HARMONYTRANSFER: KVCACHE TRANSFER SCHEDULING FOR EFFICIENT LLM AGENT SERVING

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agent applications often contain fan-out stages whose completion depends on multiple requests. Such a stage reaches collective prefill completion only when every required request has returned its first token. Serving long-context fan-out stages efficiently requires moving the reusable KVCache from lower-tier storage back onto the GPU. That transfer runs over a link shared by every live stage. Existing systems, however, schedule KVCache transfers independently for each request, without accounting for the stage's joint completion semantics. We introduce the *Agent Prefill Group* (APG), a first-class abstraction representing the set of requests whose first-token completion jointly determines a stage's collective prefill completion, and present *HarmonyTransfer*, an APG-aware KVCache transfer scheduling system. HarmonyTransfer implements objective-specific APG scheduling policies for reducing mean completion time and improving deadline attainment. It dispatches KVCache transfers in bounded chunk-level batches, allowing the scheduler to make a new APG scheduling decision at each batch boundary rather than only at full-request transfer completion. Built on LMCache and vLLM, HarmonyTransfer reduces mean APG completion time by up to 51.6% over the best-performing baseline and improves APG deadline attainment by up to 23.0 percentage points in testbed experiments.

1 INTRODUCTION

Large language model (LLM) agent applications solve complex tasks by coordinating multiple LLM requests (Yao et al., 2023). They have demonstrated strong capabilities in code generation, scientific research, and conversational systems (Hong et al., 2024; Gao et al., 2023; Semnani et al., 2023; Park et al., 2023). Many such applications organize their execution as fan-out/fan-in stages: they issue several requests concurrently to answer subquestions, retrieve evidence, or validate information, and later synchronize their results before downstream execution continues (Besrour et al., 2025; Zhao et al., 2024; Du et al., 2025).

As Figure 1 illustrates, the three workloads fan out differently: DeepSearcher reranks retrieved chunks in parallel, LongAgent splits document segments across workers, and DocQA assigns one worker to each document. Their prompts are long, and the KVCache working set that keeps long-context prefill affordable does not fit in GPU memory. Hierarchical KVCache systems therefore keep it outside the GPU and stream it back over a shared link, which places the transfer on the critical path to the first token completion time (Section 2). The requests of different stages contend for the shared link, so the transfer schedule decides how quickly a stage can finish.

What the application actually waits for, however, is not first token of any individual member request but the last member request of a stage. We call that point the stage's *collective prefill completion*, and downstream execution cannot start before it (Section 2.2). Reaching the milestone is expensive, because the median fan-out stage reuses 46–90K cached tokens (Figure 2), enough for one stage to hold the link for a long time. The order in which its members use the shared link decides when the last of them finishes.

Existing hierarchical KVCache systems make transfer decisions for individual requests, without knowing which requests contribute to the same collective prefill milestone (Section 2). The scheduler's decision object is a request, while the application's completion objective is a stage. Section 3 quantifies what this mismatch costs in detail.

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.

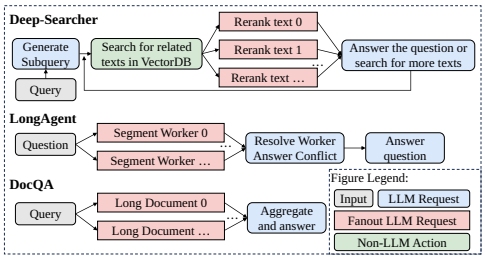


Figure 1: Fan-out/fan-in structures in three agent workloads. DeepSearcher reranks retrieved texts in parallel, LongAgent distributes document segments to multiple workers, and DocQA processes long documents concurrently. Each fan-out must synchronize its member requests before downstream execution can continue.

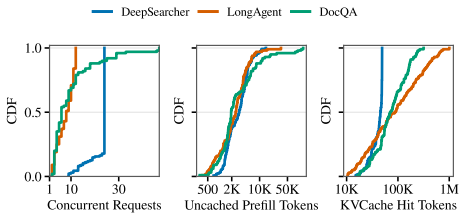


Figure 2: Distributions of fan-out workload characteristics for Llama-3.1-8B. Each sample represents one fan-out stage; number of concurrent requests, uncached prefill tokens, and KVCache hit tokens are aggregated across the requests gated by the same collective prefill completion.

This mismatch motivates treating the gating set itself as a first-class scheduling object. We call such a set an *Agent Prefill Group* (APG). Based on this abstraction, we present *HarmonyTransfer*, an APG-aware KVCache transfer scheduling system. LLM applications submit complete APGs to HarmonyTransfer, informing the scheduler which requests share the same collective prefill milestone. Objective-specific policies select an APG and one of its runnable requests from the online transfer state and a profiled prefill-demand estimate, and the dispatcher issues one bounded transfer batch. After that batch completes, the policy selects again; the next batch may belong to a different request or APG. HarmonyTransfer-Mean completes short APGs early by selecting the live APG with the smallest aggregate remaining demand. HarmonyTransfer-DDL instead projects each APG's completion over the shared transfer path, prioritizes APGs that remain feasible if served next, and demotes those APGs that are predicted to miss.

Built on LMCache and vLLM, HarmonyTransfer reduces mean APG completion time by up to 51.6% over the best-performing baseline across two models and three agent workloads. It raises APG deadline attainment by up to 23.0 percentage points, matching or exceeding the best baseline in all 18 deadline conditions. Our contributions are:

- We identify the mismatch between request-level KVCache transfer scheduling and collective prefill completion, and define the APG as the complete request set that shares this milestone.
- We design two APG-aware policies, for mean APG completion time and for deadline attainment, that rank groups by online estimates of remaining transfer and prefill demand. We also design a bounded-batch dispatcher that lets their decisions take effect on a non-preemptive transfer path.
- We implement HarmonyTransfer on LMCache and vLLM and evaluate it with fan-out traces of three agent applications.

2 Background and Problem Formulation

2.1 LLM Inference and Hierarchical KVCache Systems

LLM inference consists of a *prefill* stage, which processes the prompt tokens, generates their KVCache, and returns the first output token, and a *decode* stage, which generates the remaining tokens autoregressively. Time-to-first-token (TTFT), the elapsed time from a request's arrival to its first output token, models the responsiveness of the serving system; we use the return of the first output token as the observable marker of a request's prefill completion. The KVCache generated during prefill can be retained for reuse: for a request whose prompt shares a prefix with cached ones, only the uncached suffix requires GPU computation. Because the KVCache footprint grows with context length and GPU memory cannot hold the entire reusable working set, hierarchical KVCache systems

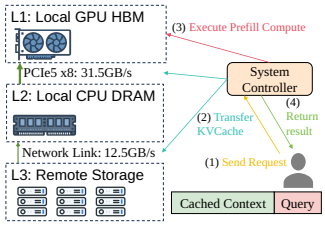


Figure 3: An example hierarchical KVCache system. The system controller fetches a cached context through the remote KVCache pool (L3), CPU DRAM (L2), and GPU HBM (L1), then dispatches prefill computation and returns the result.

combine GPU memory, host memory, and backing storage (Xie et al., 2025; Cheng et al., 2025; Qin et al., 2025).

Figure 3 summarizes both the storage hierarchy and the request path. L3 is the remote KVCache pool, L2 is CPU DRAM on the inference node, and L1 is local GPU HBM.[1] After a user sends a request containing a cached context and an uncached query, the system controller transfers the reusable KVCache along $L3 \rightarrow L2 \rightarrow L1$, dispatches prefill computation for the uncached query once the KVCache reaches L1, and returns the first output token.

Request-level transfer scheduling. Existing hierarchical KVCache systems schedule transfers around individual requests, each retrieving the cached prefix it needs before its own prefill, and recent schedulers model this transfer as part of the per-request service cost (Cheng et al., 2025; Wang et al., 2026). We call policies that rank such independent request objects *request-level scheduling*.

2.2 Collective Prefill Completion in Agent Workloads

When a downstream step depends on all requests of a fan-out stage, application semantics define them as a complete, jointly required set, and the stage finishes only when its last member has returned its complete output. That point carries the user-perceived latency of the stage, because the fan-in has nothing to consume before it; we call it *fan-out completion*. The reranking stage of DeepSearcher in Figure 1 is one instance: after decomposing a question into subquestions and retrieving candidate text chunks, it issues one LLM request per chunk, and the accepted chunk set cannot be assembled until every reranker has returned (Zilliz, 2025). Meanwhile, we define *collective prefill completion* as the point at which every member of the stage has returned its first token.

In DeepSearcher, fan-out completion and collective prefill completion coincide, because its reranker must answer with exactly one YES or NO token. The first token is therefore also the complete logical output of the request. In LongAgent and DocQA the workers still owe decode after collective prefill completion. That residual occupies a small share of the stage, so collective prefill completion still accounts for most of fan-out completion (Section E). Optimizing the prefill milestone is therefore critical for the fan-out stage latency the user perceives.

2.3 The APG Abstraction, Objectives, and Problem

To formalize this milestone, we define an *Agent Prefill Group* (APG) g as the complete set of requests $\mathcal{R}_g$ that share one such milestone.[2] The application declares the membership set $\mathcal{R}_g$, since

[1] Although we instantiate L3 as a remote KVCache pool, L3 may be any other bottlenecked storage medium, such as a local SSD.

[2] A transfer scheduler observes the stage at collective prefill completion, when every member has returned its first token, and it controls the KVCache transfer and the prefill compute up to that point. It cannot wait for fan-out completion: a member's transfer must already have completed before its first token appears. By then the transfers it can order for that stage are spent, and what remains is decode, which the inference engine schedules. Collective prefill completion is also a natural handoff boundary in prefill–decode disaggregated systems (Zhong et al., 2024; Patel et al., 2024; Qin et al., 2025; Cheng et al., 2025), and can be exposed as an interface to application-aware schedulers that optimize full-response or agent-program latency (Luo et al., 2025; Zhang et al., 2025).

application semantics determine which requests are jointly required. The complete request set becomes visible to the scheduler at APG arrival time A_g; some APGs also carry a common absolute deadline D_g. If request r returns its first token at F_r, the APG completes at

$$C_g = \max_{r \in \mathcal{R}_g} F_r. \tag{1}$$

APG completion is therefore determined by the last member to return its first token.

