Benchmarking LLM Serving Systems for Agentic AI Workloads with XPerf

Michael Wang*
University of Illinois
Urbana-Champaign, USA

Yikang Yue*
University of Illinois
Urbana-Champaign, USA

Shaobo Li
University of Illinois
Urbana-Champaign, USA

Yirui Eric Zhou
University of Illinois
Urbana-Champaign, USA

Chen Wang
IBM Research
USA

Jian Huang
University of Illinois
Urbana-Champaign, USA

Abstract

We present XPerf, a benchmarking framework that load-tests LLM serving systems with diverse agentic AI workloads. It provides detailed profiling of the serving system and hardware, enabling users to identify performance bottlenecks introduced by agentic workloads. Benchmarking LLM serving systems under agentic workloads is challenging – agentic applications rely on nondeterministic LLM outputs to guide their control flow; therefore, workload patterns vary unpredictably from run to run. XPerf minimizes this workload variation with a fine-grained trace replay approach: it enables users to easily collect traces from real agentic applications, synthesize new workloads with various patterns if needed, and reproducibly replay them on different LLM serving systems. XPerf includes eight agentic applications across diverse use cases (e.g., coding, deep research, and Q&A) by default. Our empirical study using these workloads shows that XPerf accurately replays agentic workloads, provides detailed performance breakdowns, scales to larger serving systems, and assists in serving system debugging. We will open-source XPerf on GitHub.

1 Introduction

With the advancement in large language models (LLMs), agentic AI applications are becoming pervasive. LLM agents can be deployed to make decisions and take actions to solve complex tasks [33]. Solving a task can usually require more than a single LLM call: an agent issues many LLM calls to plan [73], reason [61], reflect [68], act on external environments via tools [66], and coordinate with other agents [62]. Their superior capabilities have driven their widespread deployment across a growing number of use cases [47], including coding [4], enhanced web search [27], data center management [11], and scientific research [13, 53].

As agentic applications are increasingly deployed, understanding the performance of their backend LLM serving systems becomes essential. To rigorously evaluate LLM serving system performance, a benchmark should support diverse agentic AI workloads and enable detailed profiling of system metrics to help users identify performance bottlenecks.

Unfortunately, existing frameworks fail to achieve these goals (see Table 1). Agent evaluation harnesses [23, 28] focus on evaluating how well different agents complete tasks, rather than how the underlying serving system performs. Although they include diverse agentic applications, they do not offer configurable load generation and cannot report system performance metrics such as latency, throughput, or hardware utilization. Most LLM serving system performance benchmarks [21, 42, 51, 54] issue independent LLM calls or multi-turn conversations to the serving system, they cannot represent the typical workload patterns of agentic applications. AA-AgentPerf [6], a recent step toward benchmarking serving systems with agentic workloads, uses only proprietary coding-agent traces. In addition, neither its benchmark nor traces are publicly available.

Furthermore, it is not easy to study the serving-system performance under agentic workloads, due to the lack of reproducibility in existing benchmarks. Specifically, even for the same user request, the execution graph (i.e., the sequence of LLM calls, tool calls, and their dependencies) that an agentic application produces can differ drastically across runs, presenting the serving engine with a different number of LLM calls, each with different input and output token counts.

The variance occurs because agents decide their next action (e.g., which tool to call or whether to stop) based on the nondeterministic decoding output of LLMs. Small deviations in these outputs accumulate over execution, ultimately producing different execution graphs. We demonstrate this by running mini-SWE-agent [63] with gpt-oss-120b [45] twice on identical user requests under a fixed random seed: on average, the number of LLM calls produced by the same request varies by 70% between two runs (Figure 3). Therefore, to obtain reliable performance evaluations, system developers need a benchmarking framework that can measure serving system performance using reproducible agentic workloads.

To this end, we develop XPerf, an open-source, extensible benchmarking framework that can provide detailed profiling of serving system and GPU performance metrics under various agentic workloads. XPerf enables users to: **(1)** collect workload traces from diverse agentic applications via a generic

*Both authors contributed equally to this work.

tracing module; **(2)** construct their execution graphs; **(3)** synthesize new workloads from collected traces to model different serving scenarios; **(4)** reproducibly replay these workloads against a target serving system; and **(5)** capture detailed serving system and GPU metrics. Specifically, XPerf offers the following features.

First, XPerf supports application-agnostic tracing of diverse agentic workloads (§3.3) to accurately construct the execution graphs (§3.4). It automatically instruments and collects traces from agentic applications developed with popular frameworks such as LangChain [26] and LangGraph [2] without code modifications, and requires only single-line annotations for agentic applications built from scratch. With the traces, XPerf efficiently constructs execution graphs using a time-span-tree-based analysis to capture the dependencies among LLM and tool calls.

Second, XPerf can synthesize new workloads from collected traces to model diverse serving scenarios (§3.5). It can synthesize diverse serving system workloads such as online serving, offline batch inference, and fixed-concurrency workloads by varying the request arrival pattern over the constructed execution graphs. This enables users to benchmark a target LLM serving system under a wide range of realistic agentic workloads.

Third, XPerf can reproducibly replay traced agentic workloads to benchmark a target serving system (§3.6). For each LLM call, XPerf submits the exact recorded input token IDs to preserve prefix-cache hit behavior, and can force the serving engine to decode the recorded output tokens, for ensuring the replay accuracy. It issues these calls in the dependency order specified by each execution graph, taking the tool call duration into consideration, such that the replayed workload remains realistic even when the serving engine under test differs from the one used during tracing. This enables serving system developers to fairly and reproducibly compare different serving systems and optimizations.

Finally, XPerf can profile the serving system and underlying hardware to obtain detailed metrics (§3.7). It reports key serving-system metrics, including KV cache usage, prefix cache hit rate, and token throughput. At the hardware level, it exposes GPU metrics such as tensor core and memory bandwidth utilization. Together, these metrics provide a solid foundation for understanding how serving systems behave under agentic workloads. XPerf collects them with a lightweight profiler which adds negligible overhead.

To demonstrate the capabilities of XPerf, we use eight popular agentic applications that include deep research, coding, Q&A, and general-purpose assistance (Table 2). They serve as ready-to-use workloads in XPerf. XPerf faithfully replays agentic workloads, with a mean absolute percentage error in end-to-end request latency below 3% for both isolated and concurrent requests (§5.3).

With our accurate replay, we study the end-to-end performance of agentic applications, showing that XPerf can break down end-to-end latency (§5.4) while providing detailed serving-system and hardware metrics (§5.5). With this detailed profiling, XPerf surfaces inefficiencies such as prompt designs that fail to exploit prefix caching. It helps users diagnose, from both software and hardware perspectives, serving-system performance issues such as KV cache thrashing—where concurrent requests repeatedly evict one another's cached prefixes, incurring frequent recomputation. We further demonstrate that XPerf can model diverse serving scenarios (§5.6) and scale to benchmarking multi-engine serving systems, quantifying how LLM call routing policies affect scale-out throughput (§5.7). Additionally, XPerf assists in serving-system debugging: by minimizing agentic workload variation, it makes the effects of even small serving-system changes clearly visible (§5.8). Finally, we show that XPerf incurs minimal overhead during both trace collection and replay (§5.9). We will open-source XPerf for public use.

2 Background and Motivation

2.1 Agentic AI Applications

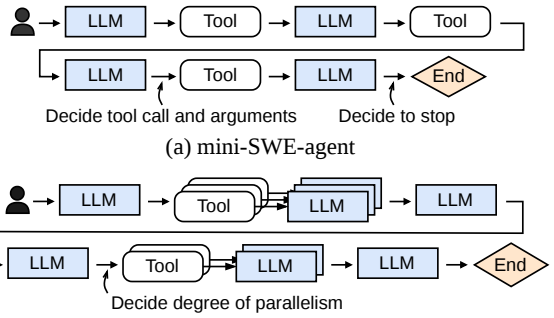
At the core of agentic applications are LLM agents, which are autonomous programs that leverage LLMs to drive their decision-making and execution. Agents can decompose and plan complex tasks [73], reflect on prior experience [68], collaborate with one another [62], and explore external environments using tools [66]. These capabilities power applications such as personal coding agents [4], autonomous data center managers [11], and scientific research agents [13, 53].

Compared to conventional chatbots, agentic applications exhibit two major differences. First, a user request to an agentic application spawns an *execution graph* of multiple dependent LLM and tool calls, rather than a single LLM call. For example, as shown in Figure 1, mini-SWE-agent [63] issues chains of alternating LLM and tool calls (i.e., ReAct [66] iterations), while Open Deep Research [27] may produce many parallel LLM calls to summarize tool call results. Second, agentic applications are significantly more dynamic, because their execution graphs are influenced by the nondeterministic decoding output of LLM calls, which can determine which tool to call, which agent to invoke next, and whether to stop execution. Thus, the execution graph structure (e.g., the number of ReAct iterations and the number of parallel LLM calls) can vary significantly even for identical user requests (see detailed discussion in §2.3).

2.2 System Infrastructure for Agentic Applications

From a system perspective, agentic applications execute atop a well-abstracted LLM serving layer, as shown in Figure 2.

Agentic Application Layer. An agentic application is a program that defines tools, LLM agents, and a workflow that controls how they interact. Its logic runs primarily on CPUs, with LLM calls dispatched to a separate serving system. Each tool is a function call (e.g., querying a database, calling a web



(a) mini-SWE-agent

(b) Open Deep Research

Figure 1. Example execution graphs of mini-SWE-agent and Open Deep Research. The graph structure is influenced by the decoding output of LLM calls.

