CACHEFLOW: EFFICIENT LLM SERVING VIA AUTOMATED 3D-PARALLEL KV CACHE RESTORATION

Anonymous authors
Paper under double-blind review

ABSTRACT

KV cache restoration is becoming a major bottleneck in long-context LLM serving, including multi-turn conversations, agentic workloads, and RAG. Existing advances largely treat restoration as a coarse per-request choice between recomputing cached states and loading them from external storage (e.g., CPU memory or remote machines), overlooking structured parallelism within the model and resource contention across batched requests. Increasingly, modern hybrid architectures complicate restoration as their interleaved attention and recurrent layers introduce heterogeneous compute costs and state footprints. We present CacheFlow, which rethinks restoration as a multi-dimensional parallel execution problem. To our knowledge, CacheFlow is the first framework to jointly exploit restoration parallelism across tokens, layers, and GPUs. CacheFlow unifies token- and layer-wise restoration as a staircase partition of the token–layer space, while lightweight boundary activations enable concurrent restoration across model shards. A batch-aware dynamic-programming planner jointly determines what to recompute and what to load across requests and layer blocks, balancing GPU computation against shared I/O resources. Across dense and hybrid models spanning single-digit to hundred-billion parameters, diverse workloads, and hardware conditions, CacheFlow reduces restoration latency by 2.24×–3.00× on average over existing advances, directly improving end-to-end serving latency by 1.64×.

1 INTRODUCTION

Rapid adoption of LLM in interactive and agentic applications is driving a shift toward long-lived contexts. Modern workloads, including multi-turn conversations (Jiang et al., 2026) and tool-augmented agents (Zhang et al., 2026a), repeatedly reuse previous context. Efficiently restoring the KV cache—the cached intermediate state representing the previous context—has become a first-order serving challenge, as restoration lies directly on the critical path of Time-To-First-Token (TTFT), a key measure of responsiveness in user-facing systems (Qin et al., 2025; Zhang et al., 2026b).

Existing KV cache restoration, however, remains fundamentally inefficient, incurring many seconds of latency for long-context requests. Existing advances often expose a coarse-grained dichotomy: they either recompute the cached prefix through prefill (Kwon et al., 2023; Liu et al., 2024a; Li et al., 2025), consuming scarce GPU compute, or load previously materialized KV states from CPU memory, SSDs, or remote machines (Cheng et al., 2025; Liu et al., 2026a), consuming I/O bandwidth. Neither resource is uniformly preferable: recomputation becomes increasingly expensive as the context grows, whereas loading is limited by storage and network bandwidth.

While existing hybrid approaches (Jin et al., 2025; Qin et al., 2025; Gao et al., 2025) partially overlap recomputation and cache transfer, they remain fundamentally request-centric and fail to capture two key properties and challenges of modern LLM serving. First, restoration costs are highly non-uniform across both tokens and layers. For full attention, recomputation cost grows superlinearly with sequence length, so the KV of later tokens is disproportionately more expensive to regenerate. Modern hybrid architectures—such as Gated Delta Networks (GDN) in Qwen3.5 (Qwen Team, 2026) and Kimi Linear (Kimi, 2026)—introduce an additional source of heterogeneity: full-attention and recurrent layers differ substantially in state size, recomputation cost, and the benefit of partial recomputation. As a result, a single recompute–load split for an entire request cannot capture the best tradeoff across the token–layer space. Second, production serving is inherently batched and

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.

distributed. Concurrent requests compete for shared GPU compute and I/O bandwidth, while model shards introduce cross-device execution dependencies. These effects create contention and stragglers, making independent per-request restoration decisions fundamentally insufficient for overall efficiency.

A decade of distributed deep learning has established a simple lesson: efficient model execution comes from exposing and exploiting its structural parallelism, from data parallelism to pipeline parallelism across devices (Li et al., 2023). KV cache restoration, however, is still largely treated as a coarse recompute-or-load operation. Our key insight is that restoration exposes an analogous structure and can itself be parallelized across the token, layer, and device dimensions. Along the *token* dimension, recomputing an early prefix while loading later KV states avoids the increasingly expensive tail of attention computation. Along the *layer* dimension, recomputing lower layers while loading higher-layer states better amortizes execution overheads (e.g., weight movement). Yet, the preferred partition varies with requests, model architecture, and available resources. This motivates a unified view of cache restoration as a *multi-dimensional parallel execution problem*.

We present CacheFlow, a KV cache restoration framework that realizes this view through coordinated 3D parallelism across tokens, layers, and GPUs. Rather than committing to a single token-wise or layer-wise split, CacheFlow generalizes both into a *staircase partition* of the token–layer space, allowing different layer blocks to recompute different portions of the cached prefix while loading the remainder. At its core, a batch-aware dynamic-programming (DP) planner efficiently searches this space by retaining, for each achievable amount of avoided cache transfer, the minimum recomputation cost required to realize it. The same formulation extends across batched requests, enabling CacheFlow to jointly balance recomputation and I/O under shared serving resources.

CacheFlow further extends restoration along the device dimension, where recomputation would normally remain sequential across model shards because each downstream shard depends on activations from the preceding one. CacheFlow exploits lightweight *boundary activations* at model's shard boundaries, allowing each GPU to independently execute the restoration plan for its local layers. Together, these mechanisms transform KV cache restoration from a coarse recompute-or-load decision into coordinated 3D parallel execution. Overall, we make the following key contributions:

- **The first multi-dimensional formulation of KV cache restoration.** To our knowledge, we are the first to formulate restoration as dependency-constrained partitioning over the token–layer space, capturing non-uniform recomputation costs, model architectures, and resource contention.
- **A unified 3D-parallel KV cache restoration framework.** We introduce token-, layer-, and GPU-level parallelism through a batch-aware dynamic-programming planner that selects recomputation boundaries across requests and layers, and enables concurrent restoration across GPUs.
- **An end-to-end system and comprehensive evaluation.** We implement CacheFlow atop vLLM and LMCache, transparently supporting existing LLM applications. We evaluate it from single-digit-billion dense models to hundred-billion-parameter sparse hybrid models (e.g., Qwen3.5-122B-A10B), across realistic chatbot and agentic workloads and diverse hardware and I/O conditions. CacheFlow reduces TTFT by 2.24×–3.00× on average and TTLT by 1.64×.

2 Related Work and Motivation

LLM serving consists of two sequential stages: a *prefill* stage that processes the input context, and a *decode* stage that autoregressively generates output tokens (Agrawal et al., 2024; Zhang et al., 2026b). Prefill produces the key and value (KV) representations reused by subsequent attention computations.

KV cache optimization. KV cache avoids repeatedly processing context, benefiting multi-turn conversations (Deshpande et al., 2025), RAG (Yao et al., 2025), and agentic workflows (Merrill et al., 2026). However, cache size grows linearly with context length and dominates GPU memory usage. Previous work reduces KV cache memory through quantization (Zandieh et al., 2026; Shen et al., 2026; Liu et al., 2026b; Hooper et al., 2024; Liu et al., 2024c), token eviction (Cai et al., 2025a;b; Kumari et al., 2024; Zhang et al., 2023), or a combination of both (Zhang et al., 2025a). Other systems offload cached states to lower memory tiers (Xie et al., 2026; Liu et al., 2026a; Cheng et al., 2025; Qin et al., 2025), or exploit the workflow structure to guide cache retention and prefetching (Li et al., 2025; Pan et al., 2025). When a needed prefix's KV cache is no longer GPU-resident, it must be restored by loading a saved copy from lower memory tiers or recomputing it (Zhang et al., 2026b; Kwon et al., 2023), placing restoration on the critical path to the first output token.

