Preprints 2026, 2026110942. https://doi.org/10.20944/preprints202611.0942.v1

Wenbo: Scaling Agent Reasoning via Dynamic DAG Scheduling

Feiyang Chen
feiyang73@sjtu.edu.cn
Shanghai Jiao Tong Univ.
Shanghai, China

Junqing Yang
jy5675@nyu.edu.cn
New York Univ.
New York, United States

Wentao Zhang
zhang_wt@sjtu.edu.cn
Shanghai Jiao Tong Univ.
Shanghai, China

Zheng Wang
wzheng@sjtu.edu.cn
Shanghai Jiao Tong Univ.
Shanghai, China

Li Pan
panli@sjtu.edu.cn
Shanghai Jiao Tong Univ.
Shanghai, China

Jianhua Li
lijh888@sjtu.edu.cn
Shanghai Jiao Tong Univ.
Shanghai, China

Abstract

Large-scale LLM agent serving suffers from resource fragmentation due to task-level resource allocation. We present **Wenbo**, a scalable agent reasoning system based on dynamic DAG scheduling. Wenbo consists of three components: a *Director* that incrementally constructs execution DAGs, a *Scheduler* that globally dispatches ready nodes based on resource requirements, and heterogeneous *Workers* that execute nodes on GPUs and CPUs. Our theoretical analysis shows that the potential speedup depends on the balance between resource availability and workload demands. For example, with one GPU and 1,024 CPUs, the potential speedup can be 1,024× for CPU-heavy workloads with suitable DAG structures. In addition, we introduce **ARGB**, an agent reasoning graph benchmark comprising real-world workloads and configurable synthetic workloads. Experiments show that Wenbo achieves 1.4–3.8× higher throughput than Flash-Searcher on real-world workloads, with even much larger gains on synthetic workloads. Our code is available at https://github.com/wenbo-project/wenbo.

CCS Concepts

• **Computer systems organization** → **Distributed architectures**; *Cloud computing*; • **Computing methodologies** → *Artificial intelligence*.

Keywords

Agent systems; DAG-based scheduling; Heterogeneous computing

1 Introduction

The remarkable reasoning capabilities of large language models (LLMs) have made LLM-based agents a powerful paradigm for solving complex Web tasks that require planning, tool use, memory access, and interaction with external environments. By combining LLM inference with external tools and control logic, agent frameworks such as LangChain, AutoGen, and BabyAGI support applications ranging from code generation to multi-step research assistance [4, 14, 25]. As these applications evolve toward large-scale Web services with massive numbers of concurrent agent tasks, efficiently managing the computation and resources required by agent reasoning becomes an increasingly important systems challenge.

Existing agent systems typically adopt a task-centric execution model, in which each task is assigned a fixed bundle of resources, such as GPUs and CPUs, for its entire lifetime [13]. This model is poorly suited to agent workloads because execution involves heterogeneous stages with different resource requirements. LLM inference is primarily GPU-intensive, whereas tool invocation, retrieval, data processing, and orchestration typically rely on CPUs. These stages alternate dynamically, leaving some allocated resources idle while others remain busy. Such coarse-grained resource ownership causes resource fragmentation and limits cluster utilization. Scaling agent services therefore requires decoupling individual computation steps from task-level resource allocation.

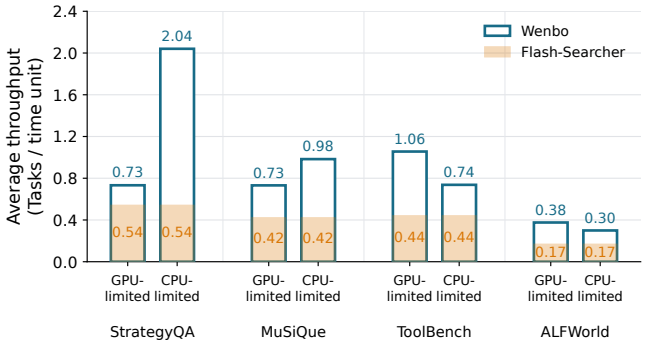


Figure 1: Throughput of Wenbo and Flash-Searcher on four datasets under GPU-limited (2 GPUs, 8 CPUs) and CPU-limited (8 GPUs, 2 CPUs) configurations.

Recent advances in LLM reasoning, exemplified by Graph-of-Thoughts [3] and subsequent work on graph-structured reasoning [28], demonstrate the benefits of organizing reasoning as interconnected steps rather than a linear sequence. Inspired by this insight, we represent agent execution as a dynamic reasoning graph, where nodes denote computation steps and edges encode dependencies. This representation exposes opportunities for *node-level parallelism*: independent steps can execute concurrently, even across different agent tasks. Figure 1 illustrates the benefits of this approach. Across real-world workloads under both GPU-limited and CPU-limited configurations, our global node-level scheduling strategy achieves 1.4×–3.8× higher throughput than Flash-Searcher [17].

Motivated by this observation, we present **Wenbo**, a DAG-centric execution system for massive agent reasoning workloads. As shown in Figure 2, Wenbo decouples agent reasoning from resource management through three major components. The *Director* transforms

Junqing Yang was an undergraduate at Shanghai Jiao Tong University during this work.

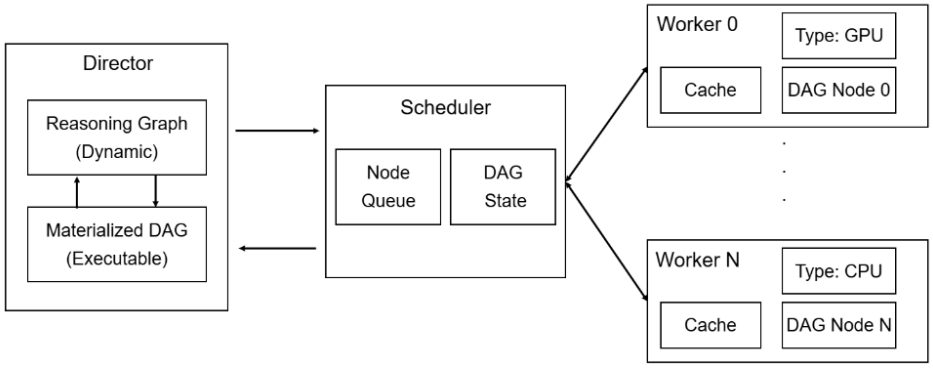


Figure 2: Architecture of Wenbo.

incoming agent tasks into dynamic reasoning graphs and incrementally materializes executable DAG flows according to intermediate reasoning states. The *Scheduler* maintains a global view of executable computation nodes across all active DAGs and dispatches them to heterogeneous GPU and CPU workers based on their computation requirements. The *Workers* execute individual computation nodes and exchange intermediate results through a shared execution layer. By separating reasoning orchestration, resource scheduling, and hardware execution, Wenbo enables fine-grained resource sharing and scalable parallel execution across heterogeneous computing resources.