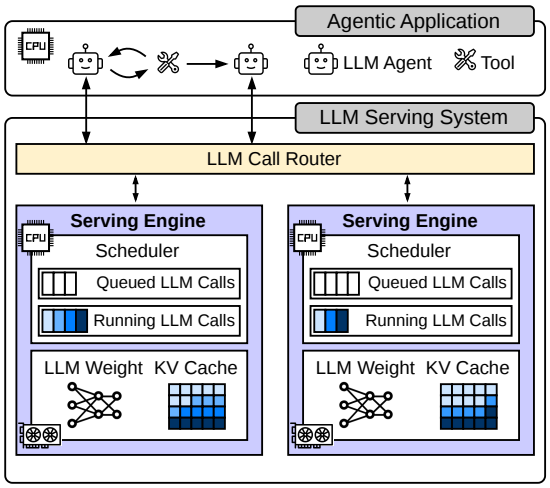
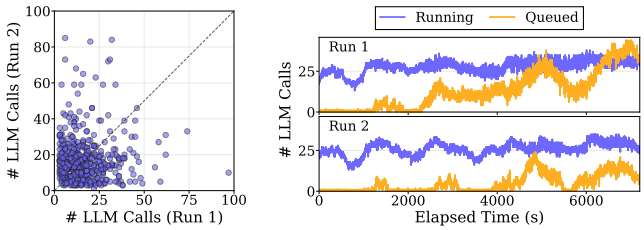


Figure 2. System infrastructure for agentic applications.

API, or executing code) and is exposed to an LLM through a dedicated prompt that lists each tool's purpose and the arguments it accepts. Each agent is a piece of code that assembles such prompts for an LLM call interface (e.g., the OpenAI API [44]): it sets a system prompt defining the agent's role, includes the descriptions of the tools, and prior context such as earlier tool outputs. The workflow, implemented either from scratch or on agentic frameworks [2, 26, 62], forms a control flow graph over agents and tools: it specifies which agent/tool to execute next and how prior outputs are passed to later ones. As the agent-specific complexity (i.e., prompt construction and output parsing) resides at the application layer, LLM serving system sees only standard LLM calls.

LLM Serving Layer. An LLM serving system contains one or more serving engines, each of which holds the model weights and KV cache. Serving engines efficiently execute LLM calls on the underlying hardware like GPUs. When an engine receives LLM calls but lacks sufficient KV cache entries to run them all, its scheduler queues the extra calls and admits them as long as enough KV cache entries become available. Systems with multiple serving engines employ a router (e.g., vLLM-Router [59]) to determine which engine should serve an LLM call based on factors such as each serving engine's load and prefix cache locality [35, 41, 55, 57, 59].



(a) LLM calls per request. Each dot represents two runs of the same request.

(b) Running and queued LLM calls in the serving system across two runs over the same two-hour time window.

Figure 3. Run-to-run variance of agentic applications, illustrated with mini-SWE-agent. Results are profiled on SWE-bench, served by gpt-oss-120b online at 4.5 requests/minute.

2.3 Benchmarking LLM Serving for Agentic AI

Today's mainstream serving systems, such as vLLM [25], are designed primarily for independent LLM calls. Consequently, they suffer from suboptimal LLM call scheduling [8, 36] and KV cache management [31**?**] when serving agentic workloads. To understand and mitigate these inefficiencies, it is essential to benchmark serving systems under realistic agentic workloads, allowing developers to accurately identify system bottlenecks and test proposed optimizations.

Conducting such benchmarks requires reproducibly stress-testing the serving system with diverse agentic applications. However, the agentic applications have dynamic execution graphs, making reproducible benchmarking fundamentally challenging. Specifically, even when identical user requests are executed with a fixed random seed, the resulting execution graphs differ substantially across runs.

For example, as shown in Figure 3a, when running mini-SWE-agent [63] on SWE-bench [19], the total number of LLM calls triggered by identical user requests varied by an average of 12.9 calls between two runs (76% of the average number of LLM calls per request). Thus, even with identical user requests, arrival times, and random seeds[1], accumulated request-level variance significantly alters the actual workload during benchmarking. Figure 3b shows the overall trend in queued LLM calls diverging significantly across two runs within a two-hour window after warm-up. This divergence yields significantly different benchmarking outcomes. Within the two-hour observation window, run 2 exhibits 65% less cumulative queuing delay across all LLM calls than run 1. Correspondingly, run 2's average end-to-end request latency is 25% lower, while the 95th percentile (P95) latency is 30%

[1]Even with a fixed random seed, decoding output may diverge across executions because batched inference can introduce minute floating-point differences [70]. Although employing batch-invariant kernels can remove this source of nondeterminism, it typically incurs substantial overhead [16].

Table 1. Comparison with existing benchmarks for LLM agents and LLM serving systems.

Category	Benchmark	Diverse Agentic Applications	Configurable Load Generator	System Metrics	Reproducibility		
					Exec. Graph	Input Tokens	Output Tokens
Agent Evaluation	Harbor [28]	✓	✗	✗	✗	✗	✗
	HAL [23]	✓	✗	✗	✗	✗	✗
Serving System Performance	InferenceX [54]	✗	✓	✓	✗	✓	✗
	LLMPerf [21]	✗	✓	✓	✗	✓	✗
	MLPerf [51]	✗	✓	✓	✗	✓	✗
	AIPerf [42]	✗	✓	✓	✗	✓	✗
	AA-AgentPerf [6]	✗	✗	✓	✓	✓	✗
	XPerf (Ours)	✓	✓	✓	✓	✓	✓

lower. It is hard to isolate the effect of system optimizations from the variance of workload patterns.

Unfortunately, no existing benchmark addresses the above challenge (see Table 1). Agent evaluation benchmarks [23, 28] focus on evaluating the ability of agents to complete tasks rather than the performance of the underlying serving system. They include a variety of agentic applications, but offer no configurable load generator for stress-testing the serving system, expose no system-level performance metrics, and cannot reproduce a consistent workload across runs.

LLM serving benchmarks [21, 42, 51, 54] offer users with configurable load generators and system profiling, but their input prompts are sampled from datasets or synthesized from user-specified distributions. Such independently drawn prompts cannot replicate the unique behavior of agentic workloads – a single request expands into many interdependent LLM calls whose prompts are assembled from earlier outputs and tool results. AA-AgentPerf [6] worked toward performance benchmarking for agentic serving systems, but it is a closed ranking service. Developers can only submit a system to the service provider and receive a published ranking, it prevents developers from reproducing results, adding new applications, and identifying system bottlenecks. Moreover, its fixed workloads may not reflect what developers deploy in their own infrastructure.

These limitations motivate us to build an open-source and extensible benchmarking framework for agentic serving.

3 Design and Implementation

3.1 Goals of XPerf

We develop XPerf with the following goals:

- **Reproducible execution.** XPerf should benchmark serving systems with reproducible agentic workloads, isolating serving system performance from the run-to-run variance of agentic applications.
- **Workload diversity.** XPerf should support a wide range of agentic applications and diverse serving scenarios such as online serving and offline batch inference.
- **System metric collection.** XPerf should provide a set of lightweight profiling tools to collect detailed system metrics across the full stack, from the agentic application to the serving system to the underlying hardware, such as end-to-end user request latency, prefix cache hit rate, GPU hardware utilization, and others.
- **Portability.** XPerf should be compatible with agent development frameworks, allowing users to easily extend it with new applications. It should also be able to evaluate diverse serving systems, as well as larger-scale deployments with multiple serving engines.
- **Ease of Use.** XPerf should minimize developer burden throughout the end-to-end benchmarking workflow, from integrating a new agentic application to generating new workload patterns and profiling system performance.

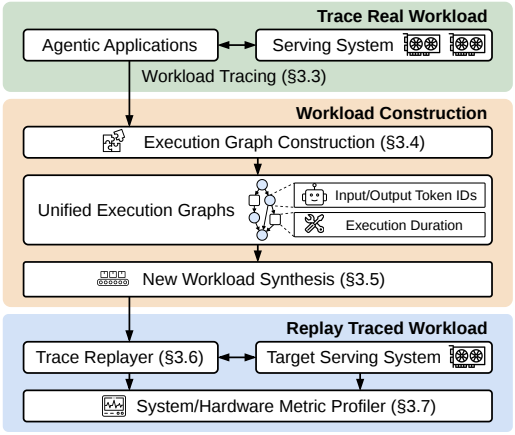


Figure 4. System overview of XPerf.

