AGENTWEAVE: Efficient Distributed Agent Serving via Flow Decomposition

Kaibin Guo[1], **Pengtu Li**[1], **Zicong Hong**[2]*, **Zhiyuan Fang**[1],
Wuhui Chen[1], **Rachid Guerraoui**[2], **Anne-Marie Kermarrec**[2]
[1]Sun Yat-sen University, [2]EPFL
{guokb,lipt5,fangzhy27}@mail2.sysu.edu.cn,chenwuh@mail.sysu.edu.cn,
{zicong.hong,rachid.guerraoui,anne-marie.kermarrec}@epfl.ch

Abstract

Agent applications are evolving into workflows composed of large language model (LLM) calls and tool invocations, termed *modules*. Existing distributed agent systems typically adopt a disaggregated architecture, deploying different modules on separate devices. While this enables pipeline parallelism, imbalanced module latencies introduce severe pipeline bubbles that limit throughput. A natural alternative is an aggregated architecture that co-deploys all modules across all devices, eliminating bubbles by keeping every device continuously active. However, this introduces *request blocking*: short requests are forced to progress in lockstep with longer ones in the same batch, and newly arriving requests must wait until the entire ongoing batch completes the full workflow. To address this, we propose AGENTWEAVE, an efficient distributed serving system for agent applications built on the aggregated architecture. Its core mechanism, *flow decomposition*, decomposes long agent requests into fine-grained execution units while preserving agent semantics, allowing short and newly arriving requests to proceed without being stalled by long-running ones. Complementing this, a *flow-adaptive scheduler* dynamically balances the throughput gains of decomposition against its scheduling overhead. Experiments across four representative agent applications demonstrate that AGENTWEAVE improves throughput by 1.28–$1.81\times$, reduces average latency by 2.08–$2.45\times$, and reduces P90 latency by 1.56–$2.04\times$ over state-of-the-art baselines.

1 Introduction

Large language model (LLM)-based agents [50] have emerged as a compelling paradigm for tackling complex real-world tasks. Yet building capable agents is non-trivial, as a single LLM [1, 8] alone is often insufficient: it may lack up-to-date knowledge [14, 39], struggle to reason over long contexts [33, 6, 57], and cannot reliably interact with external data sources [45, 29]. To overcome these limitations, modern agents augment LLMs with specialized tools that enable them to navigate external environments and solve complex tasks more effectively [45, 53, 43]. We adopt *module* as a unified abstraction encompassing both LLM calls and tool invocations; under this abstraction, a modern agent is naturally organized as a workflow of modules.

Existing distributed serving systems [47, 34, 31, 25, 59, 28] adopt a *disaggregated* architecture for agent serving, where different modules are deployed on separate devices and managed independently. This design naturally forms a pipeline that allows modules to overlap execution across requests. However, it fundamentally suffers from *pipeline bubbles*: because module latencies are highly imbalanced in practice, faster modules are forced to idle while waiting for slower ones, leaving substantial dead time in the pipeline. Figure 1(a) illustrates this architecture, where three modules

*Corresponding author.

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

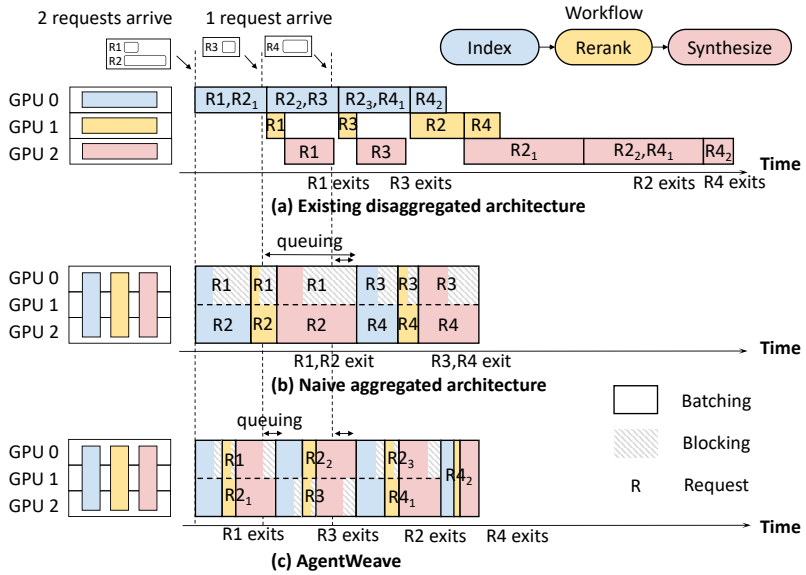


Figure 1: Comparison of execution paradigms for agent serving on simplified RAG, drawn to realistic scale, consisting of the indexing, reranking, and synthesizing modules that dominate RAG latency.

are deployed on $GPU0$, $GPU1$, and $GPU2$, respectively. When short requests $R1$, $R3$ and long requests $R2$, $R4$ arrive, modules execute in a pipelined fashion, yet the resulting pipeline bubbles severely degrade overall serving performance.

A natural alternative is the *aggregated* architecture, which co-deploys all modules across all devices, fully utilizing every device at each step of the workflow. This eliminates pipeline-induced bubbles and accelerates per-request execution. As illustrated in Figure 1(b), all three modules are deployed across all devices. $R1$ and $R2$ are batched and processed through three modules, exiting together after completing the entire agent workflow. $R3$ and $R4$ also follow the same pattern. Compared with Figure 1(a), requests complete earlier under this design, yielding lower latency and higher throughput.

Despite these advantages, the aggregated architecture introduces a critical inefficiency: *request blocking*. As shown in Figure 1(b), although short request $R1$ requires far less computation in the indexing stage, it cannot advance to reranking earlier, because all devices remain committed to indexing for the entire batch until long request $R2$ finishes. Consequently, short requests are forced to exit in lockstep with long ones, while newly arriving requests such as $R3$ must wait until the ongoing batch ($R1$, $R2$) completes the full workflow before they can begin execution.

We therefore present AGENTWEAVE, an efficient distributed serving system for agent applications. Our key idea is to decompose long requests at the agent level. As shown in Figure 1(c), we decompose $R2$ into smaller execution units $R2_1$, $R2_2$, and $R2_3$. So $R1$ can exit after executing with $R2_1$, rather than being coupled with the entire $R2$. New requests such as $R3$ can also enter between two units of a long request and complete earlier. This reduces request blocking while retaining the benefits of the aggregated architecture. To practically realize this idea, we further address two technical challenges.

Challenge 1: How to decompose a request without breaking agent semantics. This requires preserving two semantics constraints: execution workflow, i.e., the module execution structure shown in Figure 2; and complete-input requirement, where a module requires a complete input to produce correct output, e.g., reranking requires the full retrieved documents to select the global top-k results.

AGENTWEAVE introduces *flow decomposition* to decompose each request while preserving agent semantics. It develops an abstraction called *flows* (i.e., module's input or output) as the decomposable unit, and rewrites the module dependency into a flow dependency graph. It decomposes flows along valid flow dependency, and inserts supplement dependencies to satisfy complete-input requirement.

Challenge 2: How to control decomposition cost. Decomposition reduces blocking, but also introduces scheduling and flow-management costs. Over-decomposition can accumulate cost and significantly increase the latency of long requests.

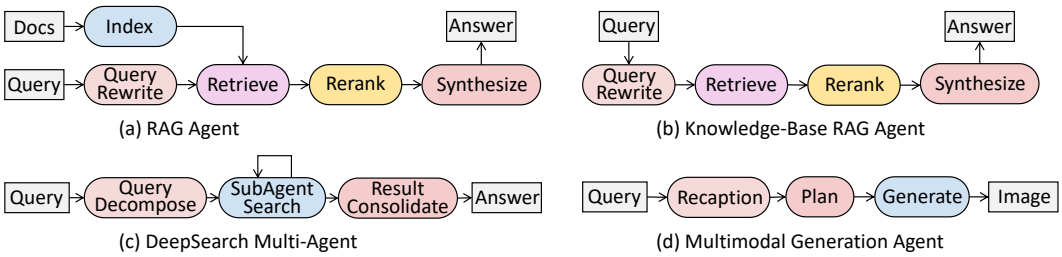


Figure 2: Representative agent applications, showing that modern agents are typically organized as a workflow of modules with inter-module execution dependencies.

AGENTWEAVE introduces a *flow-adaptive scheduler* to control this cost. Instead of using a fixed decomposition size, it determines the size online by balancing the throughput gain against the cost penalty, and then constructs the corresponding batch for execution. This guarantees the decomposition gains without increasing long-tail latency.

We implement a prototype of AGENTWEAVE on top of SGLang[59], and evaluate it on four representative agent applications, across diverse model-hardware configurations. Evaluation shows that it improves throughput by 1.28–$1.81\times$, reduces average and P90 latency by 2.08–$2.45\times$ and 1.56–$2.04\times$, respectively. The code is available at https://github.com/Guodanding/agentweave.

