RUBATO: A RUN-AHEAD ASYNCHRONOUS COMPRESSION FRAMEWORK FOR AGENTIC RAG

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic retrieval-augmented generation interleaves reasoning and retrieval, causing the model input to grow throughout a request. Prompt-level context compression limits this growth, yet existing agent pipelines place each compression task between two agent steps, blocking subsequent reasoning or retrieval. In our measurements, compression under synchronous execution accounts for 43%–54% of single-request service time. Longer service times increase worker occupancy and queueing under concurrent serving. We present Rubato, a run-ahead asynchronous framework that removes trajectory compression from the request's critical path. A *context-construction rule (CCR)* combines the latest committed compressed context with subsequent uncompressed steps, allowing the agent to advance while compression runs in the background. A *prefix-monotone criterion (PMC)* accepts a result only for an active request and only when it extends the prefix covered by the committed context, preserving consistent model inputs despite late or out-of-order completions. Across four compression methods, six question-answering datasets, and three agent backbones, Rubato reduces dataset-averaged mean single-request latency by $\mathbf{1.69–2.32}\times$ and median job completion time under concurrent serving by $\mathbf{2.17–2.71}\times$ while preserving average answer quality. These results support treating compression placement as a serving decision separate from the compression method. Our code are openly accessible to support future research.

1 INTRODUCTION

Retrieval-augmented generation (RAG) has evolved from single-shot retrieve-then-read pipelines into agentic workflows in which a language model repeatedly reasons, formulates search queries, incorporates retrieved evidence, and decides whether further retrieval is needed. Surveys of RAG and LLM agents identify this shift toward adaptive retrieval and autonomous, multi-step interaction as a central direction for knowledge-intensive applications (Gao et al., 2023; Xi et al., 2025). Search-R1 (Jin et al., 2025), R1-Searcher (Song et al., 2025), ReSearch (Chen et al., 2025), and MEM1 (Zhou et al., 2026) follow this interaction pattern for open-domain and multi-hop question answering. Each retrieval round adds evidence to the trajectory, increasing the model input as the request progresses. Without compression, every subsequent model call must process the full accumulated trajectory, increasing inference cost and ultimately approaching the model's usable context window. Prompt-level context compression mitigates this growth through trajectory folding in FoldAgent (Sun et al., 2025), recursive summarisation in ReSum (Wu et al., 2025), compact rewriting in ACON (Kang et al., 2025), and explicit memory in MemAgent (Yu et al., 2026).

These methods execute compression synchronously *between* agent steps, blocking subsequent reasoning and retrieval. We call this execution pattern synchronous compression (SC), illustrated in Figure 1. On Qwen3.6-27B, compression accounts for 43%–54% of single-request service time when pooled by method (Figure 2). This time lies on the critical path and increases job completion time (JCT), with each request treated as one job. For the ACON request with median service time on 2WikiMultiHopQA, compression occupies roughly 100 of 144 seconds (Figure 11). Under concurrent serving, each of M workers processes one request at a time. Longer service therefore increases worker occupancy and queueing near capacity (Harchol-Balter, 2013). For ACON under SC with eight workers, the arrival rate and mean service time imply a peak demand of 4.62 times avail-

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.

able worker capacity. Queueing accounts for a median 36% of JCT. Without compression, demand remains below capacity at the same arrival rates, with a median queueing share of 0% (Table 8).

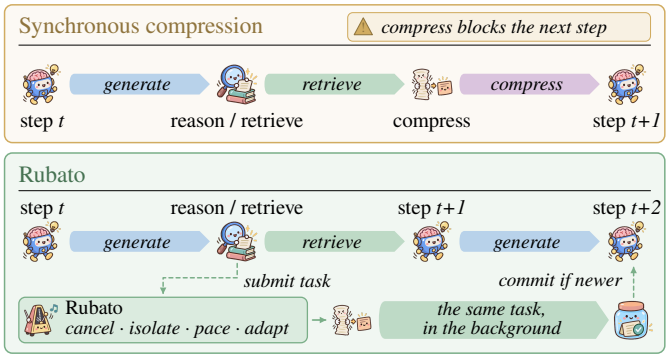

Figure 1: Execution timelines under synchronous compression (top) and Rubato (bottom).

Prior work reduces context cost through compression, preparation, and scheduling. HiAgent (Hu et al., 2025) and Parallel Context Compaction (Cim et al., 2026) improve context organisation or compression efficiency, but subsequent steps still await completion. RAPTOR (Sarthi et al., 2024) and ReadAgent (Lee et al., 2024) compress source documents. Sleep-time compute (Lin et al., 2025a) and LightMem (Fang et al., 2026) perform preparation or memory updates between requests, rather than compressing an active trajectory. InferCept (Abhyankar et al., 2024), Parrot (Lin et al., 2024), Autellix (Luo et al., 2025), and LLMCompiler (Kim et al., 2023) schedule model and tool calls subject to their dependencies, retaining the wait for compression-dependent steps. CoMem (Zhang et al., 2026) and AgentInfer (Lin et al., 2025b) overlap compression with agent inference using a trained memory model and a purpose-built compression module, respectively. These approaches either retain compression on the request's critical path or remove it through a compressor designed for overlapping execution. Enabling asynchronous execution of existing compressors would avoid retraining or redesign, but requires consistent context at every step and protection of concurrent requests from additional delay. This requirement motivates the following question.

Can existing compressors run asynchronously while preserving context and request latency?

Synchronous compression methods require a compressed context before the next agent step. For methods that compress a trajectory prefix, the corresponding raw steps remain available during compression. The agent can therefore use the latest committed compressed context and its raw suffix, then adopt a new result at a later step boundary. This execution pattern introduces two challenges. First, *consistency* requires aligning each result with the suffix beyond its coverage boundary and excluding stale results or results for terminated requests (Appendix A). An incorrect boundary can omit or duplicate steps, compromising context. Second, *contention* arises from residual tasks, blocking across requests, redundant computation, and bursts of released tasks under changing load (Table 2 and Figure 6).

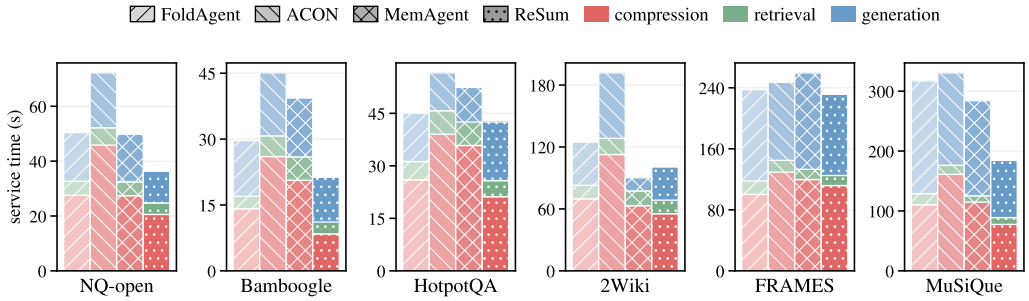

Figure 2: Mean single-request service time on Qwen3.6-27B, decomposed into compression, retrieval, and generation.

We propose **Rubato**, a Run-ahead Background Asynchronous Trajectory-compression Orchestration framework above the inference engine. The *context-construction rule (CCR)* combines the lat-

est committed compressed context with its raw suffix. The *prefix-monotone criterion (PMC)* accepts a result only if it advances the committed prefix boundary of an active request. Four orchestration rules control the resulting background workload. *Cancellation* skips queued tasks of terminated requests, while *isolation* separates task queues across requests and preserves execution order within each request. *Pacing* limits redundant computation and release bursts, and *adaptation* adjusts admission to changing load. Integration requires no additional training or changes to the compressor or inference engine.

We evaluate **Rubato** across four compression methods, six datasets, and three agent backbones. Under concurrent serving, **Rubato** reduces median JCT by $\mathbf{2.71}\times$ on Qwen3.6-27B, $\mathbf{2.17}\times$ on Gemma-4-31B-it, and $\mathbf{2.66}\times$ on GLM-4-32B. For single requests, it reduces mean latency by $\mathbf{1.69}\times$, $\mathbf{1.71}\times$, and $\mathbf{2.32}\times$, respectively. On Qwen3.6-27B, **Rubato** also reduces p95 JCT by $\mathbf{2.06}\times$ and increases the sustained completion rate at saturation by $\mathbf{52\%}$. Average answer quality is comparable to that under SC on all three backbones. The gains across these compressors support treating execution placement as a serving decision separate from context representation.

Our contributions are summarised below.

❶ ***Compression Placement.*** We quantify the blocking cost of four compression methods and establish conditions for overlapping prefix compression with subsequent agent steps without changing the compressor.

❷ ***System Design.*** We design **Rubato**, a run-ahead asynchronous framework that removes compression from the request's critical path. The *CCR* and *PMC* preserve complete, consistent model inputs, while the four orchestration rules, namely *cancellation*, *isolation*, *pacing*, and *adaptation*, regulate background compression under concurrent serving.

❸ ***Experimental Validation.*** Across four compression methods, six datasets, and three agent backbones, we show that **Rubato** preserves average answer quality while reducing both single-request latency and JCT under concurrent serving. Our analysis attributes these gains to removing blocking compression, controlling background tasks, and reducing queueing under load.

2 Related Work

Context Compression and Memory for Agents. Long-horizon agents accumulate reasoning, retrieved evidence, and tool observations in trajectories included in subsequent model calls. Prompt-level methods transform these trajectories differently, with FoldAgent (Sun et al., 2025) folding segments incrementally, ReSum (Wu et al., 2025) recursively summarising interaction history, ACON (Kang et al., 2025) rewriting trajectories compactly, and MemAgent (Yu et al., 2026) maintaining explicit memory. HiAgent (Hu et al., 2025) and Parallel Context Compaction (PCC) (Cim et al., 2026) also organise or compact long-horizon context. Other work operates below the trajectory level, compressing retrieved text (Xu et al., 2024), prompt tokens (Jiang et al., 2023), or the attention cache (Zhang et al., 2023; Xiao et al., 2024). These methods specify the retained information and its representation. **Rubato** controls compression timing during agent execution.