We further develop a theoretical model to characterize the throughput advantage of node-level scheduling over task-level scheduling. Our analysis shows that the key to throughput gains is the alignment between resource supply and workload demand: when both imbalances favor the same resource type, node-level scheduling can substantially improve resource utilization and throughput. Under idealized assumptions, the gain can approach the scale of the resource-count imbalance. For example, with one GPU and 1,024 CPUs, the gain can reach the order of 1,024× when workload demands and DAG dependencies permit effective resource utilization.

To enable systematic evaluation of agent reasoning systems, we introduce the Agent Reasoning Graph Benchmark (**ARGB**). It comprises real-world workloads (**ARGB-R**) derived from StrategyQA [5], MuSiQue [23], ToolBench/ToolLLM [18], and ALFWorld [19], as well as configurable synthetic workloads (**ARGB-S**). Our evaluation demonstrates that Wenbo not only delivers the throughput improvements described above but also achieves utilization improvement ratios of 1.36×–2.71×. Under mixed-priority workloads, it reduces P99 first-service waiting time by 2.3×–26.0× and lowers average P99 end-to-end response time by 1.55×. On synthetic workloads with highly imbalanced resource demands, Wenbo achieves even larger performance gains, further demonstrating the advantages of fine-grained scheduling under heterogeneous resource constraints.

Contributions. This paper makes the following contributions.

- We identify task-level resource allocation as a bottleneck in large-scale agent reasoning and motivate node-level parallelism.
- We implement **Wenbo**, a scalable agent reasoning system that enables efficient node-level parallelism by dynamic DAG scheduling and heterogeneous GPU/CPU execution.
- We model the throughput gains of node-level scheduling and bound its speedup by resource and workload imbalance.
- We introduce **ARGB**, a benchmark comprising real-world and synthetic agent reasoning workloads.

2 System Design

As shown in Figure 2, Wenbo consists of three major components: the *Director*, which manages the evolution of reasoning graphs and generates executable computation plans; the *Scheduler*, which performs global dependency-aware node scheduling across heterogeneous workers; and the *Workers*, which execute individual computation nodes and provide runtime feedback.

2.1 Director: Dynamic DAG Generation

The Director serves as the reasoning control plane of Wenbo. Unlike a traditional workflow planner that generates a fixed execution graph before runtime, the Director manages a dynamically evolving reasoning process, where future computation steps may depend on intermediate execution results.

Given an incoming agent task, the Director first analyzes the task objective and constructs an initial reasoning graph. Each node in the reasoning graph represents an atomic reasoning or computation step, such as LLM inference, tool invocation, retrieval, verification, or data processing, while edges represent logical dependencies among these steps. The reasoning graph captures the semantic structure of how the agent solves a problem, rather than only the execution order.

To enable distributed execution, the Director incrementally materializes executable DAG flows from the current reasoning graph.

During this process, the Director identifies executable computation nodes, determines their dependency constraints, and annotates each node with execution requirements, including computation type and resource preference. The generated DAG therefore serves as an execution representation of the current reasoning state, which can be directly consumed by the Scheduler for distributed execution.

The reasoning graph is continuously updated throughout task execution. Intermediate results produced by Workers may reveal new information, invalidate previous assumptions, or introduce additional reasoning requirements. Upon receiving execution feedback, the Director evolves the reasoning graph by creating new nodes, refining dependencies, expanding new execution branches, or terminating unnecessary computation paths. It then incrementally updates the corresponding executable DAG flows. This dynamic graph evolution enables Wenbo to support open-ended agent reasoning, where the complete execution plan is not required to be known before execution begins.

In addition to reasoning graph management, the Director maintains the global context state of each agent task. It manages the lifecycle of intermediate data generated across different computation nodes, tracks dependencies among data objects, and provides required context when new reasoning steps are generated. By maintaining both the evolving reasoning graph and the associated task context, the Director bridges high-level agent reasoning with low-level distributed execution.

2.2 Scheduler: DAG-aware Node Scheduling

The Scheduler is the execution control plane of Wenbo. Its primary responsibility is to map executable DAG nodes to appropriate heterogeneous computing resources. Based on the node types and resource requirements specified by the Director, the Scheduler dispatches each node to a suitable GPU or CPU worker. It continuously monitors resource availability and execution progress, allowing nodes from the same DAG as well as nodes from different DAGs to execute concurrently. In this way, the Scheduler separates the logical reasoning plan from its physical execution and enables Wenbo to efficiently utilize a heterogeneous cluster.

The Scheduler also provides elastic fault tolerance for distributed execution. Workers may fail due to hardware failures, resource exhaustion, network errors, or other runtime issues. When a worker failure is detected, the Scheduler identifies the affected nodes and reschedules retryable tasks to other suitable workers without restarting the entire DAG. Since the execution state and intermediate results of completed nodes are preserved, Wenbo can recover at the granularity of individual DAG nodes and avoid unnecessary recomputation. This design allows the system to continue making progress despite failures in the underlying worker pool.

In addition, the Scheduler maintains the runtime state of active DAGs, including node execution status, worker assignments, and intermediate results. It communicates execution updates to the Director, enabling the latter to refine the reasoning plan based on intermediate outcomes. When the Director introduces new nodes or modifies dependencies, the Scheduler incorporates these changes into the active execution state and continues scheduling eligible nodes. The Scheduler thus connects the evolving reasoning process managed by the Director with the physical computation performed by Workers.

2.3 Worker: Heterogeneous Node Execution

Workers are the execution layer of Wenbo. Their primary responsibility is to execute individual DAG nodes dispatched by the Scheduler. Upon receiving a task, a worker obtains the required inputs and execution context, performs the corresponding computation, and produces the node output. Workers are agnostic to the global reasoning process and focus on reliably executing the specific computation assigned to them.