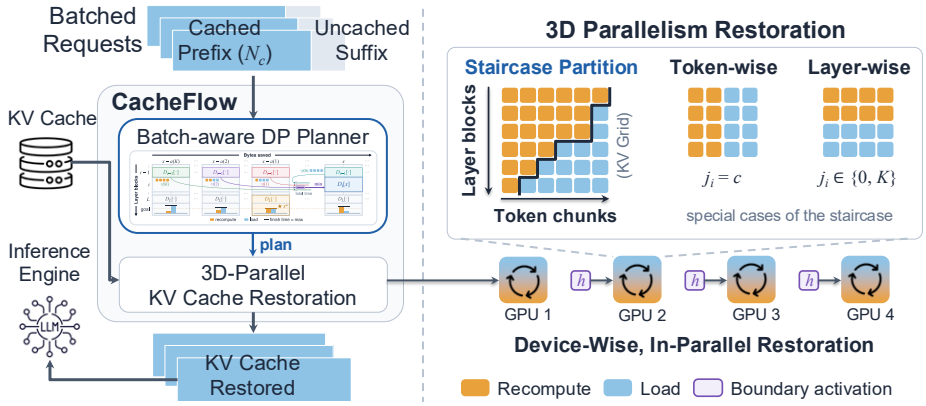


Figure 1: **CacheFlow overview.** The batch-aware DP planner partitions each request's token–layer KV grid into a staircase to maximize computation and I/O overlap across layers, tokens, and devices.

KV cache restoration. KV recomputation consumes GPU compute, while cache loading uses I/O bandwidth, motivating hybrid approaches that overlap the two operations. Cake (Jin et al., 2025) recomputes early tokens while loading the cached KV states of later tokens. Similarly, KVPR (Jiang et al., 2025) overlaps partial KV recomputation with transfer of the remaining KV during offloaded decoding. Hybrid approaches also appear in CacheBlend (Yao et al., 2025) and eLLM (Chen et al., 2026), targeting RAG contexts and partial-token caching respectively. These systems have shown the benefits of combining recomputation and transfer. CacheFlow extends this approach through joint token–layer planning across requests and concurrent restoration across devices.

Limitations of prior advances. The relative efficiency of recomputation and cache loading varies with context length, model architecture, hardware, and available I/O bandwidth. Recent hybrid architectures, as in Gated Delta Networks (GDNs) (Yang et al., 2025) in Qwen3.5 (Qwen Team, 2026) and Kimi Linear (Zhang et al., 2025b) in Kimi-K3 (Kimi, 2026), make this tradeoff substantially harder. Existing restoration techniques fail to capture two key properties in modern settings:

- *Restoration costs vary across tokens and layer types.* Full-attention layers maintain token-proportional KV states, whereas recurrent layers maintain compact fixed-size states, leading to very different recomputation–loading tradeoffs across model depth. Moreover, partial recomputation directly avoids KV transfer in full-attention layers, but does not eliminate the final recurrent state required for decoding. Existing schemes such as Cake and HCache (Gao et al., 2025) do not directly generalize to recurrent states, and/or lose their storage advantage as hidden activations are comparable to recurrent states. A single restoration policy across the model becomes insufficient.
- *Per-request optimization ignores system-wide resource sharing.* Production serving batches requests that contend for shared GPU compute and I/O bandwidth, while distributed execution introduces dependencies across model shards. Optimizing each request independently creates resource imbalance and stragglers even when each local restoration decision appears efficient.

CacheFlow rethinks restoration as a broader joint-optimization problem across tokens, layers, devices, and requests, accounting for heterogeneous costs across layers, and planning for a batch.

3 CacheFlow: Automated 3D-Parallel KV Cache Restoration

We present CacheFlow, a KV cache restoration framework that rethinks restoration as a multi-dimensional parallel execution problem. Our design is grounded in a unifying principle: *KV cache restoration can exploit parallelism along the token, layer, and device dimensions.* As shown in Figure 1, given a request with a cached prefix of N_c tokens, CacheFlow aims to minimize its KV restoration latency by jointly coordinating recomputation and cache loading.

We begin with introducing 3D restoration parallelism across tokens, layers, and devices (§3.1). Token-wise and layer-wise restoration correspond to vertical and horizontal partitions of the token–layer grid, exposing different compute–I/O tradeoffs that depend on request length and model architecture, particularly for modern hybrid-attention models. We extend these strategies across devices so that

GPUs can restore their local states concurrently. Then we generalize these partitions into a planning problem over the token–layer space and solve it for individual requests and then for a batch (§3.2).

3.1 3D Restoration Parallelism

Existing systems (e.g., vLLM and LMCache) typically restore a cached prefix in one of two ways: *recompute* its KV through forward execution, consuming GPU compute, or *load* previously materialized KV from storage, consuming I/O bandwidth. Relying exclusively on either resource leaves the other underutilized. Rather than making this choice for an entire request, CacheFlow exposes opportunities for recomputation and loading to proceed concurrently. We first exploit the token and layer dimensions within a model shard, and then extend restoration across devices.

Token-wise parallelism. Consider a request's token–layer KV grid (Figure 1). By dividing the prefix of N_c tokens into $K = \lceil N_c/C \rceil$ chunks indexed $0, 1, \ldots, \lceil N_c/C \rceil - 1$, we can recompute chunks from the first chunk of the sequence while loading the KV cache from the end. The two processes execute concurrently until they meet at an intermediate chunk. Chunking serves three purposes. First, it exposes fine-grained scheduling units, allowing recomputation to stop as soon as it meets the concurrently loaded suffix, thus avoiding redundancy. Second, choosing C to align with today's efficient GPU execution and scheduling granularities (e.g., 512 tokens per chunk in FlashAttention) preserves high recomputation efficiency (Dao et al., 2022; Jin et al., 2025).

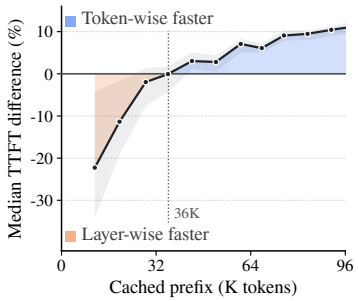


Figure 2: Paired per-request TTFT gap between token-wise and layer-wise restoration on Llama-3.1-8B. Positive means token-wise is faster.

Third, the opposite traversal directions are intentional. Under full attention, the marginal cost of recomputing a token increases with its position because it attends to an increasingly long prefix; theoretically, the cumulative recomputation cost grows quadratically with sequence length (Vaswani et al., 2017). In contrast, the KV state transferred per token is approximately constant. Therefore, token-wise restoration avoids recomputing the increasingly expensive tail.

Layer-wise parallelism. The same recomputation–loading overlap can be exploited along the layer dimension. Here, recomputation proceeds forward from the first layer, while cached states are loaded backward from the last layer. For each layer, the states for the entire cached prefix are recomputed or loaded before advancing to the next layer. The two operations run concurrently and meet at a layer boundary. Thus, layers before the boundary are recomputed for the full cached prefix.

Layer-wise restoration exposes a complementary tradeoff. Because each selected layer recomputes the entire cached prefix, it incurs the expensive attention tail that token-wise restoration avoids. In return, it better amortizes fixed execution overheads: token-wise restoration repeatedly incurs kernel-launch, synchronization, and weight-movement overheads across chunks. Layer-wise restoration pays these costs once per layer. Figure 2 illustrates the trade-off: layer-wise restoration can be preferable for shorter prefixes, whereas token-wise restoration becomes increasingly attractive as attention cost grows with context length.

Device-wise parallelism. Device parallelism is complementary to these token–layer decisions. Practical serving often partitions groups of model layers across devices, either directly or as part of a hybrid parallel deployment (Li et al., 2023). While token-wise and layer-wise restoration expose parallelism, forward recomputation across model shards remains sequential: recomputation on a downstream shard cannot begin until the preceding shard produces its input activations.

CacheFlow removes this dependency using lightweight *boundary activations*. At each shard boundary, we cache the activation that forms the input to the first layer of the downstream shard. These activations are typically much smaller than the aggregate cached states of all layers assigned to the shard, yet provide the input required to begin its local forward execution. During restoration, each GPU retrieves its boundary activation and executes the token–layer restoration plan for its local layers independently of live upstream execution. Thus, model shards depend on cached boundary states rather than one another, transforming restoration from a sequential pipeline into concurrent shard-local execution. This mechanism composes naturally with both token-wise and layer-wise

restoration and is compatible with existing tensor and expert parallelisms too (§4.3). Model shards can concurrently recompute or load their local states according to the plan derived in Section 3.2.

3.2 Batch-Aware Planning over the Token–Layer Space

The preceding mechanisms expose restoration parallelism, but leave open how much of each request should be recomputed versus loaded, especially under batch execution. The best choice varies with context length, model architecture, batch composition, and available compute and I/O bandwidth. Moreover, modern hybrid architectures interleave full-attention layers with recurrent or linear-attention layers whose state footprints and recomputation costs differ substantially. At 128K context, full-attention layers have 125× and 250× larger state footprints than a recurrent checkpoint in Qwen3.5-35B-A3B and Qwen3.5-9B, respectively, while measured per-layer recomputation time differs by 2.6–4.5× on an H100. We next formulate restoration as partitioning the token–layer space.