Agentic Retrieval and Multi-Step Execution. Search-R1 (Jin et al., 2025), R1-Searcher (Song et al., 2025), ReSearch (Chen et al., 2025), and MEM1 (Zhou et al., 2026) formulate RAG as a multi-step process that repeatedly reasons, retrieves, and updates the state passed to later calls. Their policies determine when to retrieve and what to retain, producing the growing contexts that compression must manage. Sleep-time compute (Lin et al., 2025a) shifts preparatory work to idle intervals between requests, rather than overlapping it with an active request. CoMem (Zhang et al., 2026) and AgentInfer (Lin et al., 2025b) overlap compression with agent execution through a separate memory model and a co-designed inference system, respectively. **Rubato** retains existing compressors and coordinates their background tasks across concurrent requests.

LLM Serving Systems. LLM serving systems optimise individual model calls through request scheduling in FastServe (Wu et al., 2023), VTC (Sheng et al., 2024), and Sarathi-Serve (Agrawal et al., 2024). DistServe (Zhong et al., 2024), Splitwise (Patel et al., 2024), and Mooncake (Qin et al., 2025) disaggregate prefill and decode, while SGLang (Zheng et al., 2024) and MemServe (Hu et al., 2024) reuse KV caches or prefixes. Beyond individual calls, InferCept (Abhyankar et al., 2024), Parrot (Lin et al., 2024), Teola (Tan et al., 2024), and Autellix (Luo et al., 2025) expose multi-call application structure, semantic dependencies, and tool interruptions to the serving system. **Rubato**

likewise operates at the application level but targets the dependency induced by context compression, allowing a compression task triggered by one agent step to execute during the next step, with its result becoming visible at a later step boundary. **Rubato** manages these dependencies above the inference engine, complementing systems that schedule the underlying model calls.

3 RUBATO

Figure 3 shows how **Rubato** connects the agent, background compression tasks, and the shared inference engine. The agent constructs each model input using the *CCR* (Section 3.1), while the *PMC* (Section 3.1) controls which compression results become visible at step boundaries. Four orchestration rules (Section 3.2) manage task lifetime, execution order, admission, and release. We first describe these context-consistency mechanisms and orchestration rules, then their deployment requirements and latency tradeoff (Section 3.3).

An agent request spans the processing of one user question. An agent step contains generation and retrieval, each performed through a call to the corresponding service. A compression task is a schedulable unit that invokes the compression procedure and tracks its input, execution, and result.

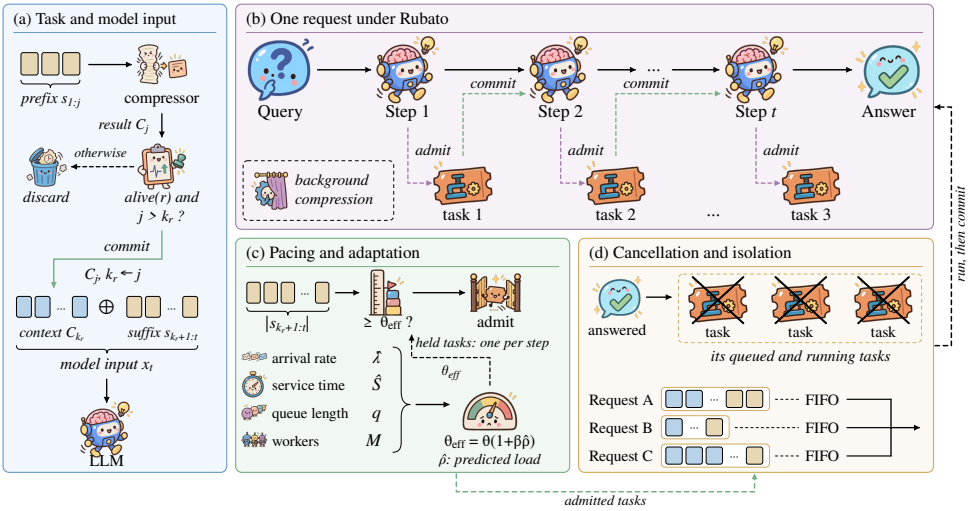

Figure 3: Overview of Rubato. The agent advances while compression runs in the background. The *CCR* and *PMC* determine model inputs and result visibility. Pacing and adaptation regulate task admission and release, while cancellation and isolation control task lifetime and execution order.

3.1 CONTEXT CONSISTENCY

The CCR constructs the context for each step, and the PMC determines whether a returned result can update it.

- **Context-Construction Rule.** A request r generates a trajectory $s_1, s_2, \ldots$, where each step s_t contains the model reasoning produced at step t and the evidence retrieved for that step. Let $s_{a:b} = s_a \oplus \cdots \oplus s_b$ denote the concatenation of steps a through b, and let $|s_{a:b}|$ denote its length in characters. Let C_k denote a compressed context covering the trajectory prefix $s_{1:k}$. This notation specifies coverage rather than the compressor's input or internal state. For each request, **Rubato** maintains the committed prefix boundary k_r, initialised to 0, and compressed context C_{k_r}, initialised to empty. At step t, the model input is
$$x_t = C_{k_r} \oplus s_{k_r+1:t}, \tag{1}$$
which concatenates the latest committed compressed context with the raw suffix generated after the prefix it summarises. Before the first result is committed, $x_t = s_{1:t}$ is the full raw trajectory. Thus every model call receives the complete trajectory regardless of background-compression state. Only the coverage boundary separating the compressed context from the raw suffix changes.
- **Prefix-Monotone Criterion.** At the start of step t, the agent reads x_t while holding the same lock used by commits. A result that arrives during the step therefore cannot alter the in-flight

model call and first becomes visible at step $t{+}1$. Compression results enter the active trajectory only at step boundaries. Each compression result identifies the prefix it summarises. When a compression task returns C_j, **Rubato** commits it only if

$$\text{commit } C_j \iff \text{alive}(r) \ \wedge \ j > k_r, \qquad \text{then } k_r \leftarrow j. \tag{2}$$

Results with $j \leq k_r$ are discarded as stale, while results for requests that have already completed are discarded as dead. Consequently, k_r is non-decreasing, stale results cannot overwrite newer committed contexts, and committing the same result twice is a no-op. A compression task that updates an existing compressed context reads the latest committed context when it starts. Serial execution in task creation order (Section 3.2) preserves dependencies between these tasks. Appendix A formalises these properties and describes the generated-schedule procedure used to validate them.

3.2 Orchestration Rules

The *CCR* and *PMC* determine model inputs and result visibility, but do not prevent background compression tasks from wasting capacity or blocking other requests. Contention can still arise from residual tasks, blocking across requests, redundant computation, and bursts of task releases as load changes. The following four orchestration rules control these sources of contention.

- **Task lifetime and execution order.** *Cancellation* targets the queued or running compression tasks of a completed or cancelled request. Such tasks would otherwise waste capacity required by active requests. When request r terminates, **Rubato** sets $\text{alive}(r) = \text{false}$ and closes its task queue. Each queued task checks this flag upon reaching the queue head and is skipped before occupying the compressor. An in-flight task is not preempted. It runs to completion, but its result fails Equation (2) and is discarded. *Isolation* prevents the head-of-line blocking caused when one long task in a shared queue delays every shorter task behind it. **Rubato** instead assigns each request a first-in-first-out lane, created when the request starts and closed when it terminates. A dedicated thread drains each lane in task creation order, limiting each task's wait to earlier tasks from the same request. The lanes access the compressor as independent streams that batching can interleave. Serial execution within each lane preserves dependencies between compression tasks.
- **Task admission and release.** *Pacing* avoids repeatedly processing nearly the same prefix at every step and dispatching tasks for short trajectories that may finish before benefiting from any result. **Rubato** admits the compression task generated at step t of request r only when the raw suffix reaches a threshold,

$$|s_{k_r+1:t}| \ \geq \ \theta, \tag{3}$$

and otherwise holds it. Held tasks are retained in task creation order and released at most one per agent step. When a step satisfies the threshold, **Rubato** dispatches its task together with the oldest held task, allowing long trajectories to drain the backlog gradually rather than in a burst. Any tasks still held when the request terminates remain undispatched, avoiding work that can no longer benefit the request. *Adaptation* addresses a fixed θ, which suits light load but diverts capacity from foreground generation under contention. **Rubato** therefore scales the threshold according to estimated agent-worker load, $\theta_{\text{eff}} = \theta\,(1+\beta\hat{\rho})$, where $\hat{\rho}$ is computed from estimated arrival rate, mean service time, and the number of requests waiting for a worker (Appendix B). Higher predicted load raises the threshold and reduces background admission. We use $\beta = 1.5$ throughout.

3.3 Deployment and Scope

Rubato supports compressors that replace a trajectory prefix with a compressed context and report its prefix boundary. Integration preserves the compression procedure, prompt, and hardware placement (Appendix C). The adaptation rule updates the admission threshold from estimated request arrival rate, mean service time, and worker-queue length (Appendix B).

For a fixed agent trajectory and retrieval times, let g_t and e_t denote generation and retrieval time at agent step t, and c_k the duration of a compression task covering the prefix $s_{1:k}$. Synchronous compression blocks on every task, yielding

$$\text{JCT}_{\text{sync}}(r) = \sum_t (g_t + e_t) + \sum_{k \in K} c_k, \tag{4}$$

Table 1: Rubato maintains SC-level answer quality and reduces single-request latency on every backbone and method. Judge scores are means over the six datasets of the concurrent runs. The difference column is the mean paired difference with a 95% bootstrap confidence interval over question-level pairs. Every lower bound exceeds the non-inferiority margin of -0.05. Speedup divides the dataset-averaged mean service time under SC by that under Rubato. The per-dataset range appears in brackets.