Workers also manage their local computing resources, including GPUs, CPUs, memory, models, and runtime environments. Each worker maintains the state of its available resources and exposes this information to the Scheduler, allowing the Scheduler to make resource-aware placement decisions. For GPU workers, this may include managing model instances, GPU memory, and inference environments; for CPU workers, it may include managing CPU capacity, memory, and execution environments. This local resource management hides hardware-specific details from the higher-level scheduling layer.

Finally, Workers continuously provide execution feedback to the Scheduler. In addition to the output of each DAG node, this feedback includes execution status, resource utilization, execution time, failures, and other runtime information. The Scheduler uses this information to update the current DAG execution state and make subsequent scheduling decisions, while the Director can use relevant execution outcomes to determine whether the reasoning DAG should be further updated. Workers therefore close the execution loop between physical computation, scheduling, and the evolution of agent reasoning.

3 Scheduling Analysis

Consider a computing pool with N_{GPU} identical GPUs and N_{CPU} identical CPUs. An infinite stream of homogeneous agent tasks arrives. Each task is pre-converted into a DAG of computational nodes. Every node is either a GPU node or a CPU node and has unit execution time. The total GPU work and total CPU work of one task are denoted by $\tau_{\mathrm{GPU}} > 0$ and $\tau_{\mathrm{CPU}} > 0$, respectively. There are no cross-task dependencies, and nodes from different tasks may execute concurrently, provided intra-task dependencies are respected.

3.1 Node-Level vs. Task-Level Scheduling

Definition 3.1 (Task-level parallelism, baseline). The system maintains a queue of tasks. At most $K = \min(N_{\mathrm{GPU}}, N_{\mathrm{CPU}})$ tasks can be in service simultaneously. Each admitted task is allocated exactly one GPU and one CPU, and it holds both resources until completion. Its DAG nodes are scheduled on this fixed resource pair as soon as their intra-task dependencies allow.

Lemma 3.2 (Execution time and throughput of task-level parallelism). *Let T_{task} be the minimum wall-clock execution time of one task under Definition 3.1. Then*

$$\max(\tau_{\mathrm{GPU}}, \tau_{\mathrm{CPU}}) \;\le\; T_{\mathrm{task}} \;\le\; \tau_{\mathrm{GPU}} + \tau_{\mathrm{CPU}}. \tag{1}$$

Moreover, the steady-state throughput of task-level parallelism is

$$\Theta_{\text{task}} = \frac{K}{T_{\text{task}}}, \tag{2}$$

where $K = \min(N_{\text{GPU}}, N_{\text{CPU}})$.

Proof of Lemma 3.2. A task receives exactly one GPU and one CPU. The GPU must spend a total of τ_{GPU} busy time and the CPU must spend a total of τ_{CPU} busy time. Even with perfect overlap, the task cannot finish before both totals have been served, hence before $\max(\tau_{\text{GPU}}, \tau_{\text{CPU}})$.

Conversely, because there are only two resources and the task is a DAG, at least one of the two resources can always be busy until the task completes. Indeed, if work remains, any source node in the remaining DAG has all dependencies satisfied and is ready to execute on either the GPU or the CPU. Therefore the makespan cannot exceed the total busy time $\tau_{\text{GPU}} + \tau_{\text{CPU}}$.

Since K tasks are in service simultaneously and each completes after T_{task}, the departure rate is K/T_{task}. □

Definition 3.3 (Node-level parallelism). All nodes of all tasks are submitted to a global scheduler that can map any GPU node to any idle GPU and any CPU node to any idle CPU as soon as its intra-task dependencies are met. There is no restriction that a task must be assigned to a fixed resource pair, and nodes from different tasks may be interleaved freely.

Lemma 3.4 (Ideal throughput of Node-level parallelism). *Under Definition 3.3, zero scheduling overhead, and perfect cross-task pipelining, the maximal system throughput is*

$$\Theta_{\text{node}} = \min\left(\frac{N_{\text{GPU}}}{\tau_{\text{GPU}}}, \frac{N_{\text{CPU}}}{\tau_{\text{CPU}}}\right). \tag{3}$$

Proof of Lemma 3.4. Let ρ denote the steady-state task completion rate. Each completed task consumes τ_{GPU} GPU busy time and τ_{CPU} CPU busy time. Therefore, $\rho\,\tau_{\text{GPU}} \le N_{\text{GPU}}$ and $\rho\,\tau_{\text{CPU}} \le N_{\text{CPU}}$. Hence

$$\rho \le \min\left(\frac{N_{\text{GPU}}}{\tau_{\text{GPU}}}, \frac{N_{\text{CPU}}}{\tau_{\text{CPU}}}\right).$$

Under the ideal assumptions, the bottleneck resource can be kept fully occupied by interleaving nodes from different tasks, so the upper bound is achievable. □

Theorem 3.5 (Throughput improvement factor). *Let* $\lambda = \Theta_{\text{node}}/\Theta_{\text{task}}$ *be the throughput improvement factor of Node-level parallelism over task-level parallelism. Then*

$$\lambda_{\min} \le \lambda \le \lambda_{\max}, \tag{4}$$

where

$$\lambda_{\min} = \frac{\max(\tau_{\text{GPU}}, \tau_{\text{CPU}})}{\min(N_{\text{GPU}}, N_{\text{CPU}})} \min\left(\frac{N_{\text{GPU}}}{\tau_{\text{GPU}}}, \frac{N_{\text{CPU}}}{\tau_{\text{CPU}}}\right), \tag{5}$$

$$\lambda_{\max} = \frac{\tau_{\text{GPU}} + \tau_{\text{CPU}}}{\min(N_{\text{GPU}}, N_{\text{CPU}})} \min\left(\frac{N_{\text{GPU}}}{\tau_{\text{GPU}}}, \frac{N_{\text{CPU}}}{\tau_{\text{CPU}}}\right). \tag{6}$$

Proof. Let

$$K = \min(N_{\text{GPU}}, N_{\text{CPU}}), \qquad D = \min\left(\frac{N_{\text{GPU}}}{\tau_{\text{GPU}}}, \frac{N_{\text{CPU}}}{\tau_{\text{CPU}}}\right).$$

From Lemma 3.2 and Lemma 3.4,

$$\lambda = \frac{\Theta_{\text{node}}}{\Theta_{\text{task}}} = \frac{D}{K/T_{\text{task}}} = \frac{T_{\text{task}}}{K}\,D.$$

By Lemma 3.2, Eq. (1),

$$\max(\tau_{\text{GPU}}, \tau_{\text{CPU}}) \le T_{\text{task}} \le \tau_{\text{GPU}} + \tau_{\text{CPU}}.$$

