cu126 (CUDA 12.6) and Triton 3.3.0. The host has two 48-core AMD EPYC 7642 sockets with 2 threads per core (96 physical cores / 192 logical CPUs) and approximately 1.0 TiB of memory. Workloads use ToolBench (Qin et al., 2024), RAGPulse (Wang et al., 2025b), and MultiDoc2Dial (Feng et al., 2021). Kernel tests use matched synthetic Q/K/V for a single attention layer, without a fixed sequence length. We test batch sizes 32, 64, and 128, query/KV heads 32/8, 32/32, 16/8, and 64/8, and head dimension 128. The 32/8 configuration is native to both models; the others vary attention geometry. PIC integration uses a single-layer Qwen3-8B configuration on ToolBench-derived workloads with 4,096 prompt tokens and 32 Decode tokens, reporting mean attention-kernel latency over repeated measurements.

Baselines. Kernel baselines are FlashInfer's paged decode attention (Ye et al., 2025) and PAT (Yi et al., 2026), which uses query packing and multi-tile execution for shared-prefix decoding. All comparisons use matched logical inputs and attention geometry on the same GPU. GPU attention latency includes attention, merging, and query-gather/output-scatter kernels.

For PIC integration, we compare CacheBlend (Yao et al., 2025a), EPIC (Hu et al., 2025), QCFuse (Yan et al., 2026), and MEPIC (Wang et al., 2025a) with and without Inlay. We use the authors' released code for CacheBlend, EPIC, and QCFuse (CacheBlend contributors, n.d.; EPIC contributors, n.d.; QCFuse contributors, n.d.), and our own reproduction of MEPIC. Each pair uses matched logical Q/K/V, repair results, sequence lengths, and attention geometry, preserving the provider's repair policy, including QCFuse's query-aware token selection. The CacheBlend, EPIC, and QCFuse baselines (*Without Ours*) materialize request-private KV and use FlashInfer. Our MEPIC baseline uses FlashInfer over shared canonical chunk KV blocks and recomputes each chunk's first block. With Inlay (*With Ours*), the resident cache uses Base–Repair–Decode storage and attention uses hybrid execution. Both variants run on the same GPU, with KV layouts and execution plans prepared before timing.

4.2 Kernel performance

Figure 3 shows median speedups of 1.32× over PAT and 1.24× over FlashInfer, computed as baseline latency divided by Inlay latency over matched configurations. For GQA, the largest gains appear at batch size 32, especially with 16/8 heads, and generally narrow at larger batch sizes. MHA retains substantial gains across the tested batch sizes, particularly on ToolBench and MultiDoc2Dial. The median speedups over PAT are 1.30×, 1.36×, and 1.27× on ToolBench, RAGPulse, and MultiDoc2Dial, respectively. Gains depend on attention geometry and shared Base volume: shared reads help when their savings outweigh query grouping and merging costs.

4.3 Integration with position-independent caching methods

Figure 4(a)–(d) evaluates Inlay with CacheBlend, EPIC, QCFuse, and MEPIC on a 64-request Qwen3-8B workload. CacheBlend and QCFuse vary the selection ratio from 5% to 50%, while EPIC

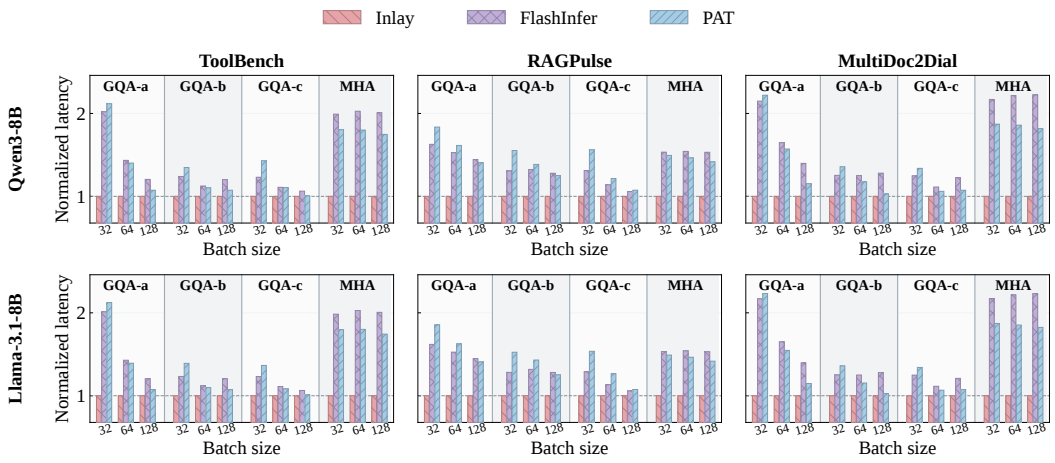

Figure 3: Normalized attention-kernel latency of Inlay, FlashInfer, and PAT using Qwen3-8B and Llama-3.1-8B configurations across three datasets. GQA-a/b/c denote 16/8, 32/8, and 64/8 query/KV heads, while MHA denotes 32/32. Results cover batch sizes 32, 64, and 128 and are normalized to Inlay (1.0); values above 1 favor Inlay.

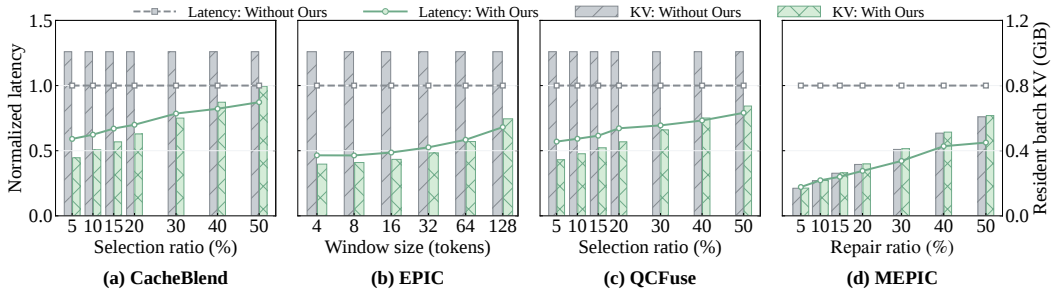

Figure 4: PIC integration with (a) CacheBlend, (b) EPIC, (c) QCFuse, and (d) MEPIC on a single-layer Qwen3-8B workload (64 requests, 4,096 prompt tokens, 32 Decode tokens). Lines show attention-kernel latency normalized per setting to *Without Ours* (left axis; baseline 1.0); bars show resident batch KV memory (GiB, right axis), excluding packed execution copies.

varies the window size from 4 to 128 tokens. For MEPIC, we fix the KV block size at 32 tokens and partition seven documents into 7–64 chunks. The first block of each chunk is recomputed and stored as request-private Repair KV. Increasing the chunk count raises the repair size from 224 to 2,048 tokens, corresponding to 5.47%–50% of the 4,096-token prompt.

At the smallest settings in panels (a)–(c), Inlay reduces kernel latency from approximately 1.335 to 0.788 ms for CacheBlend, from 1.336 to 0.621 ms for EPIC, and from 1.334 to 0.760 ms for QCFuse. The median speedups across the sweeps are 1.43×, 1.98×, and 1.49×, respectively. EPIC's lower latency at its smallest setting is consistent with its smaller Repair footprint. Larger selection ratios or windows increase private Repair KV and narrow the latency advantage, but all three remain faster than their private baselines at the largest settings.

The request-private baselines for CacheBlend, EPIC, and QCFuse each occupy 1.008 GiB throughout the sweep. With Inlay, resident KV memory is 0.356–0.795 GiB for CacheBlend, 0.318–0.595 GiB for EPIC, and 0.344–0.674 GiB for QCFuse, with median reductions of 50.1%, 63.6%, and 54.9%, respectively. Larger repair budgets reduce both memory and latency benefits.

