Characterizing How Complex Agentic AI Systems Handle General Tasks: A Trace-Based Simulation Study

Donghwan Kim[1*], Prakhar Singh[1], Younghoon Min[2], Jongryool Kim[2], Jongse Park[3], and Kiwan Maeng[1*]

[1]*The Pennsylvania State University*
[2]*SK hynix Inc.*
[3]*KAIST*
[*]*{djk6434,kvm6242}@psu.edu*

Abstract

Agentic AI completes tasks through iterative planning, tool use, and reasoning based on observed outcomes. Despite its popularity, its system-level behavior remains poorly understood—particularly for complex datasets and agent architectures—owing to highly non-deterministic execution, prohibitive evaluation costs, and limited visibility into proprietary models. This paper presents GAIATrace, the first token-level trace dataset of two state-of-the-art agentic systems (MiroThinker and OWL) running GAIA, a benchmark composed of a heterogeneous mix of general-purpose tasks. Unlike prior trace datasets, GAIATrace captures full reasoning tokens, task-level structures, and activities of every major participating LLMs, enabling in-depth systems research. Complementing the dataset, we present Vidur-Agent, a trace-driven simulator that can replay GAIATrace to perform reproducible, low-cost system evaluation across diverse simulated environments. Using both artifacts, we characterize how modern agentic systems handle general tasks and how various system design choices shape their behavior, yielding several unique findings.

1. Introduction

Agentic AI—autonomous pipelines in which LLMs plan, invoke tools, observe results, and iterate until a task is complete [19, 31, 66, 68, 83]—is a popular new paradigm with the potential to tackle open-ended tasks without human intervention. Yet, the community's understanding of its system characteristics remains thin. A recent study [37] offered valuable early insights but examined only a narrow slice of the design space (Section 2.2). More complex setups—multi-agent/model systems tackling general tasks with a broad range of tools—remain largely uncharted, because bringing up such systems and running controlled experiments are expensive and challenging.

To close this gap, we present *GAIATrace* and *Vidur-Agent*, a dataset and simulator that together enable cost-efficient and reproducible experimentation on complex agentic AI systems. GAIATrace is a trace dataset capturing the behavior of two state-of-the-art agentic systems—MiroThinker [68] and OWL [31]—as they solve a heterogeneous mix of general tasks from the GAIA benchmark [49]. Unlike existing traces [50, 57], GAIATrace preserves task-level structure (ordered queries grouped by task), full visibility into reasoning tokens, and traces from every participating LLM, including auxiliary summarization and extraction models. Vidur-Agent is a trace-driven simulator that extends the Vidur [2] LLM simulator with support for agentic execution: heterogeneous models, KV and prefix caching [40, 92], prefill–decode disaggregation [54, 93], and advanced scheduling policies. Together, they allow agentic AI system simulation to a degree that was impossible before.

Our characterization studies based on these yield several new findings, organized as takeaways throughout the paper. We highlight a few below (not a comprehensive list):

- **Agent behaviors are highly heterogeneous.** GAIATrace reveals far more diverse patterns than what prior work [37] observed, driven by heterogeneity in workloads, agents, models, and tools. Its traces deviate substantially from the simple monotonically-increasing prefill pattern observed in simpler setups (*e.g.*, ReAct [83]).
- **Popular SLA metrics are often misaligned with agentic AI.** Certain designs significantly worsen per-query metrics such as tail time-to-first-token (TTFT) or time-per-output-token (TPOT), yet improve per-task latency. Unlike chatbot LLMs—where each query faces a user and must meet strict TTFT/TPOT requirements—most tokens in agentic AI are never read by a human. This suggests that prior optimizations targeting per-query metrics [3, 75, 93] may not be optimal for agentic AI.
- **System bottlenecks shift with load.** The dominant bottleneck changes with task arrival rate, making static hardware provisioning brittle. A GPU allocation that is near-optimal at one arrival rate can be substantially worse at another, motivating dynamic reconfiguration.
- **Prefix caching is far more impactful than previously reported, but unevenly.** In our study, end-to-end task latency improved by 1.13–3.21× with prefix caching, which is substantially larger than prior reports (*e.g.*, 15.7% [37]). Yet the benefit is uneven, and some components even experience a *degraded* TTFT. The finding again shows how task latency and per-query metrics can misalign.

These findings argue that future agentic AI will exhibit characteristics that diverge from what today's simpler agents and workloads display, motivating deeper study of complex agentic systems (multi-agent, multi-model, and multi-tool) and workloads. We hope that GAIATrace, Vidur-Agent, and our initial study will be a useful foundation for future research in this direction.

2. Background and Motivation

2.1. LLM Serving Systems

Large language models (LLMs) are transformer architectures [69] that generate text autoregressively in two phases. The *prefill* phase processes the input prompt (*i.e.*, *context*) in a single pass and produces the key-value (KV) cache for each token in the input, while the *decode* phase generates output tokens one at a time, with each new token attending to the KV cache of all preceding tokens. Prefill is compute-bound, and a single long input can saturate the GPU, whereas decode is memory-bound and can benefit from batching. *Reasoning* is a special type of decode where LLMs produce an extended chain of intermediate "thinking" tokens to improve its answer. Many LLMs allow reasoning to be toggled on or off. For safety and proprietary reasons, closed-source models do not reveal the reasoning tokens.

Several techniques have emerged to optimize prefill and decode. *Prefix caching* [25, 92] avoids redundant prefill computation by reusing the KV cache across queries that share a common prefix (*e.g.*, a system prompt or a conversation history). When prefill and decode run on the same GPU [3, 4, 23, 28, 40], their distinct compute patterns interfere with each other. *Prefill-decode (PD) disaggregation* [30, 54, 93] solves this problem by running them on separate devices connected by high-bandwidth interconnects. When using PD disaggregation, each phase can use a different number of GPUs and a different parallelism strategy, such as tensor parallelism (TP) [51] or pipeline parallelism (PP) [24, 33]. A common approach is to profile a representative workload offline and statically configure the GPUs for each phase [93].

Over the years, numerous systems were proposed for efficient LLM serving [1, 3, 4, 17, 22, 23, 27, 40, 46, 47, 52, 54, 57, 62, 64, 79, 84, 85, 87, 88, 91–93] and agentic AI serving [10–12, 14, 25, 36, 42, 44, 44, 48, 53, 57, 59, 63, 67, 74, 90]. These efforts are largely orthogonal to this paper.

2.2. (Autonomous) Agentic AI

Agentic AI is an overloaded terminology in the community without a clear definition. Following the definition from Anthropic [7], we focus on *autonomous* systems that rely on the LLM to dynamically decide how to solve the given problem—generating plans, calling tools and interpreting their results, invoking other LLMs, and judging when the task is complete—without a human in the loop. We do not consider systems whose control flow is statically designed by human (which Anthropic refers to as "workflows" or "augmented LLMs" [7]).

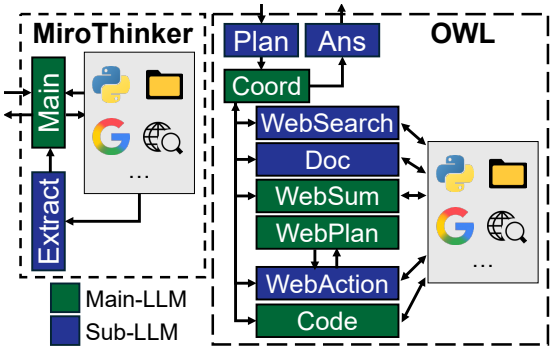


Figure 1: Two agentic AI systems, MiroThinker [68] and OWL [31]. Some details omitted for simplicity.

Design of Agentic AI. How to architect an agentic AI system remains an open question under active research. *Early agentic systems* [39, 43, 68, 77, 83, 94] adopted simple structures in which an LLM repeatedly alternates between planning, tool invocation, and replanning based on tool outcomes. At each turn, the history of prior steps (plans and tool results) is appended to the context, causing the input length to grow monotonically. We refer to such systems as *ReAct-like*, after the canonical example, ReAct [83]. *More recent agentic systems* [8, 19, 29, 31, 32, 41, 66, 78, 80, 81] explore richer architectures in which multiple agents are organized hierarchically and communicate to solve problems collaboratively. The same LLM can serve as multiple conceptually different agents (*i.e.*, role-play [41]).

Figure 1 illustrates two representative systems from each family that we will study, MiroThinker [68] and OWL [31]. *MiroThinker* [68] (left) is a simple, ReAct-like system. The main LLM ("Main" in Figure 1) generates plans, invokes tools, observes the result, and refine its plans further. It occasionally gets help from another LLM ("Extract" in Figure 1) that summarizes and extracts important data from a large, web-scraped text. While simple, MiroThinker is still more complicated than many ReAct-like systems prior works studied, as it involves multiple LLMs ("Main" and "Extract") and tools (our setup used 4 tools; Table 2). *OWL* [31] (right) has a more complex structure. There exist multiple agents that are specialized in a certain task working under the master coordinator, each generating their own sub-plan, maintaining memory, and making decisions. As different colors indicate, different LLM architectures are used for different agents. We omit the details of each system, which can be found in the respective papers [31, 68].

Agentic AI Benchmarks. Agent AI benchmarks span a wide range of domains. Many popular benchmarks consist of homogeneous tasks, such as question answering [82], function call selection [55], web browsing [95], code generation [13, 34], and scientific reasoning [26, 60]. These benchmarks can typically be solved with only one or two tools [37], and each workload exhibits a relatively homogeneous computational pattern. Other benchmarks [49, 56, 86] instead comprise mixed real-world tasks that require

a more diverse set of tools and exhibit more heterogeneous compute patterns. This paper focuses on one such general benchmark, GAIA [49].

Goal: Characterizing the Broader Landscape. A recent characterization study [37] offered valuable early insights into the system overheads of agentic AI, but it only examined a narrow slice of this vast and rapidly evolving space. In particular, the prior work [37] only studied simple ReAct-like systems on single-domain benchmarks (*e.g.*, MATH [26] and HotPotQA [82]) with only one or two tools enabled per benchmark. Such setups fail to capture the complexities of modern agentic AI: heterogeneous tasks executing concurrently on complex agentic structures with a wide variety of tools. This paper aims to close the gap by studying a hetorogeneous mixture of workloads running on complex agentic systems (multi-agent, multi-model, and multi-tool), and simultaneously develop a benchmark and simulator for future research in this direction.