Restoration as a staircase partition. Consider a model consisting of L layer blocks. For a hybrid architecture, a block contains the recurrent/linear-attention layers preceding a full-attention layer, such as 3:1 in Qwen3.5. We use the block as the planning unit because recomputing its KV requires the activations produced by the preceding layers within that block. A dense transformer is a special case in which a block contains one full-attention layer. Importantly, our formulation can be extended to heterogeneous layer blocks and architectures with multi-token prediction (MTP) modules by incorporating their respective computation costs, state sizes, and execution dependencies.

For a request with K cached chunks, let $j_i \in \{0, \dots, K\}$ denote the number of chunks recomputed in layer block i. The full-attention KV states for the remaining $K - j_i$ chunks are loaded from external storage. A valid restoration plan $\mathbb{J} = (j_1, \dots, j_L)$ must respect forward dependencies: if layer block $i+1$ recomputes a token chunk, block i must also recompute that chunk to produce its input activation. Therefore,

$$K \geq j_1 \geq j_2 \geq \cdots \geq j_L \geq 0. \tag{1}$$

Geometrically, these boundaries form a *staircase* over the token–layer grid. A constant boundary $j_1 = \cdots = j_L = c$ recovers token-wise restoration, whereas setting the first several blocks to K and the remaining blocks to 0 recovers layer-wise restoration.

Modeling recomputation and transfer. Let $f(j)$ denote the time to recompute the first j chunks through one layer block, with $f(0) = 0$. Let S_{attn} denote the full-attention KV size per chunk, S_{rec} the final recurrent-state size of the block, and B the available loading bandwidth. Recomputing j chunks in a block requires loading $d(j) = (K-j)S_{\text{attn}} + \mathbb{1}_{\{j<K\}} S_{\text{rec}}$ bytes. The first term counts the remaining KV chunks to load, and the second reflects that partial recomputation does not reconstruct the final recurrent state, unless the entire prefix is recomputed. For a plan $\mathbb{J} = (j_1, \dots, j_L)$, total computation takes $T_{\text{comp}}(\mathbb{J}) = \sum_{i=1}^{L} f(j_i)$ while loading takes $T_{\text{load}}(\mathbb{J}) = \sum_{i=1}^{L} d(j_i)/B$. These two run concurrently, so restoration finishes when the slower resource completes. Our objective is:

$$\min_{K \geq j_1 \geq \cdots \geq j_L \geq 0} \left\{ \max(T_{\text{comp}}(\mathbb{J}), T_{\text{load}}(\mathbb{J})) \right\}. \tag{2}$$

Enumerating all $(K+1)^L$ restoration plans is intractable. Our key insight is that for repeated blocks, choosing boundary j incurs the same recomputation cost $f(j)$ and avoids the same cache transfer $a(j)$ regardless of the block index. Hence, for any set of selected boundaries $\{j_1, \dots, j_L\}$, both $\sum_i f(j_i)$ and $\sum_i a(j_i)$ depend only on the multiset of boundary values, not on their assignment to model depth. Thus, permuting these boundaries leaves the restoration objective unchanged. We can thus sort them in non-increasing order, $j_1 \geq \cdots \geq j_L$, to satisfy the forward dependencies in Equation (1) without sacrificing optimality. These observations motivate a dynamic program (DP) that records the *minimum recomputation time for each achievable amount of avoided cache transfer*.

Dynamic programming over KV grid. We next introduce the DP formulation. As shown in Figure 3, for a block choosing recomputation boundary j, the avoided cache transfer is $a(j) = jS_{\text{attn}} + \mathbb{1}_{\{j=K\}} S_{\text{rec}}$, where the first term corresponds to the j recomputed KV chunks, and the second captures the recurrent state, whose loading is avoided only when the entire prefix is recomputed. Let $D_i[x]$ denote the minimum recomputation time for the first i layer blocks while avoiding x bytes of cache transfer in total. Starting from $D_0[0] = 0$ and $D_0[x] = +\infty$, we compute

$$D_i[x] = \min_{j \in \{0, \dots, K\}} \left\{ D_{i-1}[x - a(j)] + f(j) \right\}, \tag{3}$$

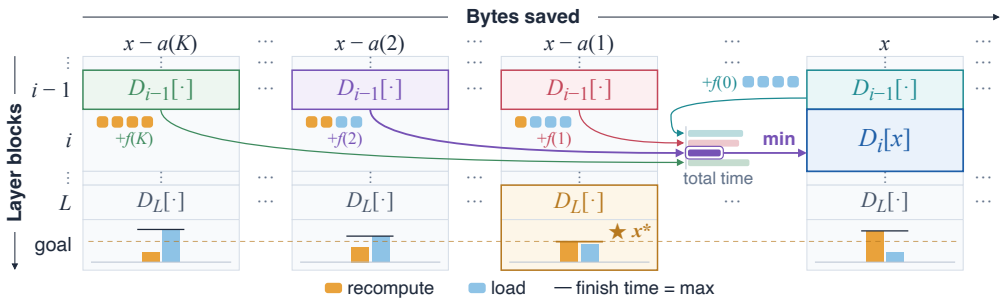


Figure 3: **Illustration of our DP planner.** $D_i[x]$ takes the minimum of $D_{i-1}[x - a(j)] + f(j)$, where boundary j recomputes j chunks in block i and loads the rest. For each final state $D_L[x]$, since recomputation and loading overlap, we pick the $x^\star$ minimizing $\max\{D_L[x], (A_0 - x)/B\}$.

with unavailable states (e.g., due to eviction) and $x - a(j) < 0$ assigned infinite cost.

After processing L layer blocks, each reachable state $D_L[x]$ represents the minimum recomputation time required to avoid x units of state transfer. Without recomputation, the total amount of state to load is $A_0 = L(KS_{\text{attn}} + S_{\text{rec}})$. As recomputation and loading execute concurrently, we select:

$$x^\star = \arg\min_x \max\left\{D_L[x], \frac{A_0 - x}{B}\right\}. \tag{4}$$

Joint planning across a batch. Optimizing requests independently can produce poor resource allocation when they compete for the same GPU compute and I/O bandwidth. Our formulation extends directly to a batch by treating every *request–block pair* as a planning item.

For a batch $\mathcal{R}$, let $M = |\mathcal{R}|L$ and let (r_m, i_m) denote the m-th request–block pair. If request r contains K_r cached chunks, choosing boundary j avoids $a_r(j) = jS_{\text{attn}} + \mathbb{1}_{\{j=K_r\}}S_{\text{rec}}$ bytes of transfer. Let $D_m[x]$ denote the minimum aggregate recomputation time after considering the first m request–block pairs while avoiding x bytes of transfer. The recurrence becomes

$$D_m[x] = \min_{j \in \{0,\ldots,K_{r_m}\}} \{D_{m-1}[x - a_{r_m}(j)] + f_{r_m}(j)\}. \tag{5}$$

The same final balancing step selects the batch-wide operating point between recomputation and cache loading. After backtracking, we group the recovered boundaries by request and reorder each request's boundaries independently to obtain a dependency-valid restoration plan.

Putting planning into practice. We make the planner lightweight in three ways. First, rather than tracking cache traffic at byte granularity, we quantize transfer savings at the same chunk granularity used for restoration (e.g., 512-token chunks). This reduces the DP state space substantially while retaining the granularity at which CacheFlow can actually partition recomputation and loading. Recurrent-state sizes are mapped to the nearest number of chunk-equivalent transfer units.

Second, recomputation costs need not be modeled analytically. For a fixed model and hardware, this cost is largely determined by the token length, agnostic to content, and LLM serving naturally executes requests across a wide range of prefix lengths. Therefore, CacheFlow maintains an online latency profile $f(j)$ by collecting recomputation times from recent serving executions and continuously updating the corresponding profile entries. Unobserved chunk counts are estimated by interpolation. These measurements are obtained from work the system already executes. Our evaluations show that such collection maintains high accuracy, introduces little planning overhead, and naturally adapts to changes in hardware and execution conditions (§4.3).