MEPIC already shares canonical chunk KV blocks across requests (Wang et al., 2025a). Its repairs fill 32-token blocks, leaving little allocation waste; resident KV memory therefore remains nearly identical with and without Inlay (Figure 4(d)). More chunks increase private Repair storage and leave less Base KV to share. Storage sharing alone does not eliminate repeated KV reads across requests. By reusing loaded Base KV across compatible queries, Inlay keeps normalized attention latency

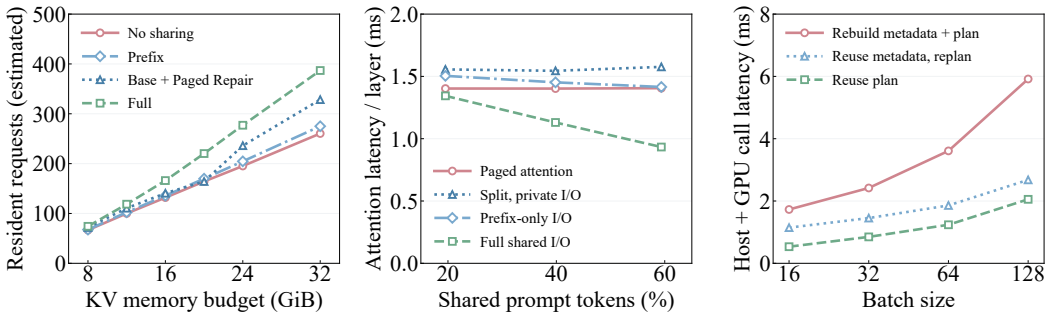

Figure 5: Inlay ablations. (a) Estimated resident-request capacity on ToolBench under fixed resident KV memory budgets for no sharing, prefix sharing, and shared Base with paged or compact token-granular Repair (Full). (b) Shared Base reads: single-layer GPU attention latency versus shared prompt fraction at batch size 64, comparing request-wise paged attention, private split attention, prefix-only sharing, and full sharing. (c) Plan reuse: synchronized host-plus-GPU call latency for rebuilding metadata and plans, reusing metadata only, or reusing both, with fixed KV topology and lengths but changing queries.

below 0.6 throughout the sweep, though the benefit narrows as the repair ratio increases.
All memory measurements cover resident KV and exclude the additional packed execution copies described in Section 3.3.

4.4 Ablation

KV capacity. Figure 5(a) compares four storage configurations on ToolBench under resident KV budgets of 8–32 GiB, with a controlled 5% Repair ratio, 32-token pages, and a 128-token Decode reservation. Capacity is estimated from the measured average per-request footprint after summing resident KV across all model layers. Budgets exclude packed execution copies and temporary workspace. At 32 GiB, shared Base with compact Repair has an estimated capacity of 386.8 resident requests, versus 260.4 for private KV, 275.0 for prefix sharing, and 328.4 for shared Base with paged Repair. These are increases of 48.5%, 40.7%, and 17.8%, respectively. The gain over private KV rises from 11.4% at 8 GiB to 48.5% at 32 GiB under the calibrated pool sizes.

Shared Base reads. Figure 5(b) uses synthetic Q/K/V with controlled prefix and non-prefix sharing, batch size 64, 32/8 query/KV heads, head dimension 128, 4096 prompt tokens, and 128 Decode tokens. The three split paths perform the same three attention calls and two merges. As the actual shared prompt fraction rises from 19.5% to 59.4%, sharing prefix and non-prefix Base reads reduces single-layer GPU attention latency from 1.344 to 0.933 ms. The reduction relative to split attention with private reads grows from 13.6% to 40.8%. At the highest sharing level, request-wise paged attention and prefix-only shared reads take 1.405 and 1.415 ms, respectively.

Plan reuse. Figure 5(c) uses the same shared-read path at a target shared prompt fraction of 40% and batch sizes 16–128. KV topology and lengths remain fixed while queries change, allowing metadata and plan reuse (Section 3.3). At batch size 128, synchronized host-plus-GPU call latency is 5.92 ms when rebuilding metadata and plans, 2.68 ms when reusing metadata while replanning, and 2.05 ms when reusing both. Joint reuse achieves 2.85–3.24× speedups, with absolute savings rising from 1.19 ms at batch size 16 to 3.86 ms at batch size 128.

5 Conclusion

We presented Inlay, a serving runtime for position-independent KV caching that combines compact Repair storage, shared KV reads, and cost-aware execution planning for each chunk. Experiments demonstrate reduced resident KV memory usage and attention latency while preserving the underlying PIC method's repair semantics.

AI DISCLOSURE

We used generative AI tools to support discussions of experimental design, cost-model formulation, and result interpretation, and to assist with code and figure preparation. We also used these tools for manuscript drafting, translation, and language editing. AI-generated suggestions informed the authors' analysis and revision process, while the authors retained responsibility for evaluating them against the implementation and experimental evidence. All final methodological choices, experimental decisions, and conclusions were determined by the authors. We take full responsibility for the accuracy and integrity of the manuscript, implementation, and reported results, including all content developed with AI assistance.

REFERENCES

Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. InferCept: Efficient intercept support for augmented large language model inference. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 81–95, 2024. URL https://arxiv.org/abs/2402.01869.

Shubham Agarwal, Sai Sundaresan, Subrata Mitra, Debabrata Mahapatra, Archit Gupta, Rounak Sharma, Nirmal Joshua Kapu, Tong Yu, and Shiv Saini. Cache-Craft: Managing chunk-caches for efficient retrieval-augmented generation. *Proceedings of the ACM on Management of Data*, 3(3): 1–28, 2025. doi: 10.1145/3725273. URL https://arxiv.org/abs/2502.15734.

Zhuohang Bian, Feiyang Wu, Chengrui Zhang, Hangcheng Dong, Yun Liang, and Youwei Zhuo. TokenDance: Scaling multi-agent LLM serving via collective KV cache sharing. *arXiv preprint arXiv:2604.03143*, 2026. URL https://arxiv.org/abs/2604.03143.

CacheBlend contributors. CacheBlend: Official source code. GitHub repository, n.d. URL https://github.com/YaoJiayi/CacheBlend. Accessed September 26, 2026.

Chuheng Du, Junyi Chen, Hanlin Tang, Kan Liu, Tao Lan, Lin Qu, Chaoyue Niu, Shengzhong Liu, Guihai Chen, and Fan Wu. C^2KV: Compressed and composable KV cache reuse for efficient LLM inference. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2026. doi: 10.1145/3770855.3817715. URL https://arxiv.org/abs/2607.17715.

EPIC contributors. EPIC: Official source code. GitHub repository, n.d. URL https://github.com/DerekHJH/epic. Accessed September 26, 2026.

Yichu Fang, Sitong Wei, Haozhe Hu, and Xiaoyu Shen. ReCache: Efficient KV cache reuse and compression for tool-augmented LLM agents. *arXiv preprint arXiv:2608.19662*, 2026. URL https://arxiv.org/abs/2608.19662.

Song Feng, Hui Wan, Chulaka Gunasekara, Siva Patel, Sachindra Joshi, and Luis Lastras. doc2dial: A goal-oriented document-grounded dialogue dataset. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pp. 8118–8128. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.emnlp-main.652. URL https://aclanthology.org/2020.emnlp-main.652/.

Song Feng, Siva Sankalp Patel, Hui Wan, and Sachindra Joshi. MultiDoc2Dial: Modeling dialogues grounded in multiple documents. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 6162–6176, 2021. doi: 10.18653/v1/2021.emnlp-main.498. URL https://aclanthology.org/2021.emnlp-main.498/.

Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-efficient large language model serving for multi-turn conversations with CachedAttention. In *2024 USENIX Annual Technical Conference*, 2024. URL https://arxiv.org/abs/2403.19708.

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt Cache: Modular attention reuse for low-latency inference. In *Proceedings of Machine Learning and Systems*, 2024. URL https://arxiv.org/abs/2311.04934.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. The Llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. MemServe: Context caching for disaggregated LLM serving with elastic memory pool. *arXiv preprint arXiv:2406.17565*, 2024. URL https://arxiv.org/abs/2406.17565.

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

Yinsicheng Jiang, Yeqi Huang, Liang Cheng, Cheng Deng, Xuan Sun, and Luo Mai. ContextPilot: Fast long-context inference via context reuse. In *Proceedings of Machine Learning and Systems*, volume 8, pp. 1639–1655, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/hash/b0131b6ee02a00b03fc3320176fec8f5-Abstract-Conference.html.

Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Shufan Liu, Xuanzhe Liu, and Xin Jin. RAGCache: Efficient knowledge caching for retrieval-augmented generation. *ACM Transactions on Computer Systems*, 44(1):2:1–2:27, 2025. doi: 10.1145/3768628. URL https://chaojin0310.github.io/files/papers/TOCS25-RAGCache.pdf.

Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-throughput LLM inference with shared prefixes. *arXiv preprint arXiv:2402.05099*, 2024. URL https://arxiv.org/abs/2402.05099.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023. doi: 10.1145/3600006.3613165. URL https://arxiv.org/abs/2309.06180.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, 2020. URL https://arxiv.org/abs/2005.11401.

Fei Li, Song Liu, Yan Liu, Jinhua Cui, Shiqiang Nie, Jinyu Wang, and Weiguo Wu. Adaptive KV cache reuse for fast long-context LLM serving. *arXiv preprint arXiv:2605.24022*, 2026. URL https://arxiv.org/abs/2605.24022.

Hanchen Li, Runyuan He, Qiuyang Mang, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Huanchen Zhang, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. Continuum: Efficient and robust multi-turn LLM agent scheduling with KV cache time-to-live. *arXiv preprint arXiv:2511.02230*, 2025. URL https://arxiv.org/abs/2511.02230.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2308.03688.

Yang Liu, Zhaokai Luo, Huayi Jin, Ruozhou He, Chenchen Hong, Zhiyong Wang, Boyu Wang, Guanjie Chen, Yifei Liu, Tao Xie, and Junhao Hu. RedKnot: Efficient long-context LLM serving with head-aware KV reuse and SegPagedAttention. *arXiv preprint arXiv:2606.06256,* 2026a. URL https://arxiv.org/abs/2606.06256.

Yifei Liu, Juntong Wu, Yang Liu, Junhao Hu, Minghao Li, Xiaoxu Chen, and Weihang Chen. HYPIC: Accelerating hybrid-attention LLM serving with position-independent caching. *arXiv preprint arXiv:2607.01299*, 2026b. URL https://arxiv.org/abs/2607.01299.

Yirui Liu, Ruoling Qi, Longwen Wang, Xuaner Wu, Jian Chen, Yuxin Jin, Jiawei Shao, and Xuelong Li. LinearKV: One cached state suffices for position-independent caching in hybrid LLMs. *arXiv preprint arXiv:2608.11231*, 2026c. URL https://arxiv.org/abs/2608.11231.

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

Bole Ma, Jan Eitzinger, and Harald Köstler. Irminsul: MLA-native position-independent caching for agentic LLM serving. *arXiv preprint arXiv:2605.05696*, 2026. URL https://arxiv.org/abs/2605.05696.

Microsoft. Microsoft 365 Copilot connectors overview, 2026. URL https://learn.microsoft.com/en-us/microsoft-365/copilot/extensibility/overview-copilot-connector. Accessed September 24, 2026.

Nathan Ordonez and Thomas Parnell. MiniPIC: Flexible position-independent caching in <100LOC. *arXiv preprint arXiv:2606.13126*, 2026. URL https://arxiv.org/abs/2606.13126.

Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. vAttention: Dynamic memory management for serving LLMs without PagedAttention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2025. URL https://arxiv.org/abs/2405.04437.

QCFuse contributors. QCFuse: Official source code. GitHub repository, n.d. URL https://github.com/uYanJX/QCFuse. Accessed September 26, 2026.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pp. 155–170. USENIX Association, 2025. URL https://www.usenix.org/conference/fast25/presentation/qin.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2307.16789.

Myunghyun Rhee, Sookyung Choi, Euiseok Kim, Joonseop Sim, Youngpyo Joo, and Hoshik Kim. MoSKA: Mixture of shared KV attention for efficient long-sequence LLM inference. *IEEE Computer Architecture Letters*, 2025. doi: 10.1109/LCA.2025.3627539. URL https://arxiv.org/abs/2511.06010.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://arxiv.org/abs/2302.04761.

Chunan Shi, Yilei Chen, Yilin Chen, Xupeng Miao, and Bin Cui. Multi-segment attention: Enabling efficient KV-cache management for faster large language model serving. *arXiv preprint arXiv:2606.02964*, 2026. URL https://arxiv.org/abs/2606.02964.

Snowflake. Cortex Agents, n.d. URL https://docs.snowflake.com/en/user-guide/snowflake-cortex/cortex-agents. Online documentation. Accessed September 24, 2026.

Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for LLM serving. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/5bc342f48de8264779952fac378f96dc-Abstract-Conference.html.

Srihari Unnikrishnan. KVBoost: Chunk-level key-value cache reuse with deviation-guided recomputation for efficient large language model inference. *arXiv preprint arXiv:2608.21362*, 2026. URL https://arxiv.org/abs/2608.21362.

Qian Wang, Zahra Yousefijamarani, Morgan Lindsay Heisler, Rongzhi Gu, Bai Xiaolong, Shan Yizhou, Wei Zhang, Wang Lan, Ying Xiong, Yong Zhang, and Zhenan Fan. MEPIC: Memory efficient position independent caching for LLM serving. *arXiv preprint arXiv:2512.16822*, 2025a. URL https://arxiv.org/abs/2512.16822.

Zhengchao Wang, Yitao Hu, Jianing Ye, Zhuxuan Chang, Jiazheng Yu, Youpeng Deng, and Keqiu Li. RAGPulse: An open-source RAG workload trace to optimize RAG serving systems, 2025b. URL https://arxiv.org/abs/2511.12979.

Zhenting Wang, Qi Chang, Hemani Patel, Shashank Biju, Cheng-En Wu, Quan Liu, Aolin Ding, Alireza Rezazadeh, Ankit Shah, Yujia Bao, and Eugene Siow. MCP-Bench: Benchmarking tool-using LLM agents with complex real-world tasks via MCP servers. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=fe8mzHwMxN.

Zhibin Wang, Rui Ning, Chao Fang, Zhonghui Zhang, Xi Lin, Shaobo Ma, Mo Zhou, Xue Li, Zhongfeng Wang, Chengying Huan, Rong Gu, Kun Yang, Guihai Chen, Sheng Zhong, and Chen Tian. CoDec: Prefix-shared decoding kernel for LLMs. *Proceedings of the ACM on Management of Data*, 4(3):151:1–151:27, 2026b. doi: 10.1145/3802028. URL https://arxiv.org/abs/2505.17694.

Bing Xie, Zhipeng Wang, Masahiro Tanaka, and Zhen Zheng. PEEK: Predictive queue-informed KV cache management for LLM serving. *arXiv preprint arXiv:2607.02525*, 2026. URL https://arxiv.org/abs/2607.02525. Preprint, under review.