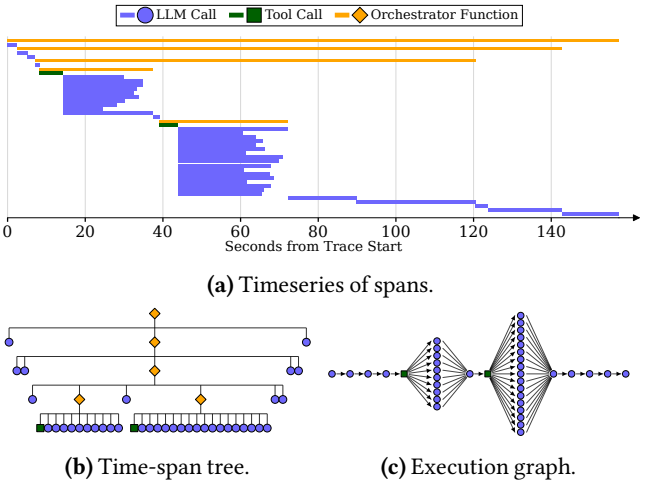
3.2 System Overview

XPerf achieves these goals via a trace replay approach, as shown in Figure 4. To collect agentic workload traces, a user first instruments the target application with XPerf. The user then runs the application while XPerf traces its LLM calls and tool calls (§3.3). Based on the trace, XPerf first constructs the execution graph for each user request (§3.4). It then generates new workload patterns (if needed) by deciding when to replay each graph, either following user-specified request arrival patterns or replaying the exact arrival times observed during tracing (§3.5). During trace replay, XPerf load-tests the target LLM serving system by replaying these execution graphs. It

mini-swe-agent_gpt-oss-120b_run42.jsonl


```
...
{   // LLM call
    "req_id": "swe-bench-verified_req025",
    "id":         "d10a86b11341c000",
    "parent_id": "1508ab21833fab78",
    "type": "llm",
    "start_time": "2026-05-01T14:20:58.529193",
    "end_time":   "2026-05-01T14:22:14.455251",
    "input_token_ids":  [200006, 17360, 200008, 3575, ...],
    "output_token_ids": [200005, 35644, 200008, 2167, ...],
}
{   // tool call
    "req_id": "swe-bench-verified_req024",
    "id":         "570cc265552120cb",
    "parent_id": "cecaec28b8edaf27",
    "type": "tool",
    "start_time": "2026-05-01T14:22:09.847307",
    "end_time":   "2026-05-01T14:22:14.487642",
}
{   // orchestrator function
    "req_id": "swe-bench-verified_req024",
    "id":         "cecaec28b8edaf27",
    "parent_id": "968a68d7e5cc0819",
    "type": "orchestrator",
    "start_time": "2026-05-01T14:20:21.390210",
    "end_time":   "2026-05-01T14:22:14.487695",
}
...
```


Figure 5. Example trace collected by XPerf.



(a) Timeseries of spans.

(b) Time-span tree. **(c)** Execution graph.

Figure 6. Execution graph construction.

traverses each graph to issue LLM calls in dependency order, such that the replayed graph reproduces the behavior of the original request (§3.6). Meanwhile, the system profiler (§3.7) monitors execution and collects system performance metrics.

3.3 Workload Tracing

XPerf employs a tracing submodule to collect the information needed for workload replay. However, real-world agentic applications are developed with different frameworks or even built from scratch, they lack a common interface to collect application traces. The XPerf tracing module generalizes across diverse applications while requiring minimal changes to their code. It decomposes the agentic application's execution into two operation types, LLM calls and tool calls, and traces each separately:

LLM calls. Agentic applications typically make LLM calls through standardized interfaces such as OpenAI APIs [44] and Anthropic SDKs [5]. XPerf traces all LLM calls issued through these interfaces via library interposition [20]. Specifically, XPerf wraps the LLM call interface in the shared client library, so every LLM call will first enter the XPerf wrapper function. Once the call returns, XPerf records the input and output token IDs the serving engine returns for each LLM call. It also employs OpenTelemetry [58] to record the start and end time of each call, as well as the user request it belongs to. Because the interposition occurs within shared client libraries, it requires no changes to the application code.

Tool calls. In contrast to LLM call interfaces, tool calls lack a standardized interface. A tool call can be a bash command [63], a web search API [27], or a custom code segment through which an agent makes a decision [73]. To support tracing different kinds of tool calls, for tool calls defined under popular agentic frameworks such as LangChain [26] and LangGraph [2], XPerf uses the same library interposition technique, wrapping the tool calls with XPerf tracing functions without any application changes. For applications developed from scratch in Python, XPerf provides a single-line annotation API that allows users to manually wrap each tool call. With these annotations, XPerf likewise traces the start and end time of each tool call as well as the user request it belongs to via OpenTelemetry. As a safeguard against missed annotations, whenever a new LLM or tool call is made, XPerf also checks whether any other call completed within the preceding window (20 ms by default). If none did, XPerf reports the gap as a likely unannotated region. This increases observability of the application by highlighting unnoticed overhead.

In addition to LLM and tool calls, we also introduce *orchestrator functions* to trace how these calls are nested, facilitating more accurate execution graph construction. We define orchestrator functions as functions that compose LLM calls, tool calls, or smaller orchestrator functions into more complex logic. They are common in applications built with agentic frameworks. For example, LangChain [26] and LangGraph [2] wrap each LLM call and tool call in a `Runnable` and compose these into a chain or graph. These chains or graphs itself is also a `Runnable` class. Because these frameworks expose all orchestrator functions through a common class such as `Runnable`, XPerf interposes on that class to trace the start and end time of every orchestrator function once it detects one of these frameworks.

XPerf also records the call hierarchy. For each LLM call, tool call, and orchestrator function, XPerf uses OpenTelemetry to record the parent orchestrator function that encloses it. For each agentic application run, XPerf logs all information to a JSONL file, as illustrated in Figure 5. The trace records the start and end time and the parent ID of every user request, LLM call, tool call, and orchestrator function. For LLM calls, we also record the input and output token IDs. This tracing incurs negligible latency overhead, which we quantify in §5.9.

Algorithm 1: Execution graph construction

Input : Root span s of a time-span tree; each span c exposes start time $c.start$, end time $c.end$, and children $c.children$
Output : Execution graph $G = (V, E)$

1 **Function** `DependentSpans`(S):
2 $P \leftarrow \{(s_1, s_2) \in S \times S : s_1.end \le s_2.start\}$;
// drop (a,b) when a reaches b transitively
3 **return** `TransitiveReduction`(P);
4 **end**
5 **Function** `InterGraphEdges`(G_1, G_2):
6 $T \leftarrow \{u \in G_1.V : \deg^{+}_{G_1}(u) = 0\}$; // sinks of G_1
7 $R \leftarrow \{v \in G_2.V : \deg^{-}_{G_2}(v) = 0\}$; // sources of G_2
8 **return** $\{(u, v) : u \in T, v \in R\}$;
9 **end**
10 **Function** `BuildGraph`(s):
11 **if** $s.children = \emptyset$ **then return** $(\{s\}, \emptyset)$; // LLM/tool call
12 $\mathcal{G} \leftarrow \emptyset$; $V \leftarrow \emptyset$; $E \leftarrow \emptyset$;
13 **foreach** $c \in s.children$ **do**
14 $\mathcal{G}[c] \leftarrow$ `BuildGraph`(c);
15 $V \leftarrow V \cup \mathcal{G}[c].V$; $E \leftarrow E \cup \mathcal{G}[c].E$;
16 **end**
17 **foreach** $(c_1, c_2) \in$ `DependentSpans`($s.children$) **do**
18 $E \leftarrow E \cup$ `InterGraphEdges`($\mathcal{G}[c_1], \mathcal{G}[c_2]$);
19 **end**
20 **return** (V, E);
21 **end**

3.4 Execution Graph Construction

Obtaining the traces shown in §3.3 itself is insufficient. If we naively replay a workload trace by issuing each LLM call with its recorded token IDs at its recorded time, it fails to reflect the dependencies among LLM calls. Whenever the serving system or the underlying hardware differs from the setup used during tracing, the replayed workload becomes unrealistic. For example, an improved engine that decodes tokens faster finishes each LLM call sooner, so the calls that depend on it should also be issued earlier. To accurately reproduce the behavior of agentic applications, it is essential to construct the dependencies between the LLM and tool calls triggered by each user request.

XPerf models the execution of each user request as a directed acyclic graph (DAG), in which each node is an LLM call or a tool call. A directed edge $a \rightarrow b$ indicates that call b depends on the result of call a and thus can execute only after a completes. To build the DAG of each user request, XPerf must infer the dependencies between calls.

Fortunately, the trace already records every operation's time span with its parent span. XPerf links each span to its parent to build a *time-span tree* that recovers the application's structure. Figure 6a shows example spans traced from Open Deep Research, and Figure 6b shows the tree built from them. LLM call and tool call spans form the leaf nodes, since they perform the actual work and invoke no further calls. Their parent spans, which are orchestrator functions (§3.3), form the internal nodes, and the root node is the span of the entire user request.

With the time-span tree, XPerf further builds the execution graph of the application request. We show the graph construction algorithm in Algorithm 1. Specifically, for each leaf node, which is either the time span of an LLM or a tool call, it produces a single-node DAG (line 11). For each non-leaf node, it first builds subgraphs for its children (line 13) and identifies dependencies between them (line 16). XPerf treats two spans as dependent when one finishes before the other starts (line 2), and keeps only direct dependencies (line 3). For each pair of dependent spans, XPerf connects their subgraphs by adding edges (line 17) from every call in the earlier span that has no successor (line 6) to every call in the later span that has no predecessor (line 7). Overall, XPerf gradually connects the subgraphs of the child subtrees into larger graphs, ultimately producing the entire execution graph for each user request. Figure 6c shows the execution graph constructed from Figure 6b.

Using the time-span tree information is important to avoid constructing false dependencies. For example, LATS [73] launches multiple independent parallel tasks, each containing one LLM call followed by a tool call. Thus, inferring dependencies without consulting the time-span tree (i.e., applying `DependentSpans` directly to all calls) introduces false dependencies: when two LLM calls in different tasks finish at nearly the same time, the tool call in one task can mistakenly treat an LLM call in another task as its predecessor. XPerf avoids this issue because the calls from different tasks will be assigned as leaf nodes under different internal nodes (e.g., different orchestrator functions) in the time-span tree. It can correctly construct the execution graph of all applications used in our evaluation.