Third, CacheFlow naturally incorporates the uncached suffix without expanding the DP state space. Suffix tokens have no cached state and are always recomputed. For block i, suffix prefill can begin only after all prefix states required by that block—its full-attention KV and final recurrent state—have been restored, whether through recomputation or loading. Once available, suffix computation is independent of how the prefix was restored, so its cost is fixed for a given request and can be obtained from the same latency profile. With block-ordered loading, suffix prefill in earlier blocks can overlap cache loading for later blocks. Thus, if T_{suf} denotes the suffix-prefill time per block, Equation 4 becomes $x^\star = \arg\min_x \max\{D_L[x] + LT_{\text{suf}}, (A_0 - x)/B + T_{\text{suf}}\}$, where the

compute term covers all L blocks' suffix prefill, and the loading term only the last block's, which must wait for all loads. The DP recurrence remains unchanged. Appendix A.1 provides the analysis.

Approximation guarantee and scaling capability. We next quantify CacheFlow's approximation gap relative to an idealized optimum T^*, where every layer may independently choose an arbitrary token-level restoration boundary (proof details in Appendix A.1):

Proposition 1 (Restoration optimality gap)**.** Consider a request with N_c cached tokens in L layer blocks and chunk size C. Let $\tilde{J}$ be the plan returned by the DP, with makespan $T(\tilde{J}) = \max\{T_{\text{comp}}(\tilde{J}), T_{\text{load}}(\tilde{J})\}$ as in Equation 2, and let e be the rounding error of $a(K)$. Assume that the plan's loading takes at least as long as the suffix prefill, the usual case since cached prefixes are typically much longer than new suffixes, so that $T(\tilde{J})$ is the plan's makespan. Let κ lower-bound the per-token recomputation time of one block, and let $\gamma = 1 + s/(B\kappa)$. Then

$$T(\tilde{J}) - T^* \le \frac{S_{\text{rec}} + Ls(C-1) + L|e|}{B} \le \gamma\, T^* \Big(\frac{S_{\text{rec}}}{LN_c s} + \frac{C-1}{N_c} + \frac{|e|}{N_c s} \Big). \tag{6}$$

The three terms capture block-level planning, chunking, and transfer quantization, whose relative impact vanishes as $O(1/N_c)$. For Qwen3.5-9B at 10 Gbps with $C = 512$, the theoretical gap is at most 19.5 ms, corresponding to 6.52%, 3.26%, and 1.63% of T^* for 32K-, 64K-, and 128K-token prefixes, respectively. For full attention at token granularity ($S_{\text{rec}} = 0$, $C = 1$, $e = 0$), the gap vanishes. Further, device-wise parallelism provides a complementary near-linear scaling benefit with the number of model shards (Appendix A.2). We empirically validate this scaling capability in §4.3.

4 Evaluation

4.1 Experimental Setup

Models and Workloads. We implemented CacheFlow atop vLLM and LMCache. We evaluated CacheFlow on Llama-3.1-8B (Grattafiori et al., 2024) (full attention), Qwen3.5-9B (Qwen Team, 2026) (hybrid attention), Qwen3.5-35B-A3B (MoE, hybrid attention), and larger Qwen3.5-122B-A10B. They expose different state structures and compute–I/O tradeoffs. Our workloads span three realistic datasets: (i) *LMSYS-Chat* (Zheng et al., 2024), which consists of real ChatGPT multi-turn conversational traces where successive turns share long prefixes; (ii) *SWE-Bench* (Jimenez et al., 2024), an agentic coding benchmark with repeated tool invocations over shared repository contexts. (iii) *OSWorld 2.0* (Yuan et al., 2026), a computer-use agentic benchmark.

Hardware and Network Conditions. We conducted experiments on NVIDIA A100s and H100s, covering both single-GPU and multi-GPU deployments. We evaluate different practical bandwidth conditions of 80 Gbps, 40 Gbps, and 10 Gbps, corresponding to typical Infiniband/RoCE speed (Jin et al., 2025), Lambda Lab SSD read speed, and Amazon Cloud inter-node bandwidth (Liu et al., 2024b), respectively. Additionally, we evaluate CacheFlow in online serving using Mooncake arrival timing (Qin et al., 2025). By default, we use 10 Gbps, with ablation studies in Section 4.3.

Baselines. We compare CacheFlow with three advances:

- *vLLM* v0.29.0 (vLLM Team, 2026): it uses recomputation-only as its default policy.
- *LMCache* v0.5.5rc5 (LMCache Team, 2026): KV cache loading without recomputation, representing the I/O-bound extreme and the state-of-the-art offloading system.
- *Cake* (Jin et al., 2025): a strong prior token-wise hybrid restoration approach that partitions each request's cached prefix, recomputing its leading portion while loading its trailing portion.

Metrics. We aim to minimize *Time-To-First-Token* (TTFT), which directly captures the latency impact of KV cache restoration on user-perceived responsiveness, and *Time-To-Last-Token* (TTLT) in online deployments. We also report GPU compute utilization and I/O utilization during restoration.

4.2 End-to-End Performance

CacheFlow reduces serving latency. Figure 4 shows the TTFT distribution across requests for LMSYS-Chat, SWE-Bench, and OSWorld 2.0. Across all workloads, CacheFlow consistently left-

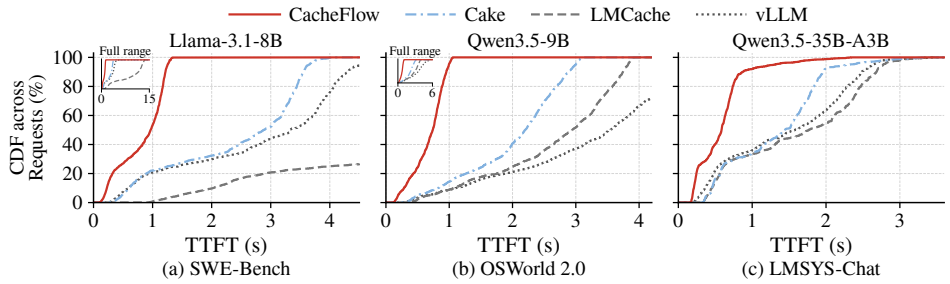


Figure 4: CacheFlow achieves lower serving latency than existing advances.

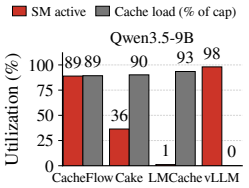


Figure 5: Breakdown of Resource utilization.

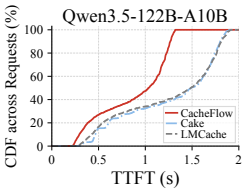


Figure 6: Large-scale MoE serving.

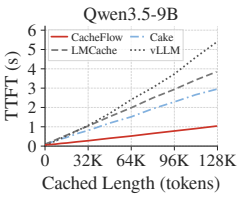


Figure 7: TTFT by input length.

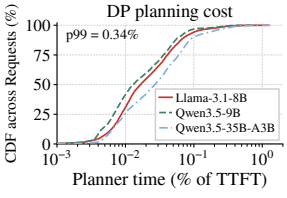


Figure 8: Planning is 0.34% of TTFT at p99.

shifts the CDF, indicating lower latency at every percentile. On average, CacheFlow achieves a 2.24×–3.00× reduction in TTFT compared to existing advances. The gains are most pronounced on LMSYS-Chat and SWE-Bench, which feature longer contexts and thus incur higher KV restoration costs. Notably, the improvement extends to the tail: CacheFlow achieves 1.77×–3.02× and 1.48×–2.95× speedups at p95 and p99, respectively. In §4.3, we show that CacheFlow improves time-to-last-token (TTLT) performance by 1.64×.

CacheFlow improves resource utilization. Figure 5 reports average GPU and I/O utilization during KV cache restoration. LMCache saturates I/O bandwidth but achieves only 1% GPU utilization, as KV transfers throttle execution. Conversely, vLLM is compute-bound with 98% GPU utilization but idle I/O. In contrast, CacheFlow achieves 89% GPU and 89% I/O utilization by effectively overlapping recomputation and I/O.