2.3. LLM Simulators

LLM serving has become a major system optimization problem, motivating the development of simulators [2, 9, 15, 16, 38, 45, 76, 89] that evaluate LLM serving efficiency without the cost of full deployment. Vidur [2] is an LLM simulator built by Microsoft Research that predicts the serving efficiency under various models, GPU hardware, parallelization strategies, batching, and scheduling policies, through a learned decision tree. The accuracy of Vidur has been verified, and several system design studies have been built on top of Vidur [3, 5, 22]. However, Vidur operates at the individual query level and does not model the multi-turn structure and inter-turn dependencies of agentic AI. We extended Vidur to build Vidur-Agent (Section 3.3).

Several other LLM simulators have also been proposed, including LLMServingSim and 2.0 [15, 16], TokenSim [76] and APEX [45]. At a different abstraction level, LLMCompass [89] and ADOR [38] focus on simulating architectural optimizations and hardware configurations. Our GAIATrace and the general methodology can be made compatible with these other simulators with proper modification.

3. GAIATrace and Vidur-Agent: Trace-based Simulation for Agentic AI System Study

Our goal is to study how complex agentic AI systems like MiroThinker [68] and OWL [31] behave when running mixed real-world tasks like GAIA [49]. First, we discuss the new trace dataset and trace-based simulator that we built. In Sections 4 and 5, we discuss the charaterization results using these dataset and simulator.

3.1. Motivation: Why Trace-based Simulation?

Studying the behavior of agentic AI systems poses two main challenges. First, their behavior is highly non-deterministic. For the same task, an agent may issue a web search in one trial—incurring tool call and a heavy prefill overheads—while in another trial, forgo the search and proceed with deep reasoning, incurring high decode overheads instead. This makes the system behavior fluctuate dramatically across trials. Fixing randomness (*e.g.*, via `random.seed(0)`) does not eliminate this stochasticity, because external tools (*e.g.*, Google search) still introduce non-determinism. Second, evaluation costs are extremely high, with each task taking minutes to run and uses LLMs with tens to hundreds of billions of parameters.

To address these challenges and lower the cost of agentic AI systems research, we built: (1) *GAIATrace*, a trace dataset containing detailed behavioral traces of two representative agentic AI systems (MiroThinker [68] and OWL [31]) tackling the GAIA [49] benchmark, and *Vidur-Agent*, a trace-driven simulator (built on top of the Vidur [2] LLM simulator) that can replay the traces and simulate GPU and tool overheads. GAIATrace and Vidur-Agent allow controlled experiments on various simulated hardware setups, enabling reproducible and rapid experimentation at low cost. We explain in Section 3.2 and Section 3.3 how we developed GAIATrace and Vidur-Agent.

3.2. GAIATrace: Trace Collection

While there are existing traces for agentic AI, none of them entirely fits our need. For example, Mooncake's Tool&Agent traces [57] only contain individual query information (*i.e.*, pair of prefill and decode tokens) and not higher task-level information (completing each *task* involves running multiple LLM *queries* in a specific order). MiroVerse traces [50] do not contain reasoning tokens and only contain traces from the main LLM—*e.g.*, trace of models annotated as "Extract" on Figure 1 is omitted. For a comprehensive study of all aspects, we built our own trace dataset: GAIATrace includes *reasoning tokens*, traces from *all the major LLMs* that participate, and contains higher *task-level information*. GAIATrace is valuable in several aspects:

1) They offer insights into how SOTA agentic AI systems behave (Section 4).
2) They can be replayed in Vidur-Agent to enable rapid experimentation and prototyping (Section 5).
3) They can also be replayed directly on real systems (*e.g.*, frameworks such as vLLM support benchmarking custom trace [72] with little engineering effort, and the LLM portion of our traces can be replayed on these systems) to more accurately estimate runtime overheads.

We collected GAIATrace on a carefully-engineered setup which balances transparency and task completion accuracy. First, we used open-source LLMs when heavy reasoning was needed to allow full inspection of the reasoning tokens. We refer to such models as *Main-LLM* throughout the text (Figure 1, green). We used gpt-oss-120b for OWL and MiroThinker-1.7-mini for MiroThinker. For non-reasoning LLMs, we used high-quality proprietary models (gpt-4o for OWL and gpt-4o-mini for MiroThinker) to ensure high accuracy. Doing so is acceptable because their behavior is determined entirely by their input and output tokens, both of which are visible to the user. We refer to such models as *Sub-LLM* (Figure 1, blue). For video- and audio-processing,

TABLE 1: Token-level trace statistics across agentic systems and models.

		Mooncake [57]	**MiroThinker (GAIA-103-Text)**		**OWL (GAIA-165-Val)**	
			Main-LLM (*MiroThinker-1.7-mini*)	Sub-LLM (*gpt-4o-mini*)	Main-LLM (*gpt-oss-120b*	Sub-LLM (*gpt-4o*)
n_{task} / Queries		– / 23608	103 / 1491	103 / 591	165 / 2669	165 / 2737
Queries per Task	[min, max]	–	[2, 48]	[1, 25]	[2, 113]	[3, 54]
	mean ± std	–	14.5 ± 10.3	7.3 ± 5.1	16.2 ± 12.7	16.6 ± 9.9
Prefill (tokens)	[min, max]	[891, 126195]	[2332, 22727]	[165, 93393]	[204, 62813]	[713, 122711]
	mean ± std	8596 ± 1101	11201 ± 5437	17248 ± 20923	4804 ± 5626	7848 ± 12908
Decode (tokens)	[min, max]	[1, 2000]	[30, 8419]	[5, 8192]	[5, 3790]	[0, 7546]
	mean ± std (total)	182 ± 242 326.7±482.8	348 ± 572	190 ± 530	324 ± 485	93 ± 173
	mean ± std (reason)	–	245 ± 556	–	149 ± 170	–
KV Cache Hit Rate (%)	intra-task / inter-task	– / 59	79.0 / 80.4	0.3 / 0.3	58.0 / 58.7	60.4 / 61.3
Reasoning Token Ratio (%)		–	70.3	–	47.0	–

we used Gemini-2.5-mini and Whisper-2, both of which are proprietary. Although their internal behaviors are opaque, these are invoked infrequently and contribute negligibly to the overall characterization.

We prevented the agentic systems from directly finding GAIA answers on the web by blocking access to URLs that contain GAIA question-answer pairs (*e.g.*, `huggingface.co/`), and manually inspected the traces to ensure they represent real-world problem solving behaviors. We ran each task at most four times. We annotated successful and failed trials, so that one can include only successful trials (representing a high-quality system with a high success rate) or all the trials (modeling the affect of failed execution as well) for simulation. OWL ran all the 165 tasks in GAIA, while MiroThinker could only run the 103 tasks that do not involve image, video, or audio. The respective success rate was 48.5% for OWL and 59.2% for MiroThinker, which were slightly lower than SOTA numbers [31, 66, 68] but roughly in a similar ballpark. We could not achieve the SOTA accuracy because (1) we partially rely on open-source models with lower quality and (2) some high accuracy numbers couldn't be reproduced even when using the default configurations (similar reproducibility issues were reported by others as well [20, 21]).

Our MiroThinker trace has 26.9M input tokens and 0.6M output tokens, and 44.0%, 44.1% of them are from failed traces, respectively. OWL trace has 34.3M input tokens and 1.1M output tokens, 63% and 62% of them are from failed traces, due to lower accuracy. The statistics indicate that many processed tokens are wasteful in current agentic AI systems (did not lead to successful task completion), which highlights the importance of improving the accuracy.

3.3. Vidur-Agent: Simulator Design

We modified Vidur [2] and built *Vidur-Agent*, which can replay GAIATrace and study its system behavior on various system configurations. First, we added support for dependent execution and tool call overhead simulation, *i.e.*, execution of a query only starts when all the dependent queries and tool calls finishes. We measured various tool calls' latency with repeated runs and used the measurements to model their delays. Second, we implemented modeling advanced system setups that are popular in agentic AI, including support for heterogneous models [31, 66], prefill-decode disaggregation [30, 54, 93], KV cache [57, 70] and prefix cache [71], load-aware query routing [25, 79], and advanced scheduling policies [48, 75]. A concurrent work proposed a simulator with similar capabilities [15], and we do not claim these additional features as a strong novelty. Our main contribution lies in the collection of GAIATrace and its characterization (Sections 4 and 5), and Vidur-Agent is one essential component enabling the characterization. With proper modifications, we believe other simulators (*e.g.*, [15]) can also be used in the place of Vidur-Agent.

4. GAIATrace Characterization Study

We first analyze several aspects of GAIATrace, showing that it captures unique behaviors of agentic AI systems.

4.1. Overall Statistics

Table 1 summarizes the token-level statistics of GAIATrace. In this table, we only include one trace per task when there were multiple trials, preferring successful traces. As discussed in Section 3.2, both OWL and MiroThinker primarily rely on two types of LLMs: a *Main-LLM* with a reasoning capability and a *Sub-LLM* without. We report token statistics for each model separately. For comparison, we also include Mooncake's Tool&Agent trace [57].

Number of Tasks and Queries. Unlike the Mooncake trace, which only reports statistics for individual LLM queries, GAIATrace additionally captures the higher-level task structure: the dataset consists of multiple high-level tasks, each handled through a series of LLM queries and tool calls. This makes GAIATrace better suited for studying task-level behavior. As shown in Table 1, MiroThinker completes 103 tasks via 1,491 Main-LLM queries, 591 Sub-LLM queries, and a number of tool calls (discussed later in Table 2). OWL completes 165 tasks via 2,669 Main-LLM queries, 2,737Sub-LLM queries, and tool calls. Again, OWL can also handle image, video, and audio inputs, covering more tasks.

On average, OWL issues more queries per task (16.2–16.6) than MiroThinker (7.3–14.5). This is due to how OWL

is designed, where multiple agents frequently communicate with each other. This illustrates how agentic AI system design significantly impacts LLM query volume. The high standard deviation in queries per task further indicates that GAIA tasks vary substantially in complexity and requires different number of LLM queries to complete.