Optimization objectives. We study two APG-level objectives. Mean APG completion time minimizes

$$\frac{1}{|\mathcal{G}|} \sum_{g \in \mathcal{G}} (C_g - A_g), \tag{2}$$

and APG deadline attainment maximizes

$$\frac{1}{|\mathcal{G}|} \sum_{g \in \mathcal{G}} \mathbf{1}[C_g \leq D_g]. \tag{3}$$

Both objectives end at collective prefill completion.

Problem and hardness. We schedule the KVCache transfers of APGs that arrive over time and share one $L3 \rightarrow L2$ link. At arrival A_g, each APG provides its complete member set $\mathcal{R}_g$ and, optionally, a deadline D_g; the scheduler orders transfers on the shared link, and the objective is Eq. 2 or Eq. 3. Two observations shape our policies. If prefill took no time, the offline single-link problem would reduce to minimizing the total completion time of jobs that each consist of several transfers, which serving the groups contiguously in order of aggregate size solves optimally; ranking APGs by aggregate remaining demand (Section 4.4) is the online counterpart of this rule. With prefill, each member is a two-stage transfer-then-compute job whose compute can overlap other members' transfers, and minimizing total completion time is NP-hard already for two-stage jobs without groups (Garey et al., 1976) (Section A.4). We therefore use online policies that act on demand estimates rather than exact schedules.

3 Motivation and Design Requirements

Request-granularity transfer schedulers mismatch APG-level objectives in two ways: they prioritize individual requests rather than APGs, and they commit transfer work too coarsely to promptly enact updated APG priorities.

3.1 Request Scheduling Fragments Collective Progress

Request-level objectives treat each first-token event independently, whereas an APG completes only after its last member completes prefill; ranking requests independently therefore interleaves work from different APGs and fragments progress toward the event the application observes.

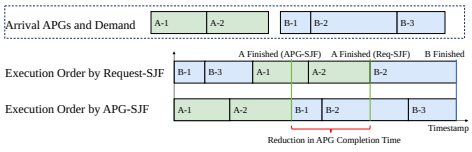


Figure 4: Toy example comparing Request-SJF and APG-SJF on the same arriving APGs and request demands. Request-SJF selects short requests independently and fragments collective progress, whereas APG-SJF selects APG A by its collective demand and completes it earlier.

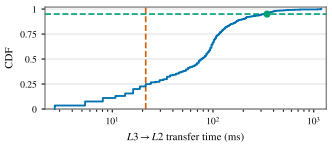


Figure 5: CDF of per-request $L3 \rightarrow L2$ KVCache transfer time for DocQA with Llama-3.1-8B at 100 Gbps. The orange vertical line marks one $b_{\max} = 8$ batch of 256-token chunks (256 MB for Llama-3.1-8B; 21.5 ms), which 75.1% of requests exceed; the green horizontal line marks p95 (344 ms), about $16\times$ the batch duration.

Figure 4 contrasts a request-level policy with an APG-aware one. Request-SJF orders the requests B-1, B-3, A-1, A-2, and B-2. Although it completes two requests of APG B first, B cannot complete until B-2 finishes at the end of the schedule. APG-SJF instead serves A-1 and A-2 consecutively before beginning APG B, so APG A reaches collective completion earlier.

APG identity alone does not determine priority. Minimizing mean completion favors APGs with little remaining work, whereas meeting deadlines requires maintaining a feasible schedule for urgent APGs. In either case, remaining demand must include both KVCache transfer and prefill, because both precede the first-token events that determine APG completion.

Requirement 1: prioritize APGs according to the objective. The system should let application frameworks submit APGs rather than standalone requests, and the scheduler should translate the objective into an APG-level policy backed by transfer and prefill estimates.

3.2 Whole-Request Transfers Prevent Fine-Grained Policy Execution

If the scheduler dispatches a request's entire remaining transfer as one non-preemptive unit, newly arriving or urgent APGs cannot alter the committed order until that transfer completes. As shown in Figure 5, the p95 whole-request transfer time is 344 ms for DocQA at 100 Gbps, about 16 times the 21.5-ms transfer time of one 256-MB batch, and 75.1% of requests exceed this fixed-size threshold. Bounded batches shorten the committed window so that the scheduler can reconsider APG priorities after each batch.

Requirement 2: bound committed transfer work. Each decision issues one bounded, non-preemptive batch; after a batch completes, the policy selects the APG and request for its replacement.

4 HarmonyTransfer

4.1 Overview and APG Submission

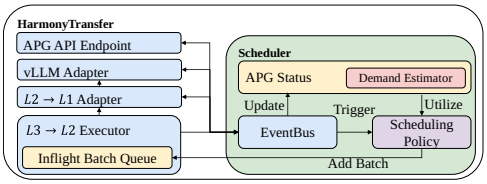


Figure 6: HarmonyTransfer system overview. The APG API and runtime adapters publish events to the EventBus, which updates APG state and invokes the scheduling policy. The policy selects a bounded transfer batch and places it in the $L3 \rightarrow L2$ executor's Inflight Batch Queue.

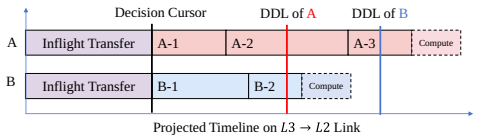


Figure 7: Finish-if-next projections after the same in-flight prefix. APG A would miss its deadline if served next, whereas APG B can still meet its deadline, so HarmonyTransfer selects B. Each row is an independent projection.

As Figure 6 shows, an application submits each APG atomically as a first-class object. A submission contains a stable APG identity, its complete request set, and an optional common deadline. HarmonyTransfer makes an APG eligible for scheduling only after the complete submission arrives, so the scheduler never observes a partial APG. Submissions and runtime progress events flow through a shared EventBus that updates scheduler state and drives $L3 \rightarrow L2$ transfer, $L2 \rightarrow L1$ loading, and prefill execution.

4.2 Batch-Level Transfer Dispatch

HarmonyTransfer stores KVCache in fixed-size *chunks*, each holding the KVCache of a few hundred tokens. Whenever the $L3 \rightarrow L2$ executor has capacity for another batch, it publishes an event to the EventBus, which invokes the scheduling policy to select an APG and one of its requests.

The selected transfer is organized as a *transfer batch* containing up to $b_{\max}$ consecutive chunks from the same request. Each batch is issued as a single non-preemptive $L3 \rightarrow L2$ command, and the scheduling policy is reinvoked at batch boundaries. By bounding the batch size instead of transferring all remaining chunks of a request at once, HarmonyTransfer lets a higher-priority APG take over the transfer path at the next batch boundary.

The $L3 \rightarrow L2$ executor processes dispatched batches in enqueue order. The batch currently using the transfer path and the batches queued behind it form the *in-flight prefix*, whose order the policy can no longer change. The Inflight Batch Queue holds at most $q_{\max}$ batches: a larger $q_{\max}$ better hides scheduling latency but commits more transfer work in advance.

4.3 APG Demand Estimation

HarmonyTransfer estimates each live request's remaining transfer and prefill time, expressing both in seconds so that the policies can compare requests with different cached-prefix and uncached-suffix lengths.

Estimating transfer time. Let $x_r(t)$ be the number of cached tokens of request r that still wait to be transferred from $L3$. The remaining link demand of r is

$$R_r(t) = \rho_r(t) + \beta x_r(t), \tag{4}$$

where $\rho_r(t)$ is the residual time of the in-flight batch belonging to r and β is the profiled transfer time per KVCache token. Appendix A details how $\rho_r(t)$ and β are obtained.

Estimating prefill compute time. For cached-prefix and uncached-suffix lengths h_r and m_r, offline profiling fits the prefill estimate $C_r = f(h_r, m_r)$ with a second-order function ($R^2 = 0.995$ and p95 absolute error 11.34 ms). The total demand of request r is then

$$w_r(t) = R_r(t) + C_r. \tag{5}$$

4.4 Objective-Specific APG Policies

Both policies select an APG and then one of its runnable requests. They share the estimator and batch dispatcher but use different APG priorities.

Optimizing APG completion time. HarmonyTransfer-Mean aggregates the live request demands of each APG:

$$W_g(t) = \sum_{r \in \mathcal{R}_g,\, r \text{ live}} w_r(t). \tag{6}$$

It selects the APG with the smallest $W_g(t)$ and then its runnable request with the smallest $w_r(t)$. This completes short APGs and their short members early, while allowing their prefill to overlap subsequent transfers.

Optimizing APG deadline attainment. EDF always favors the earliest deadline, even when that APG can no longer meet it. HarmonyTransfer-DDL instead asks a simple question before applying EDF: *could this APG still finish on time if it received the link next?*

Finish-if-next projection. Let $P(t)$ be the end of the in-flight prefix, with $P(t) = t$ when the queue is empty, and let $R_r^u(t)$ be the duration of request r's unissued transfer. To test APG g, FinishIfNext appends its unissued request transfers after $P(t)$. It orders requests by decreasing prefill time C_r, allowing a long prefill to start early and overlap the transfers that follow. When a request's transfer finishes at time L, its predicted completion is $L + C_r$; the APG prediction F_g is the latest completion among its live requests. Algorithm 1 in Appendix A.3 uses this projection as a feasibility test for the hypothetical "serve g next" schedule. Requests without unissued transfer contribute their final batch's completion time (or t if already transferred) through ReadyTime, preserving overlap with the in-flight prefix.

Feasibility-first EDF. An unexpired APG is feasible if $F_g \leq D_g$. The policy applies EDF first to feasible APGs ($\mathcal{A}$) and then to downgraded APGs ($\mathcal{D}$), placing APGs without deadlines last. Thus, in Figure 7, feasible B precedes infeasible A even though $D_A < D_B$, whereas plain EDF would

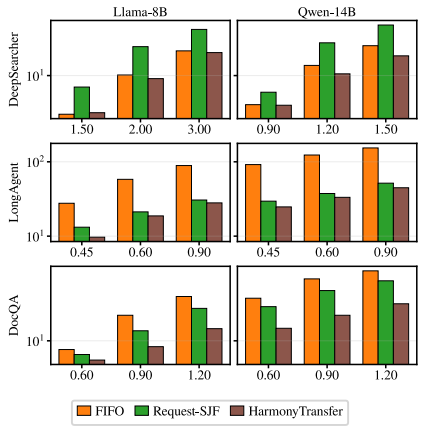


Figure 8: Mean APG completion time (seconds, log scale; lower is better) under varying arrival rates (QPS). FIFO is APG-ordered; Request-SJF is request-level.