CacheFlow is compatible with model parallelisms. We further evaluate CacheFlow on Qwen3.5-122B-A10B using eight H100 GPUs with TP=8/PP=1 and TP=4/PP=2, with expert parallelism (EP) disabled and enabled. Figure 6 and Appendix B show TTFT distributions on SWE-Bench with batch size 8 and 10 Gbps I/O bandwidth per worker. All methods use the same parallel configuration within each panel. Across these settings, CacheFlow achieves 1.21×–1.43× speedups in mean TTFT.

4.3 Performance Breakdown and Ablation Study

Breakdown by request lengths. Figure 7 breaks down CacheFlow's performance across different request length regimes on Qwen3.5-9B with OSWorld 2.0. We notice that recomputation-based methods (e.g., vLLM) incur substantially higher latency for long requests. In contrast, CacheFlow scales more gracefully by overlapping recomputation and I/O. Overall, CacheFlow achieves 3.00×, 3.96×, and 4.82× speedups in mean TTFT compared to Cake, LMCache, and vLLM, respectively.

Performance breakdown by components. Figure 9 breaks down CacheFlow's end-to-end TTFT on H100 GPUs with PP=4 and 10 Gbps I/O bandwidth per worker by progressively enabling layer-wise restoration, multi-GPU execution, and token-wise restoration. We observe that these mechanisms provide complementary benefits: together, they improve mean TTFT by 10.46×, 4.46×, and 3.73× over the single-GPU load-only baseline on Llama-3.1-8B, Qwen3.5-9B, and Qwen3.5-35B-A3B, respectively. These results highlight the effectiveness of combining restoration parallelism.

DP planner cost and accuracy. During a sweep of three models and three bandwidths, the DP planner costs 0.019% of TTFT at the median, and 0.34% at p99 (Figure 8). Our DP is not only fast but also accurate. Figure 10 shows 92% of recompute predictions and 98% of load predictions land within 10% of measured latency. We also measured how evenly the plan splits the work. Recompute and load finish within 10% of each other on 95% of restorations.

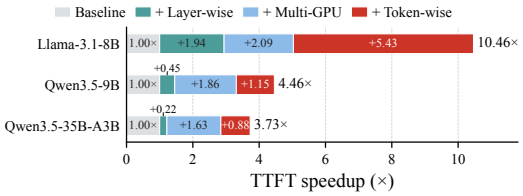


Figure 9: Performance breakdown.

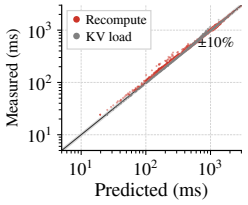


Figure 10: Predicted vs. measured latency.

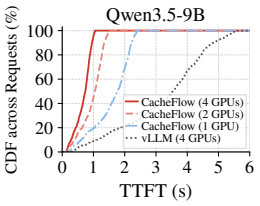


Figure 11: Strong scaling in CacheFlow.

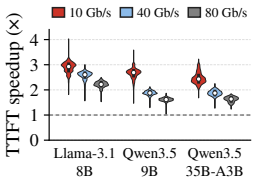


Figure 12: Impact of I/O bandwidth.

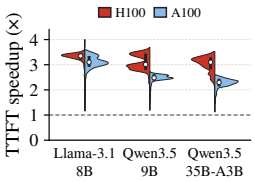


Figure 13: Impact of GPU hardware.

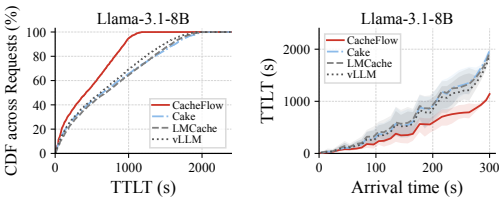


Figure 14: Online serving deployments.

Impact of GPU counts. We break down our 3D parallelism design by varying the GPU count from one to two and four GPUs. Figure 11 shows that average TTFT decreases from 1.59 s to 0.96 s and 0.67 s, respectively (a 2.36× speedup with four GPUs). Notably, even with a single GPU, CacheFlow still outperforms vLLM by 51%, highlighting the effectiveness of our batch-aware DP design.

Impact of I/O bandwidth. We evaluate CacheFlow under different I/O bandwidths from our end-to-end evaluations: 40 Gbps and 80 Gbps on our H100 cluster. Figure 12 shows that CacheFlow consistently improves mean TTFT, yielding 2.63× and 2.40× speedups over Cake at 40 Gbps and 80 Gbps, respectively. These gains stem from CacheFlow's profile-guided DP design, which balances recomputation and I/O across tokens and layers. Under higher bandwidth, more KV cache can be loaded in parallel, further reducing restoration latency.

Impact of GPU hardware. Figure 13 further ablates our previous setting by varying the hardware from H100s to A100s on Qwen3.5-35B-A3B with OSWorld 2.0, under 10 Gbps KV cache I/O transfer per worker. We observe that CacheFlow consistently outperforms Cake, achieving 2.69× and 2.07× speedups in mean TTFT on H100 and A100, respectively.

Online serving deployments. We evaluate CacheFlow in online serving of SWE-Bench using Mooncake arrival timing. Figure 14 shows that CacheFlow improves both TTFT and time-to-last-token (TTLT), achieving approximately 1.64× speedups in both p95 metrics over the fastest baseline. The TTLT timeline further shows that CacheFlow maintains lower request latency as queues build, demonstrating that its benefits extend to full-response completion.

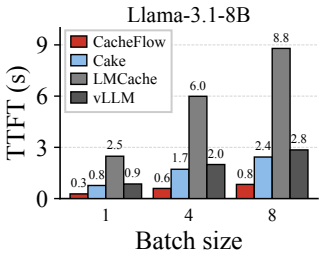


Figure 15: Impact of batch size.

Impact of batch size. Figure 15 shows that CacheFlow consistently achieves 2.8×–2.9× speedups in mean TTFT over Cake across batch sizes (1, 4, and 8 on H100, Llama-3.1-8B). This improvement increases slightly with larger batch sizes, demonstrating that CacheFlow maintains its latency advantage under batching.

5 Conclusion

Efficient restoration of cached model state is becoming essential as LLM serving shifts toward longer, repeatedly reused contexts. Modern hybrid architectures further complicate restoration: full-attention and recurrent layers exhibit different state footprints and recomputation semantics that are poorly captured by existing strategies. CacheFlow exploits 3D parallelism across tokens, layers, and devices while coordinating restoration across concurrent requests. Its staircase abstraction and batch-aware planning enable flexible compute–I/O partitioning across diverse models, workloads, and hardware configurations, reducing TTFT by 2.24×–3.00× and TTLT by 1.64×.

AI USE STATEMENT

We used generative AI tools to assist in implementing parts of our project code and to suggest possible approaches for optimizing the planning procedure. We manually derived and verified the dynamic programming framework presented in the paper. We have also reviewed and tested the AI-generated code for correctness. We reviewed all AI-assisted work and take responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work focuses on system-level optimization to accelerate LLM serving. Our contributions concern computational efficiency rather than changes to model capabilities or behavior. We do not anticipate specific ethical concerns arising from the proposed optimization methods.

REPRODUCIBILITY STATEMENT

We provide the dynamic programming formulation, its assumptions, and the planning algorithm in §3.2. Section 4.1 describes the experimental setup, including hardware and software configurations, evaluated models and workloads, baseline implementations and performance measurement procedures. We will release the source code upon acceptance. To facilitate reproducibility, we will include the configurations and scripts used to run our experiments in the code release.

REFERENCES

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. In *OSDI*, 2024.

Zefan Cai, Wen Xiao, Hanshi Sun, Cheng Luo, Yikai Zhang, Ke Wan, Yucheng Li, Yeyang Zhou, Li-Wen Chang, Jiuxiang Gu, Zhen Dong, Anima Anandkumar, Abedelkadir Asi, and Junjie Hu. R-kv: Redundancy-aware kv cache compression for reasoning models. In *NeurIPS*, 2025a.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, et al. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. In *COLM*, 2025b.

Wenyan Chen, Chengzhi Lu, Huanle Xu, Kejiang Ye, and ChengZhong Xu. High throughput and low latency llm serving via adaptive kv caching. In *EuroSys*, 2026.

Yihua Cheng, Yuhan Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. Lmcache: An efficient kv cache layer for enterprise-scale llm inference. *arXiv preprint arXiv:2510.09665*, 2025.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *NeurIPS*, 2022.