Jianxin Yan, Wangze Ni, Zhenxin Li, Jiabao Jin, Zhitao Shen, Haoyang Li, Jia Zhu, Peng Cheng, Xuemin Lin, Lei Chen, and Kui Ren. QCFuse: Query-aware cache fusion via compressed view for efficient RAG serving. *arXiv preprint arXiv:2606.05875*, 2026. URL https://arxiv.org/abs/2606.05875.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025a. URL https://arxiv.org/abs/2505.09388.

Bin Yang, Qiuyu Leng, Jun Zeng, and Zhenhua Wu. CacheClip: Accelerating RAG with effective KV cache reuse. *arXiv preprint arXiv:2510.10129*, 2025b. URL https://arxiv.org/abs/2510.10129.

Huan Yang, Renji Zhang, Mingzhe Huang, Weijun Wang, Yin Tang, Yuanchun Li, Yunxin Liu, and Deyu Zhang. KVShare: An LLM service system with efficient and effective multi-tenant KV cache reuse. *arXiv preprint arXiv:2503.16525*, 2025c. URL https://arxiv.org/abs/2503.16525v2.

Jingbo Yang, Bairu Hou, Wei Wei, Yujia Bao, and Shiyu Chang. KVLink: Accelerating large language models via efficient KV cache reuse. In *Advances in Neural Information Processing Systems*, volume 38, 2025d. URL https://arxiv.org/abs/2502.16002.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 94–109, 2025a. doi: 10.1145/3689031.3696098. URL https://arxiv.org/abs/2405.16444.

Jinwei Yao, Kaiqi Chen, Kexun Zhang, Jiaxuan You, Binhang Yuan, Zeke Wang, and Tao Lin. DeFT: Decoding with flash tree-attention for efficient tree-structured LLM inference. In *International Conference on Learning Representations*, 2025b. URL https://arxiv.org/abs/2404.00242.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024. URL https://arxiv.org/abs/2406.12045.

Lu Ye, Ze Tao, Yong Huang, and Yang Li. ChunkAttention: Efficient self-attention with prefix-aware KV cache and two-phase partition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11608–11620, 2024. doi: 10.18653/v1/2024.acl-long.623. URL https://arxiv.org/abs/2402.15220.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving. In *Proceedings of Machine Learning and Systems*, volume 7, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/hash/dbf02b21d77409a2db30e56866a8ab3a-Abstract-Conference.html.

Jinjun Yi, Zhixin Zhao, Yitao Hu, Ke Yan, Weiwei Sun, Hao Wang, Laiping Zhao, Yuhao Zhang, Wenxin Li, and Keqiu Li. PAT: Accelerating LLM decoding via prefix-aware attention with resource efficient multi-tile kernel. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2026. doi: 10.1145/3779212.3790200. URL https://arxiv.org/abs/2511.22333.

Lingfan Yu, Jinkun Lin, and Jinyang Li. Stateful large language model serving with Pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 144–158, 2025. doi: 10.1145/3689031.3696086. URL https://arxiv.org/abs/2312.05516.

Quqing Zhang, Kai Chen, Ning Liao, Zehao Lin, Bo Tang, Feiyu Xiong, Zhiyu Li, and Xiaoxing Wang. SparseX: Efficient segment-level KV cache sharing for interleaved LLM serving. *arXiv preprint arXiv:2606.01751*, 2026. URL https://arxiv.org/abs/2606.01751.

Yanqi Zhang, Yuwei Hu, Runyuan Zhao, John C. S. Lui, and Haibo Chen. DiffKV: Differentiated memory management for large language models with parallel KV compaction. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pp. 431–445, 2025. doi: 10.1145/3731569.3764810. URL https://doi.org/10.1145/3731569.3764810.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://arxiv.org/abs/2312.07104.

Yuechi Zhou, Yi Su, Jianxin Zhang, Juntao Li, Qingrong Xia, Zhefeng Wang, Xinyu Duan, and Baoxing Huai. A^3: Attention-aware accurate KV cache fusion for fast large language model serving. *arXiv preprint arXiv:2511.17560*, 2025. URL https://arxiv.org/abs/2511.17560.

APPENDIX

A EXECUTION COST MODEL

We estimate the cost of each candidate layout for the full active batch at one layer. All candidates use the same effective KV. Each candidate defines the shared and private partitions, attention descriptors, and additional operations needed for execution. Each chunk uses either Base-Repair or Private execution. Changing one chunk's execution mode can affect packing, query grouping, and merging across the batch. The model therefore evaluates the complete layout instead of adding independent savings for each chunk.

The model separates per-step execution from one-time preparation. It uses access counts, task descriptors, resource limits, and timing parameters to estimate execution time. GPU scheduling is approximated. The PIC provider's reuse and Repair decisions remain fixed when comparing layouts.

A.1 NOTATION AND WORK DESCRIPTORS

Let p denote a candidate and $\mathcal{K}(p)$ its attention launches. Launch k uses compiled configuration $t = t(k)$, which specifies the kernel variant, tile shape, head mapping, head dimension, and data types. Timing parameters depend on the GPU, backend version, configuration, and workload range used for calibration. Access volume is measured in bytes, and all times use the same unit, such as microseconds.

A logical task computes attention for query rows over a KV range. Each KV-tile iteration processes one tile in that range. The queue model maps these tasks to CTA work according to the selected backend. A runtime metadata object named CTA is not assumed to represent one physical GPU thread block; the resource model uses the backend's execution mapping.

Table 1: Main quantities in the execution cost model.

Quantity	Meaning
$D_{p,k}, B_t$	Access volume and effective access rate
n_j, s_j	KV iterations and estimated duration of logical task j
S, r_t, C_t	SM count, resident CTAs per SM, and execution-slot count
$\mathcal{J}_b, a_b$	Logical tasks assigned to CTA b and their total duration
$Q_{p,k}$	Predicted completion time of the launch's CTA work
ℓ_t, u_t, v_t	Launch cost, fixed task cost, and cost per iteration
$\widehat{C}_p$	Predicted steady-state time per decode step
I_p, H	Additional exposed preparation cost and assumed reuse steps
M_{budget}	Memory available to the cache and execution buffers

A.2 ACCESS VOLUME AND EXECUTED WORK

Let $\mathcal{A}_{p,k}$ contain the loads and stores specified by a launch's descriptors, and let b_a be the bytes accessed by event a. Then

$$D_{p,k} = \sum_{a \in \mathcal{A}_{p,k}} b_a. \tag{6}$$

Each load or store contributes to the count. If several query rows reuse one KV-tile load, that load is counted once. If another task loads the same tile, it is counted again. Masked positions are excluded only when the kernel skips their loads.

For equal K and V head dimensions d and e bytes per stored element, let R_j count token–KV-head pairs loaded by task j, including repeats. If K and V load the same pairs,

$$D^{\text{KV}}_{p,k} = 2ed \sum_j R_j, \tag{7}$$

$$D_{p,k} = D^{\text{KV}}_{p,k} + D^{\text{Q}}_{p,k} + D^{\text{meta}}_{p,k} + D^{\text{out}}_{p,k}. \tag{8}$$

K and V are counted separately when their storage or access patterns differ. Writing a partial-output row containing d elements and one log normalizer costs $db_O + b_L$ bytes, where b_O and b_L are their

storage sizes. The write is counted in attention, and the later read is counted in merging. Cache hits and memory-transaction overhead can make this byte count differ from measured HBM traffic. The effective rate B_t must therefore use the same counting rule as $D_{p,k}$.