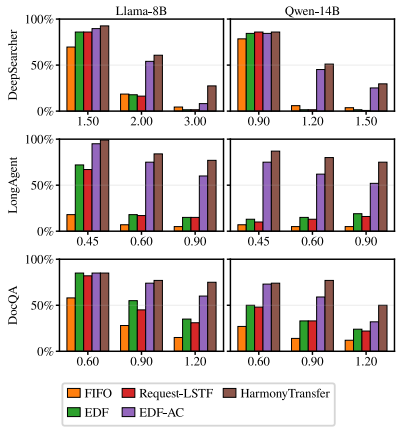


Figure 9: APG deadline attainment (higher is better) under varying arrival rates (QPS).

serve A first and waste $L3 \rightarrow L2$ bandwidth on an APG that misses anyway. Downgrading changes priority, not admission: every APG remains eligible for service. After the algorithm returns an APG, the dispatcher selects its first runnable request and issues that request's next batch.

5 Evaluation

5.1 Experiment Setup

Implementation. HarmonyTransfer is implemented on LMCache 0.4.6 (Cheng et al., 2025) and integrated with vLLM 0.19.1 (Kwon et al., 2023). Following LMCache's multi-process architecture, HarmonyTransfer runs remote KVCache transfer and CPU–GPU loading (with CUDA IPC) in a standalone process whose control path implements the APG-aware scheduler and the bounded-batch dispatcher; a thin vLLM KV connector links the two over ZeroMQ (Hintjens, 2013). Unless stated otherwise, we fix each KVCache chunk at 256 tokens, $b_{\max} = 8$ chunks, and $q_{\max} = 4$ batches for all experiments.

Hardware platform. The evaluation uses one inference node with an NVIDIA RTX PRO 6000D GPU (84 GB) and 128 GB of host DRAM. The GPU node connects to the remote KVCache pool (L3) over a 100-Gbps InfiniBand link.

Workloads. We evaluate HarmonyTransfer with two long-context LLMs: Llama-3.1-8B-Instruct (Meta AI, 2024) and Qwen2.5-14B-Instruct-1M (Yang et al., 2025). We replay fan-out stages from agent traces collected by running DeepSearcher and LongAgent (Zhao et al., 2024) on LongBench v2 (Bai et al., 2025), and DocQA on Loong (Wang et al., 2024): DeepSearcher contains 135 APGs with 3,028 member requests, LongAgent 100 APGs with 771 requests, and DocQA 100 APGs with 1,002 requests. Before evaluation, we warm up the KVCache of all contexts (retrieved texts, document segments, and long documents) in L3. As the traces do not include APG arrival intervals, we generate them as a Poisson process under different QPS settings. Arrival sequences are drawn from a fixed random seed and are identical across policies within each condition, so all comparisons are paired. The reported results use one arrival pattern per condition; Section C repeats the middle arrival rate under three Poisson patterns and confirms that both conclusions are unchanged.

Baselines. We compare multiple scheduling policies. Unless explicitly varied in the ablation study, all policies use the same demand estimator and bounded-batch dispatcher and differ only in their scheduling decisions. At each scheduling decision, FIFO selects requests in arrival order within the earliest-arriving APG, so it is already group-aware. Request-SJF selects the runnable request with the smallest remaining demand $w_r(t)$. EDF selects the uncompleted APG with the earliest absolute

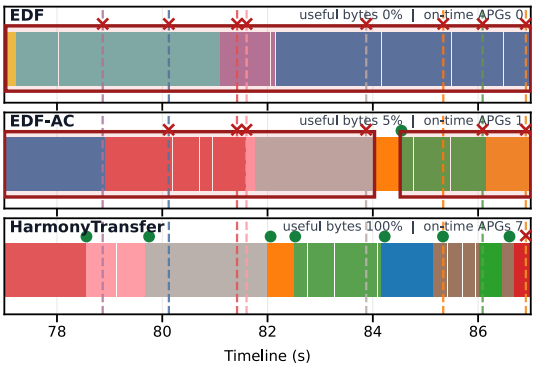


Figure 10: Remote-transfer allocation over the same 77–87 s contention window for three methods. Colored stripes are transfer batches colored by APG; vertical dashed lines mark deadlines, and green circles and red crosses mark on-time completions and misses. Red outlines highlight batches whose APGs eventually miss, and the upper-right annotations report on-time APGs and the share of bytes given to APGs that meet their deadlines.

deadline and follows request arrival order within that APG. Request-LSTF selects the runnable request with the least local slack, $D_g - t - w_r(t)$. EDF-AC selects the unexpired APG with the earliest absolute deadline and demotes an APG to a fallback class once its deadline passes; expired APGs are served only when no unexpired APG has runnable work.[3]

5.2 End-to-End Experiments

Mean APG completion time. As shown in Figure 8, HarmonyTransfer-Mean attains the lowest mean APG completion time in 17 of the 18 conditions; the exception is DeepSearcher with Llama-3.1-8B at the lightest load (1.50 QPS), where FIFO is 5% lower. Relative to the best-performing baseline in each condition, HarmonyTransfer-Mean reduces mean APG completion time by 8–26% on LongAgent, 15–52% on DocQA, and up to 29% on DeepSearcher, with the largest gain (51.6%) on DocQA with Qwen-14B at 0.90 QPS. Request-SJF orders by request size but interleaves the members of different APGs; on DeepSearcher, whose APGs contain about 22 requests each, this still leaves HarmonyTransfer-Mean cutting its mean completion time by up to 66.7%, and Request-SJF is slower than FIFO in every DeepSearcher condition. FIFO is APG-ordered but ignores demand: it is competitive on DeepSearcher, whose APGs have near-uniform demand (Figure 2), whereas HarmonyTransfer-Mean cuts its mean completion time by 27–73% on LongAgent and DocQA, whose APG demands span an order of magnitude.

APG deadline attainment. For deadline experiments, each APG's relative deadline is its interference-free APG completion time multiplied by a factor from $\{2\times, 5\times, 8\times\}$, following the settings of previous work (Gu et al., 2023). All policies share the same deadline assignment. As shown in Figure 9, HarmonyTransfer-DDL matches or exceeds the best baseline in all 18 measured conditions. Its largest gain is 23.0 percentage points with Qwen-14B on LongAgent at 0.90 QPS (75.0% versus 52.0% for EDF-AC). This improvement comes from the finish-if-next feasibility test, which identifies APGs predicted to miss their deadlines even if served next. By downgrading these APGs, HarmonyTransfer-DDL concentrates shared-link service on APGs that can finish on time.

5.3 Case Study

Figure 10 zooms into a contention window of the Llama-3.1-8B-Instruct LongAgent runs. EDF and EDF-AC direct 100% and 95.4% of their transferred bytes to APGs that eventually miss their deadlines, leaving zero and one on-time completions. HarmonyTransfer-DDL instead directs 100% of the transferred bytes to APGs that finish on time, producing seven on-time completions.

5.4 Microbenchmarks

Transfer granularity. Figure 11 compares bounded-batch transfer, which returns control to the policy after each bounded command, against request-granularity transfer, which commits all remaining

[3]CALVO's SJF and LSTF over a loading-plus-prefill cost are our Request-SJF and Request-LSTF (Wang et al., 2026); Section D.4 compares HarmonyTransfer with prior transfer-scheduling systems.

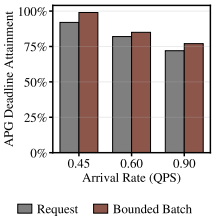


Figure 11: Bounded-batch versus request-granularity transfer on LongAgent.

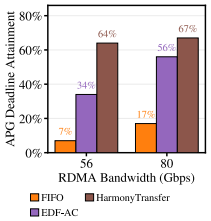


Figure 12: L3 bandwidth sensitivity on DocQA, 56 and 80 Gbps.

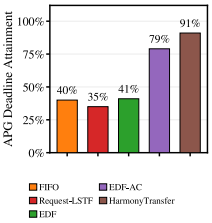


Figure 13: A uniform 50 s relative deadline for every APG on LongAgent.

KVCache at once, for HarmonyTransfer-DDL on LongAgent with Llama-3.1-8B-Instruct. Bounded Batch wins at all three arrival rates: shorter committed windows let the policy respond to updated urgency.

L3 bandwidth. We vary the RDMA link rate for Llama-3.1-8B-Instruct on DocQA at 0.9 QPS under the deadline objective. At 56 and 80 Gbps, HarmonyTransfer-DDL attains 64% and 67% of APG deadlines, EDF-AC 34% and 56%, and FIFO 7% and 17% (Figure 12).

Uniform SLO assignment. Figure 13 removes the $2\times/5\times/8\times$ multiplier and gives every APG the same relative deadline, so the distribution cannot explain the outcome. On LongAgent at 0.60 QPS with Llama-3.1-8B, the request-level baselines attain 35–41% of their deadlines, EDF-AC 79%, and HarmonyTransfer-DDL 91%, which is also the lowest mean APG completion time (49.4 s against 57.4–60.6 s). The advantage comes from APG-aware ordering, not from the deadline assignment.

6 Related Work

Hierarchical KVCache and transfer scheduling. Reusing KVCache across GPU, host and remote tiers is now standard in LLM serving (Kwon et al., 2023; Zheng et al., 2024; Cheng et al., 2025; Qin et al., 2025; Gao et al., 2024), and much of the work that follows compresses it, overlaps its loading with computation, or schedules the transfer paths of individual requests (Liu et al., 2024; Yao et al., 2025; Jin et al., 2025; Gao et al., 2025; Xie et al., 2025; Wu et al., 2026; Wang et al., 2026; Sun et al., 2026). All of them decide *how* or *how fast* a request's KVCache moves; HarmonyTransfer is complementary and decides *which* request the shared link serves next.

Coflow and group scheduling. An APG has the structure of a coflow (Chowdhury & Stoica, 2012), whose completion is defined by the last member. Coflow scheduling is NP-hard (Roemer, 2006; Mastrolilli et al., 2010; Qiu et al., 2015), so practical systems rely on online heuristics (Chowdhury et al., 2014; Chowdhury & Stoica, 2015; Agarwal et al., 2018; Wan et al., 2025). HarmonyTransfer inherits that structure but differs in demand, completion, link service, and deadlines; Section D compares the two point by point.