Prefill Statistics. The Main-LLMs of both systems process a similar number of prefill tokens (4,804–11,201) per query as Mooncake (8,596). MiroThinker's Main-LLM sees slightly longer prefills than OWL's, as its ReAct-like design more aggressively accumulates the history. The standard deviation is notably higher than Mooncake's (5,437–5,626 vs. 1,101), which we attribute to varying task complexity of the GAIA and the higher complexity in the agent architecture. We also report the ideal KV cache hit rate (assuming unbounded cache size). The Main-LLMs of GAIATrace exhibit slightly higher hit rates than Mooncake, with most hits coming from queries within the same task (intra-task). MiroThinker achieves a higher hit rate due to its ReAct-like design, which yields substantial prefill overlap between adjacent queries.

The Sub-LLM prefill statistics paint a more interesting picture. Since Sub-LLMs handle prefill-heavy tasks such as text summarization and image understanding, their average prefill lengths are higher (7,848–17,248), and their standard deviations are extreme (12,908–20,923), reflecting wide variation in the size of input texts and images. Most notably, MiroThinker's Sub-LLM suffers from an *extremely low KV cache hit rate of 0.3%*. This is because it is primarily used to summarize scraped web pages, and prefill overlap is essentially nonexistent unless the same page is scraped and summarized twice. These observations indicate that KV cache hit rates can vary significantly across agent designs.

Decode Statistics. The decode length of models in GAIATrace is roughly in the similar ballpark with Mooncake (93–348 vs. 182), but again have much higher variance. We additionally observed that nearly half of the generated tokens of reasoning models (Main-LLMs) are reasoning tokens (47.0% for OWL, 70.3% for MiroThinker). This emphasizes the importance of accurately modeling the impact of reasoning tokens, which is omitted in many existing trace datasets (*e.g.*, MiroVerse [68]).

4.2. Detailed Trace Analysis

Next, we dissect GAIATrace in more detail. Figures 2 and 3 show the prefill (input) and decode (output) token length distributions of queries to each LLM. MiroThinker has a more simplistic design than OWL, and its token distributions are correspondingly simpler. MiroThinker's Main-LLM (Figure 2, left) sees input tokens grow with the number of turns due to its ReAct-like design. The first decode (Turn 1) has a notably higher average number of tokens, which is due to the initial planning step. MiroThinker's Sub-LLM (Figure 2, right) mainly performs text summarization and thus has higher prefill tokens than decode.

In contrast, OWL exhibits more interesting behavior. Its Main-LLM (Figure 3, left) shows distinct patterns depending on which role it is playing. For example, ① when serving as a web summarizer, it is prefill-heavy; ② when serving as a coordinator, prefill tokens grow with the number of turns in ReAct-style [83]; and when serving as a web planner, it shows a bimodal behavior—③ initial planning is observation-based and prefill-heavy, while ④ replanning (after a failure) involves deeper reasoning and is decode-heavy. Similarly, OWL's Sub-LLM (Figure 3, right) also shows heterogeneous behaviors based on its role.

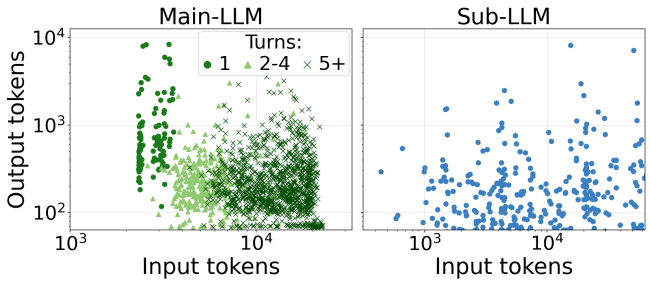


Figure 2: Token distribution (prefill vs. decode) of LLMs in MiroThinker, shown in log scale.

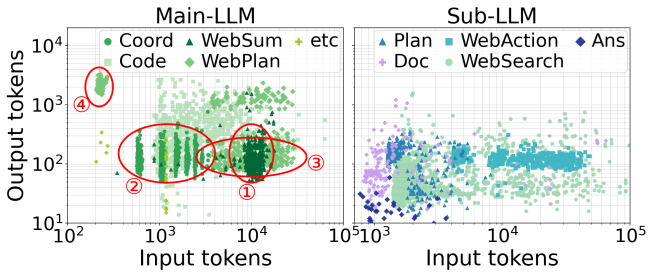


Figure 3: Token distribution (prefill vs. decode) of LLMs in OWL, shown in log scale. Even the same LLM shows different prefill/decode patterns (*e.g.*, ①–④), depending on which role it is playing.

4.3. Visualization of Representative Traces

To provide deeper insight into what the traces look like, we visualize example traces from GAIATrace in Figure 4. The example shows both systems answering the following question: *"How many at bats did the Yankee with the most walks in the 1977 regular season have that same season?"* Each bar represents an LLM query—MiroThinker performs 20 Main-LLM and 8 Sub-LLM inferences, while OWL performs 24 Main-LLM and 30 Sub-LLM inferences—and bar colors indicate the number of prefill/decode tokens processed and the number of prefill tokens reused from the KV cache.

In MiroThinker (left), being a ReAct-like system, the Main-LLM iterates through a loop of planning, tool invocation, and observation. Since this question requires heavy web search, only Google search and web scraping tools are exercised in this example. Each web scrape is followed by a Sub-LLM inference (tall blue bars), which is prefill-heavy and have almost no KV cache hits. The trace is briefly annotated in the figure. MiroThinker sequentially ① searched for the 1977 Yankee roster, ② their at-bat and walk

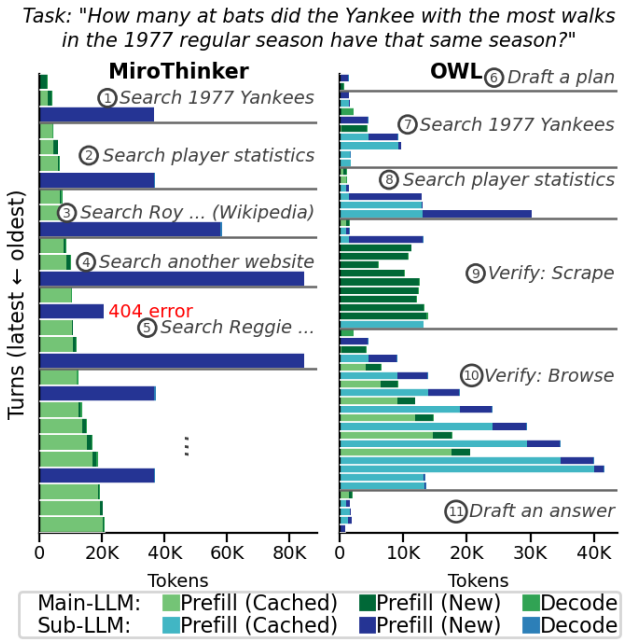


Figure 4: Example traces of MiroThinker and OWL. Older queries are at the top, and newer queries at the bottom.

statistics, and ③–⑤ detailed information of each player, switching websites or retrying with a different URL when it encounters an error (*e.g.*, a 404) or an unsatisfactory result. The overall behavior resembles ReAct [83], but does not exactly match the monotonically-increasing-prefill pattern reported by prior work [37], due to its multi-model design.

In contrast, OWL solved the problem in a more hierarchical fashion. The coordinator ⑥ first drafted an overarching plan, and specialized sub-agents executed each sub-plan. In this example, OWL ⑦ searched for the 1977 Yankees roster, ⑧ pulled relevant statistics, ⑨–⑩ performed a series of verification steps, and ⑪ finally drafted the answer. Searching (⑦–⑧) involved web browsing and scraping, producing spikes in Sub-LLM prefill. OWL also attempted two distinct verification strategies. The first (⑨) relied on web-scraped data, incurring heavy Main-LLM prefill with KV cache misses (OWL uses Main-LLM when very long web data needs to be summarized, breaking them into chunks [31]). When this approach failed to produce a satisfactory result, OWL fell back to a second strategy (⑩): browsing the web interactively (processing the rendered page and scrolling down like a human), which produced alternating inferences between Main-LLM (observe and plan) and Sub-LLM (issue action). Clearly, the two systems behave very differently even for the same task, and even conceptually similar sub-tasks (*e.g.*, web search) can exhibit markedly different system behaviors (*e.g.*, ③ vs. ⑨ vs. ⑩) depending on how they are implemented.

Figure 5 shows additional example traces from GAIATrace. As the figures illustrate, behavior varies significantly across tasks. A simple coding task (left) exhibits the ReAct-like pattern observed in prior work [37]—monotonically increasing prefill with a high KV cache hit rate. In contrast, tasks requiring iterative browsing (middle) ping-pong between Main-LLM and Sub-LLM (increasing prefills, but alternating colors), as in ⑩. Finally, tasks involving heavy web scraping and document processing (right) exhibit bursts of extremely high prefill tokens with very low KV cache hit rates. These examples illustrate that agentic AI's behavior is far from homogeneous, and prior studies [37] have captured only a limited (though important) slice of the space.

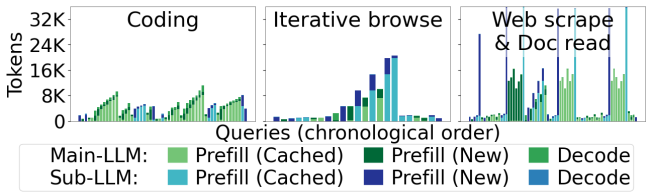


Figure 5: Additional example traces from GAIATrace for OWL. Each bar represents a query in chronological order.

TABLE 2: Per-tool execution statistics. Times are in seconds.

	Tools	Calls	Min	Max	Mean±std
MiroThinker	google_search	539	0.56	9.18	1.85±1.23
	scrape_and_extract_info	622	0.33	422.36	3.70±15.46
	run_python_code	136	0.47	32.96	1.88±2.82
	create_sandbox	19	0.92	1.98	1.10±0.18
OWL	search_google	686	0.14	5.30	0.30±0.15
	extract_document_content	419	0.00	91.08	13.78±10.61
	execute_code	262	0.02	60.07	1.48±6.92
	search_wiki	166	0.09	1.57	0.48±0.24
	search_wiki_revisions	84	0.07	0.29	0.10±0.03
	extract_excel_content	36	0.00	1.05	0.05±0.14
	ask_question_about_image	19	0.05	20.59	4.61±5.62
	search_archived_webpage	11	0.25	10.62	2.92±3.77
	ask_question_about_video	12	2.60	42.61	14.74±12.03
	ask_question_about_audio	5	1.33	4.04	2.44±0.84

Takeaway 1. GAIATrace emphasizes that complex agentic AI systems' behavior is highly heterogeneous: the statistics have extremely high variance, and behavioral patterns vary between agent architecture designs, agent roles, models, and tasks. Even the same LLM experiences different prefill-decode behaviors depending on the overall system design (*e.g.*, whether it is ReAct-like), which agent role it serves as (*e.g.*, coordinator vs. summarizer), and other surrounding contexts (*e.g.*, initial planning vs. replanning). Our results suggest that system optimizations tuned for simple agentic designs and workloads may not transfer to the more complex designs and workloads of the future, motivating further study of these more complex setups.