XPerf efficiently constructs DAGs for diverse agentic applications. The traces we collected in our evaluation contain no more than 1000 LLM/tool calls per user request. Each DAG can be constructed in less than 200 ms using a single thread on an AMD EPYC 9334 32-core CPU.

3.5 New Workload Synthesis

By default, XPerf replays each request at the exact arrival time observed during tracing. However, users often need to benchmark a serving system under serving scenarios beyond the traced one, such as online serving at a higher request rate or offline batch inference. Rather than collecting a new trace for each scenario, XPerf allows the synthesis of new workloads by varying the request arrival pattern across the same set of DAGs.

First, XPerf can generate online serving workloads at a target request rate by sampling arrival times from a Poisson process. Second, XPerf can generate an offline batch inference [56] workload by dispatching a specified number of requests at the start of execution. Third, XPerf supports maintaining a fixed number of concurrent requests (i.e., fixed-concurrency serving) to synthesize a workload in which multiple users share one serving system [6]. Finally, XPerf supports

custom request arrival patterns, specified by a user-provided configuration file that lists each request ID and its arrival time.

By default, XPerf treats tool-call durations as independent of serving system load, since tool calls typically execute on a separate machine. However, XPerf lets users configure tool call durations to study their impact on serving system behavior and end-to-end request performance.

3.6 Replaying Traces

We now describe how XPerf replays the traced workload bottom-up: first, how XPerf replays individual LLM calls, then how it replays each execution graph, and finally, how it replays the entire workload.

Replaying an LLM call. With the collected input/output token IDs, XPerf supports two LLM call replay modes. In the first mode, XPerf sends the recorded input token IDs to the serving engine to preserve the prefix-cache-hitting behavior and instructs it to ignore the `EOS` token (e.g., set `ignore_eos` for vLLM). The engine therefore decodes until it reaches `max_tokens`, which XPerf sets to the recorded output sequence length. This ensures the engine decodes the same number of tokens recorded in the trace. The second mode further forces the serving engine to generate exactly the recorded output token sequence. This benefits benchmarking when serving LLMs with mixture-of-experts [29] or native sparse attention [69], as the generated token IDs influence expert routing and attention sparsity. The first mode does not require any modification to the serving engine, while the second requires minor changes. We detail our implementation of output token ID enforcement in §4.

Replaying an execution graph. Execution graph replay follows DAG semantics: a node executes as soon as all its predecessors finish. The trace replayer therefore launches each node as an asynchronous task that suspends until all its predecessor calls complete. When awoken, an LLM call node replays its LLM call and waits for the call to finish, while a tool call node further sleeps for the traced tool call duration to reproduce its latency.

Replaying the entire workload. XPerf replays the entire workload by replaying each execution graph at the time specified by the workload synthesizer, running multiple in-flight execution graphs via multiprocessing. For workloads with many requests, holding all execution graphs in memory is costly, so XPerf loads them on the fly: a fixed-size buffer prefetches execution graphs for upcoming requests and frees memory for finished ones. Its memory footprint therefore depends only on the number of concurrent requests, not the total workload size.

3.7 System Profiling

XPerf integrates a system profiler to collect serving-system and hardware metrics. For serving-system metrics, it polls the HTTP `/metrics` endpoint exposed by most popular serving systems, capturing (1) the number of running and queued LLM calls, (2) KV cache capacity and prefix cache hit rate, and (3) input and output token throughput. For GPU metrics, the profiler uses NVIDIA's CUDA Profiling Tools Interface [43] (CUPTI). It measures utilization of (4) tensor cores, (5) streaming multiprocessors and (6) memory bandwidth. The metric sampling interval is set as 1 second for the serving system and 20 ms for the GPU and is configurable. We show the profiler has low overhead in §5.9.

4 Implementation Details

XPerf provides eight containerized, instrumented agentic applications to reduce environment setup burden (see §5.2). It is implemented in Python as four modules: application tracing, execution graph construction, workload synthesis, and trace replay. The tracing module configures the OpenTelemetry SDK [58] with automatic instrumentation to generate spans, which are exported over OTLP [46]/gRPC [15] to Jaeger [18], an OpenTelemetry-compatible tracing backend. The system profiler runs as a background process, periodically sampling serving-system metrics from each engine's HTTP endpoint and GPU metrics via CUPTI [43]. For multi-node deployments, XPerf can deploy the profiler to each node via Ansible [3] to report per-GPU metrics.

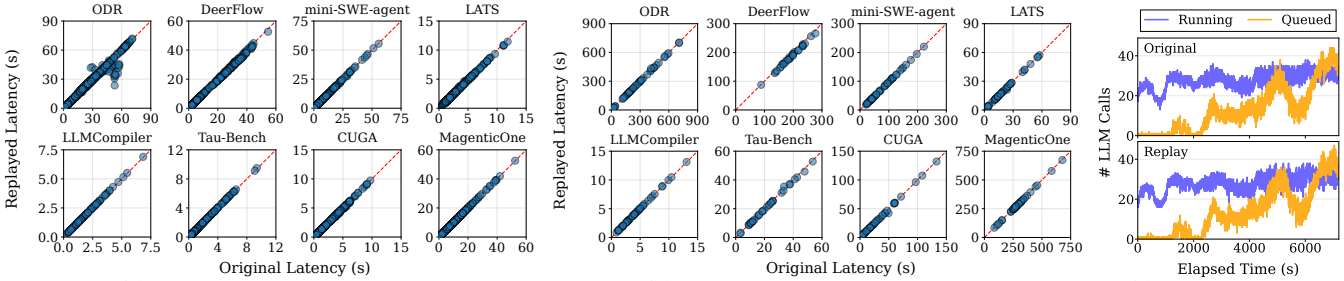
We prototype output token ID enforcement in vLLM [25]. In the messages XPerf sends to vLLM for each LLM call, XPerf appends an extra message that carries the output token IDs and assigns it a special role `replayer`, distinguishing it from the normal `system`, `user`, and `assistant` roles. vLLM moves this special message into the per-call sampling parameters and drops it from the message list. After each decoding iteration, vLLM overrides the sampled tokens with the supplied token IDs. To support speculative decoding [30], XPerf instructs the serving engine to report the number of tokens drafted successfully in each decoding iteration during trace collection. To replay, XPerf sends the per-iteration accepted-token counts through the same special message, and vLLM accepts tokens accordingly. We find no measurable overhead from output token ID enforcement.

5 Evaluation

Our evaluation shows that XPerf can (1) accurately replay the serving system workload produced by diverse agentic applications (§5.3); (2) collect end-to-end performance metrics for each user request and provide a detailed breakdown (§5.4); (3) profile detailed serving system metrics and hardware utilization metrics (§5.5); (4) synthesize workloads for diverse serving scenarios (§5.6); and (5) scale to multi-engine serving systems (§5.7). We also demonstrate how XPerf's reproducible trace replay helps developers debug serving systems for agentic applications (§5.8). Finally, we show that XPerf incurs minimal overhead during both trace collection and replay (§5.9).

Table 2. Agentic workloads used in our evaluation. LLM-call counts are per user request and profiled using gpt-oss-120b.

Application	Framework	Category	Tools	Dataset	LLM Calls	
					Avg.	P95
Open Deep Research [27]	LangGraph	Deep Research	Web search API	ResearchyQuestions [52]	69.6	143.8
DeerFlow [7]	LangGraph	Deep Research	Web search API	ResearchyQuestions [52]	14.7	19.0
mini-SWE-agent [63]	Custom	Coding	Terminal	SWE-Bench [19]	17.0	39.0
LATS [73]	LangGraph	Q&A	Web search API	HotpotQA [64]	10.6	26.0
LLMCompiler [24]	LangChain	Q&A	Web search API, Terminal	ParallelQA [24]	2.4	6.0
Tau-Bench [65]	Custom	General	Database	Airline, Retail [65]	14.9	26.0
CUGA [37]	LangGraph	General	Web search API, Terminal	AssistantBench [67]	7.3	17.0
MagenticOne [12]	AutoGen	General	Web browser, Terminal	AssistantBench [67]	43.7	54.1



(a) Individual LLM call latency. **(b)** End-to-end user request latency. **(c)** Online serving.

Figure 7. Validation of replay fidelity.

5.1 Experimental Setup

LLM. We use gpt-oss-120b [45] and Qwen3.6-35B-A3B [50] as the LLM backbones for our agentic applications. Both are state-of-the-art mixture-of-experts models explicitly post-trained for tool usage. By default, we use gpt-oss-120b.

Serving System. We use vLLM v0.19.0 [25] as the serving engine and vLLM Router [59] to route LLM calls to different engines in multi-engine deployments. We apply minor modifications to vLLM to enable CPU prefix caching for Qwen3.6-35B-A3B and output token ID enforcement.

Testbed. Our testbed is a server with dual AMD EPYC 9334 32-core CPUs with 1.5TB of DDR5 memory. The server has two NVIDIA H100 NVL 94GB GPUs, each attached to the host through a PCIe Gen5 x16 link. We use two such servers connected by a 100Gbps InfiniBand switch for scalability experiments in §5.7.

5.2 Agentic Applications

Our evaluation uses diverse agentic workloads covering a variety of popular use cases, as shown in Table 2.

