From Requests to Sessions: A Large-Scale Characterization of Human-Driven Agentic Workloads

William Nixon
University of Chicago & Harvard University

Muxin Tian
University of Toronto

Yunjia Zheng
Harvard University

Haryadi S. Gunawi
University of Chicago

Juncheng Yang
Harvard University

Abstract

Coding agents are becoming an important source of LLM inference traffic, yet their workload behavior remains poorly understood. Existing workload studies largely focus on general LLM traffic, providing limited visibility into the characteristics of agentic workloads. To bridge this gap, we study 3.7 months of production traffic from *FreeInference*, spanning 209B input tokens and 12K sessions across 13 agent harnesses, revealing agentic workload characteristics that are not captured by request-centric views of LLM serving.

Our analysis shows that coding-agent sessions are long-lived and bursty, with tight LLM–tool loops separated by longer user idle periods. Contexts grow substantially while outputs remain short, causing cache reads to dominate token volume and API cost. Tool execution dominates active-session latency, yet tool parallelism remains uncommon. Prefix reuse is high but disrupted by user idle gaps and context mutations performed by agent harnesses. A 15-minute KV-cache retention time captures most practical reuse, while longer retention provides diminishing returns at substantially higher storage cost. We will release the trace alongside the paper to enable further research on agentic workloads and serving systems.

1 Introduction

Coding agents are becoming an important workload for large language model (LLM) serving systems. Advances in reasoning and tool use have enabled agents such as Codex [36], Claude Code [5], Kilo [18], and OpenCode [38] to handle increasingly complex software-engineering tasks by interacting with external environments. Developers are adopting these agents at an increasing rate as they become more capable and dependable in day-to-day software development. As their usage grows, serving these workloads efficiently is becoming increasingly important.

Efficient serving depends on understanding the structure of the workload being served. Prior workload studies have characterized production LLM traffic to guide caching, scheduling, routing, and resource management [35, 53, 55, 58]. These studies largely focus on general or chat-oriented workloads, providing limited visibility into the session-level structure of agentic workloads. Consequently, we still have a limited understanding of how agentic workloads behave in production and how their characteristics should inform the design of efficient serving systems.

Table 1: Statistics of the *FreeInference* trace. We focus on coding-agent sessions and filter out single-request API traffic and benchmark replay, except for one SWE-Bench replay used for comparison.

Trace Summary			
Time span	3.7 months	Sessions	12,002
Requests	1,186,582	Tool calls	1,213,347
Input tokens	209.2 B	Output tokens	552.5 M
Cached tokens	197.1 B	Agent harnesses	13 (+1)

Sessions and Input Tokens by Harness

Harness	Sess.	Input Tokens	Harness	Sess.	Input Tokens
Hermes	3,953	43.1B	OpenCode	1,733	44.2B
Pi	1,499	61.2B	SWE-Bench	1,035	1.8B
Kilo	797	8.5B	Codex	763	6.1B
Oh-My-Pi	719	17.4B	Claude Code	639	8.9B
Copilot	295	5.8B	Zcode	197	9.4B
OpenClaw	189	1.0B	Cline	86	1.3B
Zoo	75	0.4B	OpenHands	22	0.1B

Recent work has begun to characterize agentic workloads using benchmarks such as SWE-Bench, Terminal-Bench, and GAIA [16, 33, 34]. These benchmarks provide useful insight into agent behavior, but they are designed primarily to run predefined tasks in controlled settings. It therefore remains unclear whether workload characteristics observed in these benchmarks reflect how coding agents behave in production, or whether serving-system evaluations based on them generalize to real-world workloads.

In this work, we aim to bridge this gap with a characterization of production coding-agent workloads using a 3.7-month provider-side trace from *FreeInference*. The trace contains 12K sessions, 1.2M requests, 209B input tokens, and 1.2M tool calls across thirteen agent harnesses. This trace provides a production-scale view of coding-agent sessions, allowing us to study both their workload characteristics and implications for LLM serving. We highlight several findings:

- **Cache reads dominate session cost.** As a session progresses, the reusable prefix grows to hundreds of thousands of tokens, while each step contributes relatively few new input and output tokens. Among requests with reported cache usage, cached input accounts for 97% of token volume and 73% of API cost.