4.4. Tool Statistics

Table 2 summarizes tool call frequencies and their execution time statistics. Tools fall into roughly three categories. First, fast tools (`search_google`, `search_wiki`, `extract_excel_content`, *etc.*) complete within seconds and do not noticeably affect end-to-end latency. Second, tools like `extract_document_content` rely on external webscraper (*e.g.* Jina [35] and Firecrawl [18]) and

exhibits much higher latency due to the website or the service being slow. These contribute non-negligibly to end-to-end latency, but little can be done about it. Finally, LLM-assisted tools (`ask_question_about_video`, `ask_question_about_audio`) use proprietary LLMs to handle inputs that Main-LLM/Sub-LLM cannot process (*e.g.*, YouTube videos are processed by Google's Gemini-2.5 via API). We currently model them as black-box tools, but they could potentially be replaced with local models (another Sub-LLM) given additional engineering effort.

Takeaway 2. Tool latency is dominated by a small subset of tools that rely on slow external services or LLMs. Their behavior has high variance cannot be accelerated easily. Also, their latencies are difficult to model systematically, posing a practical challenge for trace-driven simulation studies. The issue can be potentially eased by replacing tools using proprietary LLMs with open-source LLMs, but this requires non-negligible effort.

5. System Characterization with Vidur-Agent

Next, we show results of our system characterization study performed by running GAIATrace on Vidur-Agent under various system configurations. The following results serve two main purposes: (1) it provides us insights on the system overheads of complex agentic AI systems, and (2) it demonstrates the potential use of GAIATrace and Vidur-Agent on agentic AI system research.

5.1. Simulation Setup

We simulated a cluster consisting of two DGX A100 nodes (16×A100 (80GB) GPUs). By default, we ran four models (two Main-LLMs and Sub-LLMs), each with tensor parallelism degree of four (TP-4), and adopted prefill-decode disaggregation. That is, the default setup ran Main-LLM prefill, Main-LLM decode, Sub-LLM prefill, and Sub-LLM decode on four GPUs (TP-4) each. Section 5.6 varies this configuration. Since Vidur does not yet ship a performance model for the exact LLMs we used, we used Qwen2.5-32B-Instruct's [58] performance model instead. We expect this to still capture a reasonable first-order estimate of the overall trends, as Qwen2.5 shares its core architectural features—a decoder-only Transformer with grouped-query attention (GQA) [6] and rotary position embeddings (RoPE) [65]—with many popular LLMs. A performance model fitted to our exact LLMs would tighten the absolute numbers. In Section 5.7, we increase the size to Llama-3-70B.

We simulate the task arrival with Poisson distribution with varying rate (0.05–0.5 tasks per second, with 0.1 tasks per second being the default). Note that we control the tasks per second (TPS), not queries per second (QPS). Each task invokes multiple queries, so TPS is generally lower than QPS. Incoming tasks are randomly shuffled, and prefix caching is enabled by default, except for in Section 5.3 where it is varied. Following vLLM V1 [40, 73], KV cache is managed at block granularity (16 tokens per block); each TP-4 group holds roughly 58,848 blocks for the 32B model. and 32,870 blocks for the 70B model.

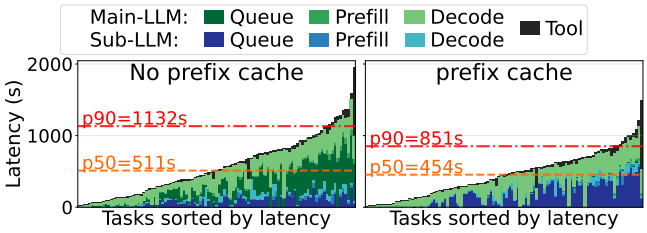


Figure 6: Effect of prefix caching in MiroThinker. Each task is sorted by its end-to-end latency.

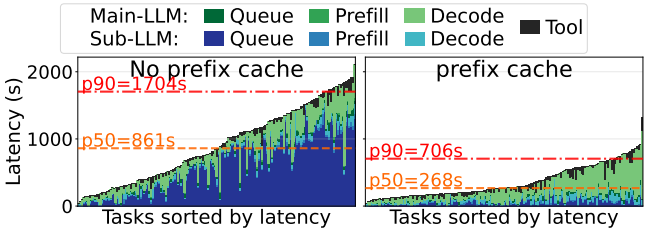


Figure 7: Effect of prefix caching in OWL. Each session sorted by its end-to-end latency.

5.2. Initial Overhead Analysis

First, we studied the overheads of prefill, decode, and tool calls. Figures 6 and 7 show the latency of MiroThinker and OWL tasks with and without prefix caching, sorted by end-to-end latency. We break down the overheads into Main-LLM and Sub-LLM's prefill, decode, queuing delay at the prefill GPUs, and tool call overhead. Decode, unlike prefill, can be batched and run in parallel, so queuing delay at decode GPUs was negligible and omitted.

First, the bottleneck varied with the system. Without prefix caching, MiroThinker was bottlenecked by Main-LLM prefill, whereas OWL was bottlenecked by Sub-LLM prefill. This is because MiroThinker's Main-LLM is ReAct-like, so prefill tokens accumulate quickly over turns, and OWL's Sub-LLM frequently processes long text or images, incurring larger prefill overheads. Second, unlike prior reports [37], tool calls were not a major overhead, because GAIA tasks require heavy text/image ingestion and long reasoning, making LLM overheads dominant. Still, extremely slow web services occasionally drove up end-to-end latency. Finally, decode of Main-LLM (dominated by reasoning) and prefill of Sub-LLM together accounted for a significant portion of the overall overhead (especially with prefix caching), neither of which is captured by existing datasets such as MiroVerse [50]. This underscores the value of GAIATrace.

Takeaway 3. System overheads depend on agent design, and our findings do not simply replicate prior observations [37]. For complex tasks like GAIA, LLM overheads dominated in most cases and tool call overheads were small, though slow external services occasionally drove up end-to-end latency. Moreover, reasoning tokens and Sub-LLM activity, both not captured by existing trace datasets, accounted for a sizable share of the total overhead.

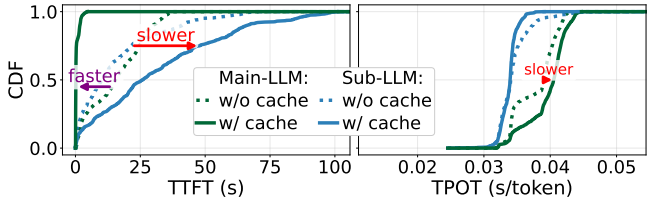


Figure 8: Cumulative distribution function (CDF) of MiroThinker's TTFT and TPOT with and without prefix caching.

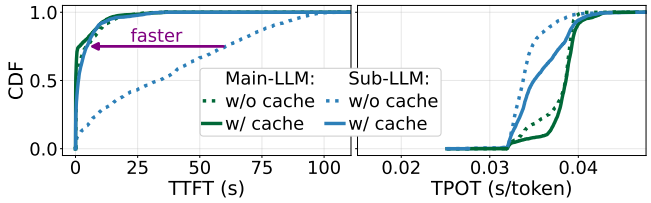


Figure 9: Cumulative distribution function (CDF) of OWL's TTFT and TPOT with and without prefix caching.

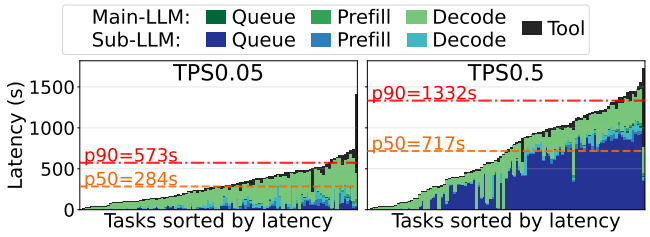


Figure 10: Task latency with different TPS in MiroThinker.

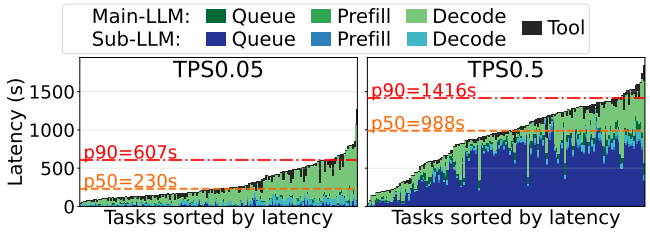


Figure 11: Task latency with different TPS in OWL.

5.3. Impact of Prefix Caching

Figures 6 and 7 show that prefix caching significantly improves both systems: p50 end-to-end latency improves by 1.13× (MiroThinker) to 3.21× (OWL), and p90 by 1.33× (MiroThinker) to 2.41× (OWL). These are larger than what prior study [37] reported (15.7% end-to-end improvement [37]), especially for OWL, for two main reasons. First, GAIA's complexity demands huge ingestion of searched data, making workloads more prefill-heavy. Second, we simulate batched inference, whereas prior work [37] studied single-batch inference. Prefill benefits less from batching as it is compute-bound, so its share of latency is even larger in our setting. Consequently, prefill (compute plus queuing) dominates without prefix caching (Figures 6 and 7, left), and its large overhead shrinks substantially once prefix caching is enabled (right). MiroThinker's prefill remains large even with prefix caching because its Sub-LLM processes long prefills with an extremely low KV-cache hit rate (Table 1); OWL, in contrast, becomes decode-bound.

Figures 8 and 9 further break down how time-to-first-token (TTFT) and time-per-output-token (TPOT) change with prefix caching. Figure 8 (left) shows that, in MiroThinker, the Main-LLM's TTFT improves (CDF shifts left) as expected, but Sub-LLM's TTFT *degrades* (CDF shifts right), which is counterintuitive. This happens because: (1) MiroThinker's Sub-LLM has an extremely low KV-cache hit rate (Table 1) and barely benefits from prefix caching, and (2) as the Main-LLM benefits from prefix caching, it submits work to Sub-LLM more frequently, lengthening its prefill queue. TPOT increases, but only slightly, because decode runs at a larger batch.