7 Conclusion

LLM-agent fan-out stages reach collective prefill completion only after every member has returned its first token, yet KVCache transfer schedulers still operate on individual requests. HarmonyTransfer exposes this milestone through the Agent Prefill Group abstraction and combines objective-specific APG scheduling with bounded transfer batches. Across two models and three workloads it reduces mean APG completion time by up to 51.6% over the best-performing baseline and raises deadline attainment by up to 23.0 percentage points. Appendix D discusses limitations and future work.

REPRODUCIBILITY STATEMENT

Section 5 describes the implementation, platform, models, workloads, baselines, scheduler parameters, and the fixed-seed arrival generation shared by all policies. Algorithm 1 specifies the deadline policy, and Appendix A gives the transfer- and prefill-estimation details together with the FINISHIFNEXT procedure. The case study in Section 5.3 analyzes a contention window at batch granularity. Appendix B documents the workload selection and the relative-deadline assignment, and Appendix C reports the additional experiments.

AI USE STATEMENT

Generative AI tools were used to polish author-written text, to draft parts of the related-work discussion from author-provided references, and to assist with code development. All AI-assisted text and code were reviewed, verified, and edited by the authors, who take full responsibility for the content.

REFERENCES

Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. Sincronia: Near-optimal network design for coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pp. 16–29. ACM, 2018. doi: 10.1145/3230543.3230569.

Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench v2: Towards deeper understanding and reasoning on realistic long-context multitasks, January 2025.

Ines Besrour, Jingbo He, Tobias Schreieder, and Michael Färber. SQuAI: Scientific question-answering with multi-agent retrieval-augmented generation. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*, CIKM '25, pp. 6603–6608, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 979-8-4007-2040-6. doi: 10.1145/3746252.3761471.

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, October 2025.

Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pp. 31–36, Redmond Washington, October 2012. ACM. ISBN 978-1-4503-1776-4. doi: 10.1145/2390231.2390237.

Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. *ACM SIGCOMM Computer Communication Review*, 45(4):393–406, September 2015. ISSN 0146-4833. doi: 10.1145/2829988.2787480.

Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pp. 443–454, Chicago Illinois USA, August 2014. ACM. ISBN 978-1-4503-2836-4. doi: 10.1145/2619239.2626315.

Zhuoyun Du, Chen Qian, Wei Liu, Zihao Xie, YiFei Wang, Rennai Qiu, Yufan Dang, Weize Chen, Cheng Yang, Ye Tian, Xuantang Xiong, and Lei Han. Multi-agent collaboration via cross-team orchestration. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 10386–10406, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.541.

Bin Gao, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, Pengfei Zuo, and Huawei Cloud. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pp. 111–126, 2024. doi: https://dl.acm.org/doi/10.5555/3724648.3724661.

Luyu Gao, Zhuyun Dai, Panupong Pasupat, Anthony Chen, Arun Tejasvi Chaganty, Yicheng Fan, Vincent Zhao, Ni Lao, Hongrae Lee, Da-Cheng Juan, and Kelvin Guu. RARR: Researching and revising what language models say, using language models. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16477–16508, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.910.

Shiwei Gao, Youmin Chen, and Jiwu Shu. Fast state restoration in LLM serving with HCache. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 128–143, Rotterdam Netherlands, March 2025. ACM. ISBN 979-8-4007-1196-1. doi: 10.1145/3689031.3696072.

M. R. Garey, D. S. Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976. doi: 10.1287/moor.1.2.117.

Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. ElasticFlow: An Elastic Serverless Training Platform for Distributed Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 266–280, Vancouver BC Canada, January 2023. ACM. ISBN 978-1-4503-9916-6. doi: 10.1145/3575693.3575721.

Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, 2013. ISBN 978-1-4493-3406-2.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework, November 2024.

Shuowei Jin, Xueshen Liu, Qingzhao Zhang, and Z Morley Mao. Compute or load KV cache? Why not both? In *The 42nd International Conference on Machine Learning (ICML)*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 979-8-4007-0229-7. doi: 10.1145/3600006.3613165.

LangChain AI. LangGraph: Building stateful, multi-actor applications with LLMs. https://github.com/langchain-ai/langgraph, 2024.

LangChain AI. Langchain-ai/open_deep_research. https://github.com/langchain-ai/open_deep_research, May 2025.

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

Yifei Liu, Zuo Gan, Zhenghao Gan, Weiye Wang, Chen Chen, Yizhou Shan, Xusheng Chen, Zhenhua Han, Yifei Zhu, Shixuan Sun, and Minyi Guo. Efficient serving of LLM applications with probabilistic demand modeling, June 2025.

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 *Proceedings of the ACM SIGCOMM 2024 Conference*, pp. 38–56, Sydney NSW Australia, August 2024. ACM. ISBN 979-8-4007-0614-1. doi: 10.1145/3651890.3672274.

Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. Autellix: An efficient serving engine for LLM agents as general programs, February 2025.

Monaldo Mastrolilli, Maurice Queyranne, Andreas S. Schulz, Ola Svensson, and Nelson A. Uhan. Minimizing the sum of weighted completion times in a concurrent open shop. *Operations Research Letters*, 38(5):390–395, 2010.

Meta AI. Introducing llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/, July 2024.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pp. 1–22, San Francisco CA USA, October 2023. ACM. ISBN 979-8-4007-0132-0. doi: 10.1145/3586183.3606763.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132, 2024. doi: 10.1109/ISCA59077.2024.00019.

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, Santa Clara, CA, February 2025. USENIX Association. ISBN 978-1-939133-45-8.

Zhen Qiu, Cliff Stein, and Yuan Zhong. Minimizing the total weighted completion time of coflows in datacenter networks. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pp. 294–303. ACM, 2015. doi: 10.1145/2755573.2755592.

Thomas A. Roemer. A note on the complexity of the concurrent open shop problem. *Journal of Scheduling*, 9(4):389–396, 2006. doi: 10.1007/s10951-006-7042-y.

Sina Semnani, Violet Yao, Heidi Zhang, and Monica Lam. WikiChat: Stopping the hallucination of large language model chatbots by few-shot grounding on wikipedia. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 2387–2413, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.157.

Yijun Sun, Xudong Liao, Songrun Xie, Hao Chen, Han Tian, Wenxue Li, Yiming Zhang, and Kai Chen. Multi-stage flow scheduling for LLM serving, March 2026.

Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. Towards end-to-end optimization of LLM-based applications with Ayo. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25. ACM, 2025. doi: 10.1145/3676641.3716278.

Xinchen Wan, Xinyu Yang, Kaiqiang Xu, Xudong Liao, Yilun Jin, Yijun Sun, Zhenghang Ren, Han Tian, and Kai Chen. Coflow Scheduling for LLM Training. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pp. 1232–1234, São Francisco Convent Coimbra Portugal, September 2025. ACM. ISBN 979-8-4007-1524-2. doi: 10.1145/3718958.3750467.

Minzheng Wang, Longze Chen, Fu Cheng, Shengyi Liao, Xinghua Zhang, Bingli Wu, Haiyang Yu, Nan Xu, Lei Zhang, Run Luo, Yunshui Li, Min Yang, Fei Huang, and Yongbin Li. Leave No document behind: Benchmarking long-context LLMs with extended multi-doc QA. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 5627–5646, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.322.

Weiye Wang, Chen Chen, Junxue Zhang, Zhusheng Wang, Hui Yuan, Zixuan Guan, Xiaolong Zheng, Qizhen Weng, Yin Chen, and Minyi Guo. CALVO: Improve Serving Efficiency for LLM Inferences with Intense Network Demands, March 2026.

Yongtong Wu, Shaoyuan Chen, Yinmin Zhong, Rilin Huang, Yixuan Tan, Wentao Zhang, Liyue Zhang, Shangyan Zhou, Yuxuan Liu, Shunfeng Zhou, Mingxing Zhang, Xin Jin, and Panpan Huang. DualPath: Breaking the Storage Bandwidth Bottleneck in Agentic LLM Inference, February 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, August 2025.

An Yang, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoyan Huang, Jiandong Jiang, Jianhong Tu, Jianwei Zhang, Jingren Zhou, Junyang Lin, Kai Dang, Kexin Yang, Le Yu, Mei Li, Minmin Sun, Qin Zhu, Rui Men, Tao He, Weijia Xu, Wenbiao Yin, Wenyuan Yu, Xiafei Qiu, Xingzhang Ren, Xinlong Yang, Yong Li, Zhiying Xu, and Zipeng Zhang. Qwen2.5-1M technical report, January 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 *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 94–109, Rotterdam Netherlands, March 2025. ACM. ISBN 979-8-4007-1196-1. doi: 10.1145/3689031.3696098.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models, March 2023.

Wei Zhang, Zhiyu Wu, Yi Mu, Ning Rui, Banruo Liu, Nikhil Sarda, Myungjin Lee, and Fan Lai. JITServe: SLO-aware LLM Serving with Imprecise Request Information, December 2025.

Jun Zhao, Can Zu, Xu Hao, Yi Lu, Wei He, Yiwen Ding, Tao Gui, Qi Zhang, and Xuanjing Huang. LONGAGENT: Achieving question answering for 128k-token-long documents through multi-agent collaboration. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 16310–16324, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.912.

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 A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3.

Zilliz. Zilliztech/deep-searcher: Open source deep research alternative to reason and search on private data. Written in python. https://github.com/zilliztech/deep-searcher, 2025.

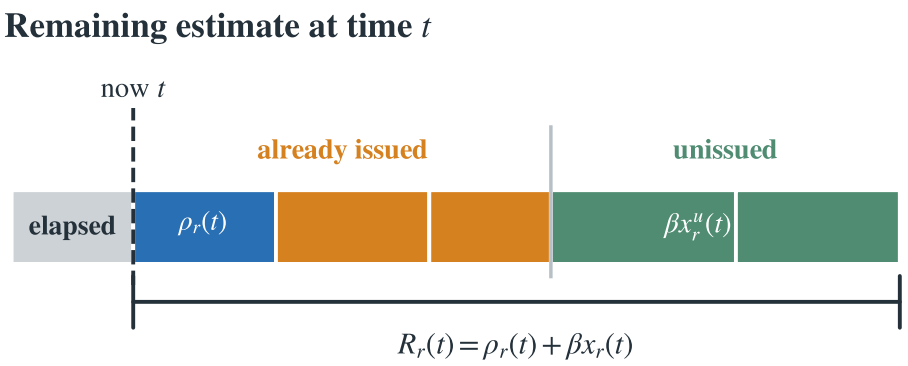


Figure 14: Remaining remote-transfer estimate at time t. The estimator excludes elapsed service and charges only the residual of the currently executing batch, the full predicted durations of already-issued queued batches, and the token-proportional duration of unissued batches.