Table 2: Comparison with prior interaction datasets and production LLM workload studies. We open-source the largest production agentic workload trace with block-level and tool information to date.

Work	Span	Tokens	Sessions	Type	Harnesses	KV	Unsampled Release
ServeGen [58]	4 mo.	–	–	Serving	–	–	–
BurstGPT [55]	4 mo.	2.18B	–	Serving	–	–	✓
Qwen-Bailian [53]	2 h	–	–	Serving	–	✓	–
Mooncake [43]	1 h	356M	–	Serving	–	✓	–
Chutes [35]	12 mo.	38.3T	–	Serving	–	–	✓
WildChat [72]	13 mo.	1.87B	1.0M	Chat	–	–	✓
LMSYS-Chat [73]	4 mo.	577M	1.0M	Chat	–	–	–
SWE-chat [7]	3 mo.	–	6.0K	Agent	Multi	–	✓
TraceLab [78]	8 mo.	55.1B	4.3K	Agent	2	–	✓
CacheWise [50]	–	10M	94	Agent	1	–	✓
Copilot [27]	1 wk	95T	13.5M	Agent	1	–	–
Ours	**3.7 mo.**	**209.2B**	**12.0K**	**Agent**	**13 (+1)**	✓	✓

- **Tool execution dominates active-session latency.** In a typical session, tool calls take longer than either prefill or decoding. Halving tool latency yields a 1.38× speedup, far exceeding the 1.09× and 1.16× gains from halving prefill and decode time, respectively.
- **Tool parallelism is underused, while synchronous subagents suffer from stragglers.** Parallel tool execution reduces repeated LLM–tool round trips, yet 84.5% of requests issue only one tool call. Subagents are used in only 10.4% of sessions and are mostly synchronous, with stragglers that can take over 5× longer than the fastest sibling.
- **Context mutations are widespread and drive substantial recomputation, yet are necessary to control serving cost.** Although only 4.0% of requests modify prior context, mutations occur in 56% of sessions, challenging the common assumption that agent context grows monotonically. Even with infinite cache retention, mutations account for 62% of newly prefilled tokens, making them a major source of recomputation. Counterintuitively, eliminating context-reduction mutations does not generally reduce prefill or cost. Instead, it increases context length, cache-read volume, and the penalty of cache misses. Under a 15 minute cache retention time, eliminating context-reducing mutations can increase prefill and API cost by up to 5%.
- **A 15-minute rule for prefix caching: balancing recomputation and storage cost.** A 15-minute retention time captures much of the available reuse while keeping provider-side storage cost low. Capturing nearly all reuse requires a day of retention, but incurs 25.6× higher storage overhead. Harness-provided session-level signals can help reclaim dead blocks and make longer retention practical.
- **Benchmark replays are different from human-driven workloads.** SWE-Bench runs a fixed task based on a long, detailed input, while humans send many short prompts to agents. Benchmark replays produce sessions that are 9.4× shorter in duration and 5.1× smaller in context than human-driven sessions, and require only 5 minutes of retention for ideal prefix reuse versus about one day in production.

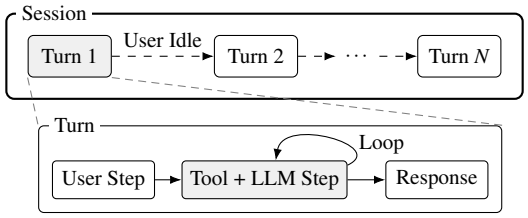


Figure 1: Coding-agent execution hierarchy. A session is a sequence of user turns separated by user idle time. Each turn begins with a user step, and continues through a repeated LLM–tool loop.

These findings show that production coding-agent workloads exhibit distinct characteristics that are not captured by request-level analyses of LLM serving. Our analysis characterizes these behaviors across diverse production coding agents and motivates treating sessions as the unit for designing and evaluating efficient agent-serving systems. To support further research, we will release the trace[1] alongside our analysis.