Unlike chatbot applications where individual TTFT and TPOT are important service-level objective (SLO), the *task completion latency* is the most important metric in agentic AI systems. End-to-end task latency still improves overall, since the Main-LLM's TTFT speedup dominates. Figure 9 shows a similar trend for OWL—its Sub-LLM's TTFT significantly improves—except its Main-LLM's TTFT does not noticeably degrade. This is because OWL's Main-LLM also enjoys a reasonable KV cache hit rate.

Takeaway 4. Prefix caching yields larger gains in our setup than in prior reports [37], but not all components benefit: while Main-LLM's TTFT improved substantially, some metrics worsen slightly—or significantly, as with MiroThinker's Sub-LLM TTFT. This is a new observation that was not reported in previous works that focused on rather simplistic agentic AI systems (*e.g.*, ReAct [37, 83]), and implies that the effect of prefix caching is model- and system-specific. When considering the end-to-end task latency, it improved with prefix caching as expected.

5.4. Impact of Task Arrival Rate

Figures 10 and 11 show how end-to-end task latency changes with the task arrival rate (tasks per second, or TPS). Compared to TPS=0.1 (Figures 6 and 7, right), per-task completion time decreased at lower TPS (0.05) and increased at higher TPS (0.5), which is expected as the system becomes more bottlenecked. The bottleneck itself also shifted with TPS. At low TPS (0.05), the bottleneck was mainly the Main-LLM's decode, while at high TPS (0.5), the Sub-LLM's prefill became the bottleneck. This is again expected because prefill is less batch-able than decode.

Takeaway 5. At low arrival rates, the reasoning model's (Main-LLM) decode was the main bottleneck. At higher arrival rates, prefill capacity dominated. This indicates that the bottleneck varies with the arrival rate, and the optimal configuration may shift accordingly.

5.5. Impact of Scheduling Policy

Next, we studied the impact of query scheduling policies. Scheduling policies primarily affect queuing delay, so we

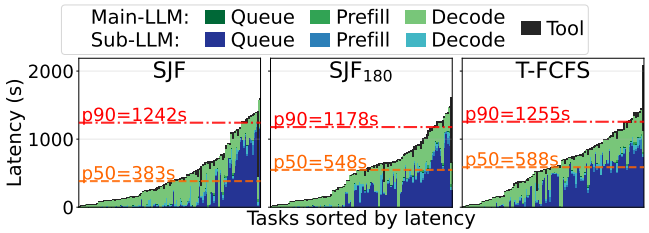


Figure 12: Different scheduling policies (MiroThinker, TPS=0.5; Figure 10 (right) is Q-FCFS for the same setup).

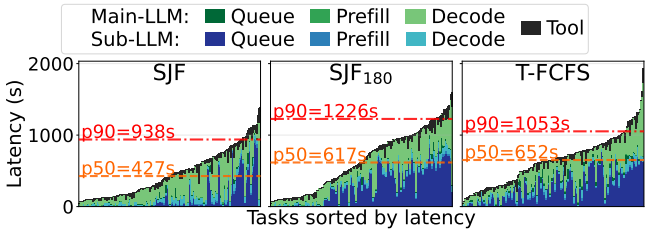


Figure 13: Different scheduling policies (OWL, TPS=0.5; Figure 11 (right) is Q-FCFS for the same setup).

focused on the setup where queuing delay was most dominant: TPS=0.5. The trend was similar at lower TPS, but less pronounced. We also restricted our study to scheduling policies on the prefill GPUs, since decode GPUs do not exhibit notable queuing delay. Prefill scheduling matters because a single long prefill can occupy the entire prefill GPUs and block every other query behind it, so the order in which queries are dequeued directly shapes tail latency. We explore the four policies listed below; all previous figures used the simplest, Q-FCFS.

1) **Query-level first-come-first-serve (Q-FCFS):** Queries are served in the order they arrive in the prefill queue.
2) **Task-level first-come-first-serve (T-FCFS):** Queries are prioritized by the arrival time of the task they belong to, so all queries from an earlier task are served before any query from a later task [42].
3) **Shortest-job-first (SJF)** [61]: Queries with the shortest remaining prefill are prioritized to reduce head-of-line blocking. Though non-clairvoyant decoding limits SJF elsewhere [48, 75], GAIATrace's huge prefills and short decodes perfectly enable this approach.
4) **SJF with timeout**: Same as SJF, but any query that has waited longer than a threshold is promoted to high priority, preventing starvation.

Figures 12 and 13 show the task latencies when using other scheduling policies. Compared to Q-FCFS (Figures 10 and 11, right), SJF significantly improved overall end-to-end latency, reducing p50 latency by 1.87× (MiroThinker) and 2.31× (OWL). This is because a long prefill query has two adverse effects under Q-FCFS: (1) it occupies the prefill GPUs and blocks all queries behind it (*i.e.*, head-of-line blocking), and (2) its large KV cache evicts other useful

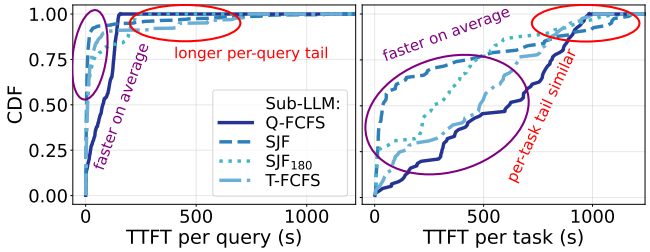


Figure 14: Per-query and per-task TTFT with different scheduling policies for Sub-LLM (TPS=0.5, MiroThinker; OWL shows similar trend and omitted). While per-query TTFT significantly worsened around the tail when using SJF (compared to the baseline Q-FCFS), per-task tail did not worsen significantly. On average (near p50), all the other scheduling policies improved upon Q-FCFS.

entries. By deferring longer prefills and running shorter ones first, SJF mitigates both effects.

Counterintuitively, deferring long prefills did *not* harm task-level tail latency—even p90 slightly improved under SJF. To understand why, we additionally plotted the CDF of per-query TTFT and per-task TTFT in Figure 14 (only Sub-LLM TTFT shown, which is the dominating bottleneck). Figure 14 (left) shows that per-query TTFT increased sharply at the tail under SJF, which is expected: queries with long prefills are repeatedly deferred and accumulate delay. However, task-level sum of TTFT at the tail did *not* worsen much (Figure 14, right). The reason is that a task issues multiple queries, and SJF's overall reduction in queuing delay also benefits the *other queries from the same task* enough to offset one query's TTFT becoming significantly worse. In our experiment, SJF almost always won at the task level, although it looks harmful in per-query TTFT.

Adding a 180-second timeout (SJF_{180}) slightly improved the tail (*e.g.*, p90) latency for MiroThinker but worsened it for OWL. The reason is similar: promoting a long-waiting query (which is, in SJF, a long-prefill query) improves TTFT for that single query, but blocks the prefill GPUs and evicts KV cache entries for everyone else—and *other queries from the same task* also hurt. So while the timeout shortened the per-query TTFT tail (in Figure 14 (left)), SJF_{180} has a much shorter tail than SJF), the end-to-end task latency almost always worsened with timeout in our experiments. Task-level FCFS (T-FCFS) showed a slight improvement over Q-FCFS, because prioritizing queries from already-active tasks increases the chance of their KV cache entries being reused before eviction. The effect, however, was marginal.

Takeaway 6. Scheduling policies that reduce prefill queuing delay, such as SJF, can significantly improve overall task latency. Surprisingly, a longer per-query tail TTFT did not always translate to a longer per-task tail TTFT or worse end-to-end latency. This suggests that prior optimizations targeting per-query TTFT/TPOT are not necessarily optimal for agentic AI systems, whose primary metric of interest is per-task latency, not per-query metrics.

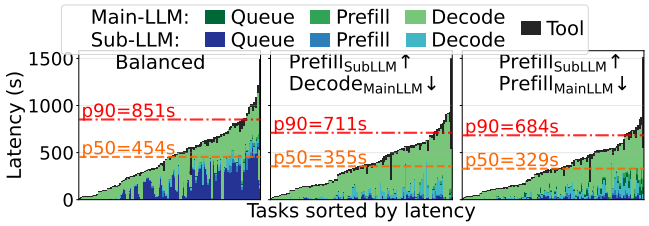


Figure 15: Different system configuration in MiroThinker with medium task arrival rate (TPS=0.1).

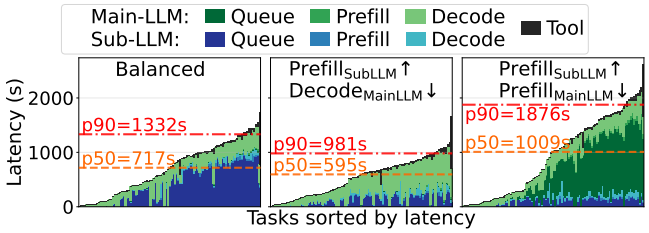


Figure 16: Different system configuration in MiroThinker with high task arrival rate (TPS=0.5).

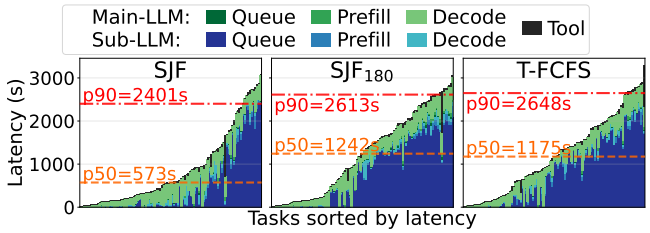


Figure 17: Different scheduling policies with 2× larger model size (MiroThinker; otherwise same with Figure 12).

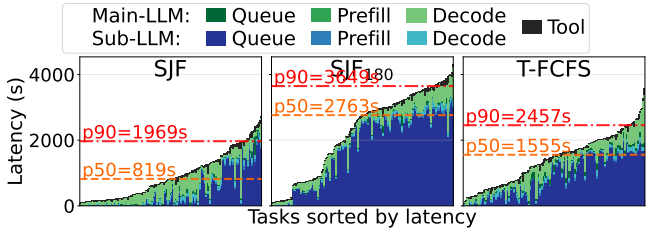


Figure 18: Different scheduling policies with 2× larger model size (OWL; otherwise same with Figure 13).