2 Background and Motivation

2.1 Agent Applications

Agents have become increasingly popular in real-world applications. Figure 2 shows four representative examples (see Appendix A for details). In a retrieval-augmented generation (RAG) agent [47, 26, 14], an LLM rewrites the query, invokes an embedding tool to index documents and retrieve relevant ones, a reranking tool to refine results, and finally synthesizes the answer, thereby enabling better knowledge-grounded generation. A knowledge-based (KB) RAG agent [26, 19] follows the RAG agent but retrieves candidates from a pre-built knowledge base instead of indexing new documents. In a DeepSearch agent [5, 21], an LLM decomposes a query into subplans, coordinates search subagents tools to collect and summarize evidence, and consolidates the results into a structured response. In a multimodal generation agent [52, 30], an LLM refines the query, plans the scene layout, and invokes a diffusion generator to produce the final image.

2.2 Inefficiency of Disaggregated Architecture

Existing distributed serving systems [34, 47, 25, 59] adopt a disaggregated architecture in Figure 1(a) for agent applications, and primarily pursue two optimization directions: workflow orchestration and module serving. The first direction [47] optimizes workflow by splitting modules into finer-grained units and overlapping their execution to shorten the latency of a request. The second direction [34, 31, 25, 59] optimizes individual modules serving, most commonly the LLM-based modules, by reducing queueing delay or eliminating redundant computation and memory.

However, both directions still suffer from pipeline bubbles [22, 40] caused by imbalanced module latencies. Figure 3 illustrates this imbalance: execution time is distributed unevenly across modules in all four agent applications, with latency skew reaching 41.5% in KB-RAG, where LLM-based synthesis accounts for 65.7% and reranking by the rerank model accounts for only 24.2%. This imbalance leaves severe bubble time in the pipeline, reducing effective overlap and limiting throughput.

2.3 Aggregated Architecture: Benefits and Inefficiency

This motivates an aggregated architecture in Figure 1(b), which co-deploys all modules of an agent across all devices. It removes pipeline and fully utilizes devices to process each module, therefore avoiding bubbles and additionally accelerating the request execution. These benefits can reduce latency and improve system throughput.

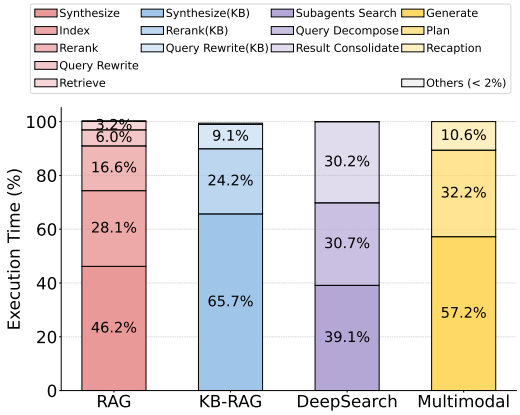


Figure 3: Execution time breakdown of representative agent applications, including RAG, KB-RAG, DeepSearch and Multimodal. The results illustrate module latency imbalance.

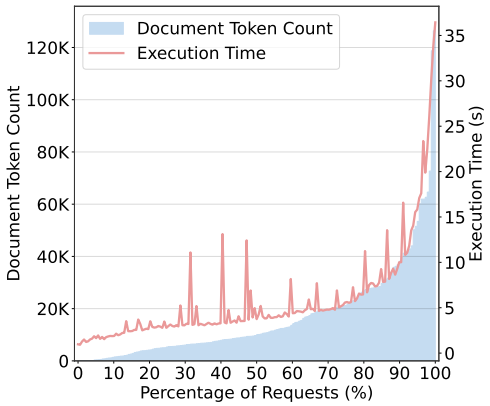


Figure 4: Execution-time heterogeneity of agent requests in RAG agent. Different requests carry different document, which lead to substantially different end-to-end execution times.

However, this architecture still suffers from request blocking. Real-world requests can vary significantly in execution time, as users or platforms choose different inputs or hyperparameters to satisfy budget, accuracy, and SLO requirements [5, 36, 7, 16]. Figure 4 shows this variation for RAG agent on TriviaQA [24], where execution time significantly differs across request percentiles, with the gap even exceeding 20×. Short and long requests can be batched and advance together, causing request blocking, and substantially limit the latency and throughput gains of the aggregated architecture.

2.4 Decomposition and Challenge

The blocking stems from executing a long request as one coarse-grained unit. A naive solution is to decompose it into smaller execution units. In this way, short requests can be executed with a small execution unit and exit sooner, rather than being coupled with the entire long ones. New requests can enter between two consecutive units of the long request, rather than waiting for its entire completion.

We observe that request execution mainly depends on the input or output data flowing through modules. This is evident in Figure 4: as the document directly determines the indexing cost and indirectly affects downstream execution by propagating through retrieval, reranking, and synthesis, it largely determines the end-to-end execution time. Thus, a long RAG request can be decomposed by splitting its documents to trigger only part of the original execution. By decomposing such data, we can decompose the corresponding executions and obtain smaller execution units.

However, decomposition presents two challenges. **First, preserving agent semantics**. Semantics impose strict execution workflow, as modules must follow the specified order; and complete-input requirement, as some modules require complete input to produce correct output. Naive decomposition may violate these constraints, causing downstream modules to execute out of order or consume partial input when complete input is required. Thus, decomposition must preserve agent semantics to avoid degrading the answer quality. **Second, controlling decomposition cost**. Although decomposition reduces blocking, it introduces scheduling and management cost. For long requests, these costs may accumulate across scheduling rounds, significantly increasing latency. Thus, decomposition must control its granularity to avoid long-tail latency.

3 Method

We present AGENTWEAVE's overall architecture (§3.1) and flow abstraction (§3.2), followed by its two key techniques: (1) flow decomposition (§3.3) to provide a semantic-preserving decomposition method, and (2) a flow-adaptive scheduler (§3.4) to guarantee decomposition gains.

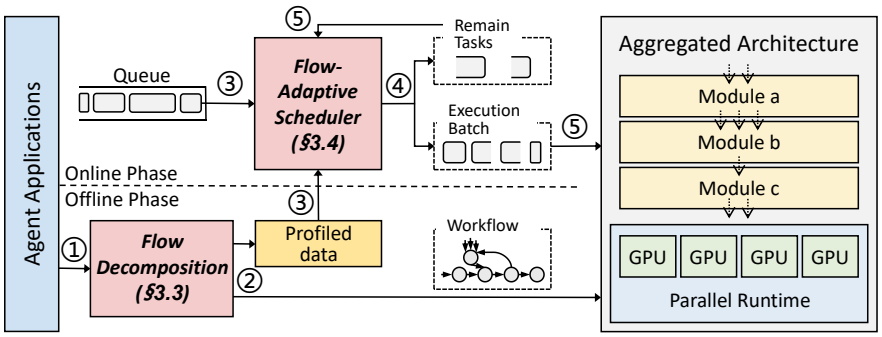


Figure 5: System Overview of AGENTWEAVE.

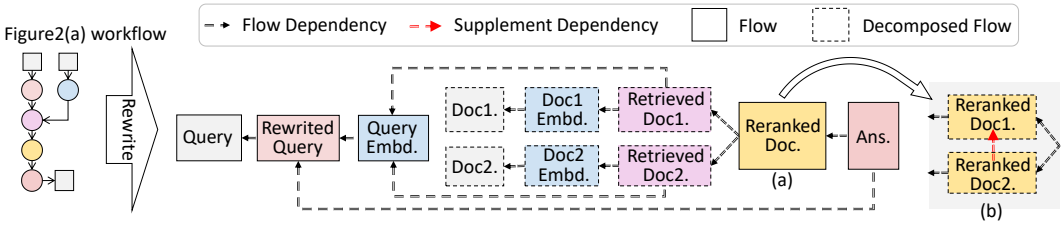


Figure 6: Flow decomposition on RAG agent in Figure 2(a). We illustrate other agents in AppendixB.

3.1 Overview

Figure 5 shows the overview of AGENTWEAVE. In the offline phase, the system registers an agent application (①), analyzes its workflow to derive a semantics-preserving decomposition method (②), and profiles module execution time and decomposition costs for online scheduling. In the online phase, the scheduler dequeues requests and determines the decomposition size based on the offline profiling results (③). It then constructs the execution batch (④), dispatching the batch to the aggregated engine and returning remaining tasks to the scheduler for later rounds (⑤).

3.2 Flow Abstraction

Flow as the Unit of Decomposition. A module's input or output data determines its required computation. We term such data a *flow*, and decompose a long request by decomposing flows into subflows for module execution. We further identify which flow can be decomposed.

A flow can be decomposed only when (1) it can be split into independent subflows, and (2) each subflow is a correct subset of the flow produced by its module on the complete input. Otherwise, the flow should be processed as a whole. In Figure 6, for example, document embeddings can be decomposed because each embeddings subflow is independent and represents a correct subset of the full embeddings produced by indexing on complete documents. In contrast, the query is not decomposable because partial queries are dependent on each other to express the original user intent.

Some flows satisfy the first condition but not the second. They can be decomposed into independent subflows, but each subflow may not be a correct subset. For reranking (a) example in Figure 6, reranked documents can be split into independent document subflows, but it requires the complete retrieved documents as input to select the global top-k results, making subflow not a correct subset result, thus not decomposable. We treat such flows as conditionally decomposable and decompose them while preserving the complete-input requirement through supplement dependency in Section 3.3.

Flow Dependency Graph. The flow dependency graph is a flow representation of the module workflow, where nodes denote data flows and edges denote how outputs are derived from inputs by modules. In this graph, each module's output flow depends on its input flows, thus, the original execution order is represented as flows dependency, as illustrated by the RAG example in Figure 6. As flow is the appropriate unit of decomposition, this reformulation makes decomposition explicit.