Because $K > 0$ and $D > 0$, multiplying the latter inequality by D/K preserves the ordering:

$$\frac{\max(\tau_{\text{GPU}}, \tau_{\text{CPU}})}{K}\,D \le \frac{T_{\text{task}}}{K}\,D \le \frac{\tau_{\text{GPU}} + \tau_{\text{CPU}}}{K}\,D.$$

Substituting the definitions of K and D yields exactly (5)–(6). □

Corollary 3.6 (Tightness of the gain range). *For any* $\tau_{\text{GPU}}, \tau_{\text{CPU}} > 0$*, the throughput gain satisfies*

$$1 \le \lambda_{\min} \le \lambda \le \lambda_{\max} \le 2\,\lambda_{\min}, \tag{7}$$

and, more precisely,

$$\frac{\lambda_{\max}}{\lambda_{\min}} = \frac{\tau_{\text{GPU}} + \tau_{\text{CPU}}}{\max(\tau_{\text{GPU}}, \tau_{\text{CPU}})} = 1 + \frac{\min(\tau_{\text{GPU}}, \tau_{\text{CPU}})}{\max(\tau_{\text{GPU}}, \tau_{\text{CPU}})} \in (1, 2]. \tag{8}$$

Proof. Immediate from dividing (6) by (5). The ratio is at most 2 because $\tau_{\text{GPU}} + \tau_{\text{CPU}} \le 2\max(\tau_{\text{GPU}}, \tau_{\text{CPU}})$, and strictly greater than 1 because $\min(\tau_{\text{GPU}}, \tau_{\text{CPU}}) > 0$. □

Corollary 3.6 shows that the lower bound is a useful predictor of the throughput gain: the actual gain is at most twice $\lambda_{\min}$. The bounds are tightest when GPU and CPU workloads are highly skewed, in which case their ratio approaches one. They are widest when the two workloads are equal, but the uncertainty remains within a factor of two. Thus, device counts and per-task resource demands provide a useful estimate of the potential benefit without requiring detailed DAG-level analysis.

3.2 Quick Estimation of Throughput Gain

Recall from Theorem 3.5 that the throughput gain of DAG-node-level scheduling over task-level scheduling is $\lambda = D T_{\text{task}}/K$. For quick estimation, we use $T_{\text{task}} \approx \max(\tau_{\text{GPU}}, \tau_{\text{CPU}})$, which provides a conservative estimate of the throughput gain under the idealized model, without requiring detailed DAG topology analysis.

Case 1: Device Counts and Task Workloads Are Known. When both device counts and per-task workloads are known, substituting the above approximation yields:

$$\lambda \approx \frac{\max(\tau_{\text{GPU}}, \tau_{\text{CPU}})}{\min(N_{\text{GPU}}, N_{\text{CPU}})} \min\left(\frac{N_{\text{GPU}}}{\tau_{\text{GPU}}}, \frac{N_{\text{CPU}}}{\tau_{\text{CPU}}}\right). \tag{9}$$

This expression can be simplified according to the relative sizes of the device pools and task workloads:

$$\lambda \approx \begin{cases} \min\left(\dfrac{N_{\text{GPU}}}{N_{\text{CPU}}}, \dfrac{\tau_{\text{GPU}}}{\tau_{\text{CPU}}}\right), & N_{\text{GPU}} \ge N_{\text{CPU}},\ \tau_{\text{GPU}} \ge \tau_{\text{CPU}}, \\ \min\left(\dfrac{N_{\text{CPU}}}{N_{\text{GPU}}}, \dfrac{\tau_{\text{CPU}}}{\tau_{\text{GPU}}}\right), & N_{\text{GPU}} \le N_{\text{CPU}},\ \tau_{\text{GPU}} \le \tau_{\text{CPU}}, \\ 1, & \text{otherwise.} \end{cases} \tag{10}$$

When device imbalance and workload imbalance are in the same direction, the estimated gain is the smaller of the two ratios. Otherwise, the estimated gain is approximately 1×. *Example:* With 1

Table 1: Task sources, graph structures, and resource composition of ARGB-R.

Dataset	Task Type	Graphs	Nodes	Edges	Avg. Depth	GPU/CPU Ratio
StrategyQA	Factual reasoning and Boolean QA	2,061	7,642	6,090	2.67	73.6% / 26.4%
MuSiQue	Multi-hop evidence reasoning	19,938	95,044	85,668	4.25	57.3% / 42.7%
ToolBench	Tool selection and invocation	26,999	124,408	131,936	3.20	41.1% / 58.9%
ALFWorld	Embodied planning and interaction	6,374	76,455	79,065	10.69	44.4% / 55.6%
Total	Four public task sources	55,372	303,549	302,759	4.42	47.8% / 52.2%

GPU, 1,024 CPUs, and per-task workloads of 1 GPU unit and 64 CPU units, the estimated gain is 64×.

Case 2: Only Device Counts Are Known. When only device counts are available, the workload imbalance is unknown. We can estimate the potential gain by assuming that the workload imbalance is in the same direction as, and at least as large as, the device imbalance. Substituting this assumption into the approximation in Case 1 yields:

$$\lambda \approx \frac{\max(N_{\text{GPU}}, N_{\text{CPU}})}{\min(N_{\text{GPU}}, N_{\text{CPU}})}. \tag{11}$$

Example: With 1 GPU and 1,024 CPUs, the potential gain can be on the order of 1,024×, assuming a CPU/GPU workload ratio of at least 1,024 and suitable DAG structures.

Case 3: Only Workload Composition Is Known. When device counts are unknown but the GPU/CPU workload composition is available, we can estimate the ideal resource provisioning ratio and the corresponding throughput gain. The ideal provisioning ratio balances GPU and CPU processing capacities. Under this ratio, the device and workload imbalances are aligned. Substituting it into the same-direction branch of Case 1 gives:

$$\frac{N_{\text{GPU}}}{N_{\text{CPU}}} = \frac{\tau_{\text{GPU}}}{\tau_{\text{CPU}}} \quad \Longrightarrow \quad \lambda \approx \frac{\max(\tau_{\text{GPU}}, \tau_{\text{CPU}})}{\min(\tau_{\text{GPU}}, \tau_{\text{CPU}})}. \tag{12}$$

Example: If GPU and CPU workloads account for 20% and 80% of the total workload, respectively, the recommended GPU/CPU provisioning ratio is 1:4, with an estimated throughput gain of 4× under this provisioning.