Method	Judge score		Difference [95% CI]	Speedup
	SC	Rubato		
Qwen3.6-27B				
FoldAgent	0.601	0.602	-0.001 [-0.019, $+0.017$]	1.68× [1.49–2.57]
ACON	0.603	0.615	$+0.012$ [-0.004, $+0.027$]	1.98× [1.70–3.84]
MemAgent	0.599	0.614	$+0.015$ [-0.004, $+0.033$]	1.59× [1.19–2.92]
ReSum	0.576	0.594	$+0.019$ [$+0.002$, $+0.036$]	1.47× [1.14–1.85]
All	**0.595**	**0.606**	$+0.012$ **[**$+0.003$**,** $+0.020$**]**	1.69× **[1.14–3.84]**
Gemma-4-31B-it				
FoldAgent	0.608	0.609	$+0.001$ [-0.013, $+0.015$]	1.71× [1.52–2.45]
ACON	0.628	0.633	$+0.004$ [-0.010, $+0.018$]	1.83× [1.37–3.57]
MemAgent	0.608	0.621	$+0.013$ [-0.006, $+0.031$]	1.67× [1.48–2.50]
ReSum	0.601	0.611	$+0.010$ [-0.004, $+0.026$]	1.61× [1.20–2.74]
All	**0.611**	**0.618**	$+0.007$ **[**-0.001**,** $+0.014$**]**	1.71× **[1.20–3.57]**
GLM-4-32B				
FoldAgent	0.578	0.574	-0.004 [-0.022, $+0.014$]	2.19× [1.87–2.52]
ACON	0.560	0.588	$+0.028$ [$+0.009$, $+0.047$]	2.53× [2.13–3.02]
MemAgent	0.554	0.578	$+0.024$ [$+0.003$, $+0.043$]	2.57× [1.66–4.22]
ReSum	0.562	0.581	$+0.018$ [-0.001, $+0.037$]	2.00× [1.68–2.23]
All	**0.564**	**0.580**	$+0.016$ **[**$+0.007$**,** $+0.026$**]**	2.32× **[1.66–4.22]**

where K contains the prefix boundaries compressed under SC. Under **Rubato**, retaining the raw suffix in the model input (Equation (1)) increases generation time to $g'_t \geq g_t$. Including task creation and enqueueing overhead w_r gives

$$\mathrm{JCT}_{\mathrm{Rubato}}(r) = \sum_t (g'_t + e_t) + w_r. \tag{5}$$

Holding resource contention fixed, the latency saving is

$$\mathrm{JCT}_{\mathrm{sync}} - \mathrm{JCT}_{\mathrm{Rubato}} = \sum_{k \in K} c_k - \sum_t (g'_t - g_t) - w_r. \tag{6}$$

Thus the eliminated blocking compression time must exceed additional generation time and orchestration overhead. Infrequent compression can yield a negative saving when the latter costs dominate. This model isolates the cost of removing blocking compression. Actual runs can differ in trajectory, executed tasks, and resource contention (Section 4.2).

4 Evaluation

4.1 Experimental Setup

We follow the official Search-R1 (Jin et al., 2025) protocol to evaluate FoldAgent (Sun et al., 2025), ACON (Kang et al., 2025), ReSum (Wu et al., 2025), and MemAgent (Yu et al., 2026) across six question-answering datasets and three agent backbones, namely Qwen3.6-27B, Gemma-4-31B-it, and GLM-4-32B, all served by the same vLLM (Kwon et al., 2023) build. Single-request runs use a closed-loop workload with one worker, so request latency equals service time. Concurrent runs use Poisson arrivals, $M{=}8$ workers, and a calibrated range of target utilisations. Each baseline retains its original prompts, compressor, and hardware placement. Answer quality is assessed by an LLM judge (Appendix D). Appendix C details the datasets, thresholds, arrival rates, and metrics.

4.2 Results

We organise the findings into five observations that separately examine answer quality, single-request latency, concurrent latency, orchestration, and robustness.

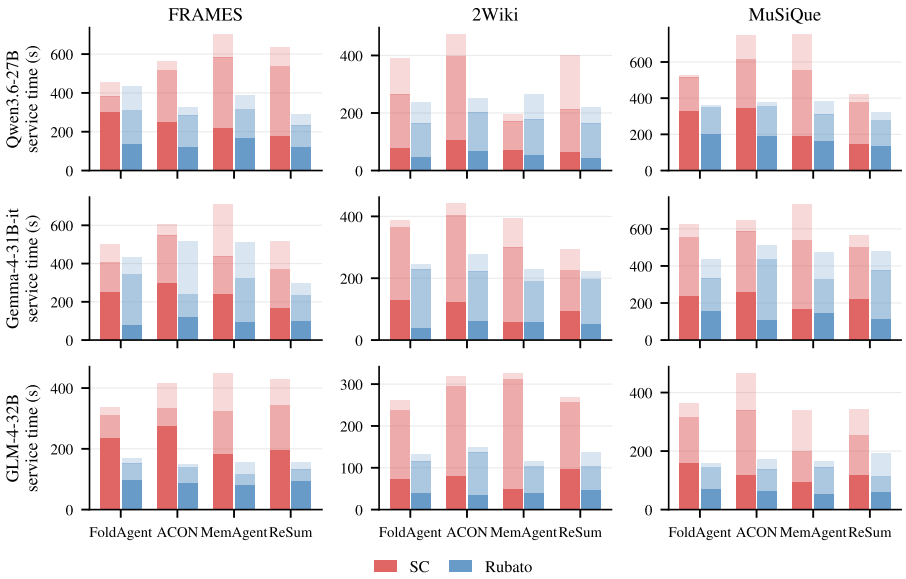

Figure 4: Rubato reduces single-request service time across backbones and datasets. Within the synchronous compression (SC) and Rubato bars, dark, medium, and light shades denote p50, p90, and p99 service-time percentiles, respectively, over 20 questions. Panels use independent scales.

Obs.❶ Rubato preserves answer quality. **Rubato** preserves answer quality across all three backbones (Table 1). Figure 7 shows the corresponding per-method intervals. The mean paired differences in judge score, measured as Rubato minus SC, are $+0.012$ $[+0.003, +0.020]$ on Qwen3.6-27B, $+0.007$ $[-0.001, +0.014]$ on Gemma-4-31B-it, and $+0.016$ $[+0.007, +0.026]$ on GLM-4-32B. Every per-method lower bound exceeds the pre-specified non-inferiority threshold of -0.05 (Appendix D). Under SC, the input combines a compressed context covering steps through k_r with the subsequent raw steps, representing the complete trajectory. **Rubato** changes only this boundary and omits no part of the trajectory.

Obs.❷ Rubato reduces single-request latency. With one request in flight, **Rubato** reduces mean single-request latency by $1.69\times$ on Qwen3.6-27B, $1.71\times$ on Gemma-4-31B-it, and $2.32\times$ on GLM-4-32B (Figure 4). Individual methods on Qwen3.6-27B achieve reductions of 1.47–$1.98\times$. Table 1 reports the per-backbone speedups, with full results in Appendix E. With only one request in flight, queueing is absent and asynchrony can remove only the compression term $\sum_{k\in K} c_k$ in Equation (6). The extra generation cost in Equation (6) offsets part of this saving, with each **Rubato** model call carrying the raw suffix $s_{k_r+1:t}$ and increasing generation time after a compression trigger (Figure 11). Orchestration overhead w_r remains below one second per request. The magnitude of the improvement reflects the amount of compression work removed from the critical path. On Qwen3.6-27B, ACON invokes compression most frequently and consequently achieves the largest reduction ($1.98\times$), whereas ReSum triggers least often and yields the smallest gain ($1.47\times$). When ReSum triggers at most once, the additional generation cost of carrying the raw suffix can exceed the blocking time eliminated by asynchronous compression, causing **Rubato** to be slower (Figure 11). On GLM-4-32B, however, ReSum triggers more frequently and its speedup rises to $2.00\times$. This crossover is consistent with the tradeoff formalised in Equation (6).

Obs.❸ Concurrency amplifies latency gains. Under concurrent serving, **Rubato** reduces median JCT by $2.71\times$ on Qwen3.6-27B, $2.17\times$ on Gemma-4-31B-it, and $2.66\times$ on GLM-4-32B, aggregated over methods, datasets, and arrival rates (Figure 5). On Qwen3.6-27B, **Rubato** also reduces p95 JCT by $2.06\times$ and sustains a $1.52\times$ higher completion rate at saturation than synchronous compression. Concurrent gains exceed single-request gains because blocking compression prolongs worker occupancy as well as request latency. At arrival rate λ, effective worker utilisation is $\rho_{\text{eff}} = \lambda \bar{S}/M$, where $\bar{S}$ denotes mean service time. Longer service increases utilisation and queueing near saturation. As Table 8 shows for Qwen3.6-27B, synchronous compression exceeds saturation in over twice as many dataset–arrival-rate combinations as **Rubato**. Shortening critical-path service time thus reduces queueing and increases the completion rate at saturation.