3.3 Flow Decomposition

Flow decomposition enables long requests to be split into smaller execution units while preserving agent semantics. The system constructs decomposition based on above abstraction.

Decomposition Construction. The system first rewrites the workflow into a flow dependency graph, then traces and splits the decomposable flows along the dependency from the final answer. In Figure 6, it decomposes the retrieved documents into subflows, and further propagates to document embeddings and raw documents. It keeps other flows intact, as they are not decomposable. This step determines which flows are decomposed and how they propagate along the valid dependency.

Supplement Dependency Construction. The system then splits conditionally decomposable flows and inserts supplement dependencies to satisfy complete-input requirement. A supplement dependency connects a later subflow to an earlier one, allowing the later subflow to receive the accumulated result from previous subflows and continue the original complete-input computation. For reranking (b) in Figure 6, the system inserts a supplement dependency from reranked document subflow 2 to subflow 1. The second reranking then consumes both the retrieved documents subflow 2 and the accumulated reranked result from subflow 1, producing the correct top-k result. This step splits conditionally decomposable flows while preserving their complete-input requirement.

Decomposition of a Request. At runtime, the system decomposes a long request by splitting the most downstream decomposed flow in the graph, i.e., documents in RAG. Each subflow is consumed to produce output subflows along the reverse direction of flow and supplement dependencies. Since each module consumes or produces only a subflow rather than the complete flow, its execution is shorter, and the execution triggered by one subflow forms a smaller execution unit for scheduling.

This runtime decomposition is controlled by the decomposition size, which sets the subflow size for each request in a scheduling round. A larger decomposition size increases per-round execution time, while a smaller one creates more execution units and introduces more overhead. Thus, the system profiles execution time and overhead under diverse decomposition sizes for online size selection.

3.4 Flow-Adaptive Scheduler

The flow-adaptive scheduler determines the decomposition size online in each scheduling round based on the current queue state and profiled data. The goal is to maximize system throughput while controlling the long tail latency caused by decomposition overhead.

Problem Formulation. We formulate decomposition size selection as an online optimization problem. At each scheduling round, let $\mathcal{R} = \{r_1, r_2, \ldots, r_N\}$ be the requests for current execution; w_i be the remaining workload of request r_i; S be a candidate decomposition size; $\mathcal{S}$ be the candidate set.

Given a decomposition size S, current round total time is $T_{\text{total}}(S) = T_{\text{exec}}(S, \mathcal{R}) + O(S)$, where $T_{\text{exec}}(S, \mathcal{R})$ is the effective execution time under decomposition size S, and $O(S)$ is the decomposition overhead introduced in this round. Both terms are estimated from the offline workload profiles.

Throughput is measured by completed requests per unit time, but this discrete metric ignores partial progress made by unfinished requests. We therefore use a continuous throughput-gain objective:

$$J_{\text{gain}}(S) = \frac{\sum_{r_i \in \mathcal{R}} \min(1, S/w_i)}{T_{\text{total}}(S)},$$

where each request contributes 1 if it completes in the current round, and otherwise contributes S/w_i to represent the fraction of its workload completed.

Decomposition incurs overhead each round, which accumulates over rounds and increases request completion time. Since the decomposition sizes of future rounds are unknown, we estimate this cost by applying the current size S to the remaining workload and summing the resulting overhead:

$$O_{\text{total}}(S) = \sum_{r_i \in \mathcal{R}} \lceil w_i/S \rceil \cdot O(S).$$

Since long requests are decomposed across more rounds, accumulated overhead can increase tail latency. The scheduler therefore selects the decomposition size S to maximize throughput gain while controlling accumulated decomposition overhead.

Algorithm 1 Flow-Adaptive Scheduling

1: **Input:** Request set R, System average execution time $\bar{T}$, Workload profiles Φ
2: **Output:** Optimal decomposition size s^*, execution batch Q_{exec}, remain tasks Q_{remain}
3: $s_{min}, s_{max} \leftarrow$ get_search_space(R)
4: $best_score \leftarrow -\infty$
5: $s^* \leftarrow s_{min}$
6: **for** s from s_{min} to s_{max} **do**
7: $\quad exec_time, overhead \leftarrow$ estimate_latency(Φ, R, s)
8: $\quad total_time \leftarrow exec_time + overhead$
9: $\quad gain \leftarrow \sum_{r \in R} \min(1, s/r.workload)$
10: $\quad sensitivity \leftarrow |R|/\bar{T}^2$
11: $\quad waste \leftarrow \sum_{r \in R}(\lceil r.workload/s \rceil \times overhead)$
12: $\quad score \leftarrow (gain/total_time) - (sensitivity \times waste)$
13: $\quad$ **if** $score > best_score$ **then**
14: $\quad\quad best_score \leftarrow score$
15: $\quad\quad s^* \leftarrow s$
16: $\quad$ **end if**
17: **end for**
18: $Q_{\text{exec}}, Q_{\text{remain}} \leftarrow$ construct(R, s^*)
19: **return** s^***,** $Q_{\textbf{exec}}$**,** $Q_{\textbf{remain}}$

Problem Solving. This scheduling problem corresponds to a multi-objective optimization problem [37], and we follow the classical scalarization method [37], as it provides trade-off-specific optimality and low decision overhead for online scheduling. We scalarize the throughput-gain and accumulated-overhead objectives into a single objective:

$$S^* = \arg\max_{S \in \mathcal{S}} \text{Score}(S), \quad \text{Score}(S) = J_{\text{gain}}(S) - \lambda \cdot O_{\text{total}}(S), \quad \lambda > 0.$$

In classical scalarization, λ is a trade-off parameter that reflects the relative preference between objectives. We can obtain a fixed λ by offline system profiling, but a fixed preference cannot adapt to the online requests, where the relative importance of throughput gain and accumulated overhead changes with the queue state. We therefore estimate λ online by quantifying the throughput loss caused by additional overhead as the current preference.

Let $\bar{T}_{\text{exec}}$ be the system average execution time. The ideal throughput is $J \approx |\mathcal{R}|/\bar{T}_{\text{exec}}$. The sensitivity of throughput to additional latency is $\partial J/\partial T = -|\mathcal{R}|/\bar{T}^2_{\text{exec}}$. We set the trade-off parameter as $\lambda = |\mathcal{R}|/\bar{T}^2_{\text{exec}}$, which converts the accumulated cost into an equivalent throughput loss, and adapts naturally to the online workload. We substitute λ to give the final scheduling rule:

$$S^* = \arg\max_{S \in \mathcal{S}} \text{Score}(S), \quad \text{Score}(S) = \underbrace{J_{\text{gain}}(S)}_{\text{Throughput Gain}} - \underbrace{|\mathcal{R}|/\bar{T}^2_{\text{exec}} \cdot O_{\text{total}}(S)}_{\text{Cost Penalization}}$$

Online Scheduling Procedure. Algorithm 1 summarizes flow-adaptive scheduling. In each scheduling round, the scheduler receives requests from the queue and searches decomposition size within the valid search space (Lines 6). For each candidate size S, the scheduler estimates the execution time and decomposition overhead from the offline workload profiles (Lines 7), computes the throughput gain and cost penalization (Lines 10–12), and evaluates the scheduling score (Lines 13). The scheduler then greedily selects the decomposition size with the highest score (Lines 14–15).

After selecting the optimal decomposition size, the scheduler constructs the corresponding execution batch and remaining tasks (Lines 19). It then dispatches the batch to the execution engine and returns remaining tasks to the scheduler for subsequent scheduling rounds. In this way, the scheduler guarantees the gains of flow decomposition by selecting a decomposition size that reduces request blocking while controlling the overhead.

3.5 Dynamic Workflow

AGENTWEAVE can be naturally extended to support dynamic, data-dependent control flow by serving the varying execution paths of a dynamic workflow via a unified static graph.

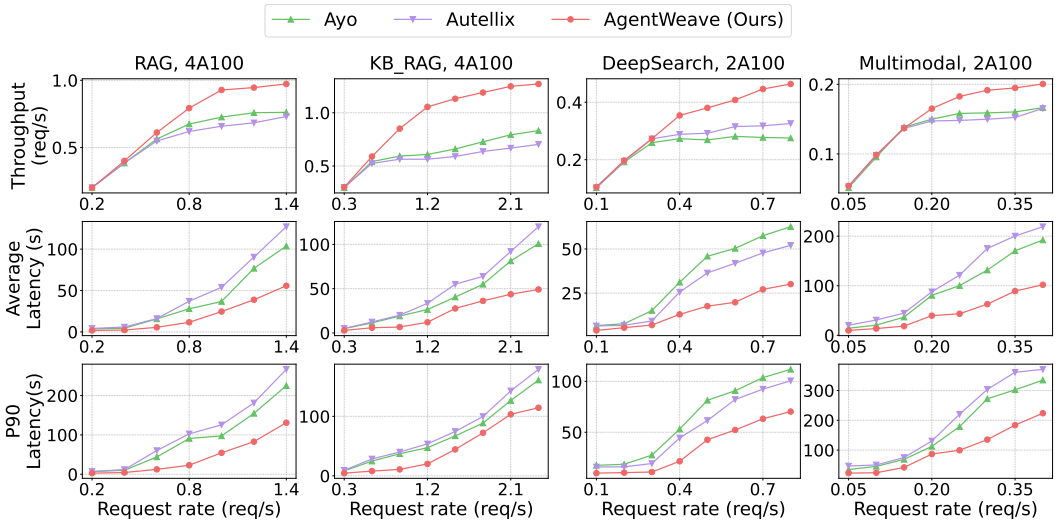


Figure 7: End-to-end performance on four agent applications under varying request rates.