4 ARGB: Agent Reasoning Graph Benchmark

We introduce **ARGB** (**A**gent **R**easoning **G**raph **B**enchmark) to standardize agent reasoning workloads and evaluate large-scale agent execution systems. ARGB represents each task as a directed acyclic graph (DAG), where nodes denote atomic operations, such as LLM inference and tool invocation, and edges encode execution dependencies. Each node specifies its resource type (GPU or CPU), memory requirements, and execution latency. The benchmark comprises two subsets: **ARGB-R**, derived from public task datasets, and **ARGB-S**, generated using a parameterized synthetic workload generator.

4.1 ARGB-R: Real-World Workloads

ARGB-R is derived from four public agent reasoning task datasets: StrategyQA [5], MuSiQue [23], ToolBench/ToolLLM [18], and ALFWorld [19]. These datasets cover factual reasoning, multi-hop evidence reasoning, tool use, and embodied interaction, respectively. Table 1 summarizes their graph statistics and resource compositions. We construct ARGB-R in three steps.

First, we extract task-specific information from each dataset, including questions and supporting facts from StrategyQA; questions and evidence passages from MuSiQue; task descriptions, available tools, and invocation traces from ToolBench; and goals, environment descriptions, and action sequences from ALFWorld.

Second, we provide the extracted information to `deepseek-v4-flash` and prompt it to construct a DAG that captures each task's reasoning and execution dependencies. Subtasks are represented as nodes, and directed edges encode prerequisite relationships. We assign LLM-intensive operations, such as reasoning, planning, and answer generation, to GPUs, while tool invocation, environment interaction, and deterministic processing are assigned to CPUs.

Finally, we normalize the generated graphs into a unified JSON format. Each graph records its source dataset, each node contains an identifier, execution prompt, and resource label, and each edge specifies a source-destination pair. This unified representation enables consistent workload processing and scheduling across different task domains.

4.2 ARGB-S: Synthetic Workloads

ARGB-S uses a parameterized generator to produce reproducible DAG workloads with configurable graph structures, resource requirements, and task concurrency. It adopts the same graph representation and JSON schema as ARGB-R, enabling controlled evaluation under diverse workload conditions.

The generator varies the number of layers, layer widths, and connectivity to produce DAGs with different sizes, dependency structures, and degrees of parallelism. By assigning uniform execution times to all nodes, these workloads isolate the effects of graph topology and dependency constraints, allowing us to evaluate how effectively schedulers exploit intra-task parallelism.

The generator also controls the proportions of CPU and GPU nodes. Because each node can execute only on a compatible resource, resource contention can delay ready nodes even when all nodes have identical execution times. These workloads therefore evaluate how schedulers coordinate dependency readiness with resource availability and how resource composition affects scheduling performance.

Finally, the generator supports batches of independent DAGs that share a common pool of CPU and GPU resources. By varying the number of concurrent DAGs, we control inter-task contention and the number of competing ready nodes. The scheduler must allocate heterogeneous resources across tasks while respecting their

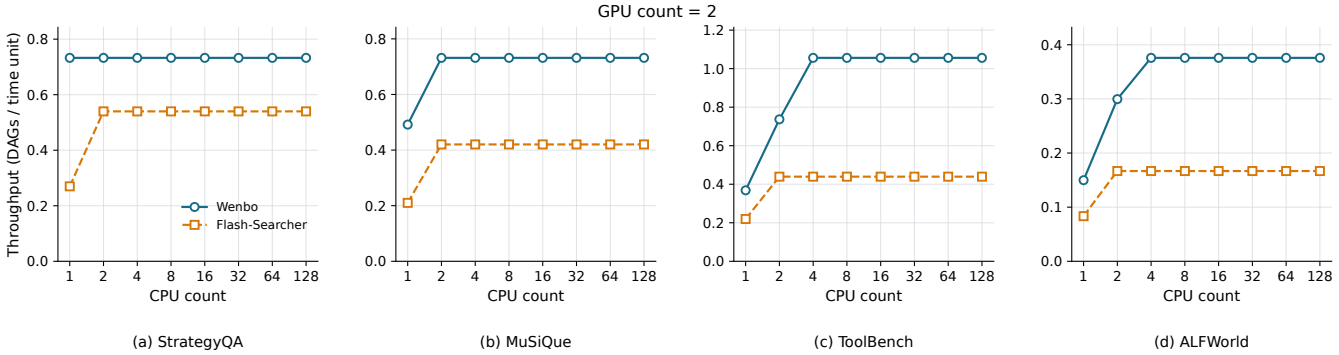


Figure 3: Throughput scaling with CPU resources on ARGB-R, with 2 GPUs fixed.

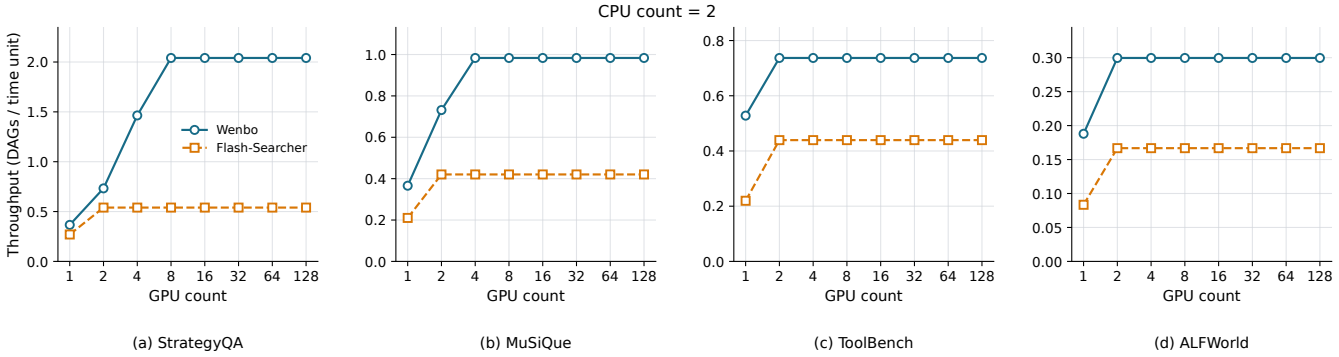


Figure 4: Throughput scaling with GPU resources on ARGB-R, with 2 CPUs fixed.

dependencies. These workloads evaluate inter-task resource allocation and scheduling efficiency as concurrency increases, reflecting scenarios in which multiple agent reasoning tasks execute simultaneously.