Deep Research. Deep research applications conduct large-scale web searches to answer complex user queries in detail. LLM calls are used to produce search queries and summarize search results. Open Deep Research (ODR) launches multiple web searches in parallel, each followed by an LLM call that summarizes its results; on average, 20.1 such search-and-summarize operations run concurrently. DeerFlow instead makes sequential LLM calls, using each call to summarize multiple search results. Compared to ODR, it produces fewer LLM calls, each with significantly longer prompts.

Coding. We use mini-SWE-agent as a representative coding agent. It follows the ReAct paradigm, alternating between LLM calls and terminal commands to resolve GitHub issues from popular open-source repositories. Its LLM calls expose substantial opportunities for prefix-cache sharing: the prompt for each LLM call concatenates the inputs and outputs of all preceding LLM and tool calls.

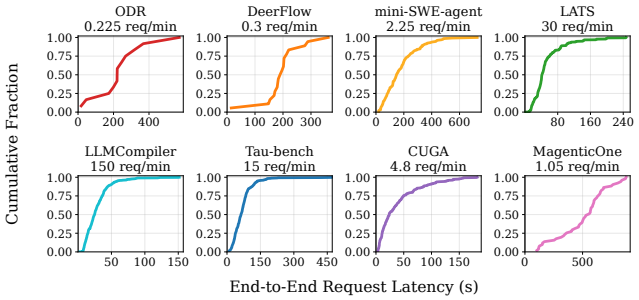


Figure 8. CDF of end-to-end latency per user request.

Q&A. We select two representative Q&A applications, LATS and LLMCompiler. LATS issues relatively more LLM calls, as it explores and evaluates many candidate answers in parallel before settling on a final one, while LLMCompiler issues fewer: a single LLM call plans a graph of downstream calls, which are then executed without further planning.

General. We also include three general-purpose applications. Tau-Bench uses a single ReAct agent for customer service, with tools to query and update customer records in a database. CUGA builds a generalist ReAct agent with code execution and web search tools. MagenticOne coordinates multiple agents, raising its LLM-call count substantially.

5.3 Replay Fidelity

We first check that XPerf traces and replays individual LLM calls accurately by replaying each user request in isolation.

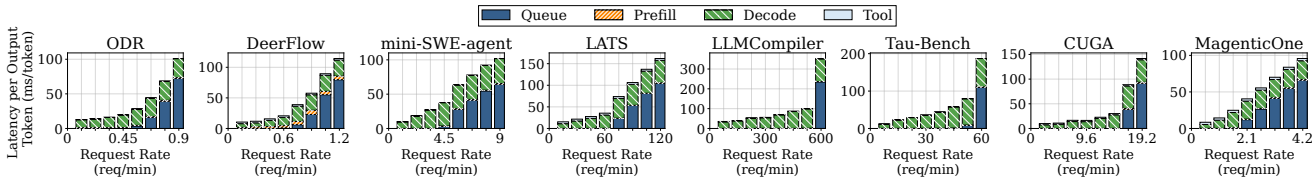


Figure 9. Latency breakdown per output token.

As shown in Figure 7a, the replayed per-LLM-call latencies closely match the original across all eight applications. The only visible gap is in ODR, where 4.6% of calls exceed a 3% relative error. This is because ODR issues multiple LLM calls in parallel. The serving engine cannot process all parallel calls at once, and the exact set of queued calls during replay differs from the set during trace collection. However, this per-call gap does not affect replay fidelity because the serving workload is determined by the dependencies among LLM calls. We further confirm this by comparing the end-to-end user request latencies in Figure 7b. For all applications, the replayed request latency has a mean absolute percentage error below 3%, including ODR. This demonstrates that XPerf replays LLM calls with accurate timing by correctly reconstructing the execution graph of each user request.

We further verified that XPerf can reproduce serving-system behaviors when multiple requests are served concurrently. Recall from Figure 3b that two independent runs of the same workload diverge significantly. We replay the first run with XPerf for 2 hours after warmup and show the comparison in Figure 7c. With XPerf's replay, the serving system states matched closely throughout this observation window. Specifically, the mean absolute error in the number of running and waiting calls (sampled every second) is 1.4 and 1.6, respectively, while the cumulative LLM call queuing time differs by only 2.9%. The mean absolute percentage error of replayed requests' latency is 1.9%. Together, these results confirm that XPerf can faithfully reproduce the original application's serving-system workload, both for individual and concurrent requests, providing a reliable foundation for evaluating system performance.

5.4 End-to-end Performance

XPerf supports profiling the end-to-end latency of each user request and provides a detailed breakdown. We demonstrate this by replaying collected traces and synthesizing online-serving workloads at varying request rates.

Figure 8 shows the distribution of end-to-end request latency for each application, with the request rate set below the serving system's saturation point. The average latency varies widely across applications, as they perform different types of tasks with different difficulties. Beyond average latency, request latency is long-tailed for most applications, with the p95 latency reaching up to 6.60× the p50 latency. This long-tail distribution is primarily caused by agents that fail to make progress yet keep issuing LLM and tool calls instead of stopping. DeerFlow and MagenticOne are notable exceptions, since DeerFlow enforces a strict limit on search iterations, while MagenticOne has an orchestrator agent that monitors other agents' progress and proactively terminates the workflow upon detecting a stall.

The two deep-research applications show a high minimum latency. Only 17% of ODR requests complete within 175 seconds, and only 6% of DeerFlow requests complete within 100 seconds. This is because a large fraction of the requests are issued by these applications as a research task, and they must complete at least one research pass and generate a long final report on the critical path.

Figure 9 provides a detailed breakdown of the end-to-end latency per output token at different request rates. Each measurement runs for two hours. For applications with parallel LLM calls, we count the output tokens on the critical path. The end-to-end latency is dominated by decoding when the serving system is not saturated, accounting for 60%-98% across the studied applications. The per-token decode latency increases with the request rate up to saturation, then stabilizes. In contrast, prefill accounts for only a small fraction of end-to-end latency: less than 23% for DeerFlow and 4% on average for the remaining applications. Prefill is most significant for DeerFlow, which has the longest LLM call input prompts among the studied applications (38.8k tokens on average). As the request rate increases, queuing delay takes a growing share of end-to-end latency, since the average queue length grows. Under heavy overloaded scenarios, queuing dominates up to 69% of the end-to-end latency at the highest request rate we tested.

- XPerf enables end-to-end tracing for each user request across diverse agentic applications and provides a detailed latency breakdown.
- Request latency in many agentic applications is long-tailed. These tail requests arise when the agent fails to converge on a solution and issues repeated LLM calls.
- At low request rates, decoding is the performance bottleneck for agentic applications. Queuing delay can become another significant source of overhead as load increases.

5.5 Serving System Performance Characterization

A key capability of XPerf is profiling detailed metrics across the serving system and underlying hardware. Together, these metrics enable us to determine when the system saturates, its throughput at a given load level, and which resources are the limiting factors. To demonstrate this, we profile the workloads

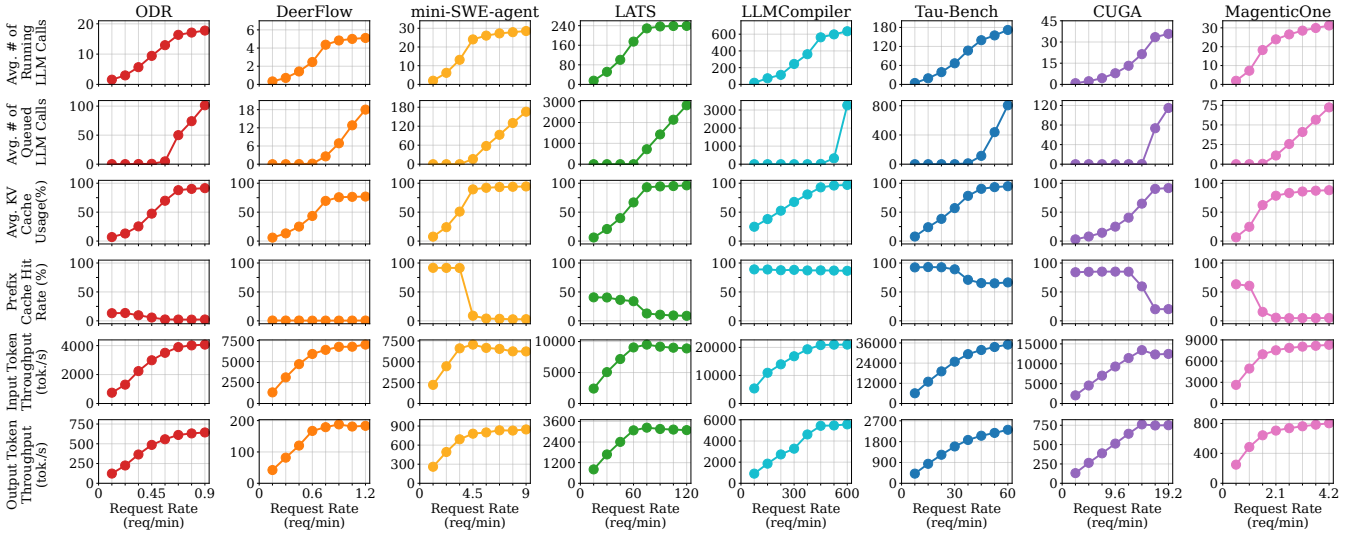


Figure 10. Serving system metrics.