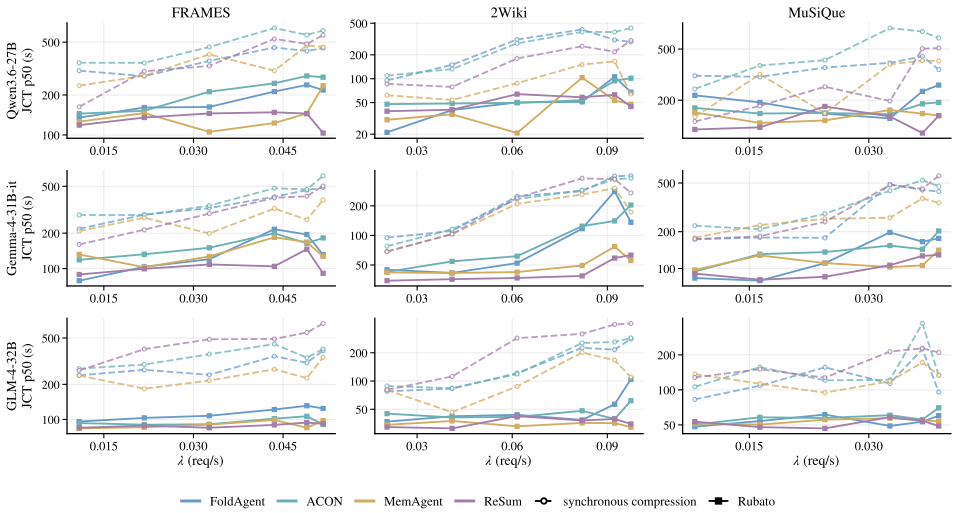

Figure 5: Under load the curves for synchronous compression rise earlier and more steeply than those for Rubato on every dataset, and the gap widens with arrival rate. Within each method colour, lighter dashed curves denote synchronous compression and darker solid curves denote Rubato. Median JCT against arrival rate at M=8.

Table 2: Ablations of the four orchestration rules, each on a workload where the problem it addresses appears. Column minima within each group are bold, including ties.

Configuration	p50 (s)	p95 (s)
Cancellation and isolation: MuSiQue, λ=0.02, M=4		
neither	215.4	455.9
isolation only	192.7	432.2
cancellation only	159.4	440.1
both (Rubato)	**132.8**	**365.2**
Pacing: MuSiQue, λ=0.02, M=4		
trigger at every step	261.3	453.1
held until the threshold (Rubato)	**230.8**	**384.2**
Adaptation: arrival ramp $\lambda \rightarrow 3.3\lambda$, M=4		
fixed threshold	19.1	167.5
fixed threshold, repeated run	18.6	160.9
adaptive threshold, β=1.5 (Rubato)	**15.7**	**144.6**
Adaptation: HotpotQA, λ: 0.2 $\rightarrow$ 0.06, M=4		
fixed threshold	**14.9**	166.1
adaptive threshold, repeated run	16.1	**133.3**
adaptive threshold, β=1.5 (Rubato)	**14.9**	137.6

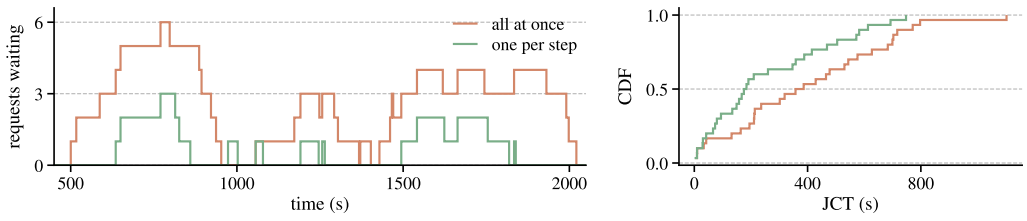

Figure 6: Releasing held compression tasks all at once makes requests queue for a worker. Releasing them one per step does not. The left panel shows requests waiting for a worker over the run on MuSiQue with $\lambda = 0.02$ and M=4. The right panel shows the JCT distribution of these two runs.

Obs.❹ Orchestration reduces contention. Table 2 evaluates each rule on a workload exposing its failure mode.[1] Cancellation, isolation, and pacing use MuSiQue at $\lambda = 0.02$ with four workers, where five to eight requests overlap and completed requests can leave residual compression tasks.

[1]Cancellation and isolation, pacing, and adaptation are separate experiments, compared only within each.

Table 3: Median JCT (s) for questions that trigger compression in 40 end-to-end runs per method and setting. Rubato entries include speedup over synchronous compression (SC). Longer retrieval reduces relative gains, while Rubato remains stable across worker counts.

	FoldAgent		ACON		MemAgent		ReSum	
Median JCT (s)	SC	Rubato	SC	Rubato	SC	Rubato	SC	Rubato
Retrieval time, M=8								
0 s	32.2	**6.7** (4.8×)	29.9	**6.6** (4.5×)	20.6	**6.9** (3.0×)	38.2	**5.6** (6.9×)
2.5 s	37.2	**17.6** (2.1×)	39.3	**12.2** (3.2×)	26.3	**12.1** (2.2×)	43.3	**14.0** (3.1×)
5 s	41.9	**24.8** (1.7×)	51.5	**17.5** (2.9×)	39.7	**18.1** (2.2×)	54.8	**22.2** (2.5×)
10 s	52.6	**35.4** (1.5×)	94.7	**31.9** (3.0×)	64.4	**32.9** (2.0×)	70.0	**32.3** (2.2×)
20 s	83.0	**55.7** (1.5×)	241.8	**85.4** (2.8×)	199.1	**87.2** (2.3×)	90.2	**52.3** (1.7×)
Workers M, retrieval 2.5 s, load per worker fixed								
M=1	35.3	**17.3** (2.0×)	29.7	**10.7** (2.8×)	25.0	**11.9** (2.1×)	43.2	**14.0** (3.1×)
M=2	38.3	**17.5** (2.2×)	76.3	**14.2** (5.4×)	44.9	**14.7** (3.1×)	50.0	**14.0** (3.6×)
M=4	36.2	**17.3** (2.1×)	52.0	**11.1** (4.7×)	37.1	**12.2** (3.0×)	43.3	**14.0** (3.1×)
M=8	37.2	**17.6** (2.1×)	39.3	**12.2** (3.2×)	26.3	**12.1** (2.2×)	43.3	**14.0** (3.1×)
M=16	40.1	**18.5** (2.2×)	38.3	**14.4** (2.7×)	25.4	**12.6** (2.0×)	43.3	**14.3** (3.0×)

Adaptation is evaluated under a mid-run arrival ramp from λ to 3.3λ and a drop to 0.3λ, testing adaptation to changing contention. Disabling both cancellation and isolation incurs a larger penalty than either alone, with cancellation removing orphaned tasks and isolation confining blocking to the originating request. Without either rule, a residual task in a shared queue can obstruct otherwise independent requests. Figure 6 isolates the release component of pacing under the same workload. Releasing all held tasks at once produces a burst that saturates the agent inference engine and slows agent steps, yielding a p50 JCT of 357 s and a p50 worker wait of 111 s. Releasing one held task per agent step eliminates this queueing and lowers p50 JCT to 175 s. Pacing therefore combines delayed admission with rate-limited release. Under the arrival drop, adaptation reduces p95 JCT from 166 s to 133–138 s across two runs, with p50 JCT of 14.9–16.1 s compared with 14.9 s for the fixed threshold.

Obs.❺ Latency gains persist across retrieval times and worker counts. Table 3 reports end-to-end measurements from the serving system while varying retrieval time and worker count. The measured runs above use 2.5 s retrieval and eight workers. Longer retrieval preserves the absolute saving but reduces its relative contribution to JCT. FoldAgent, for example, saves 17–27 s at every setting, yet its speedup decreases from $4.8\times$ at zero retrieval time to $1.5\times$ at 20 s. Varying the worker count has little effect on **Rubato**, whereas synchronous compression is slowest at two workers for ACON, MemAgent, and ReSum, when one long request occupies half of the available capacity.

5 Conclusion

Prompt-level trajectory compression is typically executed synchronously between agent steps, placing roughly half of end-to-end service time on the critical path. **Rubato** instead runs compression asynchronously while the agent advances from the latest committed compressed context and raw suffix. Across three backbones, four compression methods, and six question-answering datasets, this placement reduces mean single-request latency by 1.69–$2.32\times$ and median JCT under concurrent serving by 2.17–$2.71\times$ while preserving answer quality. These results identify execution placement as a serving decision for compressors that replace trajectory prefixes.

6 Limitations

Asynchronous execution does not eliminate the computational cost of executed compression tasks. The orchestration rules can reduce unnecessary work by controlling admission and skipping tasks after request termination. Overall gains therefore reflect both execution overlap and task orchestration. When compression is infrequent, as for ReSum on short trajectories, the additional prefill induced by a longer raw suffix can exceed the eliminated blocking latency (Equation (6)). Furthermore, our empirical scope is limited to one agent-serving node, three backbones of comparable scale, one retrieval stack, and question-answering workloads. Extending **Rubato** to other agent paradigms beyond agentic RAG and evaluating its applicability remains a direction for future empirical research.

Ethics Statement

This work evaluates serving-time orchestration on public question-answering datasets and open-weight models. It introduces no new data collection, no human subjects, and no personally identifiable information. Compression changes what context a model sees, so a compressed trajectory could in principle drop evidence that matters to a user. We therefore report answer quality alongside every latency result.

Reproducibility Statement

Our code are openly accessible to support future research.

The four compression baselines use their official prompts and templates. Their configurations, the trigger thresholds, and the hardware placement of every compressor are specified in Appendix C. Section 4.1 states the inference engine, models, judge, datasets, and load model. The contract properties of Appendix A are validated by an inference-engine-free test harness over randomised schedules, making that part of the system reproducible without any GPU. Every reported number is computed from recorded per-question and per-step traces by the scripts that also emit the tables, keeping tables, figures, and prose tied to one source.

The Use of Large Language Models

Large language models were used as writing and coding assistants for drafting and revising prose, translating the manuscript for the authors' review, generating and refining plotting and figure-layout code, and reviewing drafts for consistency. The research questions, system design, experiments, and analysis are the authors' own, and every number in the paper is produced by the scripts described in the Reproducibility Statement and was checked by the authors.