5 Experimental Evaluation

We evaluate Wenbo on real-world and synthetic workloads, focusing on throughput, device utilization, and tail latency under mixed-priority workloads.

5.1 Experimental Setup

Workload Configuration. All experiments use the datasets introduced in ARGB. For simplicity, we assign every node an execution cost of one time unit, regardless of device type, and ignore communication and other overheads. Within each comparison, all methods process the same complete dataset in the same task order and run until all inference tasks finish.

Baseline. We compare methods that support DAG-based reasoning workflows. Flash-Searcher [17] is a state-of-the-art approach to DAG-based agent reasoning, originally designed for single-task execution. We adapt it for parallel workloads by statically assigning one GPU and one CPU to each reasoning task and reserving both resources for the lifetime of its DAG.

5.2 Evaluation on ARGB-R

5.2.1 ***Throughput Scaling.*** Figures 3 and 4 show throughput as the numbers of CPUs and GPUs vary, respectively, while the other device type is fixed. We can find the following three key observations. First, Wenbo achieves 2×–4× higher throughput than Flash-Searcher across all four datasets under identical resource configurations. The reason is that: by scheduling ready nodes across DAGs, Wenbo dynamically shares resources, whereas Flash-Searcher reserves them even when tasks await dependencies. Second, adding resources of only one device type barely improves Flash-Searcher's throughput, whereas Wenbo benefits from additional resources in most configurations. This further shows that by decoupling resource allocation from task lifetime, Wenbo exposes more of the available node-level parallelism. Third, Wenbo's throughput gains eventually plateau as additional resources shift the bottleneck to the other device type. This is because that the maximum gain from node-level over task-level parallelism is bounded by the smaller of the resource imbalance ratio and workload imbalance ratio. This behavior is consistent with the resource-capacity constraints established in our theoretical analysis.

5.2.2 ***Device Utilization.*** Table 2 reports GPU utilization as the number of CPUs varies while the number of GPUs remains fixed at

Table 2: GPU utilization (%) on ARGB-R with the number of GPUs fixed at 2.

#CPUs	StrategyQA			MuSiQue			ToolBench			ALFWorld		
	Flash-Searcher	Wenbo	Gain↑	Flash-Searcher	Wenbo	Gain↑	Flash-Searcher	Wenbo	Gain↑	Flash-Searcher	Wenbo	Gain↑
1	36.833	99.940	2.71×	28.724	67.200	2.34×	20.803	34.894	1.68×	22.180	39.847	1.80×
2	73.650	99.940	1.36×	57.447	99.994	1.74×	41.604	69.785	1.68×	44.356	79.684	1.80×
4	73.650	99.940	1.36×	57.447	99.994	1.74×	41.604	99.994	2.40×	44.356	99.958	2.25×
8	73.650	99.940	1.36×	57.447	99.994	1.74×	41.604	99.994	2.40×	44.356	99.959	2.25×
16	73.650	99.940	1.36×	57.447	99.994	1.74×	41.604	99.994	2.40×	44.356	99.959	2.25×
32	73.650	99.940	1.36×	57.447	99.994	1.74×	41.604	99.994	2.40×	44.356	99.959	2.25×
64	73.650	99.940	1.36×	57.447	99.994	1.74×	41.604	99.994	2.40×	44.356	99.959	2.25×
128	73.650	99.940	1.36×	57.447	99.994	1.74×	41.604	99.994	2.40×	44.356	99.959	2.25×

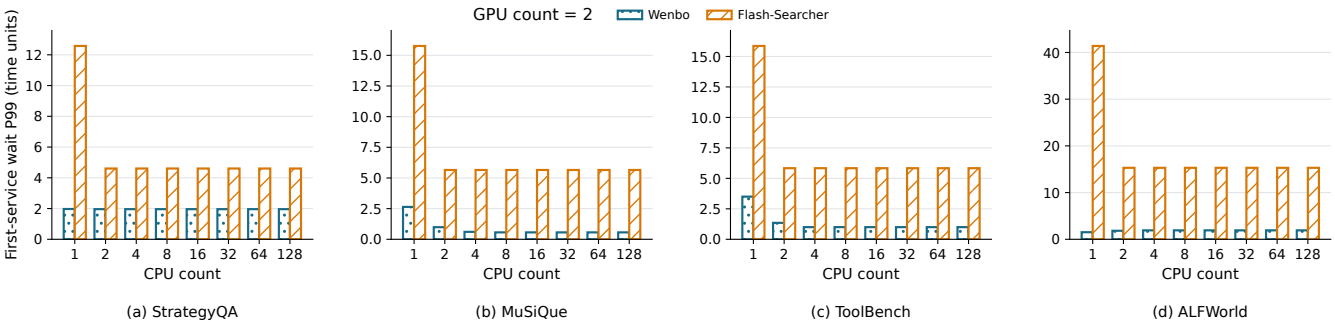


Figure 5: P99 first-service waiting time on ARGB-R as the number of CPUs varies, with the number of GPUs fixed at 2.

two.[1] Wenbo achieves higher GPU utilization than Flash-Searcher across all four datasets, with utilization improvement ratios ranging from 1.36× to 2.71×. In addition, we also find that the utilization gains vary across workloads. The magnitude of the improvement varies across workloads. For StrategyQA and MuSiQue, Wenbo achieves nearly saturated GPU utilization even with a small number of CPUs, whereas Flash-Searcher's GPU utilization remains substantially lower. For ToolBench and ALFWorld, Wenbo requires more CPU resources to approach full GPU utilization, indicating that CPU-side execution and DAG dependencies play a more prominent role in sustaining GPU demand for these workloads. Overall, these results show that the effectiveness of heterogeneous resource utilization depends on both workload structure and the scheduling granularity.

5.2.3 ***Priority Scheduling under Mixed Workloads.*** Following [6], we evaluate foreground latency under Poisson arrivals using two metrics: P99 first-service waiting time (from arrival to first-node execution) and P99 end-to-end response time (from arrival to DAG completion). We maintain 16 outstanding background requests, replenishing each upon completion. Foreground and background requests are sampled uniformly with replacement from the same workload-specific DAG set, differing only in priority. Foreground arrivals follow a Poisson process at 20% of batch-average throughput. After warming up with 1,000 foreground requests, we measure the next 10,000 and wait for all to complete.

Table 3: P99 foreground response time (units) in the same-source mixed-workload experiment on ARGB-R, with 2 GPUs, 8 CPUs, and 16 outstanding background requests.