5.6. Impact of System Configuration

We also studied the effect of varying the number of GPUs assigned to each model's prefill and decode stages. The baseline (Balanced) assigns four GPUs (TP-4) to every prefill and decode stage, as explained in Section 5.1. We additionally studied two setups that give more GPUs to the Sub-LLM's prefill stage, which is the dominant bottleneck at high TPS, by taking GPUs away from different stages. The first takes GPUs from the Main-LLM's decode (reducing to TP-2), and the second from the Main-LLM's prefill.

Figures 15 and 16 show how end-to-end task latency changes under these configurations at different TPS. We only show the results from MiroThinker for brevity; the trend was similar for OWL. As expected, at TPS=0.1 (Figure 15), allocating more GPUs to the Sub-LLM's prefill (second and third plots) slightly improved overall latency over the balanced setup (first plot). Once the Sub-LLM's prefill is no longer the bottleneck, the Main-LLM's decode becomes the next dominant cost. Consequently, taking the GPUs from the Main-LLM's prefill (third plot) yields slightly better performance than taking them from its decode (second plot), because the prefill stage has more slack to give up at this arrival rate.

The trend changes at TPS=0.5 (Figure 16). As before, taking GPUs from the Main-LLM's decode and giving them to the Sub-LLM's prefill (second plot) improves overall latency. However, taking GPUs from the Main-LLM's prefill (third plot) now has an adverse effect: end-to-end latency increases significantly, because the Main-LLM's prefill suddenly becomes the bottleneck at this higher arrival rate. The results suggest that a configuration that is optimal at one arrival rate can be highly sub-optimal at another.

Takeaway 7. The optimal GPU allocation for prefill-decode disaggregation depends on the task arrival rate—a configuration that is near-optimal at one TPS can be substantially worse at another. This suggests that dynamic reconfiguration of GPU allocation, adjusting to the current load, could be beneficial.

5.7. Simulating Larger Models

We reran the experiments from Figures 12 and 13 assuming a larger 70B model (Llama3-70B). This is possible because Vidur-Agent can simulate traces under an arbitrary model architecture, regardless of which model the trace was originally collected from—in effect, asking "what if" a similar trace had been generated by a different model. Figures 17 and 18 show that SJF with a timeout (SJF$_{180}$) gives up much of plain SJF's latency advantage. In Figure 18, it becomes substantially worse than both plain SJF and T-FCFS. As discussed in Section 5.5, promoting a long-waiting query (which is typically prefill-heavy) evicts active KV cache entries belonging to other queries. This adverse effect becomes more severe on larger models because GPU memory is relatively scarcer. The effect is more pronounced in OWL, since MiroThinker's Sub-LLM already gains little from the KV cache reuse.

Takeaway 8. Scheduling behavior also changes with model size. In particular, the relative ranking of scheduling policies is not preserved across model sizes—on larger models, timeout-based promotion for fairness becomes less effective, since KV-cache eviction costs scale with model size.

5.8. Simulator Accuracy and Cost

Vidur-Agent predicts execution time with the operator-level model introduced by Vidur [2], fitted to kernel measurements taken on the target hardware. We validate it against a real Dynamo 1.1.0 + vLLM 0.19.0 prefill/decode-disaggregated deployment on four A100 (80GB) GPUs,

TABLE 3: Real and simulated task execution time (sec) and the simulation error, MiroThinker 103 task traces on up to 4 A100.

	TPS	P50	P90	GeoMean	Total time
S1	0.1	381/354 (-7.0%)	744/738 (-0.8%)	306/282 (-7.6%)	2035/2040 (+0.3%)
S1	0.04	204/196 (-3.7%)	437/381 (-12.8%)	161/150 (-7.0%)	2745/2649 (-3.5%)
S1	0.01	157/151 (-3.8%)	323/322 (-0.2%)	125/118 (-6.3%)	9547/9207 (-3.6%)
S2	0.1	771/745 (-3.4%)	1629/1566 (-3.9%)	572/534 (-6.6%)	2783/2769 (-0.5%)
S3	0.1	2751/2669 (-3.0%)	5373/5187 (-3.5%)	1798/1738 (-3.3%)	6484/6308 (-2.7%)
S4	0.1	315/314 (-0.4%)	731/764 (+4.6%)	229/241 (+5.4%)	1847/1881 (+1.9%)

replaying the same MiroThinker trace (2082 requests, 103 tasks). All four setups (S1–S4) replay the same two-model workload. S1 uses two GPUs for prefill (TP-2) and two GPUs for decode (TP-2), with the Main-LLM and Sub-LLM sharing the same prefill/decode GPUs (four GPUs in total). S2 uses one GPU each for prefill and decode. Again, Main-LLM and Sub-LLM share the prefill/decode GPU (two GPUs in total). S3 uses the same GPU setups with S1 but uses Qwen2.5-32B instead of Llama-8B. S4 gives one GPU each for Main-LLM prefill, Main-LLM decode, Sub-LLM prefill, and Sub-LLM decode (four GPUs in total). Trace token IDs are clipped into the model's 128K-entry vocabulary so the trace can be replayed through the real tokenizer, which preserves the prefix-sharing structure and cache hit rate. We profiled every kernel the model needs under the same vLLM build the deployment runs, and release the profiles and profiling scripts so these numbers can be reproduced and regenerated for other hardware.

The real and simulated runs experience different arrival patterns because task arrival order is randomized, causing each task to experience different queuing delays. Therefore, rather than comparing per-task completion latency directly, we compare the overall latency distributions using P50 and P90 per-task completion latency, geometric mean latency, and total time to complete all tasks, under simulated and real runs. Table 3 shows that the median and geometric-mean task latencies of the simulated and real runs agree within 8%, and the total completion time differs by less than 4%. P90 latency exhibits somewhat larger variation, reaching 12.8% in one case, but remains within 5% for all other cases.

Simulation using Vidur-Agent requires only a moderate CPU and can be done with low overheads. Getting the real-run numbers in Table 3 requires running 2–4 A100s for 31–159 *minutes*, while getting the simulated numbers using Vidur-Agent takes only 173–277 *seconds* on AMD EPYC 7763 (9–35× faster, with much less compute resources).

6. Conclusion

We presented GAIATrace and Vidur-Agent, a trace dataset and trace-driven simulator that together enable the study of complex agentic AI handling a heterogeneous mix of general tasks. Our initial characterization surfaced several findings, demonstrating the value of the new dataset and simulator. Our GAIATrace and Vidur-Agent are open-sourced and hope they serve as a foundation for future work on agentic AI systems.

Acknowledgements

This work was partly supported by the U.S. National Science Foundation under award No. CNS-2349610 and CCF-2529883. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The authors used Claude for code, text, and figures, but all outputs were reviewed carefully by the authors.

References

[1] R. Abhyankar, Z. He, V. Srivatsa, H. Zhang, and Y. Zhang, "InferCept: efficient intercept support for augmented large language model inference," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.

[2] A. Agrawal, N. Kedia, J. Mohan, A. Panwar, N. Kwatra, B. Gulavani, R. Ramjee, and A. Tumanov, "Vidur: A large-scale simulation framework for llm inference," in *Proceedings of Machine Learning and Systems (MLSys)*, 2024.

[3] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, A. Tumanov, and R. Ramjee, "Taming throughput-latency tradeoff in LLM inference with sarathi-serve," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[4] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, "Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills," in *arxiv.org*, 2023.

[5] A. Agrawal, M. Yadav, S. Kumar, A. Agrawal, G. Ghai, S. Bera, E. Pinto, S. Gambhira, M. Adain, K. Sohrab, C. Antonanzas, and A. Tumanov, "Revati: Transparent gpu-free time-warp emulation for LLM serving," in *arxiv.org*, 2026.

[6] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "GQA: Training generalized multi-query transformer models from multi-head checkpoints," in *Findings of the Association for Computational Linguistics (EMNLP)*, 2023.

[7] Anthropic, "Building effective agents," https://www.anthropic.com/engineering/building-effective-agents, 2024.

[8] Z. Asgar, M. Nguyen, and S. Katti, "Efficient and scalable agentic ai with heterogeneous systems," in *arxiv.org*, 2025.

[9] A. R. Bambhaniya, H. Wu, S. Subramanian, S. Srinivasan, S. Kundu, A. Yazdanbakhsh, M. Elavazhagan, M. Kumar, M. Yu, A. Raychowdhury, and T. Krishna, "MIST: A co-design framework for heterogeneous, multi-stage LLM inference," in *arxiv.org*, 2026.

[10] Z. Bian, F. Wu, T. Ma, and Y. Zhuo, "TokenCake: A KV-cache-centric serving framework for llm-based multi-agent applications," in *arxiv.org*, 2025.

[11] A. Biswas, K. Goel, S. S, J. Mohan, A. Khare, A. Parayil, R. Ramjee, and C. Bansal, "Sutradhara: An intelligent orchestrator-engine co-design for tool-based agentic inference," in *arxiv.org*, 2026.

[12] G. I. Chaudhry, E. Choukse, H. Qiu, Íñigo Goiri, R. Fonseca, A. Belay, and R. Bianchini, "Murakkab: Resource-efficient agentic workflow orchestration in cloud platforms," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2026.

[13] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," in *arxiv.org*, 2021.

[14] Q. Chen, Z. Ye, T. Tang, P. Sun, B. Tian, G. Wang, S. Li, Y. Wen, Z. Han, and T. Zhang, "CONCUR: High-throughput agentic batch inference of LLM via congestion-based concurrency control," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2026.

[15] J. Cho, H. Choi, G. Heo, and J. Park, "LLMServingSim 2.0: A unified simulator for heterogeneous and disaggregated llm serving infrastructure," in *Proceedings of the International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2026.

[16] J. Cho, M. Kim, H. Choi, G. Heo, and J. Park, "LLMServingSim: A HW/SW co-simulation infrastructure for LLM inference serving at scale," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2024.

[17] D. Du, S. Cao, J. Cheng, L. Mai, T. Cao, and M. Yang, "BitDecoding: Unlocking tensor cores for long-context LLMs with low-bit KV cache," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2026.

[18] Firecrawl, "Firecrawl," https://www.firecrawl.dev/, 2026.