References

Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. Infercept: Efficient intercept support for augmented large language model inference. *arXiv preprint arXiv:2402.01869*, 2024.

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 *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, pp. 117–134, 2024.

Mingyang Chen, Tianpeng Li, Haoze Sun, Yijie Zhou, Chenzheng Zhu, Haofen Wang, Jeff Z Pan, Wen Zhang, Huajun Chen, Fan Yang, et al. Research: Learning to reason with search for llms via reinforcement learning. *arXiv preprint arXiv:2503.19470*, 2(3):7, 2025.

Musa Cim, Burak Topcu, Chita Das, and Mahmut Taylan Kandemir. Parallel context compaction for long-horizon llm agent serving. *arXiv preprint arXiv:2605.23296*, 2026.

Jizhan Fang, Xinle Deng, Haoming Xu, Ziyan Jiang, Yuqi Tang, Ziwen Xu, Shumin Deng, Yunzhi Yao, Mengru Wang, Shuofei Qiao, et al. Lightmem: Lightweight and efficient memory-augmented generation. In *International Conference on Learning Representations*, volume 2026, pp. 98706–98729, 2026.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023.

Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, et al. Memserve: Context caching for disaggregated llm serving with elastic memory pool. *arXiv preprint arXiv:2406.17565*, 2024.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32779–32798, 2025.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 13358–13376, 2023.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*, 2025.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. An llm compiler for parallel function calling. *arXiv preprint arXiv:2312.04511*, 2023.

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*, pp. 611–626, 2023.

Kuang-Huei Lee, Xinyun Chen, Hiroki Furuta, John Canny, and Ian Fischer. A human-inspired reading agent with gist memory of very long contexts. *arXiv preprint arXiv:2402.09727*, 2024.

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, 2024.

Kevin Lin, Charlie Snell, Yu Wang, Charles Packer, Sarah Wooders, Ion Stoica, and Joseph E Gonzalez. Sleep-time compute: Beyond inference scaling at test-time. *arXiv preprint arXiv:2504.13171*, 2025a.

Weizhe Lin, Hui-Ling Zhen, Shuai Yang, Xian Wang, Renxi Liu, Hanting Chen, Wangze Zhang, Chuansai Zhou, Yiming Li, Chen Chen, et al. Towards efficient agents: A co-design of inference architecture and system. *arXiv preprint arXiv:2512.18337*, 2025b.

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

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. IEEE, 2024.

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, 2025.

Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie, and Christopher Manning. Raptor: Recursive abstractive processing for tree-organized retrieval. In *International Conference on Learning Representations*, volume 2024, pp. 32628–32649, 2024.

Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 965–988, 2024.

Huatong Song, Jinhao Jiang, Yingqian Min, Jie Chen, Zhipeng Chen, Wayne Xin Zhao, Lei Fang, and Ji-Rong Wen. R1-searcher: Incentivizing the search capability in llms via reinforcement learning. *arXiv preprint arXiv:2503.05592*, 2025.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon llm agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025.

Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. Teola: Towards end-to-end optimization of llm-based applications. *arXiv preprint arXiv:2407.00326*, 2024.

Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, et al. Resum: Unlocking long-horizon search intelligence via context summarization. *arXiv preprint arXiv:2509.13313*, 2025.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China information sciences*, 68(2):121101, 2025.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, volume 2024, pp. 21875–21895, 2024.

Fangyuan Xu, Weijia Shi, and Eunsol Choi. Recomp: Improving retrieval-augmented lms with context compression and selective augmentation. In *International Conference on Learning Representations*, volume 2024, pp. 43478–43502, 2024.

Hongli Yu, Tinghong Chen, Jiangtao Feng, Jiangjie Chen, Weinan Dai, Qiying Yu, Ya-Qin Zhang, Wei-Ying Ma, Jingjing Liu, Mingxuan Wang, et al. Memagent: Reshaping long-context llm with multi-conv rl-based memory agent. In *International Conference on Learning Representations*, volume 2026, pp. 39458–39486, 2026.

Yuwei Zhang, Chengyu Dong, Shuowei Jin, Changlong Yu, Hejie Cui, Hongye Jin, Xinyang Zhang, Hamed Bonab, Colin Lockard, Jianshu Chen, et al. Comem: Context management with a decoupled long-context model. *arXiv preprint arXiv:2605.30842*, 2026.

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. *Advances in neural information processing systems*, 36:34661–34710, 2023.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

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

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, 2024.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Bryan Kian Hsiang Low, and Paul Liang. Mem1: Learning to synergize memory and reasoning for efficient long-horizon agents. In *International Conference on Learning Representations*, volume 2026, pp. 58413–58438, 2026.

A Context-Transition Properties and Validation

This section makes the context-transition guarantees of Section 3.1 explicit and describes the generated-schedule procedure used to validate them.

Each compression result identifies the trajectory prefix it summarises. **Rubato** adopts the result only if the request remains active and the prefix extends strictly beyond the prefix currently in use (Equation (2)). Every other case is a no-op. This rule implies the following six checkable properties.

P-perm (permutation invariance). For a fixed set of returned results with distinct prefix boundaries committed while the request remains active, the final committed context has the largest boundary regardless of commit order. This property does not imply identical intermediate model inputs or generated trajectories.

P-atomic (no torn context). The commit lock ensures that each step is represented exactly once, either in the compressed context or in the raw suffix. A reader cannot observe a partially published update.

P-cancel (termination excludes late work). Once a request is marked inactive, the alive condition in Equation (2) rejects subsequent commits for that request.

P-idem (idempotent commit). Reapplying the same commit is a no-op, preventing a retried delivery from applying a result twice.

P-mono (monotonic coverage). The committed prefix boundary never decreases. Steps already covered by the compressed context cannot re-enter the raw suffix.

P-crash (fault isolation). A failed compression task neither terminates its worker nor alters the context state.

Validation does not require an inference engine. We stub generation and feed each property randomised interleavings of task creation, completion, commit, and termination, checking the corresponding invariant after every event. All 200 generated schedules per property pass without violation. We last re-verified the current implementation on August 5, 2026.

B Adaptation

The pacing rule admits a compression task when the raw suffix reaches a fixed threshold. The adaptation rule scales this threshold using an estimate of agent-worker load,

$$\theta_{\text{eff}} \;=\; \theta\,(1+\beta\,\hat{\rho}), \qquad \hat{\rho} \;=\; \frac{\hat{\lambda}\,\hat{S}}{M} + \frac{q}{M}, \tag{7}$$

where $\hat{\lambda}$ and $\hat{S}$ are running estimates of arrival rate and mean service time, M is the number of agent workers, and q is the number of requests waiting for a worker. The serving loop maintains exponentially weighted averages of inter-arrival gaps and measured service times and reads q directly from the arrival queue. The first term estimates offered load in worker-equivalents, while the second incorporates the current backlog. Thus, $\hat{\rho}$ combines estimated demand on agent workers with the observed request backlog. **Rubato** recomputes the threshold at every admission decision, setting it to θ at $\hat{\rho} = 0$ and to 2.5θ at $\hat{\rho} = 1$ when $\beta = 1.5$. Under the mid-run arrival ramp of Table 2, the adaptive threshold lowers p50 JCT from 19.1 s and 18.6 s in two fixed-threshold runs to 15.7 s, and p95 JCT from 160.9–167.5 s to 144.6 s.

C Compression Baseline Integration

All baselines adopt a common five-stage integration process, comprising the compression condition, input construction, compression execution, result assessment, and prompt reconstruction. Each

method retains its original prompt, compressor, and hardware placement. ReSum uses its official prompt with a 1024-token trigger. MemAgent uses its official template with a 1500-character chunk budget. ACON and FoldAgent retain their published settings. The method-specific configurations are listed below.

FoldAgent. We execute its folding prompt on the agent inference engine, without a separate compressor, using the published trigger settings.

ACON. We run the released implementation and prompts on the agent inference engine with the published context threshold and retention setting. A separate Python environment isolates its dependencies.

MemAgent. We use the official memory-update template, insert the question into its designated slot, and retain the 1500-character chunk budget. Its compression inference engine uses two GPUs on a separate host, preserving the reference hardware allocation.

ReSum. We retain the official summarisation prompt and released 30B reasoning compressor, which runs on its own GPU pair as in the reference implementation. The trigger threshold is 1024 tokens and the summary budget is 4000 tokens.

The two placements use identical compression procedures, models, and hardware. Rubato changes both the blocking dependency and the admission, execution order, and cancellation of background compression tasks.

Datasets, load model, and metrics. We evaluate NQ-open, Bamboogle, HotpotQA, 2WikiMultiHopQA (2Wiki), FRAMES, and MuSiQue, using 20 questions per dataset for single-request runs and 40 for concurrent runs. Concurrent requests arrive as a Poisson process with rate $\lambda = \rho M/\bar{S}_0$, where ρ is the target utilisation and $\bar{S}_0$ is the per-dataset mean service time measured during closed-loop calibration. This normalisation gives comparable nominal load across datasets whose trajectory lengths differ by an order of magnitude. The resulting arrival rates span 0.008–1.52 requests per second.

We report judge score, median and p95 JCT, sustained completion rate (throughput), and queueing delay. The compression profile in Figure 2 and all ablations use Qwen3.6-27B. The arrival-drop ablation uses 30 questions. The orchestration-rule ablations (Table 2 and Figure 6) run FoldAgent, which compresses on the agent inference engine. Matched runs share questions, seeds, retrieval inputs, compression and model settings, and hardware allocation. Single-request speedup divides the dataset-averaged mean service time under SC by the corresponding mean under Rubato. Concurrent results use the median over method, dataset, and arrival-rate combinations. Generated interleavings separately validate the consistency of context updates (Appendix A).