2 Background

2.1 Coding Agents

Coding agents use LLMs to carry out software-engineering tasks through repeated interaction with tools and their environments [54, 64]. A user typically begins by providing a natural-language task, such as implementing a feature. The agent then executes a ReAct-style loop [67]. In each iteration, it sends the current context to the model, invokes the requested tools, incorporates their results into the context, and issues another model request. This loop is managed by an agent *harness*. The harness constructs each model request, executes tool calls, and feeds their results back into the context for the next request. A single user task can therefore expand into many model requests and tool calls. For work that spans multiple context windows, harnesses also manage compaction and persistent progress records [3, 4]. We describe this execution at three levels:

- **Session.** The highest-level unit, spanning the user's end-to-end interaction with the agent. A session contains one or more turns, accumulating context across its lifetime.
- **Turn.** One user-initiated interaction within a session. A turn begins with a user prompt, after which the agent enters an autonomous LLM–tool loop until it produces a final response and returns control to the user.
- **Step.** One model request within a turn. The first step is triggered by the user prompt. Subsequent steps are triggered by tool results and continue the LLM–tool loop until the final step returns a response and ends the turn.

2.2 LLM Workload Traces

Production traces provide realistic traffic for designing and evaluating LLM serving systems. Prior work has extensively

[1] We aim to release a trace containing user prompts and LLM responses upon approval. Otherwise, we will release an anonymized trace with hashed token blocks, tool-call information, and rich metadata.

characterized request-level traffic and multi-turn conversations [35, 41, 43, 44, 46, 53, 55, 58], but fewer studies focus on coding-agent sessions. Evaluations of agent-serving systems often use benchmark-driven workloads such as SWE-Bench, Terminal-Bench, and τ-bench [16, 33, 66]. Recent characterization studies replay such benchmarks to study requests, tool execution, context growth, and serving behavior [8, 19, 20, 70]. However, these evaluations typically run agents on controlled and isolated tasks, providing a partial view of human-driven agentic workloads.

Recent trace studies have begun to examine production coding-agent sessions. TraceLab [78], CacheWise [50], and Copilot [27] collect client-side logs from Claude Code, Codex, and Copilot, while SWE-chat [7] collects sessions shared by developers. Table 2 compares these datasets. Existing studies mainly provide client-side traces, making ours the largest provider-side characterization of human-driven agentic workloads with tool-call and block-level metadata to date.

3 Dataset and Methodology

We analyze 3.7 months of traffic from *FreeInference*, a public LLM inference service.[2] *FreeInference* serves 12 models through both local compute and remote APIs, processing more than 3 million requests and 500 billion tokens per month. Users worldwide run over 100 different agents on the service.

In this study, we focus on the most popular human-driven production agents. Table 1 summarizes the resulting dataset, which contains 12K sessions and 1.2M requests across thirteen agent harnesses. We identify the agent harness from the user-agent field when available and from the system prompt otherwise. All analyses exclude SWE-Bench unless otherwise stated. Table 3 summarizes the fields available in the trace. For each request, we record the input and output text along with their associated tool calls. We also log metadata including the timestamp, model, and anonymized user identifier, together with serving measurements such as input, output, and cached token counts, completion latency, and time to first token (TTFT). For requests forwarded externally, we use the reported latency measurements. As a result, metrics such as TTFT may include internal buffering or queueing delays.

Session Reconstruction. The inference API is stateless, and explicit session identifiers are not sent by most harnesses. We therefore reconstruct sessions from request histories. A natural approach is strict prefix matching, where a request is linked to a prior request only if it preserves the full previous input as a prefix. However, this approach splits sessions too aggressively whenever harnesses modify earlier context and break prefixes. We instead reconstruct sessions using replayed LLM outputs. Although harnesses may modify earlier context, a later request typically retains the previous LLM output in its input history. We therefore link requests by matching the earlier output against the later request's history. For tool-calling responses, we link requests using the tool-call ID. Otherwise, we match the assistant message to its replay in a later request.

[2]We analyze only data from users who explicitly consented to research data collection and sharing before using the service.

Table 3: Key fields available in the *FreeInference* production trace.

Field	Description
`request_id`	Request IDs
`user_id`	Anonymized user IDs
`model_id`	Model used
`harness`	Harness inferred from user-agent or system prompt
`session_id`	Reconstructed session ID
`messages`	Prompt and response text
`timestamps`	Request timing and latency
`token_counts`	Input, output, and cached tokens
`tool_calls`	Tool-call metadata and results

We validate the reconstructed sessions on harnesses that provide explicit session identifiers. In this subset, 95% of reconstructed sessions match the harness-provided sessions one-to-one. Most mismatches occur when a harness regenerates its session identifier after inactivity, splitting one logical conversation across multiple sessions. To preserve reuse across long idle periods, we do not impose an inactivity cutoff during reconstruction; such cutoffs can instead be applied during analysis if desired.