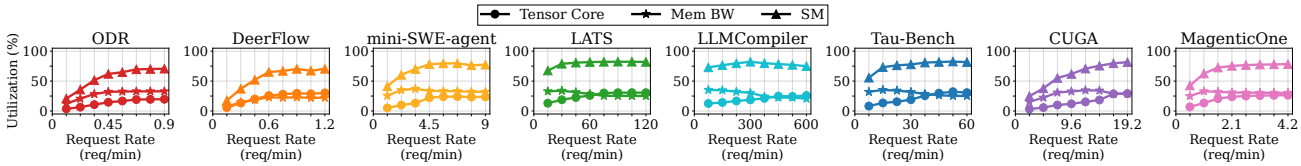


Figure 11. GPU resource utilization under different request rates.

from §5.4 under different request rates, reporting serving system metrics in Figure 10 and hardware utilization in Figure 11.

Request concurrency. The numbers of running and queued LLM calls together indicate when the serving system saturates. As the request rate increases, the number of running calls increases until the system reaches its concurrency limit, after which this number stabilizes and any additional requests wait in the queue. Therefore, the saturation point is the highest request rate the system can sustain while keeping the queue from growing without bound. For instance, with mini-SWE-agent, the number of running calls stabilizes at 24 on average, while the queue begins to grow once the request rate reaches 4.5 requests per minute.

KV cache usage. At saturation, the KV cache is nearly full for most applications, making cache capacity a key factor that limits further scaling. DeerFlow saturates at a lower usage of around 75%, since its LLM calls have an average of 40k context tokens while our tested serving system has ~213k token capacity. At this coarse granularity, once only a small amount of cache is free, the remaining space cannot admit another call, so utilization stays low. Also, under the pressure of increasing request rates, the prefix-cache hit rate drops, which can reduce both prefill and decode throughput.

Prefix-cache hit rate. At low request rates, most applications achieve substantial prefix-cache sharing, with hit rates between 41% and 91%. Deep research applications have low prefix cache sharing. Specifically, ODR has a hit rate of 14%, since its LLM call inputs mostly consist of unrelated search results. DeerFlow stays below 1% at all request rates, since it places a timestamp at the start of its system prompt, which changes the prefix of every LLM call and eliminates nearly all prefix-cache reuse. Removing the timestamp raises the hit rate to around 30% at low request rates.

For the remaining applications, the hit rate drops sharply once the request rate passes a threshold and then stabilizes at a lower value as the rate increases further. The drop reflects prefix-cache thrashing, where the serving system treats each LLM call as independent and evicts a request's accumulated context while no call is using it under KV cache pressure, even though a later call will reuse it. For example, in mini-SWE-agent, each call reuses this context as its prefix, reaching 91% at low rates but falling below 4% as the rate rises. The stabilized hit rate tracks the system prompt's share of input tokens, which stays cached across requests, settling near 88% for LLMCompiler and 61% for Tau-Bench.

Token throughput. Input and output token throughput both increase with request rate at first, then stabilize at higher rates as KV cache capacity saturates. Added load can even reduce input-token throughput by causing prefix cache thrashing, since the serving system must recompute the evicted prefixes instead of reusing them, lowering it by up to 12% for mini-SWE-agent. This recomputation also interferes with decode, so output-token throughput no longer scales with the number of running LLM calls. For instance, when mini-SWE-agent's request rate rises from 3.38 to 4.5 req/min, the number of running calls grows by 83% while output-token throughput increases by only 11%.

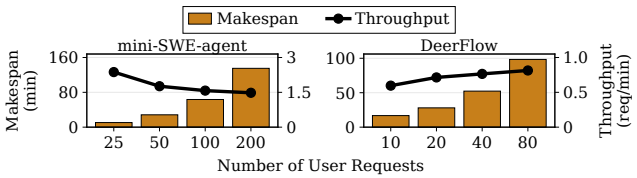


Figure 12. Offline batch inference.

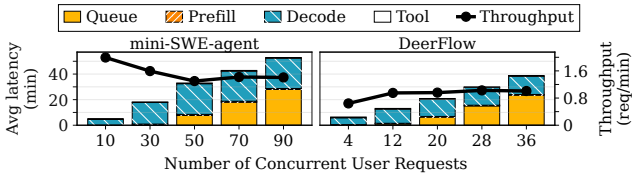


Figure 13. Fixed-concurrency serving.

GPU resource utilization. The prefix-cache thrashing is further evidenced by the GPU resource utilization. As shown in Figure 11, for applications that experience sharp drops in prefix-cache hit rate, GPU HBM bandwidth utilization decreases as the request rate approaches saturation. For example, increasing the request rate for CUGA from 14.4 to 16.8 requests per minute causes tensor core utilization to increase by 55%, while memory bandwidth utilization drops by 16%. During these periods, memory bandwidth utilization is lower than when the engine runs a full decode batch. Although tensor core utilization can continue to grow at request rates beyond the saturation point, this does not indicate improved serving efficiency; rather, it reflects wasted computation due to recomputation caused by prefix-cache thrashing.

- XPerf collects detailed metrics from the serving system and underlying hardware, enabling users to understand how their system behaves under diverse agentic workloads at different load levels.
- Agent prompts should be designed with prefix-cache sharing in mind, as prompt structure significantly affects serving performance.
- Under KV cache pressure, an agent's prefix cache may be evicted during tool calls, causing prefix-cache thrashing and reducing the hit rate. This incurs significant recomputation costs.

5.6 Support for Different Serving Scenarios

We have already demonstrated that XPerf can synthesize online serving workloads with different request rates. We now show that XPerf can evaluate serving system performance under two additional serving scenarios. For these experiments, we use Qwen3.6-35B-A3B.

Offline batch inference. In offline batch inference setups, throughput and total processing time (i.e., makespan) are the key performance metrics. As shown in Figure 12, mini-SWE-agent's throughput decreases by 38% as total requests increase from 25 to 200, primarily due to prefix-cache thrashing. DeerFlow shows the opposite trend: increasing throughput by 35% from 10 to 80 total requests. This increase has two causes.

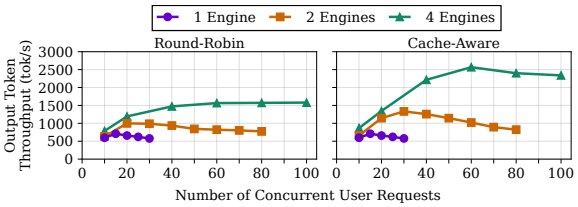


Figure 14. Decoding throughput as the number of serving engines scales, under different routing policies.

First, DeerFlow exposes almost no prefix-cache sharing and thus does not suffer from thrashing. Second, a larger request count spends proportionally less time in the final drain phase where only a few long requests remain, keeping the system at high concurrency for longer.

Fixed-concurrency serving. In fixed-concurrency setups, we maintain a fixed number of in-flight user requests for each application. We break down per-request latency to study the tradeoff between concurrency and serving quality. As shown in Figure 13, request latency is dominated by decoding at low concurrency, but the bottleneck shifts to queuing delay as concurrency rises. For throughput, mini-SWE-agent degrades by 35% as concurrency increases from 10 to 50, since higher concurrency causes prefix-cache thrashing. DeerFlow shows the opposite trend, with throughput rising from 0.64 to 0.95 req/min as concurrency increases from 4 to 36, since more concurrent requests increase the decode batch size and there is no prefix-cache to thrash.

5.7 Scalability to Multiple Serving Engines

XPerf can also benchmark serving systems with multiple engines across multiple nodes. To demonstrate this, we use Qwen3.6-35B-A3B with mini-SWE-agent to synthesize fixed-concurrency workloads. We scale the serving system from 1 to 4 engines, each holding one copy of the LLM, distributed across two of our testbed server machines. We compare the default round-robin and cache-aware routing policies provided by vLLM-Router. As shown in Figure 14, the serving system with round-robin policy scales poorly as we increase the number of serving engines, reaching only 1.41× and 2.26× the single-engine peak output token throughput at 2 and 4 engines, respectively. This is because with round-robin, the serving system cycles LLM calls for the same request across different engines, destroying prefix-cache locality. In contrast, the serving system with prefix-cache-aware policy reaches 1.88× and 3.61× improvements on output token throughput at the same engine count. The policy routes almost all LLM calls from the same request to a single engine, preserving cache reuse as the system grows. It scales decoding throughput more efficiently.

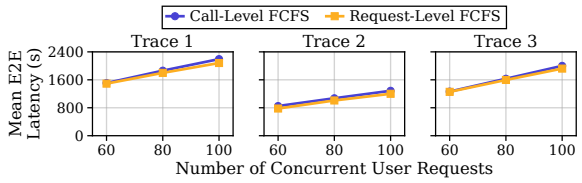


Figure 15. Average end-to-end latency of requests under various scheduling policies from replaying traces captured during different runs with the same random number seed.

- XPerf can scale to multi-engine serving systems to help users evaluate distributed LLM inference systems under agentic workloads.
- Realizing scale-out throughput gains depends critically on the LLM call routing policy, which must balance load across engines while exploiting prefix cache locality.

5.8 Debugging Serving Systems

To demonstrate how XPerf's trace replay help developers evaluate and debug serving system optimizations, we perform a case study on LLM call scheduling policies. We compare two scheduling policies, the call-level First-Come-First-Serve (FCFS) used by mainstream serving engines and the recently proposed request-level FCFS [1, 31], which prioritizes all LLM calls belonging to an earlier user request.