Dataset	Foreground P99 Response Time		Speedup↑
	Flash-Searcher	Wenbo	
StrategyQA	8.892	5.911	1.50×
MuSiQue	10.881	7.035	1.55×
ToolBench	11.458	7.251	1.58×
ALFWorld	29.631	18.837	1.57×

As shown in Figure 5, Wenbo substantially reduces P99 first-service waiting time across all workloads, achieving 2×–30× lower waiting times than the baseline. We also find that the magnitude of the improvement varies across datasets, reflecting differences in their task-level device-demand distributions and, consequently, in the devices required by the DAG nodes selected for first service. Table 3 further shows that Wenbo reduces P99 end-to-end response time by an average of 1.55× across all four datasets. The smaller gains suggest that subsequent execution and dependency constraints continue to contribute to tail latency. Overall, Wenbo improves both foreground responsiveness and end-to-end tail performance under mixed-priority workloads.

[1]The CPU-scarce scenario can be obtained by swapping the two device types, yielding similar conclusions. Due to space constraints, we focus on the GPU-scarce scenario in the remaining experiments.

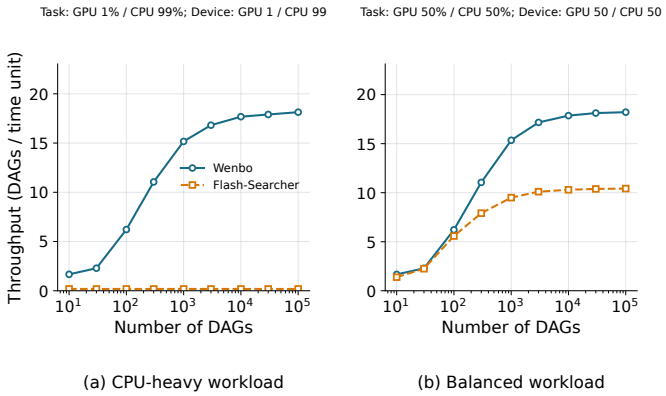


Figure 6: Throughput as the DAG number varies on ARGB-S.

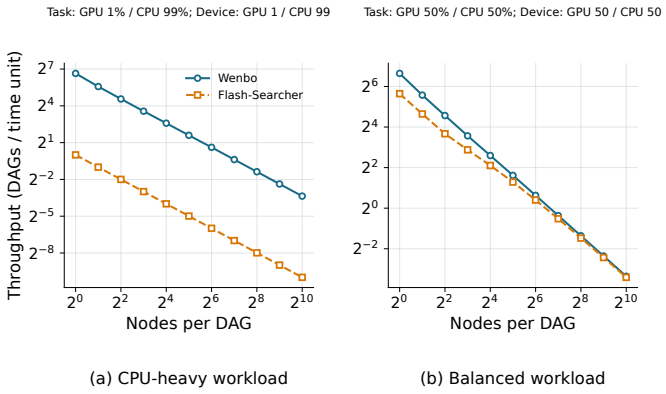


Figure 7: Throughput as the DAG size varies on ARGB-S.

5.3 Evaluation on ARGB-S

5.3.1 ***Scaling with Task Number.*** We generate synthetic DAGs that match the structural distributions of the real-world ARGB-R datasets, with 5.48 ± 2.73 nodes and a depth of 4.42 ± 2.58. We evaluate two workload configurations. As shown in Figure 6(a), under the CPU-heavy workload, the throughput speedup of Wenbo over Flash-Searcher increases from 9× to 99× as the number of DAGs grows. On the other hand, as shown in Figure 6(b), Wenbo shows a smaller but consistent throughput advantage for Wenbo under the balanced workload, with speedups increasing from 1.1× to 1.75×. The reason is that the task node-type distribution better matches device availability, reducing resource waste caused by task-level resource pairing. Overall, both results align with our theoretical analysis: the maximum speedup of node-level over task-level parallelism is bounded by the smaller of the resource imbalance ratio and the workload imbalance ratio.

5.3.2 ***Impact of DAG Size.*** To evaluate the impact of task complexity, we fix the number of inference tasks at 1,000 and vary the number of nodes per DAG, while keeping the workload distributions and device configurations unchanged from the previous experiment. As shown in Figure 7, although throughput decreases for both methods as DAG size increases, Wenbo maintains an approximately 99× throughput advantage over Flash-Searcher. This result is consistent with our theoretical analysis of the upper bound on the speedup of node-level over task-level parallelism. However, we also notice that under the balanced workload (Figure 7(b)), the performance advantage of Wenbo is less pronounced and diminishes as DAGs grow. This is because balanced tasks and devices leave less room for improvement through node-level parallelism. Moreover, larger DAGs expose more executable nodes, allowing Flash-searcher to saturate the available devices and thereby reducing the marginal benefit of Wenbo's scheduling strategy.

6 Related Work

6.1 Agent Reasoning Systems

Frameworks such as LangChain, AutoGen, and BabyAGI support agent composition, multi-agent interaction, and LLM-driven task planning [4, 14, 25]. Distributed execution frameworks such as Ray provide infrastructure for parallel task execution, while agent-serving systems target the efficient execution of agent workloads [13]. Recent systems incorporate application- or program-level context into agent serving. Parrot, Agentix, and ThunderAgent leverage request dependencies, program execution state, and workflow resource information, respectively, to improve scheduling and resource management [8, 11, 12]. However, coarse-grained task-level allocation can cause resource fragmentation because agent workflows interleave GPU-intensive LLM inference with CPU-intensive operations, such as tool execution and memory retrieval. As a result, GPUs may remain idle during CPU-bound phases, while CPU resources are underutilized during GPU-bound phases. Other work optimizes resource utilization within LLM serving. Memory management and chunked-prefill scheduling improve serving efficiency [2, 10], DistServe decouples prefill and decoding to optimize GPU allocation [30], and InferCept reduces context recomputation and GPU resource waste during external interactions [1]. In contrast, Wenbo coordinates dependent CPU- and GPU-bound operations across complete agent workflows.