Kaustubh Deshpande, Ved Sirdeshmukh, Johannes Baptist Mols, Lifeng Jin, Ed-Yeremai Hernandez-Cardona, Dean Lee, Jeremy Kritz, Willow E. Primack, Summer Yue, and Chen Xing. MultiChallenge: A realistic multi-turn conversation evaluation benchmark challenging to frontier LLMs. In *Findings of ACL*, 2025.

Shiwei Gao, Youmin Chen, and Jiwu Shu. Fast state restoration in llm serving with hcache. In *EuroSys*, 2025.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. In *NeurIPS*, 2024.

Chaoyi Jiang, Lei Gao, Hossein Entezari Zarch, and Murali Annavaram. KVPR: Efficient LLM inference with I/O-aware KV cache partial recomputation. In *Findings of ACL*, 2025.

Yinsicheng Jiang, Yeqi Huang, Liang Cheng, Cheng Deng, Xuan Sun, and Luo Mai. Contextpilot: Fast long-context inference via context reuse. In *MLSys*, 2026.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *ICLR*, 2024.

Shuowei Jin, Xueshen Liu, Qingzhao Zhang, and Z. Morley Mao. Compute or load kv cache? why not both? In *ICML*, 2025.

Kimi. Kimi K3: Open frontier intelligence, 2026. URL https://www.kimi.ai/blog/kimi-k3.

Lilly Kumari, Shengjie Wang, Tianyi Zhou, Nikhil Sarda, Anthony Rowe, and Jeff Bilmes. Bumblebee: Dynamic kv-cache streaming submodular summarization for infinite-context transformers. In *COLM*, 2024.

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 *SOSP*, 2023.

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.

Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *OSDI*, 2023.

Guangda Liu, Wenhao Chen, Chengwei Li, Zhenyu Ning, Jing Lin, Yiwu Yao, Quan Chen, Shixuan Sun, Jieru Zhao, and Minyi Guo. ECHO: Efficient KV cache offloading with lossless prefetching for serving native sparse attention LLMs. In *OSDI*, 2026a.

Jiachen Liu, Jae-Won Chung, Zhiyu Wu, Fan Lai, Myungjin Lee, and Mosharaf Chowdhury. Andes: Defining and enhancing quality-of-experience in llm-based text streaming services. *arXiv preprint arXiv:2404.16283*, 2024a.

Tengxuan Liu, Shiyao Li, Jiayi Yang, Tianchen Zhao, Feng Zhou, Xiaohui Song, Guohao Dai, Shengen Yan, Huazhong Yang, and Yu Wang. PM-KVQ: Progressive mixed-precision KV cache quantization for long-cot LLMs. In *ICLR*, 2026b.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. Cachegen: Kv cache compression and streaming for fast large language model serving. In *SIGCOMM*, 2024b.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. In *ICML*, 2024c.

LMCache Team. LMCache. https://github.com/LMCache/LMCache/releases/tag/v0.5.5rc5, 2026.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *ICLR*, 2026.

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 *NeurIPS*, 2025.

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 *FAST*, 2025.

Qwen Team. Qwen3.5: Towards native multimodal agents, 2026. URL https://qwen.ai/blog?id=qwen3.5.

Hanzhang Shen, Haoran Wu, Yiren Zhao, and Robert Mullins. Triaxialkv: Toward extreme low-precision kv-cache quantization for agentic inference tasks. *arXiv preprint arXiv:2605.17170*, 2026.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, 2017.

vLLM Team. vLLM: Release v0.29.0. https://github.com/vllm-project/vllm/releases/tag/v0.29.0, 2026.

Zhiqiang Xie, Ziyi Xu, Mark Zhao, Yuwei An, Vikram Sharma Mailthody, Scott Mahlke, Michael Garland, and Christos Kozyrakis. Strata: Hierarchical context caching for long context language model serving. In *OSDI*, 2026.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. In *ICLR*, 2025.

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 *EuroSys*, 2025.

Mengqi Yuan, Zilong Zhou, Xinzhuang Xiong, Weiming Wu, Jiayang Sun, Jiamin Song, Kaiqian Cui, Bowen Wang, Haoyuan Wu, Yitong Li, et al. Osworld2.0: Benchmarking computer use agents on long-horizon real-world tasks. *arXiv preprint arXiv:2606.29537*, 2026.

Amir Zandieh, Majid Daliri, Majid Hadian, and Vahab Mirrokni. Turboquant: Online vector quantization with near-optimal distortion rate. In *ICLR*, 2026.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *ICLR*, 2026a.

Wei Zhang, Zhiyu Wu, Yi Mu, Rui Ning, Banruo Liu, Nikhil Sarda, Myungjin Lee, and Fan Lai. Jitserve: Slo-aware llm serving with imprecise request information. In *NSDI*, 2026b.

Yanqi Zhang, Yuwei Hu, Runyuan Zhao, John CS Lui, and Haibo Chen. Diffkv: Differentiated memory management for large language models with parallel kv compaction. In *SOSP*, 2025a.

Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, Wentao Li, Enzhe Lu, Weizhou Liu, Yanru Chen, Weixin Xu, Longhui Yu, Yejie Wang, Yu Fan, Longguang Zhong, Enming Yuan, Dehao Zhang, Yizhi Zhang, Y. T. Liu, Haiming Wang, Shengjun Fang, Weiran He, Shaowei Liu, Yiwei Li, Jianlin Su, Jiezhong Qiu, Bo Pang, Junjie Yan, Zhejun Jiang, Weixiao Huang, Bohong Yin, Jiacheng You, Chu Wei, Zhengtao Wang, Chao Hong, Yutian Chen, Guanduo Chen, Yucheng Wang, Huabin Zheng, Feng Wang, Yibo Liu, Mengnan Dong, Zheng Zhang, Siyuan Pan, Wenhao Wu, Yuhao Wu, Longyu Guan, Jiawen Tao, Guohong Fu, Xinran Xu, Yuzhi Wang, Guokun Lai, Yuxin Wu, Xinyu Zhou, Zhilin Yang, and Yulun Du. Kimi linear: An expressive, efficient attention architecture. *arXiv preprint arXiv:2510.26692*, 2025b.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. In *NeurIPS*, 2023.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. In *ICLR*, 2024.

A Theoretical Analysis

A.1 Optimality and Approximation Guarantees of DP Planning

Overview. We bound how far the plan returned by CacheFlow's DP can be from T^*, the minimum restoration time allowed by the model below, including an uncached suffix (Lemma 1). The DP solves its own problem exactly (Theorems 1–2), but that problem restricts plans in three ways: (i) all layers in a block share one boundary (Theorem 2, at most R/B); (ii) boundaries lie on the chunk grid (Theorem 3, at most $Ls(C-1)/B$); and (iii) transfer is counted in chunk units with rounding error e (Theorem 4, at most $L|e|/B$). Within the model these are the only losses; Proposition 1 in §3.2 states them in relative form, and Appendix A.2 extends the model to multiple devices. The analysis does not yet cover sublayer shortcuts, additional snapshots, heterogeneous blocks, non-additive batched costs, or estimated costs; we leave these extensions to future work.

Model and assumptions. We restore one request with $N \geq 1$ cached tokens, followed by $N_s \geq 0$ uncached suffix tokens, in a serial network of $L \geq 1$ identical blocks $i = 1, \ldots, L$. Each block has $q \geq 0$ recurrent (linear-attention) layers $r = 1, \ldots, q$ followed by one full-attention layer. The parameters are:

- $s > 0$: KV bytes per token in one block's full-attention layer.
- $R_r \geq 0$: bytes in recurrent layer r's final state after all N tokens, with $R = \sum_r R_r$ per block ($R = 0$ when $q = 0$).
- $F(t)$: time to replay the first t tokens through one block, given its input activations and initial recurrent states; it is identical across blocks, nondecreasing for integer $0 \leq t \leq N + N_s$, and $F(0) = 0$.
- $B > 0$: loading bandwidth in bytes per second.

Relative to §3.2, $N = N_c$, $s = S_{\text{attn}}/C$, $R = S_{\text{rec}}$, and F, a, d are indexed by tokens rather than chunks, e.g., $f(j) = F(\min\{jC, N\})$. Network inputs are available and target caches are loadable from off device, but no internal activation checkpoints or intermediate recurrent snapshots are available (Appendix A.2 adds shard boundary activations). Replay evaluates complete layers for selected tokens from known initial states, without sublayer shortcuts. Costs are exact, nonnegative, and additive, and full-attention token costs are nondecreasing with position; chunking only restricts boundaries. One compute resource and one independent loading resource overlap without interference, with sufficient buffers.