For a continuous traversal of L_j tokens with KV tile length $B_{K,t}$,

$$n_j = \left\lceil \frac{L_j}{B_{K,t}} \right\rceil. \tag{9}$$

If each separate range starts a new tile loop, its iteration count is rounded separately before summing. If the ranges are packed into one sequence, the combined length is rounded once. The descriptor determines which case applies. Partly filled query or KV tiles still incur the instructions executed by the kernel, so their duration cannot be inferred from valid-token count alone.

A.3 Concurrency and Task Duration

The compiled configuration limits the number of resident CTAs per SM:

$$r_t = \min\{r_{\text{smem}}, r_{\text{threads}}, r_{\text{registers}}, r_{\text{hardware}}\}, \tag{10}$$

$$C_t = S r_t. \tag{11}$$

For example, the shared-memory and thread limits are

$$r_{\text{smem}} = \left\lfloor \frac{M_{\text{SM}}}{M_{\text{CTA},t}} \right\rfloor, \qquad r_{\text{threads}} = \left\lfloor \frac{T_{\text{SM}}}{T_{\text{CTA},t}} \right\rfloor.$$

These limits use the register and shared-memory allocation granularity of the compiled kernel and hardware. They determine concurrency and do not add separate latency penalties. Only configurations with $r_t \geq 1$ are eligible.

For a fixed configuration, the estimated duration of task j is

$$s_j = u_t + v_t n_j. \tag{12}$$

The coefficients represent task costs at the relevant level of concurrency. CTAs resident on the same SM share its resources. If CTA b executes the ordered task list $\mathcal{J}_b$, its estimated duration is

$$a_b = \sum_{j \in \mathcal{J}_b} s_j = |\mathcal{J}_b| u_t + v_t \sum_{j \in \mathcal{J}_b} n_j. \tag{13}$$

This expression also covers persistent CTAs that execute several tasks. The estimator preserves each CTA's assigned task list and order.

A.4 Queue Completion Time

Consider CTAs $b_1, \ldots, b_N$ in the order listed by the candidate. Initialize C_t execution slots with $\tau_c^{(0)} = 0$. For CTA b_i, assign its work to the earliest available slot:

$$c_i = \operatorname*{arg\,min}_{1 \leq c \leq C_t} \tau_c^{(i-1)}, \tag{14}$$

$$\tau_c^{(i)} = \begin{cases} \tau_c^{(i-1)} + a_{b_i}, & c = c_i, \\ \tau_c^{(i-1)}, & c \neq c_i. \end{cases} \tag{15}$$

Ties can be resolved by slot index. All CTA work finishes at

$$Q_{p,k} = \max_{1 \leq c \leq C_t} \tau_c^{(N)}. \tag{16}$$

A min-heap computes this recurrence in $O(N \log C_t)$ time after initialization. This is a scheduling approximation; the GPU may dispatch CTAs in a different order.

For N equal-duration CTAs with duration a, the estimate reduces to

$$Q_{p,k} = \left\lceil \frac{N}{C_t} \right\rceil a. \tag{17}$$

For unequal durations,

$$Q_{p,k} \geq \max\left\{\max_b a_b, \frac{\sum_b a_b}{C_t}\right\}. \tag{18}$$

The first term accounts for the longest CTA, whose work cannot be split by the estimator. The second is total work divided by slot count. If a persistent launch has $N \leq C_t$, all CTA lists fit in the modeled capacity and $Q_{p,k} = \max_b a_b$. With no work, $Q_{p,k} = 0$. A launch that is skipped has no launch cost.

Illustrative calculation. With two slots and three CTA durations of 5, 3, and 4 μs, the first two CTAs finish at times 5 and 3. The third is assigned to the second slot and finishes at time 7, giving $Q_{p,k} = 7\,\mu$s. Dividing the total work by two gives 6 μs and misses the imbalance. These values illustrate the recurrence and are not measured GPU parameters.

A.5 Launch and Per-Step Costs

For each attention launch, the model uses

$$\widehat{T}_{p,k} = \ell_t + \max\left\{\frac{D_{p,k}}{B_t}, Q_{p,k}\right\}. \tag{19}$$

Both terms inside the maximum are times. Taking their maximum approximates the overlap between data access and task execution. The effective rate B_t uses the model's access-counting rule, rather than the GPU's peak HBM bandwidth.

Let $\mathcal{U}(p)$ contain the additional stages executed each step: query gathering and position adjustment, partial-output merging, output scattering, and incremental KV updates or packing. Each stage is counted once. If stage a has $N_{p,a}$ equal work units, C_a modeled concurrent slots, and duration s_a per unit, its cost can be estimated as

$$\widehat{T}_{p,a} = \ell_a + \max\left\{\frac{D_{p,a}}{B_a}, \left\lceil\frac{N_{p,a}}{C_a}\right\rceil s_a\right\}. \tag{20}$$

Each stage defines its own work unit and duration. Equation 20 applies when those units have equal cost. Stages with unequal work can use the queue recurrence or a calibrated timing table indexed by their descriptors.

For sequential stages on the execution stream,

$$\widehat{C}_p = \sum_{k\in\mathcal{K}(p)} \widehat{T}_{p,k} + \sum_{a\in\mathcal{U}(p)} \widehat{T}_{p,a}. \tag{21}$$

Merge stages contribute to $\widehat{T}_{\mathrm{merge},p}$ in Equation 4; the other separate stages contribute to $\widehat{T}_{\mathrm{update},p}$. An operation fused into attention or merging is included in that stage and counted once. Preparing a cached prompt copy is a one-time cost, while copying new Decode KV for a packed private path is a per-step update. Requests using only Base-Repair execution append Decode KV without creating a packed execution copy. Repacking the prompt after a layout change adds a new preparation cost. For concurrent streams, execution time must follow the stage dependencies rather than a sequential sum.

A.6 Memory Feasibility and Amortized Selection

Let $\mathcal{P}$ be the generated candidate set. A feasible candidate must support the current attention geometry, masks, and position encoding. To account for memory throughout execution, define

$$M_p(x) = M_{\mathrm{resident}}(x) + M_{\mathrm{packed},p}(x) + M_{\mathrm{aux},p}(x), \tag{22}$$

$$M_p^{\mathrm{peak}} = \max_{x\in\mathcal{X}_p} M_p(x), \tag{23}$$

$$\mathcal{F} = \{p \in \mathcal{P} :\ p \text{ is compatible},\ M_p^{\mathrm{peak}} \leq M_{\mathrm{budget}}\}. \tag{24}$$

Here $\mathcal{X}_p$ includes preparation, any required compaction, plan switching, and execution. Resident memory counts allocated Base, Repair, and Decode pages, including unused capacity in retained pages. Packed memory counts actual buffer capacity only for requests using Private or hybrid execution; it is zero for requests using only Base-Repair execution. The resident representation

is retained for both execution modes. Auxiliary memory includes metadata, partial outputs, and workspace, including any temporary storage required for compaction. Each physical allocation is counted once. Old and new buffers are both counted while they coexist. The budget is the memory available after other model and runtime allocations.

The lowest predicted steady-state candidate is

$$p_{\text{steady}} = \arg\min_{p \in \mathcal{F}} \widehat{C}_p. \tag{25}$$

This selects the best evaluated candidate and does not guarantee a global optimum. Let I_p be the additional preparation time that cannot overlap with execution when switching to p. Let $H \geq 1$ be the assumed integer number of uses. Including preparation gives

$$\widehat{J}_p(H) = I_p + \sum_{h=1}^{H} \widehat{C}_p(h), \tag{26}$$

$$\overline{C}_p(H) = \frac{\widehat{J}_p(H)}{H} \simeq \widehat{C}_p + \frac{I_p}{H}. \tag{27}$$

Among feasible candidates, the planner compares $\overline{C}_p(H)$ using the same horizon H and then applies the switching rule below. Preparation includes GPU packing, position adjustment, foreground setup, and any candidate-dependent CPU work that delays execution. If activating a candidate requires compaction, its exposed cost also contributes to I_p. Search time already spent does not change the choice among returned candidates, but it still contributes to measured runtime overhead. Background work that overlaps execution is not counted again as a GPU stall.