Offline Stage: Static graph construction. Although different requests may follow different execution graphs, they can still be served using a unified static graph because the module types and their dependencies are known in advance. Offline, we record the execution paths of a set of sampled requests and merge them into a unified static graph. Specifically, we align the common parts of the sampled paths and unfold their different branches and loop iterations into a single static execution order. The resulting graph is constructed such that each sampled execution path can be mapped to an ordered subsequence of the static graph. In this way, the unified graph covers both common and less frequent execution patterns, while the observed frequencies can be used to choose an execution order that favors the common cases. AGENTWEAVE directly applies flow abstraction, flow decomposition, and the flow-adaptive scheduler to this unified static graph.

Online Stage: Online scheduling. At runtime, each request follows its own data-dependent execution path within the unified static graph. If the next module scheduled by the system does not match the module required by a request, that request waits until its required module is scheduled.

4 Evaluation

We evaluate AGENTWEAVE on representative agent applications introduced in Section 2.1. For brevity, we provide detailed implementation and experimental setup in Appendix C and Appendix D.

Baselines. We compare AGENTWEAVE with two disaggregated architecture baselines that represent the optimization directions discussed in Section 2.2. AYO [47] optimizes agent workflow by splitting the modules into fine-grained task primitives and overlapping their execution to shorten the latency. AUTELLIX [34] optimizes LLM serving by treating an agent program as the scheduling unit and prioritizing LLM calls using program-level execution context.

4.1 Performance

Throughput. The first row of Figure 7 shows that AGENTWEAVE sustains higher throughput as input rate increases, improving over the baseline by 1.33× on RAG, 1.81× on KB-RAG, 1.68× on DeepSearch, and 1.28× on Multimodal. At low request rate, all systems perform similarly because the servers are not saturated. As rate increases, the disaggregated baselines saturate earlier: Autellix performs better on LLM-dominated DeepSearch agents, while Ayo performs better on other agents, but both are limited by pipeline bubbles. AGENTWEAVE keeps devices busy and reduces request blocking, thereby sustaining higher throughput near saturation.

Average latency. The second row of Figure 7 shows that AGENTWEAVE consistently reduces average latency, improving over the baseline by 2.28× on RAG, 2.45× on KB-RAG, 2.08× on DeepSearch, and 2.15× on Multimodal. At low input rates, queueing is limited, and the improvement

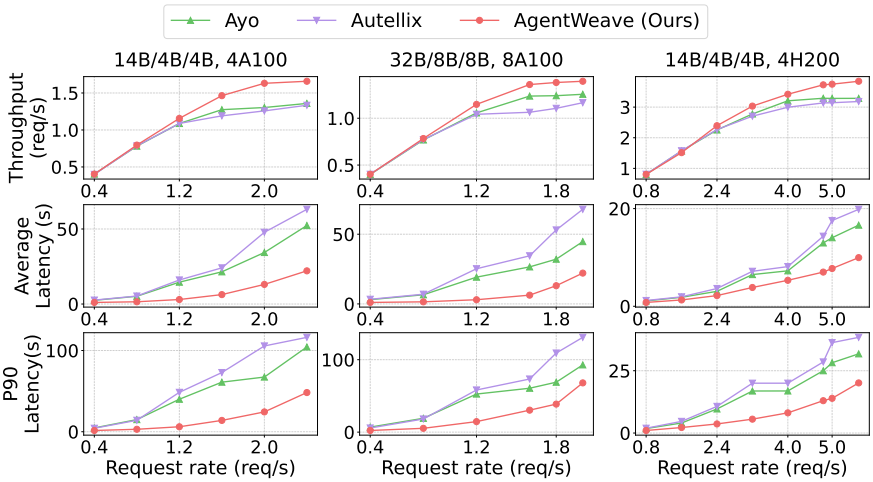


Figure 8: End-to-end RAG performance under different model and hardware configurations. Model sizes are reported as LLM/Embedding/Reranker.

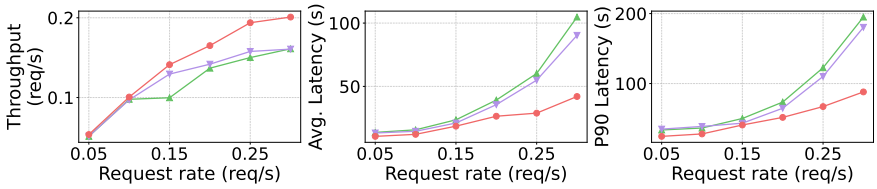


Figure 9: End-to-end performance on dynamic DeepResearch under varying request rates.

mainly comes from aggregated execution, which shortens module execution. At higher input rates, AGENTWEAVE further reduces blocking with flow decomposition, so latency grows more slowly.

P90 latency. The third row of Figure 7 shows that AGENTWEAVE reduces tail latency over the baseline by 2.04× on RAG, 1.56× on KB-RAG, 1.59× on DeepSearch, and 1.66× on Multimodal. Unlike average latency, the P90 improvement mainly comes from the aggregated architecture. Tail requests are typically long requests themselves, so decomposition does not directly shorten their execution. However, the aggregated architecture still reduces the latency of these long requests by utilizing all devices to process each module.

Different Configurations. We evaluate AGENTWEAVE on RAG under different configurations in Figure 8. AGENTWEAVE improves throughput by 1.24×, 1.20×, and 1.21×, reduces average latency by 2.85×, 3.07×, and 1.99×, and reduces P90 latency by 2.41×, 1.92×, and 1.91×, respectively. The advantage becomes more pronounced under higher request rates, where baselines suffer from pipeline bubbles, and AGENTWEAVE continues to reduce blocking in aggregated architecture. These gains show that AGENTWEAVE remains effective when scaling to larger models, more devices, and across different GPU platforms.

Dynamic Workflow. We further evaluate AGENTWEAVE on dynamic workflows using the DeepResearch prototype (see Appendix D for details) in Figure 9. AGENTWEAVE improves throughput by 1.25×, reduces average latency by 2.15× and reduces P90 latency by 2.05×. These gains show that its performance benefits extend to dynamic workflows with execution structures evolving at runtime.

Other Performance. We also evaluate AGENTWEAVE under multiple arrival request sequences and request mixes to show the performance gain stability in Appendix E.

4.2 Ablation Studies and Sensitivity Analysis

Sensitivity Analysis. We evaluate the effect of the trade-off parameter λ on RAG in Figure 10. We compare our λ with static baselines: $\lambda = 0.25$ selected by offline profiling, and $\lambda = 0.05$, 0.45, and 0.65 covering different preferences over throughput gain and decomposition overhead. Our λ

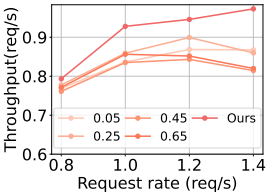


Figure 10: Throughput for different trade-off parameter λ in scheduling.

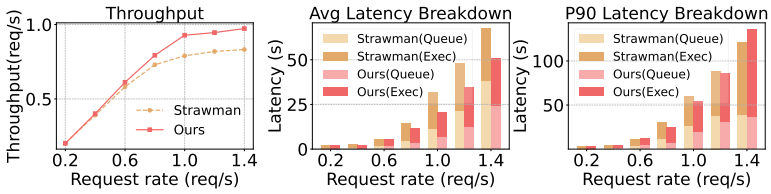


Figure 11: Ablation studies of AGENTWEAVE on RAG against the naive aggregated architecture. The figure reports end-to-end throughput, together with the breakdown of average and P90 latency.

achieves the highest throughput, improving over static settings by $1.13\times$. The advantage of our λ comes from dynamically adapting to the current queue state. Static values fit only limited regimes: $\lambda = 0.25$ performs well in most cases but becomes suboptimal at QPS $= 1.4$, while larger values over-penalize and suppress useful decomposition under high request rate.

Ablation Studies. We ablate AGENTWEAVE against a naive aggregated architecture strawman on RAG in Figure 11. Both perform similarly at low request rates, as the servers are not saturated. At high request rates, AGENTWEAVE reduces request blocking, improving throughput by $1.19\times$ and reducing average queueing and execution latency by $1.57\times$ and $1.48\times$, respectively. This confirms the gains from reducing request blocking. Although decomposition adds overhead to long requests, flow-adaptive scheduling controls the decomposition size and keeps P90 latency comparable. We provide ablation results for the other three agent applications in Appendix G. Overall, AGENTWEAVE improves throughput and average latency while limiting tail-latency overhead.

4.3 Accuracy

AGENTWEAVE improves serving efficiency with flow decomposition while preserving the semantics of the original agent workflow. Following the setup in Section 4.1, accuracy remains unchanged before and after applying AGENTWEAVE: RAG stays at 0.8470 on TriviaQA [24], KB-RAG at 0.8170 on TriviaQA [24], and DeepSearch at 0.3097 on GAIA [35]. This shows that AGENTWEAVE preserves output quality while improving serving performance. We provide more semantic-preservation evidence in Appendix F.