Uncached suffix. Every block recomputes the suffix, whose tokens attend to the whole prefix however it was restored. With additive costs, one block's suffix prefill thus takes the plan-independent time $\sigma = F(N + N_s) - F(N)$, the per-block suffix time of §3.2. Block i's suffix prefill starts only after its full-attention KV for all N prefix tokens and its final recurrent state are on device, and after block $i-1$'s suffix prefill. Restoration ends when block L finishes its suffix prefill, or, if $N_s = 0$, when all target caches are on device.

DP formulation. For chunk size $C \geq 1$, the allowed boundaries are $\mathcal{G}_C = \{0, C, \ldots, \lfloor N/C \rfloor C\} \cup \{N\}$, where N permits a partial final chunk. A plan $\mathbb{J} = (j_1, \ldots, j_L)$ with $N \geq j_1 \geq \cdots \geq j_L \geq 0$ recomputes the first $j_i \in \mathcal{G}_C$ tokens in block i. For boundary t, the bytes saved and still loaded are

$$a(t) = st + \mathbb{1}_{\{t=N\}} R, \qquad d(t) = s(N-t) + \mathbb{1}_{\{t<N\}} R,$$

since only replay to N replaces loading final recurrent states. With all-load volume $A_0 = L(Ns + R)$, the objective is

$$T(\mathbb{J}) = \max\left\{ \sum_{i=1}^{L} F(j_i) + L\sigma,\ \frac{1}{B} \sum_{i=1}^{L} d(j_i) + \sigma \right\},$$

which Lemma 1 relates to the makespan. Let $D_i[x]$ be the minimum compute time of i boundary choices saving exactly x bytes, with $D_0[0] = 0$ and all other initial or unreachable states, including

negative indices, set to $+\infty$. Then

$$D_i[x] = \min_{t \in \mathcal{G}_C} \{D_{i-1}[x - a(t)] + F(t)\},$$
$$T_{\mathrm{DP}} = \min_{x: D_L[x] < \infty} \max\{D_L[x] + L\sigma, (A_0 - x)/B + \sigma\}.$$

Backtracking from a minimizing state and sorting the boundaries in nonincreasing order gives a feasible plan with the same costs, since blocks are identical.

Lemma 1 (Suffix overlap)**.** Every execution with prefix-replay and loading totals P and Q takes at least $\max\{P + L\sigma, Q + \sigma\}$. For a sorted plan $\mathbb{J}$ whose loads are issued in block order, with ready suffix work run before remaining prefix replay, the makespan is at most $\max\{T(\mathbb{J}), Q(\mathbb{J})/L + L\sigma\} \le T(\mathbb{J}) + Q(\mathbb{J})/L$, and it equals $T(\mathbb{J})$ when $Q(\mathbb{J}) \ge L\sigma$.

Proof. The compute resource performs $P + L\sigma$ work, and the block whose loads finish last, no earlier than Q, still needs its suffix prefill. For the upper bound, let Q_k be when block k's loads finish. The compute resource idles only when all prefix replay is done and the next suffix block awaits its loads. Without idling, the makespan is $P + L\sigma$. Otherwise, after its last idle period, ending at some Q_k, it prefills only blocks $k, \ldots, L$ and finishes at $Q_k + (L - k + 1)\sigma$. Sorting makes $d(j_i)$ nondecreasing, so $Q_k \le kQ/L$, and the finish time is at most $\max\{Q/L + L\sigma, Q + \sigma\}$, its values at $k = 1$ and $k = L$. This is at most $T(\mathbb{J}) + Q/L$ because $L\sigma \le T(\mathbb{J})$, and at most $Q + \sigma$ when $Q \ge L\sigma$. □

Theorem 1 (Full attention, $C = 1$)**.** For $q = 0$, T_{DP} lower-bounds the makespan of every complete-layer execution, and the DP plan attains it whenever its loading time is at least $L\sigma$. In particular, the DP is optimal when $N_s = 0$.

Proof. Let U_i be the prefix tokens recomputed in layer i; activation dependencies give $U_{i+1} \subseteq U_i$. Replacing each U_i by its first $|U_i|$ positions preserves nesting and transfer volume without increasing compute, and by Lemma 1 the resulting prefix plan's objective is at most the original makespan. Conditioning on the last boundary then gives the recurrence for $D_i[x]$; at fixed x, loading is fixed, so minimum compute suffices, and sorting preserves both costs. This DP argument applies to any $\mathcal{G}_C$ and $R \ge 0$. Attainment follows from Lemma 1 because the DP plan is sorted. □

Theorem 2 (Loss from shared block boundaries)**.** The DP is exact among block-prefix plans. Let $T^*_{\mathrm{block},1}$ and $T^*_{\mathrm{layer},1}$ be the token-level optima with shared block boundaries and independent layer boundaries, the latter lower-bounding every complete-layer execution. Then

$$0 \le T^*_{\mathrm{block},1} - T^*_{\mathrm{layer},1} \le R/B.$$

Proof. Block exactness follows from Theorem 1. Without intermediate snapshots, recurrent replay forms a prefix, so Theorem 1's prefix replacement yields an optimal layer plan with recurrent boundaries $k_{i,r}$ and full-attention boundaries j_i. For $q > 0$,

$$k_{i,1} \ge \cdots \ge k_{i,q} \ge j_i, \qquad j_i \ge k_{i+1,1} \quad (i < L).$$

Replacing each $k_{i,r}$ by j_i keeps all required activations, so the resulting block plan $\widehat{\mathbb{J}}$ is feasible, uses no more compute, and leaves full-attention traffic unchanged. It loads an extra final state only when $k_{i,r} = N > j_i$. Let b be the first block with $j_b < N$; earlier blocks have $j_i = N$, and later recurrent boundaries are at most $j_b < N$, so only block b adds loading, at most R bytes. With the layer plan's totals P^*, Q^*,

$$T^*_{\mathrm{block},1} \le T(\widehat{\mathbb{J}}) \le \max\{P^* + L\sigma, Q^* + \sigma + R/B\} \le T^*_{\mathrm{layer},1} + R/B.$$

The lower bound holds because block plans are also layer plans. □

Theorem 3 (Loss from chunk boundaries)**.** Let $T^*_{\mathrm{block},C}$ be the exact DP value on $\mathcal{G}_C$. Then

$$0 \le T^*_{\mathrm{block},C} - T^*_{\mathrm{block},1} \le \frac{Ls(C-1)}{B}.$$

Proof. Round a token-optimal block plan $\mathbb{J}^*$ down to the grid, $\widehat{j}_i = \max\{g \in \mathcal{G}_C : g \le j_i^*\}$, so that $0 \le j_i^* - \widehat{j}_i \le C-1$. Rounding preserves order and cannot increase compute. Since $N \in \mathcal{G}_C$, a boundary equals N after rounding exactly when it did before, so $d(\widehat{j}_i) - d(j_i^*) \le s(C-1)$ and loading grows by at most $Ls(C-1)/B$. The chunk DP is no worse than $\widehat{\mathbb{J}}$, and the lower bound follows from $\mathcal{G}_C \subseteq \mathcal{G}_1$. □

Strict loss is possible: with $L=1$, $N=C=2$, $N_s=R=0$, $s/B=1$, and $F(t)=t(t+1)/2$, the token optimum is 1 at $t=1$, whereas the chunk optimum is 2 at $t=0$, attaining the bound.

Theorem 4 (Loss from unit rounding)**.** Suppose the planner indexes x in units of $S_{\text{attn}} = Cs$ bytes and rounds $a(N) = sN + R$ to $a(N) + e$, so that $|e| \le S_{\text{attn}}/2$ under nearest rounding. Then its plan $\tilde{\mathbb{J}}$ satisfies $T(\tilde{\mathbb{J}}) \le T^*_{\text{block},C} + L|e|/B$.