A Solution Details

This section gives the details of the transfer- and prefill-time estimation used by the HarmonyTransfer policies.

A.1 Transfer Time Estimation

HarmonyTransfer uses a linear model to estimate the duration of a transfer batch. For a batch containing x_b cached tokens, its predicted transfer duration is

$$\widehat{T}_{\text{batch}}(x_b) = \beta x_b, \tag{7}$$

where β is the transfer time per KVCache token. Specifically,

$$\beta = \frac{s_{\text{KV}}}{B_{\text{L3}}}, \tag{8}$$

where s_{KV} is the model's KVCache bytes per token and B_{L3} is the profiled L3 payload bandwidth in bytes per second.

Figure 14 shows how the remaining remote-transfer time at time t is estimated from the residual service time of the active batch, the predicted durations of queued issued batches, and the unissued tokens. Let $x_r(t)$ be the number of cached tokens of request r that are still waiting for transfer. This count includes tokens in already-issued queued batches and in unissued batches, but excludes the batch currently using the link. Let $x_r^u(t)$ denote the number of unissued tokens. Their remaining transfer duration is

$$R_r^u(t) = \beta x_r^u(t). \tag{9}$$

HarmonyTransfer accounts for the progress of the batch currently using the link separately. It subtracts the elapsed service time from Eq. 7 and clamps the residual at zero:

$$\rho_r(t) = \max\{0, \widehat{T}_{\text{batch}}(x_b) - (t - t_{\text{start}})\}, \tag{10}$$

where t is the current time, t_{start} is the start time of the active batch, and x_b is its token count. We set $\rho_r(t) = 0$ if that batch does not belong to r. Because $x_r(t)$ already combines all queued and unissued tokens, the request's complete remaining transfer estimate is

$$R_r(t) = \rho_r(t) + \beta x_r(t). \tag{11}$$

Term	Llama-3.1-8B	Qwen2.5-14B
c_0 (ms)	14.79	28.06
c_h	0.772	1.221
c_m	101.42	191.75
c_{hh}	-2.35×10^{-3}	-2.15×10^{-3}
c_{hm}	3.675	6.740
c_{mm}	1.855	3.476

Table 1: Coefficients of Eq. 12 for the two evaluated models. The hit and uncached counts h and m are expressed in thousands of tokens and the output in milliseconds.

Model	R^2 (10-fold)	p50 abs. err. (ms)	p95 abs. err. (ms)	MAE (ms)
Llama-3.1-8B	0.9952	3.36	11.34	4.49
Qwen2.5-14B	0.9950	5.28	20.83	7.73

Table 2: Prefill-profile accuracy per model, from a 10-fold cross-validation over the 4,801 requests profiled once each. Errors are absolute differences between predicted and measured prefill-compute time.

A.2 Prefill Compute Time Estimation

For each model, we conduct offline profiling to record the prefill-compute time of requests with different reusable-prefix and uncached-suffix lengths. Let h and m denote the cache-hit and cache-miss token counts of a request, respectively. We fit the profile with the following function:

$$\widehat{T}_{\text{prefill}}(h, m) = c_0 + c_h\, h + c_m\, m + c_{hh} h^2 + c_{hm} hm + c_{mm} m^2. \tag{12}$$

Prefill profile. The prefill estimate of Eq. 12 is fitted per model on offline probes that cover the hit and uncached lengths observed in the three workloads. The profiling set contains every one of the 4,801 member requests of the three workloads, measured once each, plus 21 pure-prefill probes at seven suffix lengths repeated three times. Table 1 lists the coefficients loaded at runtime, with h and m in thousands of tokens, and Table 2 reports the fit quality for the two evaluated models. Because R^2 grows with the spread of the profiled durations, the absolute error is the informative figure. A 10-fold cross-validation over the 4,801-request profiling set gives a p95 absolute error of 11.3 ms for Llama-3.1-8B and 20.8 ms for Qwen2.5-14B, small against the transfer and prefill durations they are added to. Refitting the same six-term form after adding the pure-prefill probes, which is what produces the deployed coefficients, moves an individual prediction by at most 2.2 ms at the p95 and 7.4 ms at the maximum for Llama-3.1-8B. The same refit moves it by at most 4.7 ms and 17.3 ms for Qwen2.5-14B.

A.3 Finish-if-Next Projection

Algorithm 1 gives the APG selection procedure of HarmonyTransfer-DDL described in Section 4.4.

Algorithm 1 HarmonyTransfer-DDL APG selection

Require: complete live APGs $\mathcal{G}$, time t, in-flight prefix ending at $P(t)$
$\mathcal{A}, \mathcal{D} \leftarrow \emptyset$
for all $g \in \mathcal{G}$ **do**
  $F_g \leftarrow$ FINISHIFNEXT(g)
  if D_g exists, $t < D_g$, and $F_g \leq D_g$ **then**
    add g to $\mathcal{A}$
  else
    add g to $\mathcal{D}$
  end if
end for
sort $\mathcal{A}$ and $\mathcal{D}$ in EDF order (APGs without deadlines last)
for all g in $\mathcal{A}$ followed by $\mathcal{D}$ **do**
  if g has runnable work **then**
    return g
  end if
end for
return NONE

Algorithm 2 details the FINISHIFNEXT procedure used by HarmonyTransfer-DDL. Given an APG g, the procedure advances one link cursor L through the unissued transfer of each live member in decreasing prefill-time order. For an unissued transfer, it sets $T_r = L + R_r^u(t)$; otherwise, it obtains T_r from READYTIME, which returns the completion time of the request's final batch if that batch is already in the in-flight prefix and t if the transfer has completed. The returned prediction is the latest prefill-completion time among live members.

Algorithm 2 FINISHIFNEXT projection for HarmonyTransfer-DDL

Require: APG g, time t, in-flight prefix ending at $P(t)$
function FINISHIFNEXT(g)
  $L \leftarrow P(t)$; $F \leftarrow t$
  for all live $r \in \mathcal{R}_g$ in decreasing C_r **do**
    if r has unissued transfer **then**
      $L \leftarrow L + R_r^u(t)$; $T_r \leftarrow L$
    else
      $T_r \leftarrow$ READYTIME(r)
    end if
    $F \leftarrow \max\{F, T_r + C_r\}$
  end for
  return F
end function

A.4 GROUP ORDERING WITHOUT PREFILL

Section 2.3 states that, if prefill took no time, serving APGs contiguously in order of aggregate size would be optimal. We make this precise.

Lemma 1. *Consider a single link and a set of APGs available at time* 0*, where APG* g *consists of transfers of total size* S_g *and completes when its last transfer finishes. If prefill takes no time, serving the APGs contiguously in non-decreasing order of* S_g *minimizes the total (equivalently, the mean) APG completion time.*

Proof sketch. Contiguity. Take any schedule and let g be the APG that completes first, at time C_g. Move all transfers of g to the front of the schedule, shifting the other transfers that preceded C_g later by the same total amount. g now completes no later than before, and every other APG completes at the same time as before, because its last transfer lies after C_g and is not moved. Applying the same step to the remaining APGs in order of completion yields a contiguous schedule that is no worse.

Order. In a contiguous schedule each APG behaves as a single job of length S_g on one machine, and the total completion time is minimized by shortest-processing-time-first, by the standard adjacent-exchange argument. □

With prefill, each member becomes a two-stage transfer-then-compute job, and the compute stage of one member can overlap the transfers of others. Already without groups this is the two-machine flow shop with the total-completion-time objective, which is NP-hard (Garey et al., 1976); online arrivals and deadlines add further difficulty. HarmonyTransfer-Mean applies the rule of Lemma 1 online, with S_g replaced by the aggregate remaining demand $W_g(t)$ of Eq. 6, which also accounts for the prefill term.

A.5 Application Interface

Job format. An application submits one fan-out stage as one `FanOutJob`, `FanOutJob(job_id, members, deadline)`. Each entry of `members` is a `FanOutMember(request_id, prompt, max_tokens)`, where `request_id` is the key the downstream fan-in joins on and `max_tokens` is the member's output length. Construction rejects an empty `job_id` and an empty member set. A stage therefore cannot enter the system half-declared, so the scheduler never observes a partial APG. `deadline` is optional and relative to the stage's interference-free completion time; without it the stage runs under the mean objective.

Agent workflow frameworks already expose a fan-out stage as one unit: LangGraph's map-reduce branch spawns one worker per sub-query and joins them at a collector (LangChain AI, 2024; 2025).

Gateway change. The gateway adds three objects and changes nothing else: the `ApgJob` scheduling unit, its handle, and the streamed prefill-to-decode handoff. `gateway.submit(fanout)` converts the front-end job into the repository's existing `PrefillGroupJob` and returns an `ApgJobHandle`. The driver submits that payload through `submit_job`, the call the repository already uses, so the gateway itself never talks to the scheduler.

Grouping is a switch rather than a second implementation. With grouping the gateway returns the stage as one atomic group job. Without it the gateway returns one single-member job per member. The handle, the streamed decode and the group milestone are shared by both paths, so an application that does not declare grouping behaves as it did before.

The handle carries two views. Each member's first token releases a `PrefillRequestHandle` to the decode side as soon as it exists, which keeps per-member streaming unchanged. A group-level result reports the collective milestone, once every member is prefilled, together with whether the stage met its deadline.

B Experiment Details

B.1 SLO Assignment and Workload

A deadline experiment needs a per-APG target before it can say whether an APG met it. Following earlier work (Gu et al., 2023), we use *relative* deadlines: an APG's target is its own interference-free completion time scaled by a slack factor. We measure each APG once in isolation—one APG in flight at a time, with no contention on the $L3 \to L2$ path—and record that duration as the APG's optimal time t_g. Its deadline is then

$$D_g = m_g\, t_g, \qquad m_g \in \{2\times, 5\times, 8\times\}, \tag{13}$$

where the multiplier m_g is assigned per APG. The three values give a spectrum of slack: $2\times$ leaves little room for queueing, whereas $8\times$ is met by every policy under light load.