[19] A. Fourney, G. Bansal, H. Mozannar, C. Tan, E. Salinas, Erkang, Zhu, F. Niedtner, G. Proebsting, G. Bassman, J. Gerrits, J. Alber, P. Chang, R. Loynd, R. West, V. Dibia, A. Awadallah, E. Kamar, R. Hosn, and S. Amershi, "Magentic-One: A generalist multi-agent system for solving complex tasks," in *arxiv.org*, 2024.

[20] Github, "Cannot reproduce the performance on gaia"?" https://github.com/camel-ai/owl/issues/503, 2025.

[21] Github, "Is it possible for someone to reproduce the pass@1 score of the code named "gaia69"?" https://github.com/camel-ai/owl/issues/587, 2025.

[22] K. Goel, J. Mohan, N. Kwatra, R. S. Anupindi, and R. Ramjee, "QoServe: Breaking the silos of LLM inference serving," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2026.

[23] Y. Gyeong-In, J. Joo Seong, K. Geon-Woo, K. Soojeong, and C. Byung-Gon, "Orca: A distributed serving system for transformer-based generative models," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[24] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "PipeDream: Fast and efficient pipeline parallel dnn training," in *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2019.

[25] W. He, Y. Jiang, P. Zhao, Q. Xu, E. Yoneki, B. Cui, and F. Fu, "Efficient multi-round LLM inference over disaggregated serving," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2026.

[26] D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, and J. Steinhardt, "Measuring mathematical problem solving with the math dataset," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[27] M. Hidayetoglu, A. Qiao, M. Wyatt, J. Rasley, Y. He, and S. Rajbhandari, "Shift parallelism: Low-latency, high-throughput LLM inference for dynamic workloads," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2026.

[28] C. Holmes, M. Tanaka, M. Wyatt, A. A. Awan, J. Rasley, S. Rajbhandari, R. Y. Aminabadi, H. Qin, A. Bakhtiari, L. Kurilenko, and Y. He, "DeepSpeed-FastGen: High-throughput text generation for LLMs via MII and DeepSpeed-Inference," in *arxiv.org*, 2024.

[29] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "MetaGPT: Meta programming for a multi-agent collaborative framework," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

[30] C. Hu, H. Huang, L. Xu, X. Chen, C. Wang, J. Xu, S. Chen, H. Feng, S. Wang, Y. Bao, N. Sun, and Y. Shan, "ShuffleInfer: Disaggregate LLM inference for mixed downstream workloads," in *ACM Transactions on Architecture and Code Optimization (TACO)*, 2025.

[31] M. Hu, Y. Zhou, W. Fan, Y. Nie, B. Xia, T. Sun, Z. Ye, Z. Jin, Y. Li, Q. Chen, Z. Zhang, Y. Wang, Q. Ye, B. Ghanem, P. Luo, and G. Li, "OWL: Optimized workforce learning for general multi-agent assistance in real-world task automation," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[32] Y. Hu, R. Ma, Y. Fan, J. Shi, Z. Cao, Y. Zhou, J. Yuan, S. Zhang, S. Feng, X. Yan, S. Zhang, W. Zhang, L. Bai, and B. Zhang, "FlowSearch: Advancing deep research with dynamic structured knowledge flow," in *Proceedings of the ACL (Association for Computational Linguistics)*, 2026.

[33] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "GPipe: efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[34] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "SWE-bench: Can language models resolve real-world github issues?" in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

[35] Jina, "Jina.ai," https://jina.ai/, 2026.

[36] H. Kang, Z. Li, X. Yang, W. Xu, Y. Chen, J. Wang, B. Chen, T. Krishna, C. Xu, and S. Arora, "ThunderAgent: A simple, fast and program-aware agentic inference system," in *arxiv.org*, 2026.

[37] J. Kim, B. Shin, J. Chung, and M. Rhu, "The cost of dynamic reasoning: Demystifying ai agents and test-time scaling from an ai infrastructure perspective," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2026.

[38] J. Kim, H. Lee, G. Ko, G. Choi, S. Ham, S. Hong, and J.-Y. Kim, "ADOR: A design exploration framework for LLM serving with enhanced latency and throughput," in *Proceedings of the International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2025.

[39] S. Kim, S. Moon, R. Tabrizi, N. Lee, M. Mahoney, K. Keutzer, and A. Gholami, "An LLM compiler for parallel function calling," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.

[40] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2023.

[41] G. Li, H. A. A. K. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem, "CAMEL: Communicative agents for "mind" exploration of large language model society," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[42] H. Li, R. He, Q. Mang, Q. Zhang, H. Mao, X. Chen, H. Zhou, A. Cheung, J. Gonzalez, and I. Stoica, "Continuum: Efficient and robust multi-turn LLM agent scheduling with KV cache time-to-live," in *arxiv.org*, 2026.

[43] X. Li, G. Dong, J. Jin, Y. Zhang, Y. Zhou, Y. Zhu, P. Zhang, and Z. Dou, "Search-o1: Agentic search-enhanced large reasoning models," in *Findings of the Association for Computational Linguistics (EMNLP)*, 2025.

[44] C. Lin, Z. Han, C. Zhang, Y. Yang, F. Yang, C. Chen, and L. Qiu, "Parrot: Efficient serving of LLM-based applications with semantic variable," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[45] Y.-C. Lin, W. Kwon, R. Pineda, and F. N. Paravecino, "APEX: An extensible and dynamism-aware simulator for automated parallel execution in LLM serving," in *arxiv.org*, 2024.

[46] Z. Lin, H. Xu, G. Chen, Z. Chen, Y. Lu, and X. Zhang, "Bullet: Boosting gpu utilization for LLM serving via dynamic spatial-temporal orchestration," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2026.

[47] Y. Liu, Y. Cheng, J. Yao, Y. An, X. Chen, S. Feng, Y. Huang, S. Shen, R. Zhang, K. Du, and J. Jiang, "LMCache: An efficient KV cache layer for enterprise-scale LLM inference," in *arxiv.org*, 2025.

[48] M. Luo, X. Shi, C. Cai, T. Zhang, J. Wong, Y. Wang, C. Wang, Y. Huang, Z. Chen, J. E. Gonzalez, and I. Stoica, "Autellix: An efficient serving engine for LLM agents as general programs," in *arxiv.org*, 2025.

[49] G. Mialon, C. Fourrier, C. Swift, T. Wolf, Y. LeCun, and T. Scialom, "GAIA: A benchmark for general ai assistants," in *arxiv.org*, 2023.

[50] MiroMind, "MiroVerse-v0.1," https://huggingface.co/datasets/miromind-ai/MiroVerse-v0.1, 2025.

[51] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient large-scale language model training on gpu clusters using megatron-LM," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[52] X. Pan, E. Li, Q. Li, S. Liang, Y. Shan, K. Zhou, Y. Luo, X. Wang, and J. Zhang, "InstAttention: In-storage attention offloading for cost-effective long-context LLM inference," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2025.

[53] Z. Pan, A. Patel, Z. Hu, Y. Shen, Y. Guan, W.-L. Li, L. Qin, Y. Wang, and Y. Ding, "KVFlow: Efficient prefix caching for accelerating LLM-based multi-agent workflows," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[54] P. Patel, E. Choukse, C. Zhang, A. Shah, Íñigo Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative LLM inference using phase splitting," in *arxiv.org*, 2024.

[55] S. G. Patil, H. Mao, C. Cheng-Jie Ji, F. Yan, V. Suresh, I. Stoica, and J. E. Gonzalez, "The Berkeley Function Calling Leaderboard (BFCL): From tool use to agentic evaluation of large language models," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2025.

[56] Y. Qi, H. Peng, X. Wang, A. Xin, Y. Liu, B. Xu, L. Hou, and J. Li, "AGENTIF: Benchmarking instruction following of large language models in agentic scenarios," in *arxiv.org*, 2025.

[57] R. Qin, Z. Li, W. He, J. Cui, F. Ren, M. Zhang, Y. Wu, W. Zheng, and X. Xu, "Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot," in *USENIX Conference on File and Storage Technologies*, 2025.

[58] Qwen, :, A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei, H. Lin, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Lin, K. Dang, K. Lu, K. Bao, K. Yang, L. Yu, M. Li, M. Xue, P. Zhang, Q. Zhu, R. Men, R. Lin, T. Li, T. Tang, T. Xia, X. Ren, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Wan, Y. Liu, Z. Cui, Z. Zhang, and Z. Qiu, "Qwen2.5 technical report," in *arxiv.org*, 2025.

[59] D. Raghavan, K. Santhanam, M. S. Rahman, N. Modugula, L. G. Schroeder, M. Cura, H. Liu, P. Thaker, P. Levis, and M. Zaharia, "Alto: Orchestrating distributed compound ai systems with nested ancestry," in *arxiv.org*, 2025.

[60] D. Rein, B. L. Hou, A. C. Stickland, J. Petty, R. Y. Pang, J. Dirani, J. Michael, and S. R. Bowman, "GPQA: A graduate-level google-proof q&a benchmark," in *First Conference on Language Modeling*, 2024.

[61] L. Schrage, "A proof of the optimality of the shortest remaining processing time discipline," *Operations Research*, 1968.

[62] J. She, Z. Li, H. Du, S. Wu, W. Zheng, E. Xing, Z. Liu, H. Yao, J. Xue, and Q. Ho, "LAPS: A length-aware-prefill llm serving system," in *Proceedings of Machine Learning and Systems (MLSys)*, 2026.

[63] Y. Sheng, S. Cao, D. Li, B. Zhu, Z. Li, D. Zhuo, J. E. Gonzalez, and I. Stoica, "Fairness in serving large language models," in *arxiv.org*, 2023.

[64] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "FlexGen: high-throughput generative inference of large language models with a single gpu," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.

[65] J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu, "RoFormer: Enhanced transformer with rotary position embedding," in *arxiv.org*, 2023.

[66] S. Su, S. Xing, X. Dong, M. Zhong, B. Wang, X. Zhu, Y. Chen, W. Wang, Y. Deng, P. Zhu, Z. Liu, T. Li, J. Yu, Z. Chen, L. Bing, and J. Dai, "MiroFlow: Towards high-performance and robust open-source agent framework for general deep research tasks," in *arxiv.org*, 2026.

[67] X. Tan, Y. Jiang, Y. Yang, and H. Xu, "Towards end-to-end optimization of LLM-based applications with ayo," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2025.