5 Conclusion

This paper presents AGENTWEAVE, an efficient distributed serving system for agent applications. AGENTWEAVE adopts an aggregated architecture and addresses its key challenge, request blocking, through decomposition. It exposes flow as the decomposition unit and deploys flow decomposition to safely decompose a request. It also deploys a flow-adaptive scheduler to balance decomposition gains against overhead. Experiments on representative agents show that AGENTWEAVE achieves better serving performance over existing baselines while preserving accuracy. We clarify the abstraction differences from AYO in Appendix H. We discuss the target workload and applicability in Appendix I. We finally discuss limitations and future directions in Appendix J.

Acknowledgments and Disclosure of Funding

This work was supported in part by the Major Key Project of PCL under Grant PCL2025A03, the National Natural Science Foundation of China (62472459), Guangdong S&T Program (2025B0101080001), and Shandong Provincial Key Research and Development Program (No. 2022CXGC020106). This work was also supported in part by the Swiss National Science Foundation (SNSF) under Grant 200021_200477.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] 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)*, pages 117–134, 2024.

[3] Amazon Web Services. Supported connectors – Amazon Q Business. `https://docs.aws.amazon.com/amazonq/latest/qbusiness-ug/connectors-list.html`, 2026. Accessed: 2026-05-07.

[4] Amazon Web Services. Amazon Bedrock Knowledge Bases. `https://docs.aws.amazon.com/bedrock/latest/userguide/knowledge-base.html`, 2026. Accessed: 2026-05-07.

[5] Anthropic. How we built our multi-agent research system, June 2025. URL `https://www.anthropic.com/engineering/multi-agent-research-system`. Anthropic Engineering blog. Published June 13, 2025. Accessed April 11, 2026.

[6] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench: A bilingual, multitask benchmark for long context understanding. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3119–3137, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.172. URL `https://aclanthology.org/2024.acl-long.172/`.

[7] Cohere. Agentic multi-stage rag with cohere tools api. `https://docs.cohere.com/page/agentic-multi-stage-rag`, 2026. Cohere documentation. Accessed: 2026-04-15.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL `https://aclanthology.org/N19-1423/`.

[9] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. *IEEE Transactions on Big Data*, 2025.

[10] Patrick Esser, Sumith Kulal, Andreas Blattmann, Rahim Entezari, Jonas Müller, Harry Saini, Yam Levi, Dominik Lorenz, Axel Sauer, Frederic Boesel, Dustin Podell, Tim Dockhorn, Zion English, and Robin Rombach. Scaling rectified flow transformers for high-resolution image synthesis. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[11] Jiarui Fang, Jinzhe Pan, Xibo Sun, Aoyu Li, and Jiannan Wang. xdit: an inference engine for diffusion transformers (dits) with massive parallelism. *arXiv preprint arXiv:2411.01738*, 2024.

[12] Zhiyuan Fang, Yuegui Huang, Zicong Hong, Yufeng Lyu, Wuhui Chen, Yue Yu, Fan Yu, and Zibin Zheng. Klotski: Efficient mixture-of-expert inference via expert-aware multi-batch pipeline. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 574–588, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400710797. doi: 10.1145/3676641.3716261. URL `https://doi.org/10.1145/3676641.3716261`.

[13] Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. Precise zero-shot dense retrieval without relevance labels. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1762–1777, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.99. URL `https://aclanthology.org/2023.acl-long.99/`.

[14] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, Haofen Wang, et al. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2(1):32, 2023.

[15] Google. Google Cloud Search Connector Directory. `https://developers.google.com/workspace/cloud-search/docs/connector-directory`, 2026. Accessed: 2026-05-07.

[16] Google Cloud. Choose a design pattern for your agentic ai system. `https://docs.cloud.google.com/architecture/choose-design-pattern-agentic-ai-system`, 2025. Google Cloud Architecture Center documentation, last reviewed 2025-10-08, accessed 2026-04-15.

[17] Jonathan Ho and Tim Salimans. Classifier-free diffusion guidance. *arXiv preprint arXiv:2207.12598*, 2022.

[18] Zicong Hong, Yuyan Chen, Haoyue Zhang, Peng Li, Wuhui Chen, Song Guo, and Xiaowei Shen. Efficient multimodal serving via module multiplexing. In *Proceedings of the 21st European Conference on Computer Systems*, EUROSYS '26, page 2292–2307, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400722127. doi: 10.1145/3767295.3769389. URL https://doi.org/10.1145/3767295.3769389.

[19] Zhengding Hu, Vibha Murthy, Zaifeng Pan, Wanlu Li, Xiaoyi Fang, Yufei Ding, and Yuke Wang. Hedrarag: Co-optimizing generation and retrieval for heterogeneous rag workflows. In *Proceedings of the ACM SIGOPS 31st symposium on operating systems principles*, pages 623–638, 2025.

[20] Kaiyi Huang, Kaiyue Sun, Enze Xie, Zhenguo Li, and Xihui Liu. T2i-compbench: A comprehensive benchmark for open-world compositional text-to-image generation. *Advances in Neural Information Processing Systems*, 36:78723–78747, 2023.

[21] Lisheng Huang, Yichen Liu, Jinhao Jiang, Rongxiang Zhang, Jiahao Yan, Junyi Li, and Wayne Xin Zhao. ManuSearch: Democratizing deep search in large language models with a transparent and open multi-agent framework. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng, editors, *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 2403–2417, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-335-7. doi: 10.18653/v1/2025.findings-emnlp.130. URL https://aclanthology.org/2025.findings-emnlp.130/.

[22] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[23] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LongLLMLingua: Accelerating and enhancing LLMs in long context scenarios via prompt compression. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1658–1677, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.91. URL https://aclanthology.org/2024.acl-long.91/.

[24] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1601–1611, 2017.

[25] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

[26] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

[27] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in neural information processing systems*, 36:51991–52008, 2023.

[28] Hanchen Li, Qiuyang Mang, Runyuan He, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. Continuum: Efficient and robust multi-turn llm agent scheduling with kv cache time-to-live. *arXiv preprint arXiv:2511.02230*, 2025.

[29] Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 3102–3116, 2023.

[30] Long Lian, Boyi Li, Adam Yala, and Trevor Darrell. LLM-grounded diffusion: Enhancing prompt understanding of text-to-image diffusion models with large language models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=hFALpTb4fR. Featured Certification.

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

[32] Yexiong Lin, Shanshan Ye, Yu Yao, Zhen Fang, Bo Han, and Tongliang Liu. Squidagent: Parallelize wisely, coordinate efficiently, 2026. URL https://arxiv.org/abs/2610.08647.

[33] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173, 2024.

[34] 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. Accepted to NSDI'26.

[35] Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: a benchmark for general AI assistants. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=fibxvahvs3.

[36] Microsoft. Best practices for using azure openai on your data (classic). https://learn.microsoft.com/en-us/azure/foundry-classic/openai/how-to/on-your-data-best-practices, 2026. Microsoft Learn documentation. Accessed: 2026-04-15.

[37] Kaisa Miettinen. *Nonlinear multiobjective optimization*, volume 12. Springer Science & Business Media, 1999.

[38] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.

[39] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

[40] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.

[41] NVIDIA. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/, 2026. Accessed: 2026-05-05.

[42] NVIDIA. NVIDIA H200 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/h200/, 2026. Accessed: 2026-05-05.

[43] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565, 2024.

[44] Sebastián Ramírez. Fastapi. https://github.com/fastapi/fastapi, 2026. Software, MIT License. Documentation: https://fastapi.tiangolo.com.

[45] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

[46] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[47] 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*, pages 1302–1316, 2025.

[48] Haodong Wang, Qihua Zhou, Zicong Hong, and Song Guo. D2moe: Dual routing and dynamic scheduling for efficient on-device moe-based llm serving. In *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*, ACM MOBICOM '25, page 574–588, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711299. doi: 10.1145/3680207.3723493. URL https://doi.org/10.1145/3680207.3723493.

[49] Jize Wang, Zerun Ma, Yining Li, Songyang Zhang, Cailian Chen, Kai Chen, and Xinyi Le. Gta: a benchmark for general tool agents. *Advances in Neural Information Processing Systems*, 37:75749–75790, 2024.

[50] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

[51] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[52] Ling Yang, Zhaochen Yu, Chenlin Meng, Minkai Xu, Stefano Ermon, and Bin Cui. Mastering text-to-image diffusion: Recaptioning, planning, and generating with multimodal llms. In *Icml*, volume 3, page 7, 2024.

[53] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[54] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX symposium on operating systems design and implementation (OSDI 22)*, pages 521–538, 2022.

[55] Yue Yu, Wei Ping, Zihan Liu, Boxin Wang, Jiaxuan You, Chao Zhang, Mohammad Shoeybi, and Bryan Catanzaro. Rankrag: Unifying context ranking with retrieval-augmented generation in llms. *Advances in Neural Information Processing Systems*, 37:121156–121184, 2024.

[56] Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, et al. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025.

[57] Zhenyu Zhang, Runjin Chen, Shiwei Liu, Zhewei Yao, Olatunji Ruwase, Beidi Chen, Xiaoxia Wu, and Zhangyang Wang. Found in the middle: How language models use long contexts better via plug-and-play positional encoding. *Advances in Neural Information Processing Systems*, 37:60755–60775, 2024.