Hardware placement. ACON and FoldAgent compress on the agent inference engine. MemAgent and ReSum use the same pair of A6000 GPUs on a second host in separate runs. ReSum uses its released 30B compression model. vLLM serves the agent backbone with tensor parallelism across two RTX A6000 GPUs. Retrieval runs as a shared service on a third host, and the judge uses hardware with no experiment traffic.

D Judge Protocol and Quality Analysis

One fixed model scores every answer using the LLM-as-a-judge protocol (Zheng et al., 2023). Keeping the judge constant makes scores comparable across experimental phases. Changing it would require rescoring the entire matrix. Retaining every question-level verdict supports both the paired analysis below and the answer-rate decomposition in Table 5.

Judge model, decoding, and prompt. We use Qwen3-4B-Instruct-2507 as the judge, served with vLLM 0.21.0 (Kwon et al., 2023) in float16 with prefix caching and a 20,480-token context window. The server runs on a single Quadro RTX 5000 on a dedicated host with no experiment traffic. Each verdict is produced by one chat completion containing a single user turn and no system prompt. We use greedy decoding (temperature 0) and allow at most four completion tokens. The model does not expose a thinking mode, which we nevertheless explicitly disable in the request. Before constructing the prompt, we truncate the question to 600 characters and the prediction to 300 characters and insert

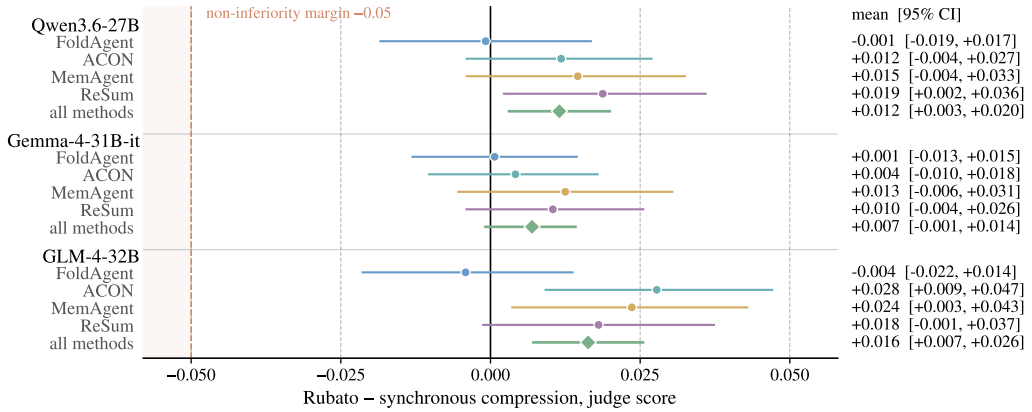

Figure 7: Every method's lower confidence bound exceeds the non-inferiority margin on every backbone. Points show mean paired judge-score differences (Rubato minus synchronous compression) with 95% bootstrap confidence intervals over question-level pairs from concurrent runs. The dashed line marks the margin. Diamonds pool the four methods within each backbone.

the gold answers as a list. No retrieved evidence is shown. The judge compares the prediction only against the gold answers. We assign a verdict of 1 when the lower-cased completion contains `yes`, and 0 otherwise. Empty predictions are assigned a verdict of 0 without a judge call. This convention separates answer rate from judge-on-answered in Table 5. A run's judge score is the mean verdict across its questions. The prompt is reproduced verbatim below.

```
You are grading a question-answering system.
Question: {q}
Gold answer(s): {golds}
Model answer: {pred}
Does the model answer match any gold answer in meaning
(ignore wording, casing, date format differences)? Reply
with exactly one word: yes or no.
```

Confidence intervals and non-inferiority. Within each method, dataset, and arrival-rate condition, every question appears once under synchronous compression and once under **Rubato**. This produces 5,560, 5,760, and 5,760 paired verdicts for Qwen3.6-27B, Gemma-4-31B-it, and GLM-4-32B, respectively. The estimand is the mean paired difference (**Rubato** − synchronous compression), with percentile-bootstrap intervals from 5,000 question-level resamples. We pre-specify the margin at $\delta = 0.05$ and declare non-inferiority when the lower confidence bound exceeds $-\delta$.

After pooling within each backbone, the differences are $+0.012$ $[+0.003, +0.020]$, $+0.007$ $[-0.001, +0.014]$, and $+0.016$ $[+0.007, +0.026]$. Every lower bound exceeds the margin, and two lower bounds exceed zero. Across the twelve method–backbone pairs, all lower bounds exceed -0.022. The widest interval is that of FoldAgent on GLM-4-32B (Figure 7). The judge-score difference between the SC and Rubato columns in Table 1 ($+0.011$) uses weights over method and dataset combinations, whereas the paired estimate weights individual questions. MemAgent on NQ-open is the only Qwen3.6-27B method–dataset combination outside the margin ($0.78 \rightarrow 0.72$). Every combination on Gemma-4-31B-it and GLM-4-32B remains inside it. Aggregate scores by method and backbone appear in Table 4.

E Full Results

Tables 6 and 7 break down the latency results from Section 4.2 by backbone and method for concurrent and single-request serving, respectively.

Speedup relative to the compression share. Pooled per method on Qwen3.6-27B, on-path compression accounts for 43–54% of service time (Figure 2), which bounds the single-request speedup

Table 4: Mean judge score under the same questions and judge configuration, per backbone and method, averaged over the six datasets and the arrival rates of the concurrent experiment. The Rubato column shows the difference from SC in parentheses.

Backbone	Method	SC	Rubato
Qwen3.6-27B	FoldAgent	0.601	0.602 (+0.001)
	ACON	0.603	0.615 (+0.012)
	ReSum	0.576	0.594 (+0.019)
	MemAgent	0.599	0.614 (+0.015)
	All	0.595	0.606 (+0.011)
Gemma-4-31B-it	FoldAgent	0.608	0.609 (+0.001)
	ACON	0.628	0.633 (+0.004)
	ReSum	0.601	0.611 (+0.010)
	MemAgent	0.608	0.621 (+0.013)
	All	0.611	0.618 (+0.007)
GLM-4-32B	FoldAgent	0.578	0.574 (−0.004)
	ACON	0.560	0.588 (+0.028)
	ReSum	0.562	0.581 (+0.018)
	MemAgent	0.554	0.578 (+0.024)
	All	0.564	0.580 (+0.016)

Table 5: Quality decomposed as raw judge = answer rate × judge on answered. Judge-on-answered is a selected subset (a placement that answers more questions answers harder ones). Bold = better placement on raw judge.

		raw judge		answer rate		judge on answered	
Dataset	Method	synchronous compression	Rubato	synchronous compression	Rubato	synchronous compression	Rubato
NQ-open	reference	0.704		98%		0.722	
	FoldAgent	0.688	**0.696**	95%	98%	0.727	0.714
	ACON	0.750	**0.767**	91%	99%	0.826	0.773
	MemAgent	**0.775**	0.725	98%	99%	0.791	0.731
	ReSum	0.629	**0.683**	95%	99%	0.662	0.689
HotpotQA	reference	0.654		99%		0.662	
	FoldAgent	**0.725**	0.692	98%	98%	0.740	0.706
	ACON	**0.754**	0.713	98%	99%	0.774	0.718
	MemAgent	0.713	**0.721**	98%	98%	0.728	0.739
	ReSum	**0.692**	0.654	99%	99%	0.700	0.662
Bamboogle	reference	0.800		100%		0.803	
	FoldAgent	0.808	**0.817**	98%	99%	0.829	0.827
	ACON	**0.800**	0.783	95%	97%	0.846	0.807
	MemAgent	**0.796**	0.792	97%	99%	0.823	0.802
	ReSum	**0.833**	0.825	97%	98%	0.858	0.839
2Wiki	reference	0.637		95%		0.671	
	FoldAgent	0.596	**0.604**	85%	90%	0.701	0.674
	ACON	0.600	**0.650**	80%	89%	0.746	0.732
	MemAgent	0.554	**0.646**	76%	92%	0.731	0.705
	ReSum	0.492	**0.604**	73%	92%	0.670	0.656
FRAMES	reference	0.429		70%		0.609	
	FoldAgent	0.442	**0.475**	64%	77%	0.693	0.620
	ACON	**0.400**	0.383	64%	67%	0.627	0.571
	MemAgent	0.429	**0.450**	72%	76%	0.599	0.590
	ReSum	0.421	**0.438**	75%	70%	0.564	0.621
MuSiQue	reference	0.283		76%		0.374	
	FoldAgent	**0.354**	0.329	74%	76%	0.480	0.434
	ACON	0.312	**0.392**	79%	78%	0.397	0.505
	MemAgent	0.329	**0.350**	79%	80%	0.416	0.438
	ReSum	**0.388**	0.362	87%	78%	0.445	0.463

at 1.8–2.2× if both placements followed the same trajectory. The measured 1.47–1.98× stays below this bound. Per dataset, five of the 24 method–dataset cells exceed their own bound, and in each of them **Rubato** takes fewer agent steps per question. ACON on NQ-open achieves 3.84× against a bound of 2.74× while its mean step count falls from 3.55 to 3.00, and MemAgent on Bamboogle achieves 2.92× against 2.10× with 3.15 versus 2.65 steps. With 20 questions per cell, such differences in trajectory length dominate the per-dataset ratio.

Table 6: Ratio of SC to Rubato for median JCT under concurrent serving, reported by backbone and method over six datasets and six arrival rates. The table reports the median and range across those runs. Values above one favour Rubato. Gemma-4-31B-it and GLM-4-32B use the shared compression inference engine one at a time to avoid asymmetric contention between simultaneous runs (Appendix E).