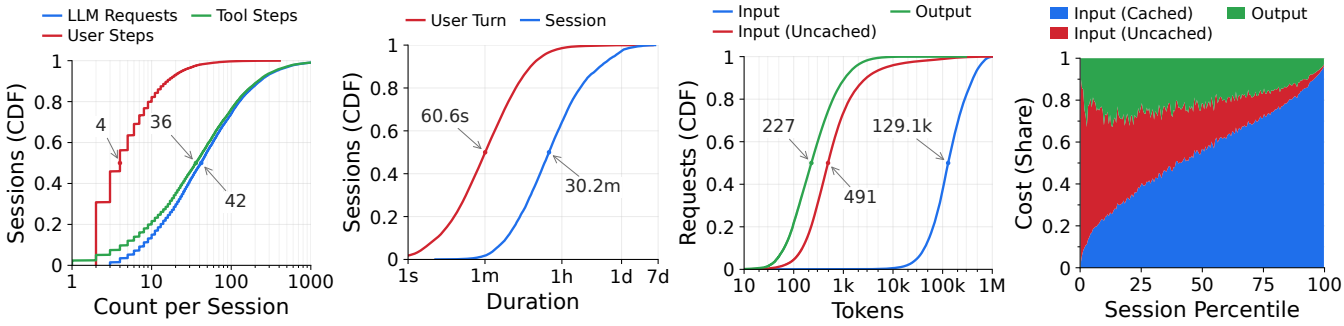
4 Sessions

We first study coding-agent sessions in terms of turns and the steps within them. We then examine how context grows, where time is spent, and how users prompt agents across turns.

4.1 Session Structure

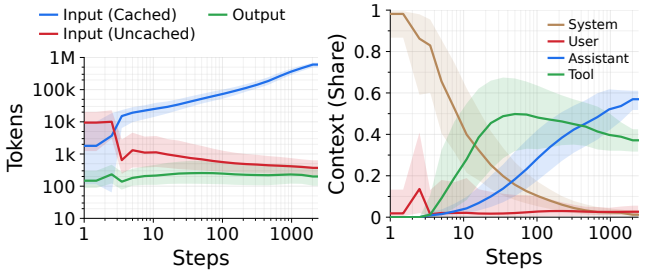
Sessions are mostly tool and LLM-driven. Figure 2a shows that sessions contain only a few user steps but tens to hundreds of steps in total, with some sessions reaching thousands. Sessions have a median of 4 user steps, but 36 tool and 42 LLM steps per session. LLM and tool steps closely track each other at nearly a one-to-one ratio, showing that most activity within a session is driven by the LLM–tool loop.

Sessions last for hours, while each user turn runs for minutes. A session consists of multiple user turns, each beginning with a user prompt followed by an LLM–tool loop. Figure 2b compares the duration of individual user turns with the sessions that contain them. The median user turn lasts about one minute, while the median session lasts 30 minutes, roughly 30$\times$ longer. The longest 10% of sessions exceed 7.7 hours. This gap is driven primarily by idle time between turns rather than active execution. Once a turn completes, users may spend substantial time thinking, typing, or stepping away before issuing the next request. Session duration is therefore made up of two distinct components: fast LLM–tool execution loops within turns and longer idle gaps between *multiple user turns*.



(a) Counts per session **(b)** Turn and session duration **(c)** Tokens per request **(d)** Token cost share by session

Figure 2: (a) Sessions contain a few user steps but tens to hundreds of LLM requests, most of them triggered by tools. (b) A single user turn lasts from a few seconds to more than 24 hours, while a typical session lasts from a few minutes to almost one week. (c) Requests carry long inputs, with most tokens being cached. (d) The cost of most sessions is dominated by prefix-cache hits.



(a) Token growth by step **(b)** Context share by step

Figure 3: Context composition changes as a session progresses. Shaded areas show the P25–P75 ranges. (a) Cached tokens accumulate rapidly across steps, while uncached input and output remain small. (b) Earlier turns are dominated by the system prompt, middle turns are dominated by tool results, and later turns are dominated by LLM responses.