We collect three traces from different runs of the mini-SWE-agent under the same fixed random seed and replay each trace under both policies. As shown in Figure 15, with all three traces, request-level FCFS performs comparably to or slightly better (<8%) than call-level FCFS. However, the results vary substantially across three traces. Both scheduling policies achieve lower end-to-end latency (by 720 to 910 seconds under 100 concurrent requests) on trace 2 compared to traces 1 and 3. This is because trace 2 contains fewer long requests with only 4.5% of requests exceeding 40K total output tokens, versus 12% and 13.5% for traces 1 and 3.

Without XPerf, developers may be misled due to this run-to-run variation. If a developer happens to test call-level FCFS on the run producing trace 1 and request-level FCFS on the run producing trace 2, they would significantly overestimate the benefit of the request-level policy on this typical application. Even if they measured the call-level FCFS policy on trace 1 and the request-level FCFS policy on trace 3, the calculated benefit would be 12%. In contrast, XPerf's replay mechanism offers a fair comparison across different policies with the same trace, isolating the performance impact. They can also replay problematic runs multiple times to investigate the reason and debug the policy.

5.9 Overhead Analysis

We evaluate XPerf's overhead by measuring its resource consumption and its interference with running workloads. During trace recording, the OpenTelemetry tracer adds only 0.1% to end-to-end request latency and generates 5 KB/s of network traffic to export each request's traces over its lifetime. The tracing backend that aggregates these traces is configured with a 1 GB buffer for incoming traces and consumes, on average, under 5% of a CPU core and 2 GB of memory, writing 80 KB/s to disk. Logging input and output token IDs adds a further 250 KB/s. The trace size is 4.6 MB per request on average, with the largest requests from DeerFlow and ODR occupying up to 27.46 and 12.3 MB, respectively.

During replay, XPerf profiles detailed system and hardware metrics. We show the overhead by comparing the average decoding iteration time with and without profiling: interference is negligible, increasing per-iteration time by <0.1%. Since we must fetch the metrics from the GPU and persist them to disk, keeping this traffic small is essential to avoid contending with KV cache offloading. Even at a 20 ms sampling interval, the monitor uses only 16 KB/s of PCIe bandwidth, negligible relative to the link speed of 64 GB/s.

6 Discussion and Future Work

XPerf conducts benchmarking and performance profiling of LLM serving systems under agentic workloads. The host, which runs the application control logic and tool calls, falls outside this scope by design. The tools of different agentic applications vary in their interfaces and execution environments. Faithfully reproducing their execution is a distinct problem from benchmarking the serving system.

Replay remains important when the host is the object of study. Tool calls' arguments, duration, and execution environment vary across runs, and this variance propagates into any host-side measurement taken without a fixed reference trace. By enabling workload tracing, execution-graph construction, and faithful replay, XPerf lays the foundation that both serving-engine and host benchmarking require.

For future work, a natural extension is to capture tool-call arguments for tools with well-defined interfaces, such as RPC calls and shell commands dispatched to Docker, where the interface boundary is explicit and the arguments are cleanly serializable. Recording these arguments allow XPerf to replay tool calls with their original inputs, extending faithful reproduction from the serving engine outward to the host that runs the application.

7 Related Work

Performance Benchmarking for LLM Serving Systems. Existing works [21, 40, 42, 51] evaluate LLM serving systems on independent LLM calls or multi-turn conversations, reporting metrics such as TTFT, TPOT, and decoding throughput. More recently, AA-AgentPerf [6] measures serving performance with ReAct-style coding agent traces under varying numbers of concurrent sessions. In contrast, XPerf evaluates serving systems with diverse agentic applications, configurable load generation, detailed system-level metrics, and fully reproducible workloads.

System Optimization for Agentic Applications. Many studies have been conducted to optimize serving system performance for agentic applications. Parrot [34], Ayo [39], Kairos [8], and Agentix [36] optimize schedule policy to reduce user request latency. Regarding KV cache management, InferCept [1], Continuum [31], CONCUR [9], and ThunderAgent [22] reduce recomputation overhead by preventing unnecessary cache evictions during tool calls, whereas KVFlow [48], PBKV [72], and ScaleSim [49] proactively swap the cache to CPU memory based on predicted execution in multi-agent workflows. Pie [14] and AIOS [38] ease the deployment of such optimizations by making the serving system more programmable. XPerf enables reproducible benchmarking of these optimizations.

Framework for Developing Agentic Applications. Early frameworks such as LangChain [26], LangGraph [2], AutoGen [62], and MetaGPT [17] expose APIs for building agentic workflows, but require developers to hardcode the workflow structure. Recent frameworks [10, 32, 60, 71] go further by automating the generation of the agentic workflow. The diversity of these frameworks makes it time-consuming to collect agentic applications for benchmarking. XPerf addresses this with a unified execution graph abstraction for collecting benchmarking traces, requiring only minor code annotations to the application.

8 Conclusion

We develop XPerf, a benchmarking framework that can evaluate LLM serving systems under diverse agentic workloads. XPerf provides an end-to-end pipeline to collect, synthesize, and replay agentic workloads with minimal effort required from users. Our experiments demonstrate that XPerf can accurately replay agentic workloads while capturing detailed system metrics at low overhead.

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 *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 3, 15 pages.

[2] LangChain AI. 2024. *LangGraph: A low-level orchestration framework for building stateful agents.* https://github.com/langchain-ai/langgraph Accessed: 2026-05-24.

[3] Ansible Project Contributors. 2026. Introduction to Ansible. https://docs.ansible.com/projects/ansible/latest/getting_started/introduction.html. Accessed: 2026-06-09.

[4] Anthropic. 2025. Claude Code. https://www.claude.com/product/claude-code. Accessed: 2025-10-19.

[5] Anthropic. 2026. Python SDK. https://platform.claude.com/docs/en/api/sdks/python. Claude API Documentation. Accessed: 2026-06-05.

[6] Artificial Analysis. 2026. AA-AgentPerf Methodology. https://artificialanalysis.ai/methodology/agentperf. Accessed: 26 May 2026.

[7] ByteDance, Daniel Walnut, and Henry Li. 2025. DeerFlow: Deep Exploration and Efficient Research Flow. https://github.com/bytedance/deerflow. Open-source long-horizon super agent harness. Accessed: 2026-05-28.

[8] Jinyuan Chen, Jiuchen Shi, Quan Chen, and Minyi Guo. 2025. Kairos: Low-latency Multi-Agent Serving with Shared LLMs and Excessive Loads in the Public Cloud. arXiv:2508.06948 [cs.DC] https://arxiv.org/abs/2508.06948

[9] Qiaoling Chen, Zhisheng Ye, Tian Tang, Peng Sun, Boyu Tian, Guoteng Wang, Shenggui Li, Yonggang Wen, Zhenhua Han, and Tianwei Zhang. 2026. CONCUR: High-Throughput Agentic Batch Inference of LLM via Congestion-Based Concurrency Control. arXiv:2601.22705 [cs.DC] https://arxiv.org/abs/2601.22705

[10] crewAI, Inc. 2023. crewAI: Framework for orchestrating role-playing, autonomous AI agents. https://github.com/crewAIInc/crewAI. Accessed: 2026-06-02.

[11] Aaron Erickson. 2024. Optimizing Data Center Performance with AI Agents and the OODA Loop Strategy. NVIDIA Technical Blog. https://developer.nvidia.com/blog/optimizing-data-center-performance-with-ai-agents-and-the-ooda-loop-strategy/ Accessed: 2026-05-23.

[12] Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang, Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. 2024. Magentic-One: A Generalist Multi-Agent System for Solving Complex Tasks. arXiv:2411.04468 [cs.AI] https://arxiv.org/abs/2411.04468

[13] Ali Essam Ghareeb, Benjamin Chang, Ludovico Mitchener, Angela Yiu, Caralyn J Szostkiewicz, Dmytro Shved, Gavin J Gyimesi, Jon M Laurent, Samantha M Wright, Muhammed T Razzak, et al. 2026. A multi-agent system for automating scientific discovery. *Nature* (2026), 1–3.

[14] In Gim, Zhiyao Ma, Seung-seob Lee, and Lin Zhong. 2025. Pie: A Programmable Serving System for Emerging LLM Applications. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 415–430. doi:10.1145/3731569.3764814

[15] gRPC Authors. 2026. gRPC. https://grpc.io/. Accessed: 2026-06-09.

[16] Horace He and Thinking Machines Lab. 2025. Defeating Nondeterminism in LLM Inference. *Thinking Machines Lab: Connectionism* (2025). doi:10.64434/tml.20250910 https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

[17] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, Vol. 2024. 23247–23275.

[18] Jaeger Authors. 2026. Jaeger: Open Source, Distributed Tracing Platform. https://www.jaegertracing.io/. Accessed: 2026-06-09.

[19] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[20] Michael B. Jones. 1993. Interposition agents: transparently interposing user code at the system interface. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, USA) *(SOSP '93)*. Association for Computing Machinery, New York, NY, USA, 80–93. doi:10.1145/168619.168626

[21] Waleed Kadous, Kyle Huang, Wendi Ding, Liguang Xie, Avnish Narayan, and Ricky Xu. 2023. Reproducible Performance Metrics for LLM inference. *Anyscale Blog* (November 2023). https://www.anyscale.com/blog/reproducible-performance-metrics-for-llm-inference Accessed: 2025-10-20.