[68] M. Team, S. Bai, L. Bing, C. Chen, G. Chen, Y. Chen, Z. Chen, Z. Chen, J. Dai, X. Dong, W. Dou, Y. Deng, Y. Fu, J. Ge, C. Han, T. Huang, Z. Huang, J. Jiao, S. Jiang, T. Jiao, X. Jian, L. Lei, R. Li, G. Luo, T. Li, X. Lin, Z. Liu, Z. Li, J. Ni, Q. Ren, P. Sun, S. Su, C. Tao, B. Wang, W. Wang, H. Wang, J. Wang, J. Wang, J. Wang, L. Wang, S. Wang, W. Wang, Z. Wang, J. Xu, S. Xing, C. Yang, H. Ye, J. Yu, Y. Yu, M. Zhong, T. Zhao, X. Zhu, Y. Zhou, Y. Zhang, and Z. Zhu, "Mirothinker: Pushing the performance boundaries of open-source research agents via model, context, and interactive scaling," in *arxiv.org*, 2026.

[69] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[70] vLLM, "KV Cache Aware Routing," https://docs.vllm.ai/projects/production-stack/en/vllm-stack-0.1.5/tutorials/kvaware.html, 2024.

[71] vLLM, "Prefix Aware Routing," https://docs.vllm.ai/projects/production-stack/en/vllm-stack-0.1.5/tutorials/prefixaware.html, 2024.

[72] ——, "vLLM Benchmark CLI," https://docs.vllm.ai/en/stable/benchmarking/cli/, 2024.

[73] ——, "vLLM-V1," https://docs.vllm.ai/en/stable/usage/v1_guide/, 2024.

[74] S. Woo, J. Kil, H. Kim, M. Kim, J. Kim, A. Seo, S. Lee, M. Jo, J. Ryu, B. park, S. J. Kwon, and D. Lee, "ICaRus: Identical cache reuse for efficient multi-model inference," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2026.

[75] B. Wu, Y. Zhong, Z. Zhang, G. Huang, X. Liu, and X. Jin, "Fast distributed inference serving for large language models," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2026.

[76] F. Wu, Z. Bian, G. Duan, T. Xu, J. Wu, T. Ma, Y. Yao, R. Gong, and Y. Zhuo, "TokenSim: Enabling hardware and software exploration for large language model inference systems," in *arxiv.org*, 2025.

[77] J. Wu, B. Li, R. Fang, W. Yin, L. Zhang, Z. Tao, D. Zhang, Z. Xi, G. Fu, Y. Jiang, P. Xie, F. Huang, and J. Zhou, "WebDancer: Towards autonomous information seeking agency," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2026.

[78] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "AutoGen: Enabling next-gen llm applications via multi-agent conversation," in *arxiv.org*, 2023.

[79] T. Xia, Z. Mao, J. Kerney, E. J. Jackson, Z. Li, J. Xing, S. Shenker, and I. Stoica, "Skywalker: A locality-aware cross-region load balancer for LLM inference," in *arxiv.org*, 2025.

[80] Z. Xie, H. Kang, Y. Sheng, T. Krishna, K. Fatahalian, and C. Kozyrakis, "AI Metropolis: Scaling large language model-based multi-agent simulation with out-of-order execution," in *Proceedings of Machine Learning and Systems (MLSys)*, 2024.

[81] Y. Yang, H. Chai, S. Shao, Y. Song, S. Qi, R. Rui, and W. Zhang, "AgentNet: Decentralized evolutionary coordination for LLM-based multi-agent systems," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[82] Z. Yang, P. Qi, S. Zhang, Y. Bengio, W. W. Cohen, R. Salakhutdinov, and C. D. Manning, "HotpotQA: A dataset for diverse, explainable multi-hop question answering," in *Findings of the Association for Computational Linguistics (EMNLP)*, 2018.

[83] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

[84] J. Yi, Z. Zhao, Y. Hu, K. Yan, W. Sun, H. Wang, L. Zhao, Y. Zhang, W. Li, and K. Li, "PAT: Accelerating LLM decoding via prefix-aware attention with resource efficient multi-tile kernel," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2026.

[85] D. Yoon, Y. Min, H. Kim, S. H. Noh, and J. Kim, "TraCT: Disaggregated LLM serving with CXL shared memory KV cache at rack-scale," in *arxiv.org*, 2025.

[86] O. Yoran, S. J. Amouyal, C. Malaviya, B. Bogin, O. Press, and J. Berant, "AssistantBench: Can web agents solve realistic and time-consuming tasks?" in *Findings of the Association for Computational Linguistics (EMNLP)*, 2024.

[87] S. Yu, J. Xing, Y. Qiao, M. Ma, Y. Li, Y. Wang, S. Yang, Z. Xie, S. Cao, K. Bao, I. Stoica, H. Xu, and Y. Sheng, "Prism: Unleashing gpu sharing for cost-efficient multi-LLM serving," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2026.

[88] C. Zhang, K. Du, S. Liu, W. Kwon, X. Mo, Y. Wang, X. Liu, K. You, Z. Li, M. Long, J. Zhai, J. Gonzalez, and I. Stoica, "Jenga: Effective memory management for serving LLM with heterogeneity," in *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2025.

[89] H. Zhang, A. Ning, R. B. Prabhakar, and D. Wentzlaff, "LLMCompass: Enabling efficient hardware design for large language model inference," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2024.

[90] W. Zhang, Z. Wu, Y. Mu, R. Ning, B. Liu, N. Sarda, M. Lee, and F. Lai, "JITServe: SLO-aware LLM serving with imprecise request information," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2026.

[91] Y. Zhao, S. Yang, K. Zhu, L. Zheng, B. Kasikci, Y. Qiao, Y. Zhou, J. Xing, and I. Stoica, "BlendServe: Optimizing offline inference with resource-aware batching," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2026.

[92] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "SGLang: Efficient execution of structured language model programs," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[93] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[94] A. Zhou, K. Yan, M. Shlapentokh-Rothman, H. Wang, and Y.-X. Wang, "Language agent tree search unifies reasoning acting and planning in language models," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

[95] S. Zhou, F. F. Xu, H. Zhu, X. Zhou, R. Lo, A. Sridhar, X. Cheng, T. Ou, Y. Bisk, D. Fried, U. Alon, and G. Neubig, "WebArena: A realistic web environment for building autonomous agents," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

Appendix

1. Abstract

This artifact studies how agentic AI workloads behave on an LLM serving cluster. GAIATrace is a request-level execution traces recorded from two agent systems, OWL [31] and MiroThinker [68], running the GAIA [49] benchmark, plus the scripts that derive the paper's characterization results from them. Vidur-Agent replays those traces against a simulated serving cluster, extending the Vidur simulator with heterogeneous replica groups, prefill–decode disaggregation, dependency-released multi-turn sessions.

2. Artifact check-list (meta-information)

- **Data set:** GAIA 2023 validation set
- **Run-time environment:** Ubuntu 24.04, Python 3.10, uv
- **Hardware:** x86-64
- **Output:** Figures, text tables, metrics
- **Experiments:** Trace characterization and simulation
- **How much disk space required (approximately)?:** 10GB
- **How much time is needed to prepare workflow (approximately)?:** 20 minutes
- **How much time is needed to complete experiments (approximately)?:** 20 minutes
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache-2.0 (GAIATrace), MIT(Vidur-Agent)
- **Data licenses (if publicly available)?:** CC-BY-4.0 (traces)
- **Archived (provide DOI)?:** Yes

3. Description

3.1. How to access. Both GAIATrace and MiroThinker are published in Zenodo as sibling directories:
(https://doi.org/10.5281/zenodo.22133686)

3.2. Hardware dependencies. Any x86-64 machine; no GPU, since operator execution times come from the shipped profiling data. The sweeps run concurrently and predictor fitting uses all cores.

3.3. Software dependencies. We used Ubuntu 24.04, Python 3.10, detailed instruction in Appendix D.

3.4. Datasets. This artifact ship are the traces recorded from the two agent systems represented in token_id. GAIA benchmark is not shipped. To make validation set not be republished in crawlable form, text verbatim files are left out. What ships is still enough to reproduce every result.

4. Installation

Install from Zenodo:
(https://doi.org/10.5281/zenodo.22133686)

```
pip install -r GAIATrace/owl/requirements.txt
cd Vidur-Agent
uv venv && source .venv/bin/activate && uv sync
```

5. Experiment workflow

```
# Stage 1 (GAIATrace): Trace gen + Characterization
cd GAIATrace
./GAIATrace/owl/run_all.sh
./GAIATrace/mirothinker/run_all.sh

# Stage 2 (Vidur-Agent): Trace Simulation
cd ../Vidur-Agent
# Fit predictors once
bash warmup_predictors.sh

# Simulator runners
bash run_miro.sh && bash run_owl.sh
bash run_miro70B.sh && bash run_owl70B.sh
bash analysis/run_gantt_compare.sh
bash analysis/run_plot_ttft_tpot.sh
```

Predictor fitting (`warmup_predictors.sh`) saturates every core, so it trains one configuration at a time, covering every model and parallelism the sweeps use. The sweep scripts launch their runs concurrently by default; on machines with few cores or small DRAM, sequentialize them to fit the available capacity.

6. Evaluation and expected results

GAIATrace writes to each tree's `analysis/outputs/`, generating Figure2–Figure5, Table1 and Table2. Vidur-Agent writes per-run metric CSVs. Running `analysis/*.sh` gives `analysis/gantt_compare/*.pdf` and `analysis/ttft_tpot/*.pdf` corresponds to Figure 6–Figure 18. Simulation is deterministic given a trace, cluster configuration and seed, so the outputs reproduce exactly.

7. Experiment customization

- **Workload** — any CSV with `num_prefill_tokens` and `num_decode_tokens` are accepted. Additional columns like `session_id`, `dep`, `inter_request_latency` and `token_ids` add multi-turn structure, dependencies, tool time and token-exact prefix caching.
- **HW Configuration:** `replica_groups_configs/` files declare replica groups (role, count, model, parallelism, device) and, per model, which replicas prefill and which decode. Need additional warmup when testing uncached hardware configurations.
- **Scheduling policy:** Request scheduling policies like session priority and starvation prevention for per-replica admission.

8. Notes

The READMEs at the repository root and in each subdirectory document the pipeline, trace format, cluster configuration schema.