4.2 Tokens and Cost

Fresh prefill and output per request are small; API cost is dominated by prefix-cache hits. Figure 2c shows that requests carry large contexts but add relatively few new tokens. The median request contains roughly 129K input tokens, yet only about 500 are uncached, roughly 260× fewer than the total input. Outputs are small at about 200 tokens per request. Thus, agentic workloads are less prefill-heavy than their raw input sizes suggest. Figure 2d shows the consequence at the workload level using Claude Fable 5 pricing [6], where cached input is priced at $1/M tokens, uncached input at $10/M, and output at $50/M. Token and cost shares use requests with reported cache usage. Cached input accounts for 97.4% of all tokens and 72.9% of API cost, while uncached input and output make up only 17.4% and 9.7% of cost respectively. Thus, although cached tokens cost 1/10$^{\text{th}}$ of the input price, the high volume of reused input in agentic workloads makes prefix cache hits a substantial component of total API cost.

Prefill concentrates in the first few steps. Figure 3a shows that requests carry the most uncached input near the beginning of a session, while deeper requests increasingly reuse an already cached prefix. Median uncached input is about 10K tokens in the first few steps due to the long system prompt and repository exploration (Section 7), but falls to around 1K and continues declining. Meanwhile, the cached prefix grows from a few thousand tokens to hundreds of thousands, but sublinearly: it adds about 2.5K tokens per step over the first ten steps, and only about 330 per step after a hundred. Output remains roughly constant at about 200 tokens per request. Early requests are therefore more prefill-heavy, whereas later requests are more cache-heavy.

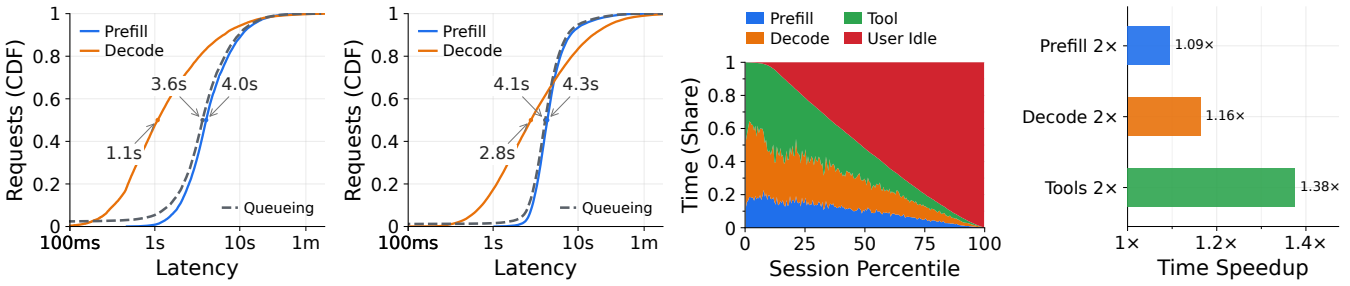
Tool history peaks and is pruned, while assistant history continues to grow. Figure 3b shows that context composition shifts substantially over the course of a session. Early steps are dominated by the system prompt and initial user input. Tool results then accumulate rapidly: assistant and tool history together grow from 32% of context tokens at 10 steps to 92% at 1K steps. However in longer sessions tool history peaks and subsequently declines, whereas assistant history continues to grow and eventually becomes the largest component. This divergence reflects a common harness policy: pruning older tool outputs while retaining assistant messages, including their tool calls.

Finding 1: Agentic requests repeatedly reuse large and growing contexts while adding short new input and output tokens. Prefix-cache reads dominate coding-agent session costs.

Serving systems should optimize the cost of serving prefix-cache hits. **Harnesses** should aim to reduce the number of steps rather than input per turn; **Users** should pay more attention to cache-read pricing.

4.3 LLM and Tool Latency

Request latency is TTFT-bound, largely due to serving overheads. Figures 4a and 4b break down TTFT and decode time for two externally routed models. We estimate pure prefill time from each model's throughput on cold long-context misses and subtract it from TTFT. The remaining time, la-



(a) Minimax-M2.7 **(b)** Kimi-K2.7-Code **(c)** Session time breakdown **(d)** What-if analysis

Figure 4: Request and session latency have different bottlenecks. Request latency is strongly shaped by TTFT serving overheads. (a, b) For externally routed models like Minimax-M2.7 and Kimi-K2.7-Code, TTFT is dominated by these overheads rather than pure prefill. (c) At the session level, tool execution and user idle time account for much of the session time. (d) With user idle time excluded and each tool gap capped at 15 minutes, accelerating tools yields larger gains than accelerating prefill or decode.