[58] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[59] 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.

A Agent Applications

This appendix introduces the four agent applications considered in this paper, covering their use cases and how they execute as workflows involving multiple LLM or tool invocations [49, 32].

A.1 RAG Agent

Retrieval-augmented generation (RAG) [26, 14] in Figure 2(a) is one of the most widely used agents for accurate answers generation with retrieved knowledge. A standalone LLM answers from its parametric knowledge and the prompt context, which can be insufficient [14, 39] when the answer depends on newly added documents, private files, domain-specific corpora, or information that is too long to directly place in the prompt. A RAG agent addresses this limitation by coupling the LLM with retrieval tools. Instead of asking the LLM to answer directly, the agent first identifies relevant evidence and then asks the LLM to synthesize an answer conditioned on that evidence.

Given a user query and a set of documents, the RAG agent executes a workflow that combines LLM calls with tool calls. It first invokes an LLM to expand or rewrite the query [13, 23], producing a clearer search intent for downstream retrieval. The agent then calls an embedding tool to encode the documents and update the vector index. Using the rewritten query, it invokes a retrieval tool to search the index and obtain candidate documents. Since the retrieved candidates may be noisy or only partially relevant, the agent further invokes a reranking tool to rescore them and select the most relevant evidence. Finally, the agent invokes the LLM again to synthesize the answer from the reranked documents [55].

Modern RAG extends retrieval-augmented generation into a multi-stage agent workflow. It combines query expansion, document indexing, vector retrieval, reranking, and answer synthesis to improve knowledge-grounded response generation. Advanced RAG therefore represents a common agent workflow that relies on multiple LLM and tool invocations for knowledge-intensive tasks.

A.2 Knowledge-Based RAG Agent

Knowledge-based RAG [14, 19] in Figure 2(b) is an agent for answering questions over a persistent knowledge source [4]. It follows the same general principle as RAG: the LLM should answer using retrieved evidence rather than relying only on its internal knowledge. Different from advanced RAG, the documents are processed and indexed offline into a local knowledge base, which may contain enterprise documents, product manuals, internal wikis, troubleshooting guides, code repositories, or other domain-specific resources [4, 3, 15]. This setting is widely used when user queries target private, domain-specific, or frequently updated corpora.

Given a user query, the knowledge-based RAG agent executes a workflow that combines LLM calls with retrieval tools. It first invokes an LLM to expand or rewrite the query, producing a clearer search intent for the knowledge base. The agent then calls a retrieval tool to search the pre-built database and obtain candidate documents or passages. Since the retrieved candidates may still contain irrelevant or weakly related evidence, the agent further invokes a reranking tool to rescore them and select the most relevant passages. Finally, the agent invokes the LLM again to synthesize the answer from the reranked evidence.

Knowledge-based RAG removes the online document encoding stage and focuses on query expansion, knowledge-base retrieval, reranking, and answer synthesis over a persistent corpus. This makes it a practical agent workflow for domain-specific question answering, where LLM reasoning is coupled with retrieval from private or enterprise knowledge sources.

A.3 DeepSearch Multi-Agent

DeepSearch multi-agent [5, 21] in Figure 2(c) is an agent for complex information-seeking tasks. Unlike RAG-style agents, which usually retrieve evidence from a fixed corpus or index, DeepSearch actively explores external sources through multiple coordinated agents. It is designed for questions that require broad investigation, where the answer may depend on collecting evidence from different directions and synthesizing them into a structured report.

Given a user query, the DeepSearch multi-agent executes a workflow that combines LLM agents with search subagent tools [27]. It first invokes a lead LLM agent to analyze the query and decompose it into multiple subquestions or research directions. The lead agent then dispatches these subquestions to multiple search subagents. Each subagent invokes search tools, reads retrieved pages or documents, refines its search when needed, and summarizes the evidence for its assigned direction. After the subagents finish, a consolidation agent aggregates their findings, removes redundant information, resolves inconsistent evidence, and generates the final response with supporting citations.

DeepSearch organizes complex information seeking as a multi-agent research process. By decomposing a query into multiple directions, collecting evidence in parallel, and consolidating intermediate findings, it supports open-ended questions that require broader exploration than standard retrieval-based workflows.

A.4 Multimodal Generation Agent

Multimodal generation [52, 30, 18] in Figure 2(d) is an agent for generating images from text prompts. A diffusion model [10] can synthesize an image directly from a prompt, but short user prompts often underspecify visual details such as object attributes, spatial relations, background, lighting, or style. A multimodal generation agent addresses this limitation by using LLM calls to refine and structure the prompt before invoking the image generator. This improves prompt understanding, scene composition, and controllability.

Given a user prompt, the multimodal generation agent executes a workflow that combines LLM calls with a generative tool. It first invokes an LLM-based recaptioning module to rewrite the prompt into a richer and more explicit description. The agent then invokes an LLM-based planning module to decompose the refined description into structured scene plans, such as objects, regions, layouts, or visual constraints. Finally, the agent calls a diffusion-based image generation tool to synthesize the final image from the planned description.

Multimodal generation extends agent workflows beyond text-only question answering. It uses LLM-based recaptioning and planning to prepare structured inputs for a diffusion-based generator, connecting language reasoning with specialized visual generation tools.

A.5 Common Properties

Although the four applications target different tasks, they share several properties that distinguish agent serving from serving standalone LLM requests [12, 48, 25, 59]. First, they execute as module workflows. The final response is produced through multiple LLM calls and tool invocations, such as retrieval, reranking, search, database access, or image generation. Second, these modules are connected by data dependencies. Each module consumes intermediate data produced by earlier modules, so the workflow can only progress when the required data becomes available. Third, requests can have substantially different execution costs. Different requests carry different documents, retrieved candidates, subquestions, search results, or scene plans, which change the amount of work required by each module and lead to large differences in execution time.

These properties make agent serving a workflow-level systems problem. Optimizing a single LLM call or an individual tool is insufficient, because overall performance also depends on how intermediate data moves through the workflow and how different modules are scheduled together. Therefore, an agent serving system should optimize the execution of dependency-constrained module workflows rather than treating each module invocation as an isolated request.

B Other Decomposition

Figure 12 illustrates flow decomposition for the other three agent applications in Figure 2. These decompositions follow the same construction rule as Section 3.2 and Section 3.3: we rewrite the module workflow into a flow dependency graph, split decomposable flows along valid dependencies, decompose conditional decomposable flows, and insert supplement dependencies to satisfy the complete-input requirement.

Knowledge-Base RAG Agent. The KB-RAG agent in Figure 2(b) follows the retrieval, reranking, and synthesis stages of RAG, but retrieves documents from a pre-built knowledge base instead of

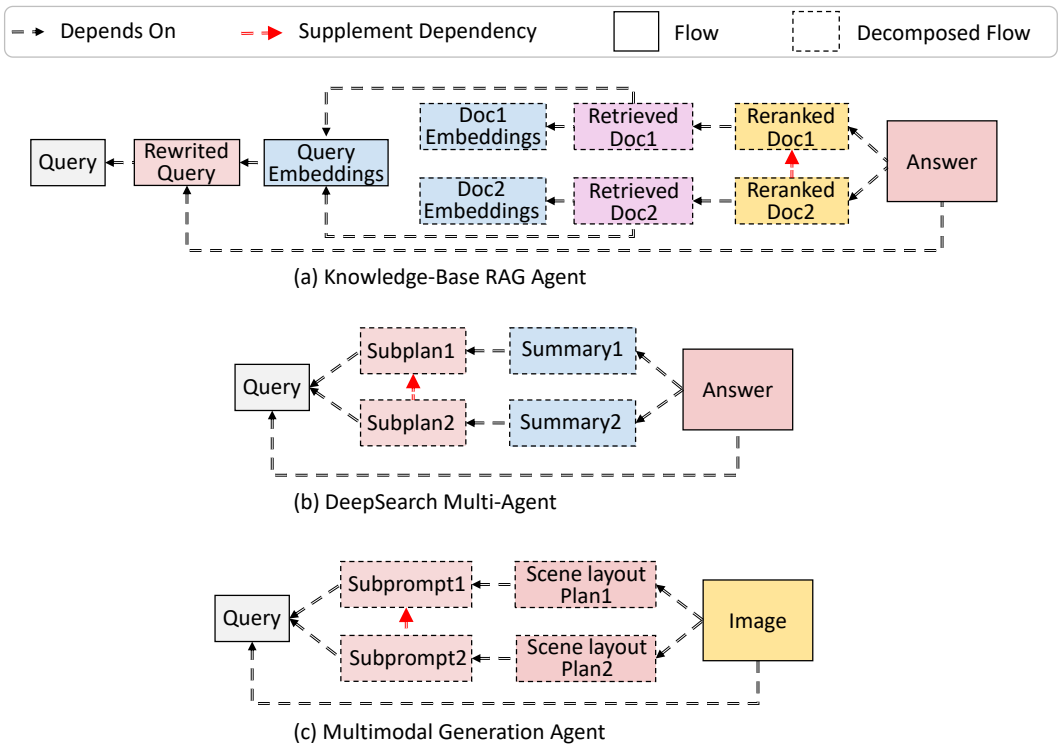


Figure 12: Flow decomposition on the KB-RAG, deepsearch and multimodal agent in Figure 2(b)(c)(d).

indexing newly provided documents. The rewritten query and query embedding are kept intact, since they describe a single query intent and cannot be decomposed into independent subflows. The retrieved documents and document embeddings are decomposable, thus split into independent subflows. Reranked documents are conditionally decomposable, since reranking needs the complete retrieved documents to select global top-k documents. We decompose them and insert supplement dependencies to satisfy the complete-input requirement.