Backbone	Method	Median	Range
Qwen3.6-27B	FoldAgent	3.00×	1.32–9.16
	ACON	4.06×	1.41–10.06
	ReSum	2.35×	1.10–6.80
	MemAgent	2.41×	0.92–10.25
	All	2.71×	0.92–10.25
Gemma-4-31B-it	FoldAgent	2.38×	1.00–4.78
	ACON	2.17×	1.13–3.81
	ReSum	1.51×	0.58–9.84
	MemAgent	2.06×	1.49–5.27
	All	2.17×	0.58–9.84
GLM-4-32B	FoldAgent	2.30×	1.49–5.88
	ACON	2.52×	1.40–6.69
	ReSum	3.56×	1.56–11.75
	MemAgent	2.78×	1.24–21.85
	All	2.66×	1.24–21.85

Table 7: Single request (closed loop, one question at a time, 20 questions per dataset). Ratio of SC to Rubato for per-question service time per backbone and method, with the median and range over the six datasets.

Backbone	Method	Median	Range
Qwen3.6-27B	FoldAgent	1.84×	1.49–2.57
	ACON	2.17×	1.70–3.84
	MemAgent	1.67×	1.19–2.92
	ReSum	1.52×	1.14–1.85
	All	1.78×	1.14–3.84
Gemma-4-31B-it	FoldAgent	1.97×	1.52–2.45
	ACON	1.84×	1.37–3.57
	MemAgent	1.75×	1.48–2.50
	ReSum	1.56×	1.20–2.74
	All	1.75×	1.20–3.57
GLM-4-32B	FoldAgent	2.20×	1.87–2.52
	ACON	2.45×	2.13–3.02
	MemAgent	2.75×	1.66–4.22
	ReSum	1.94×	1.68–2.23
	All	2.22×	1.66–4.22

Across backbones. **Rubato** yields lower median JCT for every method on all three backbones, although the ranking of methods changes. On Qwen3.6-27B, ACON achieves the largest reduction ($4.06\times$) and ReSum the smallest ($2.35\times$). On GLM-4-32B, ReSum achieves the largest ($3.56\times$). As on Qwen3.6-27B, ReSum has the smallest reduction on Gemma-4-31B-it ($1.51\times$) and is slower under **Rubato** on several short-dataset runs, with a minimum ratio of $0.58\times$. Every other method–backbone pair contains at most one such reversal. As discussed in Section 4.2, these changes correspond to how often each method triggers compression on each backbone.

Effective utilisation. Table 8 reports measured ρ_{eff} and the fraction of JCT spent queueing under each placement on Qwen3.6-27B. These measurements support the utilisation argument in Section 4.2. Synchronous compression exceeds saturation in 8–20 of each method's 36 dataset–arrival-rate combinations, and **Rubato** exceeds it in 0–9. The median queue share under **Rubato** is zero because most of its combinations stay below saturation. In those that exceed it, requests also queue, with a median queue share of 35% for ACON and 37% for MemAgent.

Table 8: Effective utilisation $\rho_{\text{eff}} = \lambda \bar{S}/M$ per placement on Qwen3.6-27B, over the 36 dataset–arrival-rate combinations of each method at $M{=}8$. The nominal arrival rate is identical under both placements by construction. ρ_{eff} differs because SC increases service time. Columns report the range of ρ_{eff}, the number of combinations beyond saturation ($\rho_{\text{eff}}{>}1$), and the median share of JCT spent queueing.

	ρ_{eff}		past saturation		queue share	
Method	SC	Rubato	SC	Rubato	SC	Rubato
FoldAgent	0.13–3.17	0.04–1.16	20/36	4/36	27%	0%
ACON	0.18–4.62	0.04–1.54	20/36	9/36	36%	0%
MemAgent	0.12–4.45	0.06–2.09	18/36	7/36	32%	0%
ReSum	0.06–1.92	0.03–0.97	8/36	0/36	9%	0%
No compression	0.04–0.91	–	0/36	–	0%	–

Table 9: Sustained completion rate (requests per second) at saturation on Qwen3.6-27B, per dataset and method, under no compression, SC, and Rubato, with the ratios of Rubato to SC and to no compression.

Dataset	Method	no compression	SC	Rubato	Rubato/SC	Rubato/none
NQ-open	FoldAgent	0.193	0.093	0.166	1.78	0.86
	ACON		0.069	0.181	2.63	0.94
	MemAgent		0.084	0.126	1.51	0.65
	ReSum		0.098	0.180	1.84	0.93
HotpotQA	FoldAgent	0.052	0.075	0.098	1.31	1.89
	ACON		0.055	0.121	2.18	2.33
	MemAgent		0.046	0.077	1.67	1.49
	ReSum		0.120	0.081	0.68	1.57
Bamboogle	FoldAgent	0.133	0.082	0.123	1.50	0.92
	ACON		0.051	0.110	2.16	0.82
	MemAgent		0.047	0.144	3.09	1.08
	ReSum		0.157	0.187	1.19	1.40
2Wiki	FoldAgent	0.054	0.041	0.063	1.52	1.16
	ACON		0.032	0.048	1.48	0.88
	MemAgent		0.051	0.064	1.26	1.18
	ReSum		0.037	0.065	1.75	1.20
FRAMES	FoldAgent	0.032	0.023	0.034	1.47	1.08
	ACON		0.020	0.033	1.63	1.02
	MemAgent		0.023	0.034	1.52	1.08
	ReSum		0.023	0.037	1.63	1.16
MuSiQue	FoldAgent	0.025	0.022	0.026	1.18	1.06
	ACON		0.019	0.027	1.40	1.09
	MemAgent		0.022	0.028	1.27	1.12
	ReSum		0.022	0.025	1.16	1.02

Throughput at saturation. Table 9 reports the sustained completion rate for every method and dataset on Qwen3.6-27B. **Rubato** exceeds synchronous compression in every cell except ReSum on HotpotQA. For every method, it also exceeds the no-compression rate on HotpotQA, FRAMES, and MuSiQue. It remains below that rate on NQ-open for all methods, on Bamboogle for ACON and FoldAgent, and on 2Wiki for ACON. Across saturated runs, the median improvement over synchronous compression is $1.52\times$. Over all loads, the ratio falls to $1.40\times$, because, below capacity, throughput under both placements is determined by the arrival rate.

One backbone at a time on the shared compression inference engine. Two methods use a separate compression inference engine shared across backbones. Our initial Gemma-4-31B-it and GLM-4-32B experiments accessed this inference engine concurrently, and their drivers traversed the same experiment grid in the same order. The runs therefore remained in phase, with a run using synchronous compression overlapping another such run 41% of the time, compared with 12% for two **Rubato** runs.

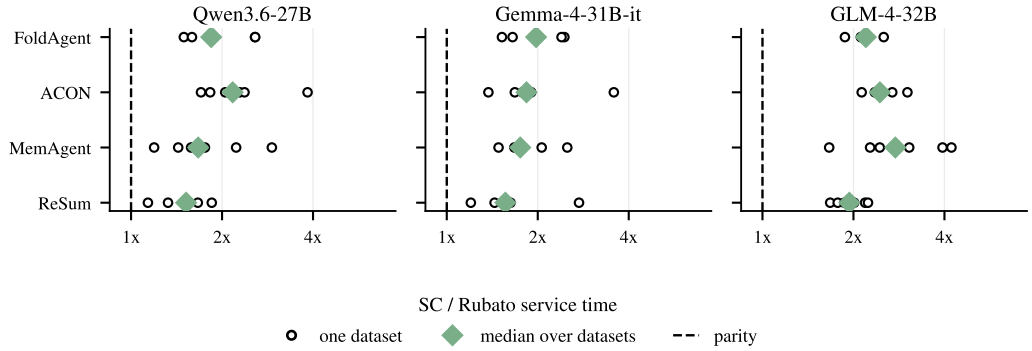

Figure 8: Single-request speedup varies more across datasets than across methods and exceeds parity on every dataset. Each hollow point is the ratio of synchronous compression to Rubato for mean per-question service time on one dataset (closed loop, 20 questions). Filled diamonds show the median ratio across datasets. The dashed line marks parity. SC denotes synchronous compression.

Table 10: Median ratio of SC to Rubato for JCT under concurrent serving, reported by dataset with the range over arrival rates in parentheses. Values above one favour Rubato.

Method	NQ-open	HotpotQA	2Wiki	Bamboogle	MuSiQue	FRAMES
FoldAgent	3.0 (2.3–9.2)	3.5 (3.2–3.6)	4.4 (2.9–8.2)	4.0 (2.6–5.5)	1.7 (1.3–2.7)	2.1 (1.7–2.3)
ACON	3.3 (3.1–10.1)	5.0 (4.1–6.4)	4.2 (2.3–7.2)	4.3 (4.0–6.0)	2.9 (1.4–4.7)	2.3 (2.0–2.6)
ReSum	1.5 (1.1–4.5)	2.1 (1.8–2.8)	3.2 (1.9–6.8)	2.6 (2.5–3.8)	1.4 (1.2–4.5)	2.8 (1.4–5.5)
MemAgent	3.5 (2.1–10.2)	2.2 (2.0–5.1)	1.8 (1.3–4.2)	2.5 (2.3–5.2)	2.3 (0.9–2.7)	2.2 (1.9–3.8)

Table 11: Answer quality per dataset, shown as judge scores under synchronous compression and Rubato. Averaged over the arrival rates of the concurrent experiment.