Table 4: Example commands in each shell category, most frequent first. ∗ denotes a shell-command category, as in Table 5.

Category	Example commands
∗Interpreter	`python`, `code`, `node`, `npm run`, `php`
∗Search	`grep`, `sed`, `rg`, `find`, `Select-String`, `awk`, `fd`, `jq`
∗File Ops	`ls`, `rm`, `ps`, `cp`, `mkdir`, `Get-Content`, `Get-ChildItem`
∗Git/VCS	`git diff`, `git log`, `git status`, `git commit`, `git show`
∗Network	`ssh`, `curl`, `netstat`, `ss`, `scp`, `Invoke-WebRequest`
∗Wait/Poll	`sleep`, `nohup`
∗Build/Test	`pytest`, `cargo test`, `npx tsc`, `bun test`, `dotnet build`
∗Packages	`pixi`, `pip`, `cargo fmt`, `cargo clippy`, `uv pip`

beled queueing, includes both queueing and other serving overheads. Both models process prefill at roughly 8K tokens/s, while the median request contains fewer than 1K uncached tokens, making pure prefill typically less than 0.1 s. As a result, serving overheads account for most of the roughly 4 s median TTFT. Requests generate only hundreds of output tokens, keeping median decode time to roughly 1–3 s. Thus, serving overheads dominate request latency for these models. Although we show only two models, we observe the same pattern across all externally routed models.

Most time is spent outside the LLM. Figure 4c breaks session wall-clock time into prefill, decode, tool execution, and user idle time. We define user idle time as the time between the final response of one turn and the next user prompt. On average, user idle time accounts for 50.3% of session time, followed by tool execution at 21.7%, decode at 18.1%, and prefill at 9.9%. Decode dominates prefill because most requests are served by locally hosted models with little queueing and high prefix reuse. Together, tool execution and user idle time account for 72.0% of session time. In Section 6.1 we will show that user idle time is the biggest source of cache misses.

Tool execution is the new bottleneck. While substantial effort has been devoted to improving LLM inference throughput, tool execution has emerged as the dominant bottleneck in agentic systems. Figure 4d quantifies the effect of independently doubling prefill, decode, and tool-execution throughput on end-to-end latency. With tool gaps capped at 15 minutes, doubling tool-execution throughput provides a 1.38× speedup, compared with speedups of only 1.09× and 1.16× from doubling prefill and decode throughput, respectively.

> **Finding 2:** Serving overheads such as queueing dominate per-request TTFT, whereas tool execution dominates active session time. User idle time dominates end-to-end session duration.
>
> **Serving systems** should jointly optimize routing and load balancing to preserve prefix reuse while minimizing queueing delays. **Hardware designers** should optimize for emerging tool-execution workloads. To shorten session duration, **harnesses** should select lower-latency tools, minimize the need for user intervention, and potentially request user clarification promptly and asynchronously.

5 Tools

Tool execution accounts for a substantial share of active session time, as agents autonomously invoke tools throughout each turn. In this section, we examine how agents use tools. Agents have two types of tools: (1) native tools provided by the harness and (2) a bash tool that performs more diverse functions. We normalize tool names across agents and bash by intended function. For compound commands, we use the final command in the chain; for pipelines, we use the first command. Table 5 lists the resulting categories, and Table 4 gives representative shell commands for each category.