DeepSearch Multi-Agent. The DeepSearch workflow in Figure 2(c) contains query decomposition, subagent search, summary generation, and result consolidation. The original query is still kept intact. The summaries are decomposable, because they can be split to independent summary on different plan, and each is a correct subset of the full summaries. The plan is conditionally decomposable, because each subplan describes a separate search target, but the plan still requires complete input. We decompose it and insert supplement dependencies, which corresponds to decomposing the output of LLM generation.

Multimodal Generation Agent. The multimodal generation workflow in Figure 2(d) contains query recaption, scene planning, and image generation. The original query is kept intact, since it describes the user intent. The scene-layout plans are decomposable, because they can be split into independent plans, and each is a correct subset of the full scene-layout flow. The subprompts are conditionally decomposable, because each subprompt describes an independent visual generation target, but the subprompt flow still requires the complete query as input. Also, we decompose it and insert supplement dependencies.

C System Implementation

AGENTWEAVE is implemented as a distributed serving system for agent workflows, with roughly 8.9K lines of Python code, based on FastAPI [44] for the control plane, Ray [38] for distributed scheduling and execution, SGLang [59] for LLM generation, embedding, and reranking, FAISS [9] for retrieval, xDiT [11] for diffusion-based generation.

Frontend. The frontend is an asynchronous FastAPI service that exposes a stateless request interface, while the runtime tracks each request as a long-lived stateful flow. After admission, a request enters an FCFS queue that stores remaining subflows, intermediate state, and completion status. This separates the user-facing API from decomposition state, allowing AGENTWEAVE to pause and resume requests across scheduling rounds without changing application semantics.

Execution engine. AGENTWEAVE wraps SGLang's asynchronous engine interface as the LLM executor, with continuous batching [54], chunked prefill [2] and prefix caching [59] techniques. AGENTWEAVE keeps decomposition and scheduling decisions outside the LLM engine.

Tool backends. Beyond the LLM executor, AGENTWEAVE implements embedding, reranking, retrieval, web search, and multimodal generation as resident tool services, avoiding repeated initialization during decomposed execution.

Scheduler. The scheduler is an event-driven loop over active request states. It is independent of tool internals and agent-specific module code. In each scheduling round, it uses profiled data to estimate execution time and scheduling overhead, selects the decomposition size, constructs the execution batch, and returns remaining tasks to the queue.

Agent executors. Each application uses a common executor interface to define its flow semantics, including which flows can be decomposed, which context remains intact, and which intermediate results are carried as supplement dependencies. Our prototype implements this interface for RAG, KB-RAG, DeepSearch, and multimodal generation. Thus, application-specific decomposition rules remain inside each executor, while the frontend, scheduler, LLM executor, and tool backends are shared across agents.

D Experimental Setup

We evaluate AGENTWEAVE on four representative agent applications: RAG, KB-RAG, DeepSearch and multimodal generation. We additionally evaluate DeepResearch agent to assess its performance on dynamic workflows. For all experiments, we use the same hardware for AGENTWEAVE and the baselines. The disaggregated baselines place different modules on separate GPU groups, while AGENTWEAVE adopts the aggregated architecture and co-deploys modules across all available GPUs.

RAG. We evaluate RAG on TriviaQA [24] with two model configurations. The 14B/4B/4B configuration uses Qwen3-14B [51] as the LLM, Qwen3-Embedding-4B [56] as the embedding model, and Qwen3-Reranker-4B [56] as the reranker. The 32B/8B/8B configuration uses Qwen3-32B [51], Qwen3-Embedding-8B [56], and Qwen3-Reranker-8B [56]. We evaluate 32B/8B/8B on 4 A100 GPUs [41] in Figure 7, 14B/4B/4B on 4 A100 GPUs, 32B/8B/8B on 8 A100 GPUs, and 14B/4B/4B on 4 H200 GPUs [42] in Figure 8. For the 4-GPU settings, the disaggregated architecture deploys the embedding model and reranker on one GPU each, and deploys the LLM with tensor parallelism [46] across the remaining two GPUs. For the 8-GPU setting, the disaggregated architecture deploys the embedding model and reranker with tensor parallelism across two GPUs each, and deploys the LLM with tensor parallelism across remaining four GPUs. For AGENTWEAVE, all models are deployed with tensor parallelism across all available GPUs.

KB-RAG. KB-RAG follows the RAG workflow but retrieves candidate knowledge from a pre-built knowledge base instead of indexing documents at request time. We evaluate KB-RAG on TriviaQA using Qwen3-32B, Qwen3-Embedding-8B, and Qwen3-Reranker-8B on 4 A100 GPUs in Figure 7. For the disaggregated architecture, the embedding model and reranker are deployed on one GPU each, and the LLM is deployed with tensor parallelism across two GPUs. For AGENTWEAVE, all models are deployed with tensor parallelism across all four GPUs.

DeepSearch. We evaluate DeepSearch on GAIA [35]. We use Qwen3-14B as the main LLM and Qwen3-4B [51] as the subagent SLM in Figure 7. Experiments are conducted on 2 A100 GPUs. For the disaggregated architecture, the main LLM and subagent model are deployed on one GPU each. For AGENTWEAVE, both models are deployed with tensor parallelism across all GPUs.

Multimodal Generation. We evaluate multimodal generation on T2I-CompBench [20]. We use Qwen3-14B for recaptioning and planning, and Stable Diffusion 3 [10] as the diffusion generator in Figure 7. Experiments are conducted on 2 A100 GPUs. For the disaggregated architecture, the LLM and diffusion generator are deployed on one GPU each. For AGENTWEAVE, the LLM is deployed

with tensor parallelism across all GPUs, and the diffusion generator with classifier-free guidance parallelism [17] and PipeFusion-style parallelism [11].

DeepResearch. To evaluate AGENTWEAVE's serving performance when workflows is dynamic, we implement a DeepResearch prototype following Anthropic's multi-agent Research architecture [5]. The LeadResearcher iteratively plans the task, spawns a query-dependent number of Subagents, aggregates their findings, and decides at runtime whether another research round is needed. Each Subagent independently follows a dynamic loop, selecting search tools and iterating until sufficient evidence is collected; the CitationAgent runs only after the research loop terminates. For a fair comparison, we similarly extend AYO to dynamic workflows using an offline-profiled static graph. We evaluate the performance of this prototype on GAIA using Qwen3-32B for the LeadResearcher and CitationAgent and Qwen3-14B for the Subagents, on 4 A100 GPUs.

E Stability of Performance Gains

We repeated the performance experiment under multiple arrival request sequences and request mixes. The improvements remain stable across these runs in Table 1, where values denote mean speedup $\pm$ standard deviation.

Table 1: Performance improvements across repeated runs.

	RAG	KB-RAG	DeepSearch	Multimodal Generation	DeepResearch
Throughput	$1.33 \pm 0.04\times$	$1.81 \pm 0.06\times$	$1.68 \pm 0.06\times$	$1.28 \pm 0.03\times$	$1.25 \pm 0.08\times$
Average latency	$2.28 \pm 0.25\times$	$2.45 \pm 0.27\times$	$2.08 \pm 0.13\times$	$2.15 \pm 0.21\times$	$2.15 \pm 0.35\times$
P90 latency	$2.04 \pm 0.22\times$	$1.56 \pm 0.16\times$	$1.59 \pm 0.10\times$	$1.66 \pm 0.16\times$	$2.05 \pm 0.32\times$

F Semantic-Preservation Evidence

We reanalyzed 1,000 RAG requests. We provide three stronger forms of evidence: an output-equivalence analysis over 1,000 RAG requests, a concrete failure avoided by supplement dependencies, and a case where decomposition may be unsafe.

Output equivalence rates. Using LLM-as-a-Judge [58] with Qwen-32B, 986/1,000 (98.6%) decomposed outputs were semantically equivalent to non-decomposed outputs. Since both answer synthesis and LLM-as-a-Judge are LLM-based, the remaining 1.4% may reflect their combined generation and evaluation variability and does not directly indicate decomposition-induced semantic violations.

Examples of failures avoided by supplement dependencies.

> Query: "In which 1993 film does Jonah write to Annie … to meet his father at the Empire State Building on Valentine's Day?"

In this RAG request, the 1024-token document subflows contain complementary evidence: an earlier subflow identifies "Sleepless in Seattle", while a later one contains the "Jonah–Annie–Empire State Building" evidence. Without the supplement dependency, later reranking loses the earlier result and the LLM answers "The Mask". Carrying the accumulated result forward answers "Sleepless in Seattle", matching the non-decomposed execution and ground truth.

Cases where decomposition may be unsafe.

> Query: "We're No Angels centred on three escaped convicts. Peter Ustinov and Aldo Ray were two … who played the third?"
>
> Naive split: (1) "We're No Angels centred on three escaped convicts"; (2) "Peter Ustinov and Aldo Ray were two … who played the third?"