Proof. Each $a(t)$ with $t < N$ is already a multiple of S_{attn}, so each plan's estimated load errs by $e\,\nu(\mathbb{J})/B$ with $\nu(\mathbb{J}) = |\{i : j_i < N\}| \le L$. The estimated objective $\tilde{T}$ thus lies in $[T, T + L|e|/B]$ if $e \ge 0$ and in $[T - L|e|/B, T]$ otherwise. For $e \ge 0$ and $\mathbb{J}^*_C$ attaining $T^*_{\text{block},C}$, $T(\tilde{\mathbb{J}}) \le \tilde{T}(\tilde{\mathbb{J}}) \le \tilde{T}(\mathbb{J}^*_C) \le T(\mathbb{J}^*_C) + L|e|/B$; the case $e < 0$ is symmetric. □

Magnitude of the approximation terms. Write $T^* = T^*_{\text{layer},1} > 0$ and let $\kappa > 0$ satisfy $F(t) \ge \kappa t$. A plan recomputing $Z = \sum_i j_i$ full-attention positions needs at least κZ compute and $s(LN - Z)/B$ loading, and the suffix only adds to both. With $\gamma = 1 + s/(B\kappa)$,

$$T^* \ge \min_{0 \le Z \le LN} \max\Big\{\kappa Z, \frac{s(LN-Z)}{B}\Big\} = \frac{LNs}{B\gamma},$$

so dividing each additive term by this bound gives

$$\varepsilon_R = \frac{R}{BT^*} \le \frac{\gamma R}{LNs}, \qquad \varepsilon_C = \frac{Ls(C-1)}{BT^*} \le \frac{\gamma(C-1)}{N}, \qquad \varepsilon_e = \frac{L|e|}{BT^*} \le \frac{\gamma|e|}{Ns}.$$

All three are $O(1/N)$ and assume nothing about F beyond $F(t) \ge \kappa t$.

Combined guarantee (Proposition 1). Theorems 2–4 give

$$T^* \le T(\tilde{\mathbb{J}}) \le T^* + \frac{R + Ls(C-1) + L|e|}{B}, \qquad \frac{T(\tilde{\mathbb{J}})}{T^*} \le 1 + \varepsilon_R + \varepsilon_C + \varepsilon_e.$$

Since T^* lower-bounds every complete-layer execution and, by Lemma 1, $T(\tilde{\mathbb{J}})$ is the DP plan's makespan whenever $Q(\tilde{\mathbb{J}}) \ge L\sigma$, the bounds then compare CacheFlow with the best execution.

Long suffixes. If instead the suffix prefill outlasts loading, Lemma 1 allows the makespan to exceed $T(\tilde{\mathbb{J}})$ by up to $Q(\tilde{\mathbb{J}})/L < \sigma$, since waiting for early loads delays the suffix chain. The bounds then gain at most one block's suffix prefill, and the suffix, not restoration, dominates the time to first token.

Qwen3.5-9B at 10 Gbps. Qwen3.5-9B has $L = 8$ blocks, each with $q = 3$ Gated DeltaNet layers (a $32 \times 128 \times 128$ FP32 state and an 8192×3 BF16 convolution state each) and one full-attention layer (4 BF16 KV heads of dimension 256). Thus $s = 4096$ B and $R = 6{,}438{,}912$ B; with $B = 1.25$ GB/s and $C = 512$, R rounds to $3S_{\text{attn}}$ and $|e| = 147{,}456$ B. The H100 dense BF16 peak $\Pi = 989$ TFLOP/s gives the valid bound $\kappa = 2P_{\text{blk}}/\Pi \approx 1.75\,\mu$s for $P_{\text{blk}} \approx 8.65 \times 10^8$ parameters per block, so $\gamma \approx 2.87$. Assuming at most 50% MFU, an empirical assumption rather than a hardware limit, gives $\kappa \approx 3.50\,\mu$s and $\gamma \approx 1.94$. The additive gap is at most 19.5 ms either way. The relative bounds below hold for any N_s and are rounded upward (1K = 1024 tokens):

		Peak ($\gamma \approx 2.87$)				50% MFU ($\gamma \approx 1.94$)			
Context	N	ε_R	ε_C	ε_e	Total	ε_R	ε_C	ε_e	Total
32K	32,768	1.73%	4.49%	0.32%	6.52%	1.17%	3.03%	0.22%	4.40%
64K	65,536	0.87%	2.25%	0.16%	3.26%	0.59%	1.52%	0.11%	2.20%
128K	131,072	0.44%	1.13%	0.08%	1.63%	0.30%	0.76%	0.06%	1.10%

A.2 Near-Linear Speedup of Device-wise Parallelism

Setup. We split the L blocks of Appendix A.1 evenly into S contiguous shards on identical devices, each with loading bandwidth B. Every shard after the first loads a cached boundary activation of h bytes per token before replay, in place of the network input. The suffix traverses the shards in order, and we ignore transferring its activations between devices. Let $T^{\star}_{\text{single}}$ be the minimum of $T(\mathbb{J})$ over single-device plans, and T_{3D} the concurrent restoration time with per-shard DP plans under the best schedule.

Theorem 5 (Near-linear speedup)**.** With $\Delta = \max\{F(N), (sN+R)/B\} + hN/B$,

$$T^{\star}_{\text{single}}/S \;\le\; T_{\text{3D}} \;\le\; T^{\star}_{\text{single}}/S + \Delta + (1+1/S)L\sigma.$$

Proof. Lower bound: let shard s have prefix-replay and loading totals P_s, Q_s, excluding its boundary activation. It also prefills the suffix of its L/S blocks, and its last load precedes at least one block's suffix prefill, so $T_{\text{3D}} \ge \max\{P_s + L\sigma/S,\, Q_s + \sigma\}$. Concatenating and sorting the shard plans gives a single-device plan, so $T^{\star}_{\text{single}} \le \max\{\sum_s P_s + L\sigma, \sum_s Q_s + \sigma\} \le S\,T_{\text{3D}}$. Upper bound: sort an optimal suffix-free single-device plan, $j_1 \ge \cdots \ge j_L$, with totals P, Q, so that $\max\{P, Q\} \le T^{\star}_{\text{single}}$, and give shard g the boundaries $j_g, j_{g+S}, \ldots$. Since $F(j_i)$ is nonincreasing, each term after the first is at most the mean of the S terms before it, so shard g's compute is at most $P/S + F(N)$. Since $d(j_i)$ is nondecreasing, comparing each term but the last with the S terms after it bounds its loading by $Q/S + (sN+R)/B$. Shard g's DP objective, and hence its prefix-restoration time, is thus at most $T^{\star}_{\text{single}}/S + \max\{F(N), (sN+R)/B\} + L\sigma/S$. Loading its boundary activation first adds at most hN/B, and prefilling the suffix through all L blocks afterwards adds $L\sigma$. □

Interpretation. Δ does not grow with L, whereas the suffix term is serial because the suffix traverses every shard. Since $T^{\star}_{\text{single}} \ge LNs/(B\gamma)$ (Appendix A.1),

$$S \;\ge\; \frac{T^{\star}_{\text{single}}}{T_{\text{3D}}} \;\ge\; S\Big(1 - \frac{S}{L}\cdot\frac{\gamma B\Delta}{Ns} - (S+1)\frac{\gamma B\sigma}{Ns}\Big).$$

The speedup is near-linear when each shard holds many blocks, boundary activations are small relative to per-block KV, and a block's suffix prefill is short compared with loading its prefix KV.

B Additional Results

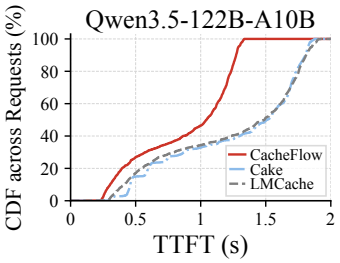
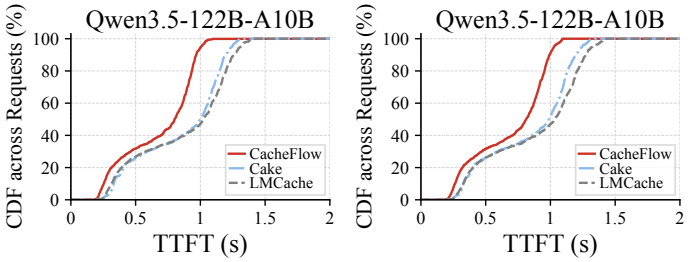


Figure 16: Large-scale MoE serving (TP=8, PP=1, EP disabled).



(a) EP disabled. (b) EP enabled.

Figure 17: Large-scale MoE serving (TP=4, PP=2).