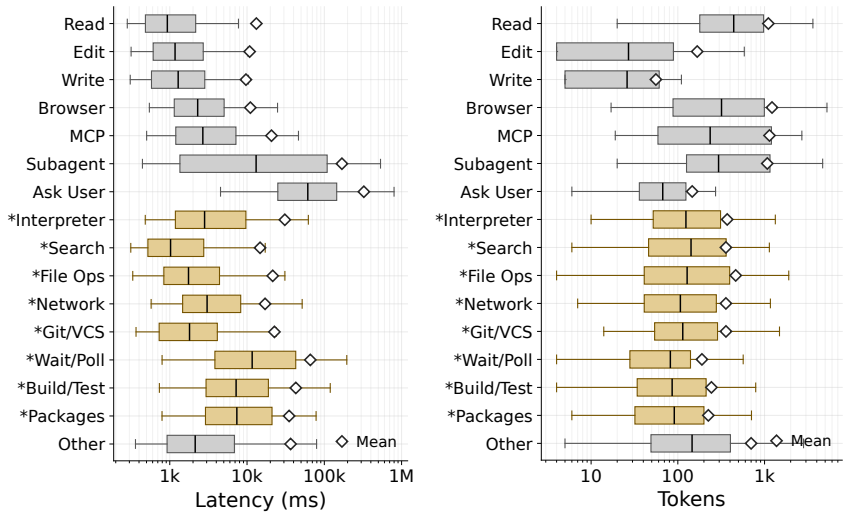
We measure the *effective latency* of tools as the elapsed time between when the LLM finishes a response and the tool result returns (excluding the estimated network round-trip time).

5.1 Tool Overview

Tool usage is concentrated in a small set of broadly used operations. Native tools account for 47% of all tool calls, while shell commands account for the remaining 53%. Calls

	Share (%)					Batched (%)	
Category	**Calls**	**Sess.**	**Tokens**	**Time**	**Fail**	**Same**	**Mixed**
Read	16.3	68.2	33.9	7.2	1.0	23.9	5.5
Edit	12.2	55.5	3.8	5.6	2.1	8.1	1.3
Write	4.3	46.8	0.5	1.8	1.0	8.0	2.9
Browser	2.6	23.2	5.9	0.8	1.7	28.7	8.2
MCP	1.5	5.7	3.3	1.2	4.2	15.8	6.4
Subagent	0.4	10.4	0.8	2.0	1.5	29.8	6.5
Ask User	0.4	18.2	0.1	5.8	2.2	1.2	0.7
∗Interpreter	12.4	60.9	8.6	16.9	5.0	2.7	3.0
∗Search	11.2	56.7	7.5	6.9	1.9	4.8	6.8
∗File Ops	9.0	70.6	7.6	6.9	3.3	9.7	14.2
∗Network	3.7	31.2	2.5	2.7	6.3	5.7	5.2
∗Git/VCS	3.6	28.2	2.4	3.3	2.2	6.7	7.7
∗Wait/Poll	3.3	28.7	1.2	10.0	8.1	0.3	1.2
∗Build/Test	2.5	18.9	1.1	4.7	3.6	1.2	4.0
∗Packages	0.9	15.3	0.4	1.4	4.8	2.0	6.7
Other	15.6	80.8	20.4	22.9	5.5	9.0	5.6
All	**100.0**		**100.0**	**100.0**	**3.3**	**10.0**	**5.5**

Table 5: Tool overview. ∗ denotes bash tools. Batched indicates how often a tool is batched with itself (same) or others (mixed) in one LLM response.



(a) Effective latency **(b)** Output tokens

Figure 5: Tool categories differ substantially in both latency and output size. Boxes show the P5–P95 range, and diamonds denote the mean. (a) Effective tool-execution latency. (b) Number of tokens added to the context by tool execution.

are concentrated in a few categories, with Read, Interpreter, Edit, Search, and File Ops together accounting for over 60% of calls. These categories are also widely used across sessions, with File Ops appearing in 71% of sessions and Read, Interpreter, and Search each appearing in roughly 60%. In contrast, more specialized categories such as Browser, MCP, Ask User, and Subagent account for fewer calls and appear in fewer sessions. For example, MCP and Subagent appear in only 5.7% and 10.4% of sessions, respectively.

Tool frequency hides large differences in cost and execution behavior. Call frequency can differ substantially from result-token and wait-time share. Read, for example, accounts for 34% of all result tokens despite representing only 16% of calls, while Interpreter contributes 17% of total tool wait time from 12% of calls. The gap is even larger for some infrequent operations, where Ask User represents only 0.4% of calls but contributes 5.8% of total tool wait time. Tool categories also differ in how reliably and how concurrently they execute. Failure rates are low for common native tools such as Read and Write, but higher for environment-dependent operations such as Wait/Poll, Network, and Interpreter. Likewise, Read, Browser, and Subagent are more often issued in batches, whereas most shell commands are issued serially.