Method	NQ-open	HotpotQA	2Wiki	Bamboogle	MuSiQue	FRAMES
FoldAgent	0.69→0.70	0.72→0.69	0.60→0.60	0.81→0.82	0.35→0.33	0.44→0.47
ACON	0.75→0.77	0.75→0.71	0.60→0.65	0.80→0.78	0.31→0.39	0.40→0.38
ReSum	0.63→0.68	0.69→0.65	0.49→0.60	0.83→0.82	0.39→0.36	0.42→0.44
MemAgent	0.78→0.72	0.71→0.72	0.55→0.65	0.80→0.79	0.33→0.35	0.43→0.45

This asymmetry biases the comparison because contention occurs mainly while synchronous compression is active, and the resulting queueing delay is attributed to that baseline. We therefore reran every affected combination with at most one backbone using the shared compression inference engine and report these isolated measurements. For ReSum on Gemma-4-31B-it, the median ratio of synchronous compression to **Rubato** falls from $1.84\times$ to $1.51\times$, and **Rubato** becomes slower on three additional runs. On GLM-4-32B, the ratio falls from $3.92\times$ to $3.56\times$, and the single reversal no longer occurs. Contention on the shared compression inference engine thus inflated the gain on one backbone and introduced a spurious reversal on the other. Every Gemma-4-31B-it and GLM-4-32B entry in Table 6 comes from the one-backbone-at-a-time reruns.

Per-dataset breakdowns. These breakdowns show where the aggregate results arise. Figure 8 compares closed-loop speedups across all datasets, while Figures 9 and 10 extend Figures 4 and 5 to the three omitted datasets. The corresponding per-dataset concurrent JCT, judge score, and compression cost appear in Tables 10 to 12.

Retrieval time and worker count. The absolute savings behind Table 3 are 17–27 s for FoldAgent and 29–38 s for ReSum at every retrieval time. For ACON and MemAgent the saving grows with retrieval time, from 23 s to 156 s and from 14 s to 112 s, reflecting a steep rise in their synchronous-compression JCT at 10 and 20 s. Across worker counts, synchronous compression is slowest at $M{=}2$ for ACON, MemAgent, and ReSum, while FoldAgent stays within 35–40 s.

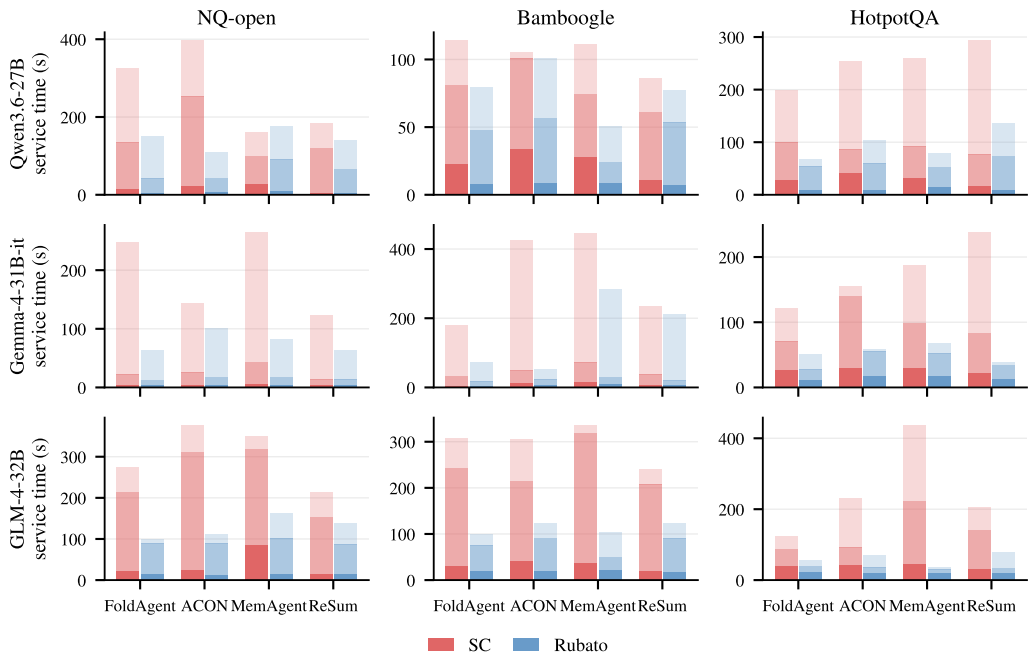

Figure 9: The three datasets omitted from Figure 4 exhibit the same relative performance pattern. Within the synchronous compression (SC) and Rubato bars, dark, medium, and light shades denote p50, p90, and p99 service-time percentiles, respectively. Panels use independent scales.

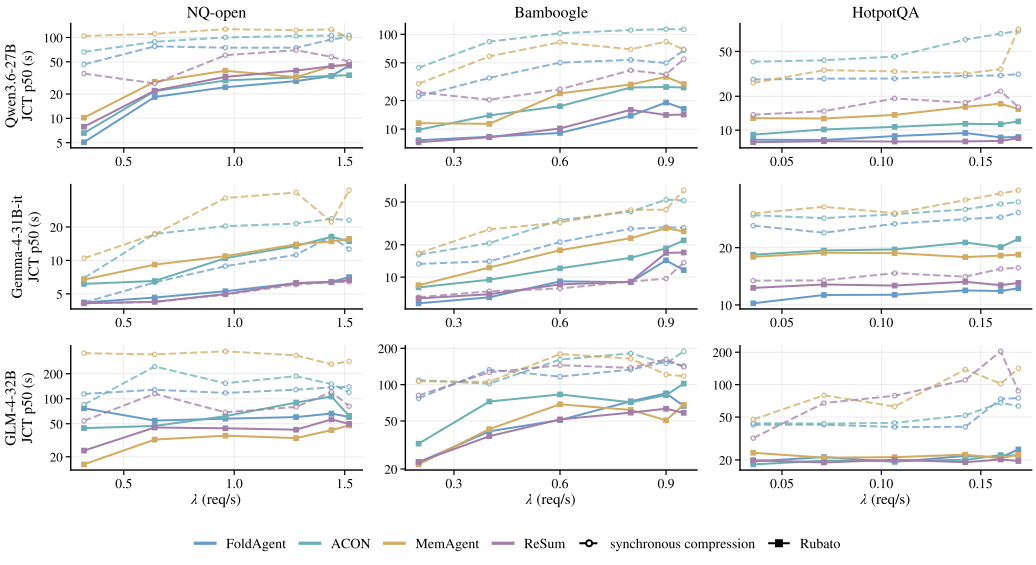

Figure 10: On the three datasets omitted from Figure 5, the gap also widens with load. Within each method colour, lighter dashed curves denote synchronous compression and darker solid curves denote Rubato. Panels use independent scales.

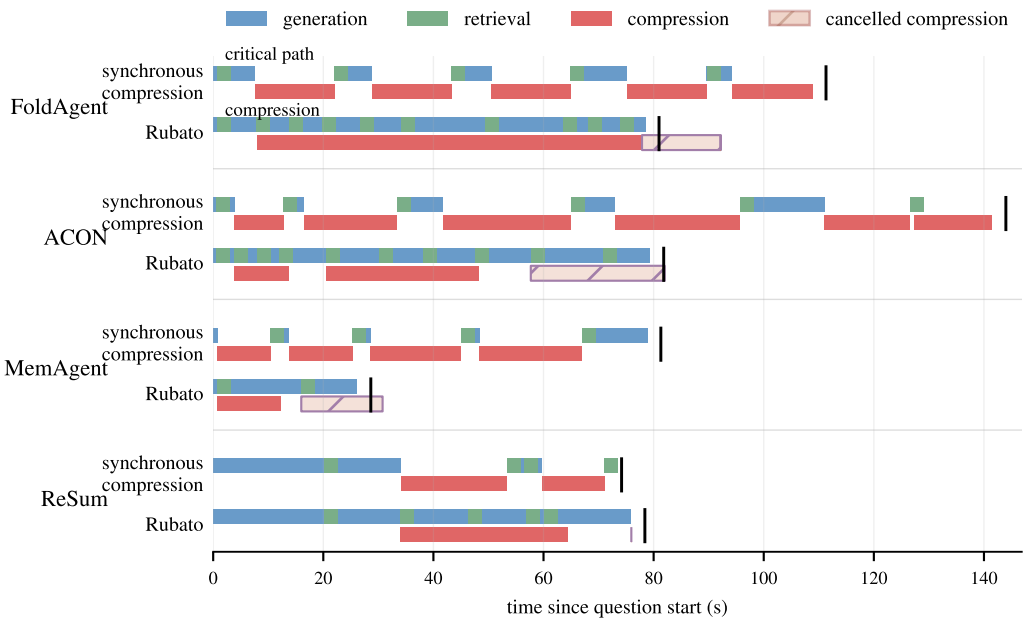

Figure 11: Retaining the raw suffix in the model input lengthens generation after a trigger under Rubato, yet requests usually finish earlier once compression no longer blocks an agent step. Each timeline depicts the 2WikiMultiHopQA question with median service time for its method, aligned at the first step. Lanes with synchronous compression place it *between* generation steps. Rubato runs it *underneath*, with cancelled superseded tasks hatched. The longer post-trigger generation corresponds to the second term of Equation (6). ReSum takes one additional step and finishes slightly later. Its rare trigger leaves little blocking time to remove.

Table 12: Compression cost by dataset, reported as blocking compression time per request (s) / compression tasks per request and averaged over concurrent arrival rates.

Method	NQ-open	HotpotQA	2Wiki	Bamboogle	MuSiQue	FRAMES
FoldAgent	40 / 2.4	22 / 1.7	90 / 5.5	20 / 1.5	98 / 6.5	117 / 7.6
ACON	53 / 2.4	45 / 2.4	113 / 5.5	44 / 2.2	166 / 6.3	175 / 6.9
ReSum	38 / 0.8	26 / 0.7	103 / 2.4	23 / 0.6	139 / 3.1	182 / 3.5
MemAgent	42 / 2.7	38 / 2.9	76 / 5.8	30 / 2.4	116 / 6.5	143 / 7.7