6.2 DAG-based LLM Reasoning

Tree of Thoughts explores alternative reasoning paths through tree-based search, while Graph of Thoughts generalizes reasoning to graph structures for more flexible composition of intermediate states [3, 27]. These approaches primarily improve reasoning quality rather than optimize execution across concurrent workflows. Other systems exploit operation dependencies to expose parallelism. LLM-Compiler identifies dependencies among function calls and executes independent calls concurrently [9]. Ayo decomposes LLM applications into fine-grained primitives and constructs dataflow graphs to identify cross-module optimization opportunities [22]. Plato represents subquestion dependencies as a DAG and parallelizes inference across nodes whose prerequisites are satisfied [7]. Agent execution DAGs pose additional challenges: dependencies may evolve at runtime, nodes vary in granularity and resource demands, and execution spans heterogeneous CPU and GPU workers. Wenbo dynamically schedules dependency-ready nodes across concurrent workflows, coordinating execution while exploiting available parallelism.

6.3 Heterogeneous Resource Scheduling

Prior work has extensively studied resource allocation for cluster computing and deep learning. TetriSched uses mixed-integer linear programming to schedule jobs with diverse resource requirements, Gandiva improves GPU utilization through time-slicing and migration, and Optimus dynamically adjusts resource allocation based on predicted job performance [16, 24, 26]. Gavel accounts for accelerator heterogeneity through an effective-throughput abstraction [15]. These systems primarily target jobs or training tasks rather than fine-grained operations in dynamically evolving agent workflows. Model-parallel systems such as Alpa decompose model computation into operators and map them across devices [29], addressing a different execution structure. For LLM serving, Llumnix balances load through request migration [21], while Preble jointly optimizes KV-cache reuse and computational load balancing [20]. However, these approaches do not directly address dependency-aware scheduling of heterogeneous operations across complete agent workflows. Wenbo integrates fine-grained execution with cross-workflow resource scheduling. Specifically, its global scheduler dispatches dependency-ready nodes to appropriate CPU or GPU workers, exploiting parallelism both within individual workflows and across concurrent agent trajectories.

7 Conclusion

We presented **Wenbo**, a DAG-centric execution system for scaling LLM-based agent reasoning. Wenbo decouples reasoning orchestration from resource management by incrementally materializing dynamic reasoning graphs into executable DAGs and scheduling individual computation steps across heterogeneous GPU and CPU resources. We further developed a theoretical framework to characterize the throughput benefits of DAG-level scheduling and evaluated the system on synthetic and realistic agent workloads, demonstrating significant improvements over task-level scheduling. Our work highlights DAG-centric execution as a promising approach to reducing resource fragmentation and improving the scalability of large-scale agent services.

References

[1] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. 2024. InferCept: Efficient Intercept Support for Augmented Large Language Model Inference. In *International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*. PMLR, 81–95.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 117–134.

[3] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *AAAI Conference on Artificial Intelligence*, Vol. 38. 17682–17690.

[4] Harrison Chase. 2022. LangChain. https://github.com/langchain-ai/langchain.

[5] Mor Geva, Daniel Khashabi, Elad Segal, Tushar Khot, Dan Roth, and Jonathan Berant. 2021. Did aristotle use a laptop? a question answering benchmark with implicit reasoning strategies. *Transactions of the Association for Computational Linguistics* 9 (2021), 346–361.

[6] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like clockwork: Performance predictability from the bottom up. In *USENIX Symposium on Operating Systems Design and Implementation*. 443–462.

[7] Shuowei Jin, Xueshen Liu, Yongji Wu, Haizhong Zheng, Qingzhao Zhang, Atul Prakash, Matthew Lentz, Danyang Zhuo, Feng Qian, and Z. Morley Mao. 2025. Plato: Plan to Efficiently Decode for Large Language Model Inference. In *Conference on Language Modeling*.

[8] Hao Kang, Ziyang Li, Weili Xu, Xinyu Yang, Yinfang Chen, Junxiong Wang, Beidi Chen, Tushar Krishna, Chenfeng Xu, and Simran Arora. 2026. ThunderAgent: A simple, fast and program-aware agentic inference system. *arXiv preprint arXiv:2602.13692* (2026).

[9] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. 2024. An LLM Compiler for Parallel Function Calling. In *International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*. PMLR, 24370–24391.

[10] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Symposium on Operating Systems Principles*. 611–626.

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

[12] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E Gonzalez, et al. 2026. Agentix: An Efficient Serving Engine for LLM Agents as General Programs. In *USENIX Symposium on Networked Systems Design and Implementation*. 2443–2459.

[13] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging AI applications. In *USENIX Symposium on Operating Systems Design and Implementation*. 561–577.

[14] Yohei Nakajima. 2023. BabyAGI. GitHub repository. https://github.com/yoheinakajima/babyagi_archive Accessed: 2026-08-18.

[15] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 481–498.

[16] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *European Conference on Computer Systems*. 1–14.

[17] Tianrui Qin, Qianben Chen, Sinuo Wang, Xinyu Yang, Kang Zhu, He Zhu, Dingfeng Shi, Xinxin Liu, Ge Zhang, Jiaheng Liu, Xitong Gao, Yuchen Eleanor Jiang, and Wangchunshu Zhou. 2026. Flash-Searcher: Fast and Effective Web Agents via DAG-Based Parallel Execution. In *International Conference on Learning Representations*, C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (Eds.), Vol. 2026. 45026–45066.

[18] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *International Conference on Learning Representations*, B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (Eds.). 9695–9717.

[19] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2020. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. *arXiv preprint arXiv:2010.03768* (2020).

[20] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. 2025. Preble: Efficient Distributed Prompt Scheduling for LLM Serving. In *International Conference on Learning Representations*, Vol. 2025. 37057–37082.

[21] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 173–191.

[22] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. 2025. Towards end-to-end optimization of llm-based applications with ayo. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1302–1316.

[23] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022. MuSiQue: Multihop questions via single-hop question composition. *Transactions of the Association for Computational Linguistics* 10 (2022), 539–554.

[24] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2016. TetriSched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters. In *European Conference on Computer Systems*. Association for Computing Machinery, 35:1–35:16. doi:10.1145/2901318.2901355

[25] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang (Eric) Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Ahmed Awadallah, Ryen W. White, Doug Burger, and Chi Wang. 2024. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. In *Conference on Language Modeling*.

[26] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu

Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation*. 595–610.

[27] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems* 36 (2023), 11809–11822.

[28] Yuanhe Zhang, Ilja Kuzborskij, Jason Lee, Chenlei Leng, and Fanghui Liu. 2026. DAG-Math: Graph-of-thought guided mathematical reasoning in LLMs. In *International Conference on Learning Representations*, Vol. 2026. 74347–74389.

[29] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *USENIX Symposium on Operating Systems Design and Implementation*. 559–578.

[30] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 193–210.