The assignment is balanced rather than independent. The planner lays the three multipliers out round-robin over a workload's APGs, and then shuffles that list with the same random generator that samples the Poisson arrival offsets. Round-robin gives 34 APGs at $2\times$, 33 at $5\times$ and 33 at $8\times$ in the 100-APG workloads, and 45 of each in the 135-APG workload. Two consequences matter for reading the results. First, the multiset of multipliers is fixed, so different seeds present the same mix of tight and loose deadlines and only the pairing with APGs changes. Second, within one seed every

			Members per APG			Uncached prefill tok. (K)			KVCache hit tok. (K)		
Workload	APGs	Requests	p50	p90	max	p50	p90	max	p50	p90	max
DeepSearcher	135	3,028	24	24	24	3.1	6.3	14.4	45.6	49.2	49.2
LongAgent	100	771	8	12	12	2.5	6.0	34.4	90.1	436.5	1,003.0
DocQA	100	1,002	6	23	47	1.9	10.9	123.9	72.7	174.8	316.7

Table 3: Stage shapes for Llama-3.1-8B. "Members per APG" is the barrier width; the token columns are per stage and sum the members that one barrier gates. Percentiles are nearest-rank over the APGs of each workload, the convention used by the workload builder.

scheduling policy receives the identical arrival offsets and the identical per-APG multipliers, because the plan depends only on the model, workload, arrival rate and seed and never on the scheduler. The comparison between policies is therefore paired at every seed: a difference in deadline attainment comes from the ordering decisions, not from one policy being handed an easier set of deadlines.

The same construction fixes what a deadline number can be compared across. Because the multipliers follow the seed, the same job does not carry the same deadline in two different seeds, so cross-seed comparisons are meaningful only for aggregates over a cell; Section C.1 reports them.

Workloads. The evaluation replays fan-out stages from three agent workloads. DeepSearcher reranks retrieved chunks in parallel, LongAgent distributes document segments to a pool of workers, and DocQA assigns one worker to each document of a long input. The three workloads contain 135 APGs and 3,028 member requests for DeepSearcher, 100 and 771 for LongAgent, and 100 and 1,002 for DocQA. The traces record the jobs and their member requests but not the intervals between successive APG arrivals. Arrivals are therefore simulated with a Poisson process at the rate under test, and every policy replays the same arrival plan.

B.2 Workload Construction

Fan-out extraction. We replay recorded stages of real agent runs rather than a synthetic request mixture. A run records the *barriers* at which the application issues requests in parallel and joins them before its next step. The members of one barrier are therefore exactly the requests whose first tokens must all return before the stage can proceed. From the DeepSearcher runs we take the rerank barrier, which scores the retrieved chunks concurrently, and from the LongAgent runs the worker barrier, which distributes document segments to the worker pool. Barriers with a single member carry no fan-out and are dropped, and every member prompt is validated to start with one of the precomputed context prefixes recorded in the same run. The two traces contribute more stages than we need. DeepSearcher keeps the 135 with the largest joint transfer and compute demand, and LongAgent keeps 100 chosen by farthest-point sampling over the plane of $\log(\text{transfer bytes})$ against $\log(\text{uncached tokens})$. Sampling this way makes the replayed set span both the cheap and the expensive corner of the demand plane instead of concentrating in one of them.

DocQA. The workload is built from Loong (Wang et al., 2024): 100 of the dataset's 1,600 questions are drawn uniformly at seed 0, and every document listed for a question becomes one worker request, so each question contributes a per-document barrier. The resulting 1,002 member requests are tokenized, cached, and replayed under the same protocol as the two traced workloads, and the generation cap that the published agent applies to a single document is preserved.

Cached-token accounting. Each member prompt is tokenized with the target model and split into the matched context prefix and the remaining suffix. The prefix is the only reusable part, and it is warmed in L3 and stored in 256-token chunks, so a member's cache-hit tokens are its prefix length aligned down to a chunk boundary. Its uncached prefill tokens are its prompt length minus the hit tokens. Table 3 summarizes the stage shapes that result.

Table 3 shows that the three workloads stress different axes. DeepSearcher has a fixed, wide barrier with near-uniform demand, whereas LongAgent and DocQA vary both the barrier width and the per-stage demand by more than an order of magnitude. That spread is what separates a group-aware policy from a request-level one.

Model	Workload	QPS	FIFO (s)	Request-SJF (s)	HarmonyTransfer (s)	Gain (%)
Llama-3.1-8B	DeepSearcher	1.50	**2.66**	6.74	2.80	-5.4
		2.00	10.23	27.09	**9.01**	+11.9
		3.00	23.39	48.93	**22.12**	+5.4
Qwen2.5-14B	DeepSearcher	0.90	3.70	5.67	**3.62**	+2.1
		1.20	14.20	30.83	**10.65**	+25.0
		1.50	27.92	56.85	**19.76**	+29.2
Llama-3.1-8B	LongAgent	0.45	27.68	13.21	**9.71**	+26.5
		0.60	58.00	21.22	**18.79**	+11.4
		0.90	88.98	30.62	**28.03**	+8.5
Qwen2.5-14B	LongAgent	0.45	91.57	29.51	**24.82**	+15.9
		0.60	123.32	37.50	**33.23**	+11.4
		0.90	154.00	51.57	**44.73**	+13.3
Llama-3.1-8B	DocQA	0.60	7.78	6.71	**5.71**	+14.9
		0.90	21.15	13.41	**8.45**	+37.0
		1.20	36.43	25.77	**14.28**	+44.6
Qwen2.5-14B	DocQA	0.60	34.70	27.09	**14.46**	+46.6
		0.90	60.95	43.47	**21.05**	+51.6
		1.20	77.03	57.55	**29.43**	+48.9

Table 4: Detailed results behind Figure 8. Every cell reports the mean APG completion time in seconds for one (workload, model, arrival-rate) condition. The best value of each row is set in bold; "Gain" is the percentage reduction of HarmonyTransfer-Mean relative to the better of FIFO and Request-SJF in that row.

B.3 Detailed Results Behind E2E Experiments

Table 4 and Table 5 list the individual numbers plotted in Figure 8 and Figure 9, so that the claims of Section 5.2 can be checked without reading bar heights off a logarithmic axis.

In the mean-completion table, HarmonyTransfer-Mean attains the lowest value in 17 of the 18 conditions; the exception is DeepSearcher with Llama-3.1-8B at 1.50 QPS, where FIFO is 5.4% lower. The largest reduction over the better of FIFO and Request-SJF in a condition is 51.6% (DocQA with Qwen2.5-14B at 0.90 QPS), and the reduction reaches at least 5% in 16 of the 18 conditions.

In the deadline table, HarmonyTransfer-DDL matches or exceeds the best baseline in all 18 conditions. It wins outright in 16, and ties at 85.9% (Qwen2.5-14B, DeepSearcher, 0.90 QPS, with Request-LSTF) and at 85.0% (Llama-3.1-8B, DocQA, 0.60 QPS, with EDF and EDF-AC). Its largest margin is 23.0 percentage points over EDF-AC (Qwen2.5-14B, LongAgent, 0.90 QPS).

C Additional Experiments

C.1 Policy Robustness across Arrival Patterns and Deadline Assignments

We replay the middle arrival rate of every workload under three seeds, one per arrival pattern, over the same job count and the same $2\times$ / $5\times$ / $8\times$ multiplier multiset. Within a seed all methods share the arrival offsets and the per-APG deadline assignment, so the comparisons in Table 6 and Table 7 are paired, and a seed changes only the pattern and the pairing of multipliers with APGs. Seed 0 replays the arrival vector of the main evaluation in the restarted environment used for this batch, so its columns track the corresponding main-text cells rather than reproducing them exactly.

C.1.1 Average Objective: Only the Arrival Pattern Varies

Table 6 reports the average objective under the three patterns, and HarmonyTransfer attains the lowest mean APG completion time in every column. The raw spread across seeds is large for the baselines—FIFO on LongAgent runs $58.69 \rightarrow 73.73 \rightarrow 53.87$ s—but it cancels once each pattern's metrics are expressed relative to its own HarmonyTransfer result: the baselines then stay between

Model	Workload	QPS	FIFO	EDF	Request-LSTF	EDF-AC	HarmonyTransfer	Gain (pp)
Llama-3.1-8B	DeepSearcher	1.50	69.6	85.9	85.9	89.6	**92.6**	+3.0
		2.00	18.5	17.8	16.3	54.1	**60.7**	+6.7
		3.00	4.4	1.5	1.5	8.1	**27.4**	+19.3
Qwen2.5-14B	DeepSearcher	0.90	78.5	84.4	**85.9**	84.4	**85.9**	+0.0
		1.20	5.9	1.5	1.5	45.2	**51.1**	+5.9
		1.50	3.7	1.5	0.7	25.2	**29.6**	+4.4
Llama-3.1-8B	LongAgent	0.45	18.0	72.0	67.0	95.0	**99.0**	+4.0
		0.60	7.0	18.0	17.0	75.0	**84.0**	+9.0
		0.90	5.0	15.0	15.0	60.0	**77.0**	+17.0
Qwen2.5-14B	LongAgent	0.45	7.0	13.0	10.0	75.0	**87.0**	+12.0
		0.60	5.0	15.0	13.0	62.0	**80.0**	+18.0
		0.90	5.0	19.0	16.0	52.0	**75.0**	+23.0
Llama-3.1-8B	DocQA	0.60	58.0	**85.0**	82.0	**85.0**	**85.0**	+0.0
		0.90	28.0	55.0	45.0	74.0	**77.0**	+3.0
		1.20	15.0	35.0	31.0	60.0	**75.0**	+15.0
Qwen2.5-14B	DocQA	0.60	27.0	50.0	48.0	73.0	**74.0**	+1.0
		0.90	14.0	33.0	33.0	59.0	**77.0**	+18.0
		1.20	12.0	24.0	22.0	32.0	**50.0**	+18.0

Table 5: Detailed results behind Figure 9. Every cell reports APG deadline attainment in percent for one (workload, model, arrival-rate) condition. The best value of each row is set in bold; "Gain" is the difference in percentage points between HarmonyTransfer-DDL and the best of the four baselines in that row.