[22] Hao Kang, Ziyang Li, Xinyu Yang, Weili Xu, 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:2602.13692 [cs.OS] https://arxiv.org/abs/2602.13692

[23] Sayash Kapoor, Benedikt Stroebl, Peter Kirgis, Nitya Nadgir, Zachary S Siegel, Boyi Wei, Tianci Xue, Ziru Chen, et al. 2025. Holistic Agent Leaderboard: The Missing Infrastructure for AI Agent Evaluation. *arXiv preprint arXiv:2510.11977* (2025).

[24] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael Mahoney, Kurt Keutzer, and Amir Gholami. 2024. An LLM Compiler for Parallel Function Calling. In *Forty-First International Conference on Machine Learning*. https://github.com/SqueezeAILab/LLMCompiler

[25] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[26] LangChain. 2023. LangChain Classic API Reference: SequentialChain. https://reference.langchain.com/python/langchain-classic/chains/sequential/SequentialChain. Accessed: 2026-04-15.

[27] LangChain. 2025. *Open Deep Research*. https://github.com/langchain-ai/open_deep_research

[28] Laude Institute and Stanford. 2025. Harbor: A Framework for Running Agent Evaluations and RL Environments. https://github.com/harbor-framework/harbor. Accessed: 2026-05-25.

[29] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. {GS}hard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *International Conference on Learning Representations*. https://openreview.net/forum?id=qrwe7XHTmYb

[30] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*. PMLR, 19274–19286.

[31] Hanchen Li, Qiuyang Mang, Runyuan He, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. 2025. Efficient and Robust Multi-Turn LLM Agent Scheduling with KV Cache Time-to-Live. *arXiv preprint arXiv:2511.02230* (2025).

[32] Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and Yongfeng Zhang. 2024. Autoflow: Automated workflow generation for large language model agents. *arXiv preprint arXiv:2407.12821* (2024).

[33] Guannan Liang and Qianqian Tong. 2025. LLM-Powered AI Agent Systems and Their Applications in Industry. arXiv:2505.16120 [cs.AI] https://arxiv.org/abs/2505.16120

[34] 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 *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 929–945. https://www.usenix.org/conference/osdi24/presentation/lin-chaofan

[35] llm-d Community. 2025. llm-d: Kubernetes-native Distributed Inference at Scale. https://github.com/llm-d/llm-d. Accessed: 2026-05-25.

[36] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. 2026. Agentix: An Efficient Serving Engine for LLM Agents as General Programs. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 2443–2459. https://www.usenix.org/conference/nsdi26/presentation/luo

[37] Sami Marreed, Alon Oved, Avi Yaeli, Segev Shlomov, Ido Levy, Offer Akrabi, Aviad Sela, Asaf Adi, and Nir Mashkif. 2025. Towards Enterprise-Ready Computer Using Generalist Agent. arXiv:2503.01861 [cs.DC] https://arxiv.org/abs/2503.01861

[38] Kai Mei, Xi Zhu, Wujiang Xu, Mingyu Jin, Wenyue Hua, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. 2025. AIOS: LLM Agent Operating System. In *Second Conference on Language Modeling*. https://openreview.net/forum?id=L4HHkCDz2x

[39] NetX-lab et al. 2025. Towards End-to-End Optimization of LLM-based Applications with Ayo. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[40] NVIDIA. 2024. GenAI-Perf. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/perf_analyzer/genai-perf/README.html. Accessed: 2025-10-20.

[41] NVIDIA. 2025. Dynamo: A Datacenter Scale Distributed Inference Serving Framework. https://github.com/ai-dynamo/dynamo. Accessed: 2026-05-25.

[42] NVIDIA. 2026. AIPerf: A comprehensive benchmarking tool that measures the performance of generative AI models. https://github.com/ai-dynamo/aiperf. GitHub repository.

[43] NVIDIA Corporation. 2026. CUPTI Documentation. https://docs.nvidia.com/cupti/. Accessed: 2026-06-09.

[44] OpenAI. 2024. OpenAI API. https://openai.com/api/ Accessed: 2024-05-25.

[45] OpenAI. 2025. gpt-oss-120b & gpt-oss-20b Model Card. arXiv:2508.10925 [cs.CL] https://arxiv.org/abs/2508.10925

[46] OpenTelemetry Authors. 2026. OpenTelemetry Protocol Specification. https://opentelemetry.io/docs/specs/otlp/. Accessed: 2026-06-09.

[47] Melissa Z. Pan, Negar Arabzadeh, Riccardo Cogo, Yuxuan Zhu, Alexander Xiong, Lakshya A Agrawal, Huanzhi Mao, Emma Shen, Sid Pallerla, Liana Patel, Shu Liu, Tianneng Shi, Xiaoyuan Liu, Jared Quincy Davis, Emmanuele Lacavalla, Alessandro Basile, Shuyi Yang, Paul Castro, Daniel Kang, Joseph E. Gonzalez, Koushik Sen, Dawn Song, Ion Stoica, Matei Zaharia, and Marquita Ellis. 2026. Measuring Agents in Production. arXiv:2512.04123 [cs.CY] https://arxiv.org/abs/2512.04123

[48] Zaifeng Pan, Ajjkumar Patel, Yipeng Shen, Zhengding Hu, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. 2025. KVFlow: Efficient Prefix Caching for Accelerating LLM-Based Multi-Agent Workflows. In *Advances in Neural Information Processing Systems*. https://openreview.net/forum?id=5Iw1nDtYmT

[49] Zaifeng Pan, Yipeng Shen, Zhengding Hu, Zhuang Wang, Aninda Manocha, Zheng Wang, Zhongkai Yu, Yue Guan, and Yufei Ding. 2026. ScaleSim: Serving Large-Scale Multi-Agent Simulation with Invocation Distance-Based Memory Management. *arXiv preprint arXiv:2601.21473* (2026).

[50] Qwen Team. 2026. Qwen3.6-35B-A3B: Agentic Coding Power, Now Open to All. https://qwen.ai/blog?id=qwen3.6-35b-a3b

[51] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, et al. 2019. MLPerf Inference Benchmark. *arXiv preprint arXiv:1911.02549* (2019).

[52] Corby Rosset, Ho-Lam Chung, Guanghui Qin, Ethan C. Chau, Zhuo Feng, Ahmed Awadallah, Jennifer Neville, and Nikhil Rao. 2024. Researchy Questions: A Dataset of Multi-Perspective, Decompositional Questions for LLM Web Agents. arXiv:2402.17896 [cs.CL]

[53] Samuel Schmidgall and Michael Moor. 2025. AgentRxiv: Towards Collaborative Autonomous Research. arXiv:2503.18102 [cs.AI] https://arxiv.org/abs/2503.18102

[54] SemiAnalysis. 2026. InferenceX by SemiAnalysis: Open Source AI Inference Benchmark. https://inferencex.semianalysis.com/. Accessed: 26 May 2026.

[55] SGLang Team. 2024. SGLang Router (Model Gateway): Rust-based Data-parallel Routing for SGLang. https://github.com/sgl-project/sglang/tree/main/sgl-router. Accessed: 2026-05-25.

[56] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*. PMLR, 31094–31116.

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

[58] The OpenTelemetry Authors. 2026. OpenTelemetry. https://opentelemetry.io. Accessed: 2026-06-04.

[59] vLLM Project. 2025. vLLM Router: A High-Performance and Light-weight Router for vLLM Large-scale Deployment. https://github.com/vllm-project/router. Accessed: 2026-05-25.

[60] Yanbo Wang, Zixiang Xu, Yue Huang, Xiangqi Wang, Zirui Song, Lang Gao, Chenxi Wang, Xiangru Tang, Yue Zhao, Arman Cohan, et al. 2025. DyFlow: Dynamic Workflow Framework for Agentic Reasoning. In *Advances in Neural Information Processing Systems*.

[61] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1800, 14 pages.

[62] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. 2024. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework. In *COLM*.

[63] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://arxiv.org/abs/2405.15793

[64] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 conference on empirical methods in natural language processing*. 2369–2380.

[65] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. arXiv:2406.12045 [cs.AI] https://arxiv.org/abs/2406.12045

[66] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*.

[67] Ori Yoran, Samuel Joseph Amouyal, Chaitanya Malaviya, Ben Bogin, Ofir Press, and Jonathan Berant. 2024. AssistantBench: Can Web Agents Solve Realistic and Time-Consuming Tasks? arXiv:2407.15711 [cs.CL] https://arxiv.org/abs/2407.15711

[68] Xiao Yu, Baolin Peng, Vineeth Vajipey, Hao Cheng, Michel Galley, Jianfeng Gao, and Zhou Yu. 2025. Exact: Teaching ai agents to explore with reflective-mcts and exploratory learning. In *International Conference on Learning Representations*, Vol. 2025. 65157–65184.

[69] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, et al. 2025. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 23078–23097.

[70] Jiayi Yuan, Hao Li, Xinheng Ding, Wenya Xie, Yu-Jhe Li, Wentian Zhao, Kun Wan, Jing Shi, Xia Hu, and Zirui Liu. 2026. Understanding and mitigating numerical sources of nondeterminism in llm inference. *Advances in Neural Information Processing Systems* 38 (2026), 169819–169851.

[71] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. 2025. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, Vol. 2025. 34040–34077.

[72] Haoyu Zheng, Fangcheng Fu, Jia Wu, Binhang Yuan, Yongqiang Zhang, Hao Wang, Yuanyuan Zhu, Xiao Yan, and Jiawei Jiang. 2026. Efficient Serving for Dynamic Agent Workflows with Prediction-based KV-Cache Management. *arXiv preprint arXiv:2605.06472* (2026).

[73] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406* (2023).