For a currently valid reference path p_0, an amortized switching rule with margin $0 \leq \epsilon < 1$ can be written as

$$\overline{C}_p(H) < (1 - \epsilon)\overline{C}_{p_0}(H). \tag{28}$$

If an arrival invalidates the old plan, the reference must be a feasible path for the updated batch. With $I_{p_0} = 0$, approximately constant saving $\Delta T = \widehat{C}_{p_0} - \widehat{C}_p > 0$, and no margin, the break-even reuse count is

$$H_{\text{BE}} = \max\left\{1, \left\lceil \frac{I_p}{\Delta T} \right\rceil \right\}. \tag{29}$$

This is the first integer horizon with nonnegative net savings. The strict switching test in Equation 28 requires positive savings when $\epsilon = 0$, so equality at break-even does not trigger a switch. If Decode growth changes the saving, the corresponding break-even condition is

$$\sum_{h=1}^{H} \big(\widehat{C}_{p_0}(h) - \widehat{C}_p(h)\big) \geq I_p. \tag{30}$$

The runtime supplies a conservative integer horizon $H \geq 1$. It caps the configured horizon by any known bound on remaining compatible uses; if no reuse allowance is available, it uses $H = 1$. The same horizon is used for all candidates in one comparison and is reconsidered when batch state or memory constraints change. This horizon is a planning assumption, not a prediction of how many tokens a request will still generate. A packed buffer can be reused as long as it remains valid. Rebuilding the batch plan therefore does not always require repacking unchanged prompt KV.

A.7 Parameter Calibration and Reproducibility

For each compiled configuration, calibration records access counts, iteration counts, task lists, resource limits, and measured GPU time. Samples should cover limited parallelism, multiple execution waves, different task lengths, and sustained throughput. Additional stages are profiled over their own work sizes. Measurements must use the same timing scope as the model's launch and preparation terms.

For samples indexed by m, with measured launch time $y_m > 0$, one fitting procedure is

$$\min_{\substack{\ell_t, u_t, v_t \geq 0 \\ B_t > 0}} \sum_m \left[\frac{\ell_t + \max\{D_m/B_t, Q_m(u_t, v_t)\} - y_m}{y_m} \right]^2. \tag{31}$$

The recurrence recomputes Q_m for each parameter set. Samples dominated by data access cannot determine task costs, while samples dominated by task execution cannot determine the access rate. For equal-length CTAs, each executing one task with n iterations, task-dominated samples over w waves approximately satisfy

$$y \simeq \ell_t + wu_t + wv_t n. \tag{32}$$

Varying w and n helps separate launch, per-task, and per-iteration costs. CTAs with multiple tasks instead use Equation 13. Parameters that remain undetermined need independent measurements or more calibration samples. Validation uses separate samples to check timing error and whether the model ranks candidates correctly.

B CPU Planning and Prediction Validation

Planning on batch changes. Inlay reuses an execution plan across attention calls while the batch structure and execution requirements remain compatible with the plan. For an ongoing request, Base references and logical Repair positions remain fixed during decoding; each step appends Decode KV and updates the associated lengths and visibility metadata. Within the plan's supported capacity and configuration, these updates preserve the selected shared/private layout without repeating candidate search. New request arrivals trigger replanning when admission changes the batch's sharing groups or execution layout. Request retirement, page-layout changes, Repair-slot relocation, or budget changes require checking and updating affected state. Repair compaction changes physical addresses without changing logical Repair positions; descriptors that reference moved entries must be rebuilt before reuse. Full layout search is triggered when the selected layout is no longer valid or needs reconsideration; ordinary attention calls reuse the cached plan.

CPU search. The planner groups Base references by their physical page-ID sequences using a hash table, retaining each request's position offsets and Repair masks. It passes an immutable metadata snapshot to a C++ search routine that evaluates shared, private, and hybrid layouts with the calibrated cost model. Six CPU workers explore different attention schedules, starting from shared, private, and owner-count-based assignments and refining them with bounded greedy subset moves. Each move is evaluated for the full batch because changing a group assignment can affect packing, task grouping, and output merging elsewhere in the plan. Cached group summaries and candidate descriptors reduce repeated work. The search returns candidate plans for final memory-budget and preparation-amortization checks.

Background planning and publication. CPU search uses the metadata snapshot and the calibrated model, so it can run in the background while the GPU executes attention with a valid foreground plan. The search constructs candidates without modifying metadata or buffers currently in use. A candidate takes effect at a decode-step boundary only if it matches the current batch, configuration, page and slot mappings, and memory budget, and any required GPU-side preparation is complete. The foreground thread discards stale candidates and attaches the latest Decode metadata before execution. Initial execution requires a valid plan, and GPU-side preparation for a new layout is scheduled separately from CPU search.

Evaluation setup. Figure 6 evaluates prediction accuracy and the GPU latency of the proposed layouts. We use synthetic BF16 Q/K/V tensors on controlled ToolBench, RAGPulse, and Multi-Doc2Dial layouts, with respective prefix/non-prefix duplication fractions of 27.0/43.0%, 17.1/28.7%, and 36.1/42.7%; RAGPulse uses the archived layout surrogate. For each workload, we combine batch sizes 32/128, query/KV-head pairs 16/8, 32/8, 64/8, and 32/32, and Decode lengths 96/160, giving 16 configurations per workload and 48 in total. We run each configuration ten times, yielding 480 evaluation runs in total and 160 per workload. All configurations use 4,096 prompt tokens, head dimension 128, and 32-token pages. We measure candidate plans on an A100-SXM4-40GB, with the model parameters fixed before evaluation. Within each evaluation run, each candidate latency is the median of three technical trial means, each computed from three calls. GPU timing includes per-step packing updates, attention, gather/scatter, and merging; it excludes CPU search, CPU launch gaps, and initial preparation.

Prediction accuracy. Figure 6(a) compares the predicted and measured GPU times of each candidate. Most points lie close to the equality line across the three datasets and the measured latency

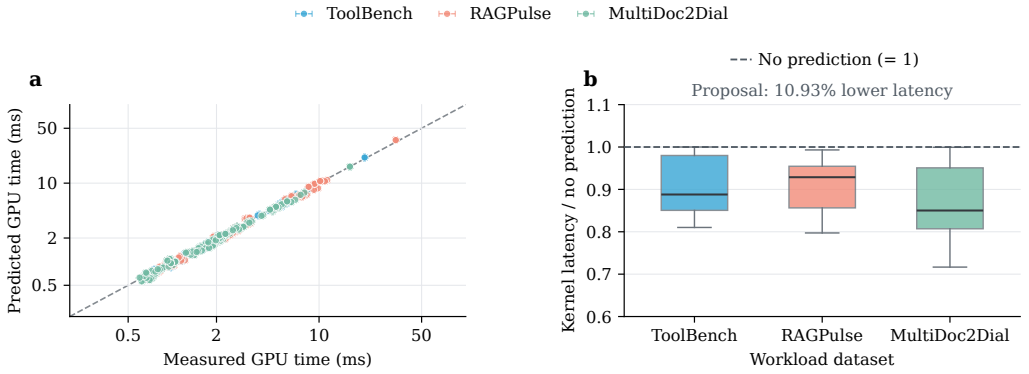

Figure 6: Prediction accuracy and GPU latency of model-guided proposals. (a) Predicted versus measured GPU latency of the evaluated candidates. Horizontal error bars show the sample standard deviation of three technical trial means; the diagonal denotes equality. (b) Proposal latency normalized to a fixed owner-count heuristic without cost prediction (dashed line at one). Each box summarizes 160 paired runs for one workload, showing the median, interquartile range, and whiskers extending to the most extreme observations within 1.5 times the interquartile range of the quartiles. The annotation reports the geometric-mean latency reduction over all 480 paired runs. Proposals are evaluated before preparation-amortization checks. CPU planning and initial preparation are excluded.

range. Across the evaluated candidates, the median absolute percentage error is 4.86% and the mean absolute percentage error is 5.73%. These results support using the calibrated model to estimate candidate execution costs.