The first subflow lacks an answer target, while the second loses the movie identity and the antecedent of "the third". AGENTWEAVE therefore keeps the query intact, answering "Humphrey Bogart" under both decomposed and non-decomposed execution, matching the ground truth. When safe decomposition cannot be established, AGENTWEAVE conservatively disables it.

G Ablation Studies

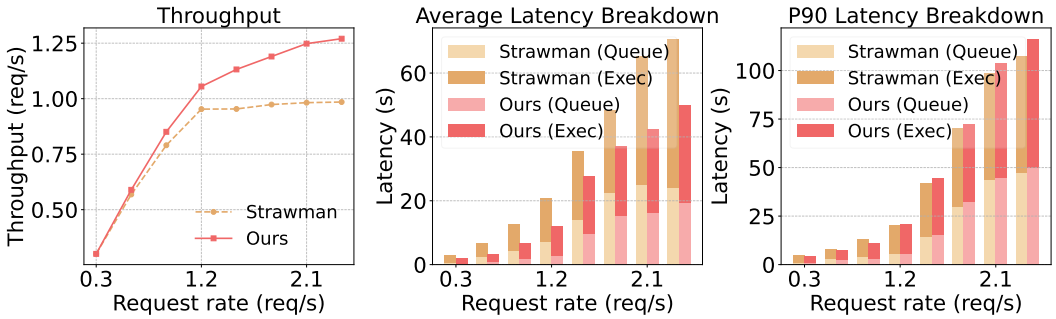


Figure 13: Ablation studies of AGENTWEAVE on Knowledge-Based RAG Agent against the naive aggregated architecture. The figure reports end-to-end throughput, together with the breakdown of average and P90 latency.

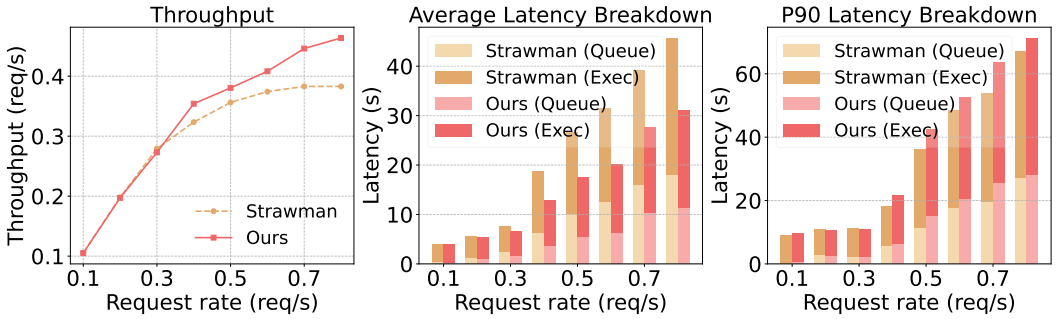


Figure 14: Ablation studies of AGENTWEAVE on DeepSearch Multi-Agent against the naive aggregated architecture. The figure reports end-to-end throughput, together with the breakdown of average and P90 latency.

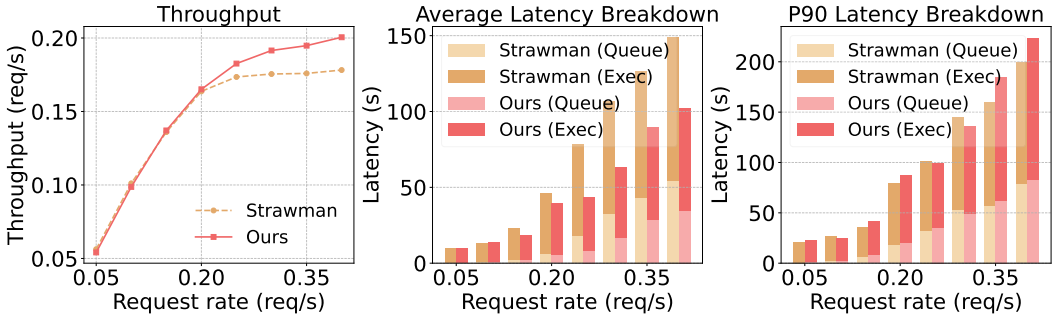


Figure 15: Ablation studies of AGENTWEAVE on Multimodal Generation Agent against the naive aggregated architecture. The figure reports end-to-end throughput, together with the breakdown of average and P90 latency.

H Abstraction Differences from AYO

AGENTWEAVE adopts the flow graph, whereas AYO adopts primitive graph. Although both systems use graphs to expose finer-grained execution, their graph optimizations address different scheduling problems and consequently differ in their abstractions and resulting execution units.

First, their graph optimizations address different scheduling problems. AYO's graph optimization asks how a request can finish sooner: when two operations of that request do not need to wait for

each other, AYO executes them in parallel or overlaps them through pipelining. AGENTWEAVE's flow-graph optimization asks how multiple requests of different lengths can share the aggregated engine without being forced to progress in lockstep, while preserving the semantics of each request. It divides long requests into semantically valid, interruptible execution units so that shorter or newly arrived requests can execute between them.

Second, their graphs represent different entities and dependencies. The different scheduling problems require AYO's primitive graph to represent computation operations and their execution dependencies, whereas AGENTWEAVE's flow graph represents inter-module data flow and its semantic dependencies. An operation graph can expose parallelism, but does not by itself determine whether a large input can be partitioned into independently valid subflows. AGENTWEAVE makes data flow the scheduling abstraction because its size determines module execution cost, and explicitly captures its decomposability, complete-input requirements, and supplement dependencies, enabling long requests to be safely decomposed without changing their semantics.

Third, their graph decompositions produce different execution units. AYO produces an optimized operation graph that exposes more concurrent operations within a query. AGENTWEAVE produces semantically equivalent subflow executions that can be scheduled across multiple rounds and interleaved with other requests. Their granularity is selected online according to the concurrent workload, allowing AGENTWEAVE to reduce request blocking while controlling decomposition overhead.

I Target Workload and Applicability

AGENTWEAVE's primary target is workflows containing multiple GPU-intensive modules or model invocations that compete for GPU resources, rather than the workflow where tool execution is largely CPU- or API-bound. It targets private agent services on single-node multi-GPU systems, especially for SMEs and enterprise teams constrained by data locality and limited GPU and operational budgets. This is a broad practical setting; for example, HPE Private Cloud AI, and Lenovo Hybrid AI 221 with Canonical provide integrated small-scale private AI platforms.

J Limitations and Future Work

AGENTWEAVE has several limitations. First, it currently requires each agent application to specify its flow semantics, including decomposable flows and complete-input dependencies. Automatically deriving such semantics from general agent programs remains future work. Second, the flow-adaptive scheduler uses offline profiles to estimate execution time and decomposition overhead; online profile adaptation could further improve robustness under workload and hardware changes. Finally, our evaluation focuses on representative agents. Extending AGENTWEAVE to more dynamic production agents, external-tool failures, and multi-tenant deployments is an important future direction.

NeurIPS Paper Checklist

1. **Claims**

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

   Answer: [Yes]

   Justification: The abstract and introduction state the paper's scope and main contributions: an aggregated architecture, flow decomposition, and a flow-adaptive scheduler for agent serving. The experimental claims are supported by the evaluation in Section 4.

   Guidelines:

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

2. **Limitations**

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

   Answer: [Yes]

   Justification: We discuss the main limitations of AGENTWEAVE and directions for future work in Appendix J.

   Guidelines:

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

3. **Theory assumptions and proofs**

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

Answer: [N/A]

Justification: The paper does not present formal theoretical results.

Guidelines:

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

4. **Experimental result reproducibility**

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

   Answer: [Yes]

   Justification: We provide implementation details and experimental settings in Appendix C and Appendix D.

   Guidelines:

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

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

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

Answer: [Yes]

Justification: We release code and provide details for reproducibility.

Guidelines:

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

6. **Experimental setting/details**

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

   Answer: [Yes]

   Justification: We specify workloads, models, hardware, baselines, request rates, and metrics in Section 4 and Appendix D.

   Guidelines:

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

7. **Experiment statistical significance**

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

   Answer: [No]

   Justification: We do not report error bars or significance tests.

   Guidelines:

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

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

8. **Experiments compute resources**

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

   Answer: [Yes]

   Justification: We report the GPU configurations, model placements, and experimental setup in Appendix D.

   Guidelines:

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

9. **Code of ethics**

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

   Answer: [Yes]

   Justification: The work uses public datasets, models, and tools, and does not involve human-subject experiments or sensitive data.

   Guidelines:

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

10. **Broader impacts**

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

    Answer: [Yes]

    Justification: The work can improve the efficiency of agent serving.

    Guidelines:

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

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

11. **Safeguards**

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

    Answer: [N/A]

    Justification: The paper does not release new models, generators, or scraped datasets with high misuse risk.

    Guidelines:

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

12. **Licenses for existing assets**

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

    Answer: [Yes]

    Justification: They are open-sourced.

    Guidelines:

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

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

13. **New assets**

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

    Answer: [N/A]

    Justification: The paper does not release new datasets or models.

    Guidelines:

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

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

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

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or human-subject studies.

    Guidelines:

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

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

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

    Answer: [N/A]

    Justification: The paper does not involve human-subject research.

    Guidelines:

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

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

16. **Declaration of LLM usage**

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

    Answer: [Yes]

    Justification: LLMs are used as evaluated modules in the agent workflows, as described in the paper and appendix.

    Guidelines:

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