Workload	Method	Seed 0	Seed 1	Seed 2	Mean	Std.
DeepSearcher	FIFO	10.01	16.91	11.02	12.65	3.72
DeepSearcher	Request-SJF	27.48	38.94	26.98	31.14	6.76
DeepSearcher	HarmonyTransfer	**9.12**	**16.08**	**9.42**	11.54	3.94
LongAgent	FIFO	58.69	73.73	53.87	62.10	10.36
LongAgent	Request-SJF	21.27	25.67	19.78	22.24	3.06
LongAgent	HarmonyTransfer	**18.90**	**23.68**	**18.19**	20.25	2.99
DocQA	FIFO	20.93	32.78	20.26	24.66	7.04
DocQA	Request-SJF	13.55	23.35	13.49	16.80	5.67
DocQA	HarmonyTransfer	**8.53**	**13.22**	**8.10**	9.95	2.84

Table 6: Average objective: mean APG completion time in seconds under three Poisson arrival patterns; lower is better. Bold marks the best value in each (workload, seed) column.

$1.10\times$ and $3.06\times$ HarmonyTransfer, with a cross-pattern standard deviation of at most 0.31. The residual spread sits on the baselines already closest to HarmonyTransfer, not on the winner.

C.1.2 Deadline Objective: the Arrival Pattern and the Deadline Assignment Vary Together

Table 7 reports the deadline objective, and here the seed reshuffles both the arrival pattern and the pairing of the $2\times$ / $5\times$ / $8\times$ deadlines with APGs. The mix of tight and loose deadlines is therefore unchanged, which inflates the across-seed spread of every method—EDF-AC on DeepSearcher moves $54.1 \rightarrow 25.9 \rightarrow 48.9\%$—without favouring any one of them. HarmonyTransfer attains the highest deadline attainment in all nine (workload, pattern) pairs; normalised by its own result, the baselines span 0.09 to 0.93 of it with a cross-pattern standard deviation of at most 0.17. Because that spread mixes arrival and deadline variation, it is an upper bound on what arrival variation alone would produce, and the dispersion again concentrates on the baselines closest to HarmonyTransfer (EDF and Request-LSTF on DocQA) rather than on the winner.

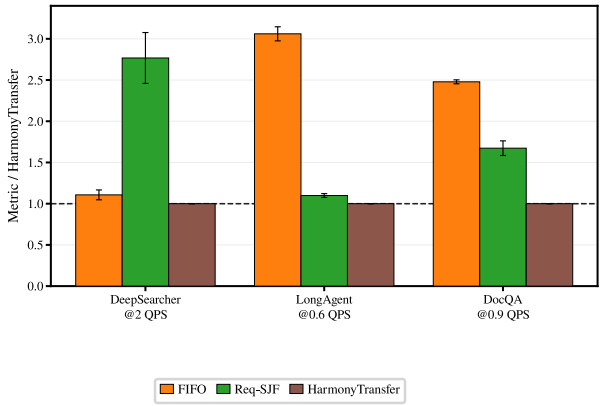


Figure 15: Robustness of the average objective, where the seed varies only the arrival pattern. Each pattern's mean APG completion time is normalised by that same pattern's HarmonyTransfer result, so a bar of one means parity with HarmonyTransfer. Bars are the mean over the three patterns and whiskers are one standard deviation over them; HarmonyTransfer is the normalisation reference and therefore carries no spread. Every baseline stays above one for all three workloads, and the whiskers are small relative to the distances, so the ranking and the size of the advantage both survive the change of arrival pattern.

Workload	Method	Seed 0	Seed 1	Seed 2	Mean	Std.
DeepSearcher	FIFO	19.3	9.6	20.0	16.3	5.8
DeepSearcher	EDF	17.8	5.2	12.6	11.9	6.3
DeepSearcher	Request-LSTF	17.0	5.2	12.6	11.6	6.0
DeepSearcher	EDF-AC	54.1	25.9	48.9	43.0	15.0
DeepSearcher	HarmonyTransfer	**60.7**	**41.5**	**61.5**	54.6	11.3
LongAgent	FIFO	7.0	7.0	8.0	7.3	0.6
LongAgent	EDF	18.0	11.0	25.0	18.0	7.0
LongAgent	Request-LSTF	18.0	9.0	21.0	16.0	6.2
LongAgent	EDF-AC	72.0	72.0	81.0	75.0	5.2
LongAgent	HarmonyTransfer	**84.0**	**85.0**	**87.0**	85.3	1.5
DocQA	FIFO	28.0	15.0	24.0	22.3	6.7
DocQA	EDF	55.0	27.0	46.0	42.7	14.3
DocQA	Request-LSTF	49.0	23.0	45.0	39.0	14.0
DocQA	EDF-AC	73.0	64.0	70.0	69.0	4.6
DocQA	HarmonyTransfer	**77.0**	**70.0**	**76.0**	74.3	3.8

Table 7: Deadline objective: APG deadline attainment in percent under three Poisson arrival patterns; higher is better. Bold marks the best value in each (workload, seed) column.

C.2 Uniform Deadline Assignment

Figure 13 changes the deadline assignment and nothing else. It replays the LongAgent cell of Section 5.2 on the same arrival vector, with the same 100 APGs and the same five methods, but gives every APG one shared relative deadline $D = 50$ s instead of a $2\times$ /$5\times$ /$8\times$ multiplier. The value sits just below the cell's median standalone completion time, so the request-level baselines are not excluded by construction and still attain 35–41%. Under the same deadline, HarmonyTransfer-DDL finishes its p90 APG at 48.8 s and attains 91%, and EDF-AC attains 79%; because only the assignment differs, the gap cannot come from the mix of tight and loose deadlines.

C.3 Bandwidth Sensitivity Cells

Figure 12 varies one quantity, the Mooncake RDMA rate, at 56 and 80 Gbps, with the model, workload, objective, arrival rate, seed, and deadline assignment held fixed and all six cells drawn from

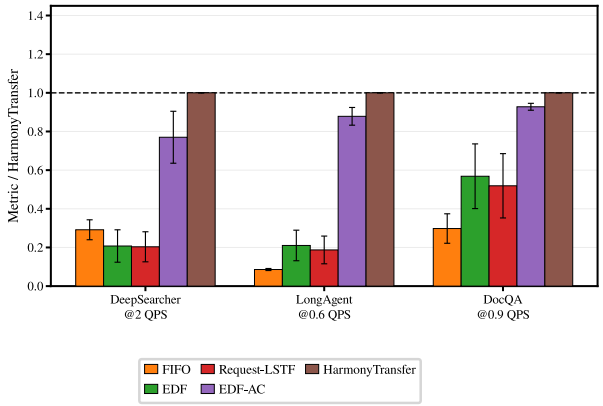


Figure 16: Robustness of the deadline objective, where the seed varies both the arrival pattern and the assignment of $2\times/5\times/8\times$ deadlines to APGs. Each pattern's deadline attainment is normalised by that same pattern's HarmonyTransfer result, so a bar of one means parity with HarmonyTransfer. Bars are the mean over the three patterns and whiskers are one standard deviation over them. Every baseline stays below one for all three workloads; because each whisker mixes the two sources of variation, it overstates what arrival variation alone would produce.

Policy	Live APGs	Live requests	Median (ms)	p95 (ms)	p99 (ms)	p99 / batch
FIFO	100	1,483	2.68	2.73	2.75	0.13×
FIFO	300	4,200	9.43	9.52	9.56	0.45×
FIFO	1,000	13,738	34.08	34.30	34.44	1.60×
EDF	100	1,483	2.97	3.08	3.15	0.15×
EDF	300	4,200	9.86	10.03	10.11	0.47×
EDF	1,000	13,738	38.74	39.04	39.17	1.82×
HarmonyTransfer-Mean	100	1,483	3.59	3.80	3.88	0.18×
HarmonyTransfer-Mean	300	4,200	10.88	11.17	11.31	0.53×
HarmonyTransfer-Mean	1,000	13,738	44.72	45.24	45.45	2.12×
HarmonyTransfer-DDL	100	1,483	0.07	0.07	0.08	**0.004**×
HarmonyTransfer-DDL	300	4,200	0.24	0.26	0.27	**0.012**×
HarmonyTransfer-DDL	1,000	13,738	0.93	0.96	0.97	**0.045**×

Table 8: Cost of one scheduling decision as the live APG population grows. "p99 / batch" compares the 99th percentile with the 21.5-ms transfer-batch budget. Only measured cells are shown; the population is held fixed while the decisions are timed.

one rerun batch. HarmonyTransfer-DDL leads on both metrics at both rates: mean APG completion time and deadline attainment are 50.5 s/7% (FIFO), 61.7 s/34% (EDF-AC), and 38.5 s/64% (HarmonyTransfer-DDL) at 56 Gbps, and $31.1/17$, $27.5/56$, and $20.7/67$ at 80 Gbps. Lowering the rate raises every method's latency without changing the ranking.

C.4 Scheduler Decision Cost

A scheduler that spends longer choosing a batch than the batch takes to transfer cannot keep the link busy, so we measure the decision itself. One decision is a single call into the native scheduler core on a batch-completion event, against a batch budget of 21.5 ms, the time one bounded transfer costs at 100 Gbps ($b_{\max} = 8$ chunks of 256 tokens). The harness drives the same native core that all experiments use. APG shapes are resampled from the recorded Llama-3.1-8B token statistics of the three workloads, and deadlines follow Eq. 13. L1 admission is left unconstrained, so every live APG remains a candidate, which is the worst case for the ordering search. Each cell takes 100 warm-up and 1,000 timed decisions at a fixed population, on one x86-64 core of a 24-core machine running Python 3.10.12.

Our workloads hold 135, 100 and 100 APGs, so the 100-APG row is the operating point of the evaluation. There a decision costs 0.07–3.6 ms, between 0.4% and 18% of the batch it schedules, and no policy comes near starving the link.