Benefit of model-guided planning. Figure 6(b) compares the proposal with the lowest predicted cost against a fixed policy that makes chunks with fewer than two owners private, keeps the remaining chunks in Base-Repair mode, and uses a fixed attention schedule. Both use the same kernel implementations. The planner jointly selects the layout and attention schedule using the cost model, and its proposal is chosen before GPU timing. Across all 480 paired runs, the geometric mean of the proposal-to-baseline latency ratio is approximately 0.890, corresponding to an 11.0% reduction. The geometric-mean reductions for ToolBench, RAGPulse, and MultiDoc2Dial are 9.53%, 9.18%, and 14.19%, respectively. These measurements quantify the per-step GPU benefit of selecting layouts and attention schedules with the cost model.

C Plan Updates When New Requests Arrive

Adding a request changes the batch's sharing topology and execution work. Here we consider a request admitted to decoding after the PIC provider has supplied its Base references, request- and layer-specific Repair KV, and position metadata. A request may arrive before it is admitted to the active batch.

Updating the planning state. The runtime registers the new request's Base references, logical positions, offsets, Repair masks, and Decode state. References to the same physical Base pages can join an existing group; other references form new groups. Each owner retains its own visibility mask and position offset. The added request can change sharing opportunities, query grouping, and packing costs. Candidate costs are therefore evaluated for the updated batch, even when some topology metadata can be reused.

Reusing metadata, plans, and buffers. Topology metadata, backend attention plans, and packed prompt buffers have different reuse conditions. If the Base/Repair topology is unchanged and Decode grows within existing pages, a shared execution group can update its visibility metadata. A private path may need an updated backend plan when query or KV lengths change, even if its packed prompt data is still valid. Changes to batch membership, page mappings, or Repair-slot addresses require

checking the affected descriptors. Repair compaction alone does not change effective prompt KV, so a valid packed copy need not be rebuilt solely because its source entries moved; descriptors that read those entries must be updated. A cached layout can remain correct while another layout becomes faster.

Foreground execution and background search. The foreground updates a compatible plan incrementally while Decode grows. When the batch changes, it first establishes a valid execution path for the updated batch. If no reusable plan is compatible, it prepares a Base-Repair fallback without creating packed prompt copies. The fallback must satisfy the current masks, position conventions, and memory budget, including its workspace. A new request is admitted only after this path is ready; if it cannot fit, admission waits while the existing batch continues on its valid plan.

Candidate search can then run on a background CPU thread using a fixed metadata snapshot. Before using a completed candidate, the foreground checks the batch generation, request IDs, configuration, page and Repair-slot mappings, and current memory budget. It discards stale candidates and attaches the latest Decode metadata to a valid one. A candidate is adopted only after the amortized-cost check and any required GPU preparation have completed. The switch occurs at a decode-step boundary.

Preparing and activating an execution path. The runtime separates CPU topology construction, execution-path selection, and GPU buffer preparation. C++ search workers use CPU snapshots and do not modify CUDA state. The foreground checks whether to adopt the plan and whether memory is available before preparing GPU buffers. Packing, required key-position adjustment, and attention are submitted in order on the same CUDA stream. If Repair compaction is needed, affected GPU operations finish before entries move, and updated address mappings are published before reuse. The background thread produces a CPU plan; GPU preparation remains a separate step.

Preparation cost after an arrival. An arrival may add planning, metadata construction, compaction, or repacking time to the execution path. Valid cached prompt buffers can be reused, while a changed layout or insufficient capacity may require new buffers and repacking. The amortized comparison in Section A.6 depends on how long the resulting plans and buffers remain usable. Any exposed delay from memory management, waiting for a plan, or using a fallback belongs in the runtime measurement; timing a prepared kernel alone does not capture it.

D Cache Management When New Requests Arrive

Registering shared Base. The PIC provider determines which cached chunks a request can reuse and which positions need private KV. The runtime records references to resident canonical Base ranges, together with their logical positions and offsets. A new request can reference an existing Base without copying its unchanged KV. If a required Base is not resident, its data must be loaded before attention reads it. Each occurrence retains its own metadata, including when one request uses the same Base more than once.

Shared Base remains immutable. Request-specific replacements are stored in Repair, and the request's mask excludes the Base entries they replace. Repair also stores private prompt positions with no reusable Base. A new request therefore does not change the KV seen by existing requests.

Page allocation and Repair slots. Base, Repair, and Decode acquire physical pages on demand under the same KV memory budget. Base pages can be shared by request references, while Decode pages are private. Repair uses a common page-backed slot space across requests and layers, including Layer 0, and allocates slots according to each request's repair volume at each layer.

One Repair slot holds a token's K/V for one layer, including all KV heads and their dimensions. K and V use separate arrays. For h_{KV} KV heads, equal K/V head dimension d, and e bytes per element, each slot stores

$$b_{\mathrm{slot}} = 2h_{\mathrm{KV}}de. \tag{33}$$

Each allocation receives a contiguous interval in the logical slot space, with its start, length, and mapping to physical pages recorded in metadata. Allocations need not start at page boundaries, and different requests and layers can share a physical Repair page. Logical token positions are stored separately, so sparse logical positions do not require gaps in the slot space.

Let $k_{r,l}$ be the number of Repair tokens allocated for request r at layer l, and let B be the page size in tokens. Separate paged allocations reserve

$$M_{\text{paged}} = b_{\text{slot}} \sum_{r,l} B \left\lceil \frac{k_{r,l}}{B} \right\rceil, \tag{34}$$

while the compact layout's live data occupies

$$M_{\text{live}} = b_{\text{slot}} \sum_{r,l} k_{r,l}. \tag{35}$$

These expressions assume a uniform slot size. Compact Repair still acquires whole physical pages. With N_{Repair} allocated Repair pages,

$$M_{\text{Repair,pages}} = b_{\text{slot}} B N_{\text{Repair}} \geq b_{\text{slot}} B \left\lceil \frac{\sum_{r,l} k_{r,l}}{B} \right\rceil. \tag{36}$$

The right-hand side is a compacted lower bound, not the actual pool reservation. Allocated pages, allocator metadata, and temporary buffers enter the peak-memory check in Section A.6. If a request-layer pair uses several separate paged allocations, each is rounded separately in Equation 34.

Tail allocation and compaction. New Repair allocations append at the tail of the allocated slot space. Released slots can leave holes before the tail. If the remaining tail space is insufficient, the allocator first compacts live Repair entries to consolidate the released slots. It acquires additional pages within the KV budget only if the resulting space is still insufficient. Compaction moves entries after their affected GPU operations finish and updates slot-to-page mappings and execution descriptors before reuse. Logical positions, Repair masks, and effective KV remain unchanged. Live Repair entries are preserved; compaction does not evict data still required by an active request.