At 1,000 APGs, an order of magnitude beyond any trace we replay, the policies separate. HarmonyTransfer-DDL reaches 0.93 ms, 4.5% of a batch, while the other three take 34–45 ms and exceed a whole batch by 1.6–2.1×. The reason lies in what each decision has to build. The three order the shared link by materialising the complete order of live requests. They sort every live APG and then every live request, and the key they sort on is the demand estimate itself, so the estimate is recomputed throughout the sort. HarmonyTransfer-DDL instead evaluates one projection per APG and orders only the accepted and downgraded classes it has just formed. The marginal cost per live APG is 35.0, 39.8 and 45.7 us for FIFO, EDF and HarmonyTransfer-Mean against 1.0 us for HarmonyTransfer-DDL, and per live request 2.57, 2.95 and 3.41 us against 76 ns.

We read this as an implementation cost, not a property of the objectives. All four policies run in the same native core, but HarmonyTransfer-Mean evaluates the same aggregate-demand key as the baselines and is routed through their ordering routine; only the deadline policy has an APG-level path that avoids materialising the global order. At the scales we evaluate, every decision fits comfortably inside the batch it schedules, and the APG-level policy keeps that property an order of magnitude further out.

D Related Work and Limitations

D.1 Hierarchical KVCache and Transfer Scheduling

Reusing KVCache across GPU, host, and remote tiers is now standard in LLM serving (Kwon et al., 2023; Zheng et al., 2024; Cheng et al., 2025; Qin et al., 2025; Gao et al., 2024). Much of the work that follows lowers the cost of bringing that KVCache back to the GPU in one of five ways. It compresses and streams the cached KVCache (Liu et al., 2024); it recomputes part of the prefix so that loading overlaps computation (Yao et al., 2025; Jin et al., 2025); it restores hidden states instead of KVCache (Gao et al., 2025); it pipelines loading across the memory tiers (Xie et al., 2025); or it schedules the transfer paths of individual requests (Wu et al., 2026; Wang et al., 2026; Sun et al., 2026).

All of these decide *how* or *how fast* a given request's KVCache moves. HarmonyTransfer asks a different question: which request the shared link serves next, given the group that gates a stage. The two are complementary, and each mechanism above remains available underneath the APG-level order.

D.2 Application-Aware LLM Serving

Parrot (Lin et al., 2024), Ayo (Tan et al., 2025), Autellix (Luo et al., 2025), Hermes (Liu et al., 2025), and JITServe (Zhang et al., 2025) expose application structure such as dependencies, dataflow graphs, and program-level demand to the serving engine and use it to schedule GPU execution, treating the KVCache transfer path as a fixed per-request cost. The structure they carry is the dependency graph between the application's own steps; the resource they schedule is the GPU. APG carries the same kind of application knowledge one level down, to the transfer path, and schedules the shared lower-tier link rather than the GPU. The two are complementary along both axes. A system such as Parrot or Autellix could submit APGs for its fan-out steps and let HarmonyTransfer order the transfers that feed them. The demand signal those systems already track is what an APG membership declaration would be derived from.

D.3 Where HarmonyTransfer Departs from Coflow Scheduling

An APG is the structure of a coflow (Chowdhury & Stoica, 2012), so the question is what the coflow model does not cover. Four differences matter.

- *Demands are estimated, not declared.* A coflow scheduler is given the size of every flow. An APG arrives with member prompts and cached prefixes, and its remaining demand must

be inferred online as the sum of a predicted transfer time and a predicted prefill-compute time (Eq. 5). The estimate changes while the APG is in flight, which is why the bounded batch, not the job, is the unit at which the policy re-evaluates.

- *Completion follows a compute stage that overlaps transfer.* A coflow completes when its last byte arrives. An APG completes when its last member returns a first token, and that member's prefill computation can overlap the transfers of other members or of other APGs. This is why HarmonyTransfer-Mean chooses an APG and then its shortest member rather than its shortest transfer, and why FINISHIFNEXT advances one link cursor and orders members by prefill time.
- *The link is non-preemptive and served in bounded batches.* Coflow schedulers assume fluid, preemptible rate allocation and can give each flow a rate. Our $L3 \rightarrow L2$ path executes one bounded command at a time, so priority is enacted by choosing what to issue next and can only be revised at a batch boundary.
- *Deadlines are handled by projection, not admission.* Varys admits a deadline coflow only if it can finish on time and defers the rest. Under the two properties above, an APG cannot simply be refused: it has already arrived, its members are already qualified for execution, and the only decision left is ordering. HarmonyTransfer-DDL therefore projects feasibility continuously and demotes the APGs predicted to miss, without cancelling them.

The coflow literature also supplies the hardness backdrop for Section A.4. Coflow scheduling on a datacenter fabric is NP-hard (Roemer, 2006; Mastrolilli et al., 2010; Qiu et al., 2015), and the two-stage structure of an APG turns the scheduling problem into a two-machine flow shop rather than a set of independent jobs.

D.4 RELATION TO REQUEST-LEVEL SCHEDULING

CALVO (Wang et al., 2026) and DualPath (Wu et al., 2026) both aim to accelerate KVCache movement, but both schedule individual requests and neither is aware of an APG's collective prefill completion. At the ordering-policy layer, which is the layer that produces our baselines, their decision rules are already present in Section 5. Table 9 maps each one to the baseline that instantiates it: CALVO maps to Request-SJF and Request-LSTF, and DualPath's request ordering maps to FIFO. The end-to-end comparison is therefore APG scheduling against the request-level policies of both systems, and porting them as extra baselines would re-measure rules the paper already reports against. What remains outside the ordering layer—CALVO's dispatcher–executor pipeline and DualPath's path and engine placement—is a runtime and deployment mechanism rather than a competing ordering rule, and the latter has a single feasible value in our single-node, single-link testbed. Neither system subsumes HarmonyTransfer: they are the request-level policies our baselines instantiate, and the APG objective is what remains missing.

Prior mechanism	Baseline that instantiates it	Decision rule
CALVO (average TTFT)	Request-SJF	smallest $w_r(t) = R_r(t) + C_r$
CALVO (SLO attainment)	Request-LSTF	smallest slack $D_g - t - w_r(t)$
DualPath (request ordering)	FIFO	earliest-arriving APG first
DualPath (path/engine choice)	— (collapses)	one node, one $L3$ path

Table 9: HarmonyTransfer's request-level baselines already instantiate the ordering policies of CALVO and DualPath. The last row is a deployment-level decision with a single feasible value in our single-node, single-link testbed.

D.5 DISCUSSION AND LIMITATIONS

Scope of the evaluation. All experiments use one inference node and one shared $L3 \rightarrow L2$ link. When the members of an APG are served by different nodes or links, the finish-if-next projection must combine the queues of several links; we leave this coordination to future work.

APG declaration. HarmonyTransfer relies on the application to declare APG membership at submission. Frameworks with explicit fan-out primitives can do so at the call site, and dependency-

Workload	QPS	Decode tok	APG (s)	Dec. p50 (s)	Dec. p90 (s)	Share p50 / p90
DeepSearcher	1.5	1/1	2.75	0.02	0.02	1% / 1%
LongAgent	0.45	87/200	10.00	1.60	3.68	16% / 37%
DocQA	0.6	19/100	5.77	0.35	1.84	6% / 32%

Table 10: Residual fan-out decode at the lowest arrival rate of each workload (Llama-3.1-8B, HarmonyTransfer-Mean). "Decode tok" is the recorded p50/p90 member decode length, and the last column is the decode share of APG completion. The decode is charged once per group, because continuous batching advances every live member by one token per step, so a group's residual is its longest member decode rather than the sum over members.

tracking serving systems (Lin et al., 2024; Tan et al., 2025; Luo et al., 2025) can derive membership from the dependencies they already record. Partially known groups are not supported, and an APG becomes schedulable only when its complete member set has arrived.

Deadline assignment. Following prior work (Gu et al., 2023), deadlines are relative to each APG's interference-free completion time, so larger APGs receive larger deadlines. Absolute per-workload deadline budgets are not evaluated.

Committed work and estimation error. Priorities take effect only at batch boundaries and after the in-flight prefix of at most $q_{\max}$ batches drains, so a newly urgent APG can be delayed by up to $q_{\max}$ batch transfers. Both policies act on the demand estimates of Section 4.3; the prefill profile must be collected per model, and the sensitivity of the policies to estimation error has not been quantified.

Fairness and decode. HarmonyTransfer-Mean favors APGs with little remaining demand and may delay large APGs under sustained load, as any shortest-first rule does; HarmonyTransfer-DDL demotes infeasible APGs but never cancels them. Finally, APG completion is a prefill-side milestone: it coincides with fan-out stage completion for single-token outputs (as in DeepSearcher reranking), and otherwise precedes the decode phase, which HarmonyTransfer does not schedule.

E Motivation Details

E.1 Decode Residual After APG Completion in a Fan-out Stage

The testbed lacks the machines to serve decode separately at the scale of these fan-out stages, so we characterize the post-milestone decode analytically instead of measuring a decode-inclusive run. Under prefill–decode disaggregation the decode runs on the decode side, so it doesn't compete for the shared link.

Inter-token latency. We profile the per-token decode cost in isolation and average three repetitions. The measurement is decode-only: the first chunk of each request is discarded, so prefill lies outside the window. It runs on the same PRO6000D GPU, streaming from a vLLM OpenAI server with a 7,504-token prompt at concurrency 16, giving **18.4 ms/token**. Concurrency 16 exceeds the fan-out width of every stage whose decode is non-trivial (LongAgent 8, DocQA 6).

Projected decode time. The decode lengths are the `generated_tokens` recorded for the barrier each workload replays: 24-way reranking for DeepSearcher, where every member returns one token by construction, a worker barrier for LongAgent, and one request per document for DocQA. Charging each group its longest member decode at 18.4 ms/token gives Table 10 at the lowest arrival rate of each workload, which is the worst case for the share.

Table 10 shows that the residual decode is a small fraction of the milestone it follows. Taking the longest recorded decode, it reaches 16% at p50 and 37% at p90, so APG completion accounts for at least 86% of fan-out completion at p50 and 73% at p90. DeepSearcher has no residual at all, because its reranker's entire relevance decision is one token, so there the two endpoints coincide. The share falls as the arrival rate rises, because APG completion grows with transfer and prefill demand while the decode term is capped by the generation length $d_{\max}$ times the per-token cost τ. Decode is therefore additive: no transfer policy shortens it, and every policy pays it equally, so it cannot reorder the policies or overturn the comparison.