Decode growth and packed execution copies. Each request has private, append-only Decode KV backed by pages. Additional packed execution buffers are allocated only for requests using Private or hybrid execution. In a hybrid layout, selected Base segments use Private execution while the remaining segments use Base-Repair execution; each visible position contributes to exactly one attention partition. Requests using only Base-Repair execution allocate no private execution copy. The resident Base-Repair-Decode representation is retained in all cases.

A packed prompt can be reused while its content, position convention, layout, and request mapping remain valid. When buffer capacity is sufficient and Decode has been appended continuously, the private path copies only the newest Decode KV each step. Rebuilding a packed buffer may require repacking the prompt. The planner counts actual packed capacity, including page rounding and reservation, alongside resident pages and peak workspace.

Position conventions. Base K arrives with RoPE already applied at canonical positions; the write operation only stores it. Shared execution rotates each owner's Q by the negative position offset. A packed private path instead rotates the selected Base K by the positive offset. Repair and Decode K already use the request's target positions and receive no additional offset rotation. This convention supports fixed-frequency, full-head RoPE with consistent rotation rules. Any candidate whose backend cannot implement the required position adjustment is excluded by the compatibility check in Section A.6.

Releasing storage and handling memory pressure. When a request finishes, the runtime removes it from future execution and waits for its last GPU use before releasing storage. Repair slots become reclaimable, Decode pages are released, and Base reference counts are decremented. Base stays valid while another request references it. Once no references remain, the cache policy determines whether to retain or reclaim it. Execution copies and old plan buffers are also released only after their last GPU use.

Admission accounts for allocated resident pages and additional execution buffers, including any temporary coexistence during compaction or switching. For Repair, the runtime follows the compaction-before-page-growth procedure above. If no memory-feasible path can be prepared, admission waits while the active batch continues on a valid plan. Allocation is retried after eligible storage has been released. Live-slot counts alone do not establish admission capacity.

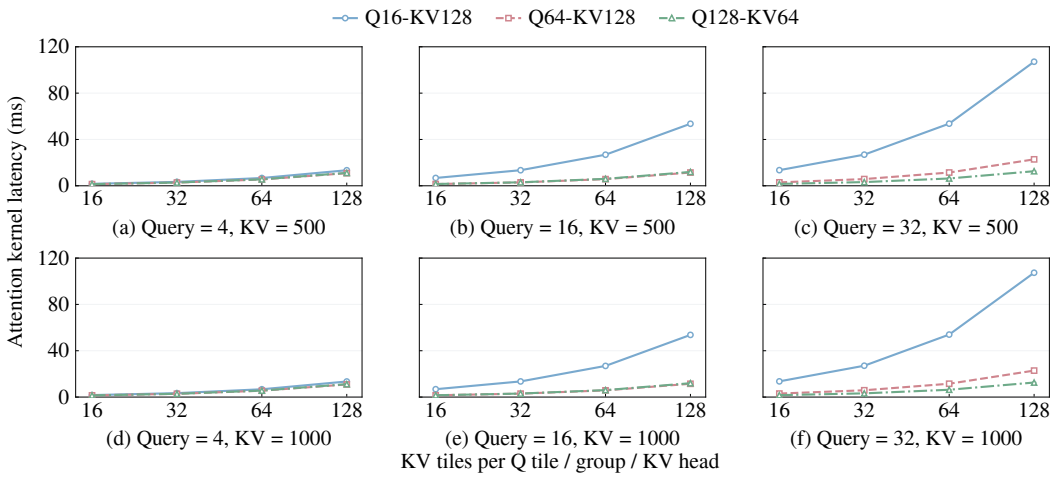

Figure 7: Kernel latency across configurations and work assignments. The three curves compare Q16-KV128, Q64-KV128, and Q128-KV64. Panels (a)–(c) use KV $= 500$ and Query $= 4$, 16, and 32; panels (d)–(f) use KV $= 1000$ with the same Query settings. The horizontal axis is KV tiles per Q tile, group, and KV head; the vertical axis is attention-kernel latency in milliseconds.

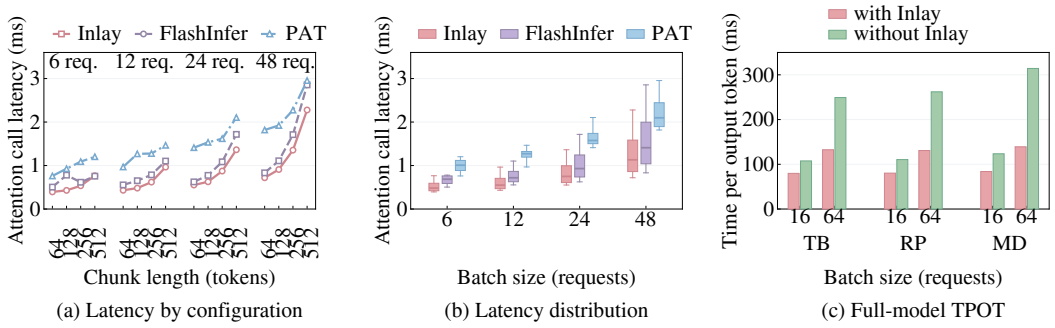

Figure 8: Additional attention-call and full-model decoding results. (a) Attention-call latency of Inlay, FlashInfer, and PAT for chunk lengths of 64–512 tokens at batch sizes 6, 12, 24, and 48. (b) Latency distributions across configurations. (c) CacheBlend TPOT with and without Inlay on ToolBench (TB), RAGPulse (RP), and MultiDoc2Dial (MD) at batch sizes 16 and 64. Lower values indicate better performance in all panels.

Resident-capacity estimates. The capacity ablation in Figure 5(a) uses a 5% Repair ratio and estimates resident-request capacity from the measured average per-request KV memory footprint across all model layers. The resident KV budget excludes packed execution copies and temporary workspace. This is a resident-capacity estimate rather than a guarantee that the same number of requests can execute under a hybrid plan. Execution feasibility is checked separately using the complete peak-memory accounting in Section A.6.

E Kernel Configuration Sensitivity

Figure 7 compares three kernel configurations across the six Query/KV settings shown in the panels. The horizontal axis varies KV tiles per Q tile, group, and KV head from 16 to 128. The curves show how measured kernel latency depends on both the configuration and the work assigned to it.

At Query $= 4$, the three configurations have similar latency in both rows. At Query $= 16$, Q64-KV128 and Q128-KV64 remain close, while Q16-KV128 becomes substantially slower as the horizontal-axis value increases. At Query $= 32$, Q128-KV64 has the lowest latency, followed by

Q64-KV128 and Q16-KV128. The same ordering appears at both KV settings. These trends motivate configuration-specific calibration of task costs and resource limits, as used in Sections A.3 and A.7.

F Additional Attention and Full-Model Results

Attention-call latency. Figure 8(a) compares Inlay, FlashInfer, and PAT at batch sizes 6, 12, 24, and 48, with chunk lengths of 64, 128, 256, and 512 tokens. Inlay has lower plotted latency than both baselines across these configurations. The gap generally becomes more visible at the larger batch sizes and longer chunks. Panel (b) summarizes latency across configurations at each batch size: Inlay has the lowest median at every tested batch size, although the distributions overlap. These results extend the kernel comparison to different chunk lengths and batch sizes.

Full-model decoding. Figure 8(c) compares CacheBlend with and without Inlay on ToolBench, RAGPulse, and MultiDoc2Dial at batch sizes 16 and 64. Integrating Inlay lowers full-model time per output token (TPOT) in all six plotted cases. The absolute reduction is larger at batch size 64 for each workload. The result shows that the benefit extends to full-model decoding in the tested CacheBlend integration. TPOT includes computation beyond attention, so its reduction need not match the attention-call gain.