Most tool operations are fast with a very long tail. Figure 5a shows effective latency across tool categories. Local repository operations such as Read, Edit, Search, and File Ops are generally fast, with median latencies between 0.9 and 1.7 s. Execution-oriented operations take longer, with median latency reaching 2.8 s for Interpreter and 7.2 s for Build/Test. Wait/Poll is slower still at 11.7 s as it explicitly waits for ongoing work to complete. Coordination operations have the highest typical latency, with Subagent taking a median of 13 s and average of 100 seconds, while Ask User reaches minutes. These distributions are also strongly long-tailed. Even fast operations such as Read have a mean latency of 13.2 s despite a median below one second, while Interpreter rises from a 2.8 s median to a 30.8 s mean. Part of this tail may come from commands that require human approval under certain harness or user configurations. The same pattern is more pronounced for slower operations, with Subagent and Ask User operations reaching 8.9 and 13.3 minutes at P95, respectively.

Table 6: Tool-call batching is widely used across sessions, but most requests issue only one tool call. Subagent use is uncommon. Most subagent calls spawn one child, and one-third return asynchronously.

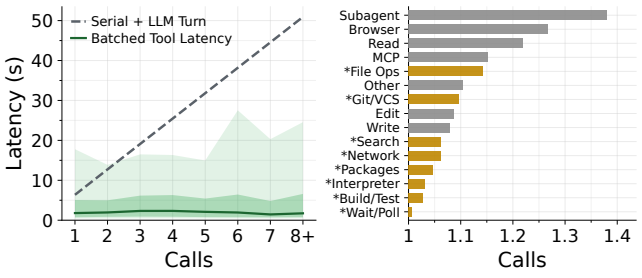
Metric	**Category**	**Share (%)**
Sessions	Batched	66.9
	Unbatched	33.1
Tool calls (per req.)	1 call	84.5
	2 calls	8.7
	3 calls	3.5
	4+ calls	3.3
Batch mix	Same tool	71.3
	Mixed	28.7

(a) Batching profile

Metric	**Category**	**Share (%)**
Sessions	No subagents	89.6
	Uses subagents	10.4
Fan-out (per req.)	1 spawn	80.6
	2 spawns	9.2
	3 spawns	5.1
	4+ spawns	5.2
Fan-in	Synchronous	67.0
	Asynchronous	33.0

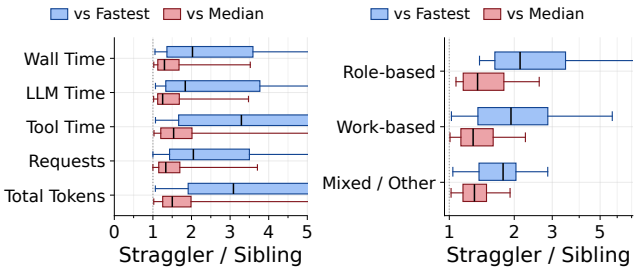
(b) Subagent profile

Read and external tools drive context growth. Figure 5b shows substantial differences in result size across tool categories. Read produces the largest output, with a median of 441 tokens, and accounts for 33.9% of all result tokens despite representing only 16.3% of calls. External and delegated operations also return large outputs, with Browser, MCP and Subagent typically returning around 300 tokens. In contrast,



(a) Latency by step **(b)** Calls per request

Figure 6: (a) Serial latency grows with the number of calls, while batched latency remains nearly flat. (b) Subagent, Browser, and Read are batched most frequently.



(a) Imbalance by metric **(b)** Wall-time imbalance by work

Figure 7: Straggler imbalance in synchronous subagent fan-out. (a) The slowest sibling is typically about 2× slower than the fastest, with the largest gap in tool time. (b) Role-based fan-outs exhibit greater wall-time imbalance than work-based and mixed fan-outs.

mutation operations such as Edit and Write return much less content, with medians below 50 tokens. Result sizes are also strongly long-tailed, with mean outputs reaching roughly 1K tokens and P95 values of several thousand tokens.

Finding 3: Tool latency and result size show distinct characteristics by category, which may provide useful signals for scheduling and prefix-cache management. However, both also vary substantially within each category, making tool type alone a weak predictor.

Serving systems may combine category with historical user- and tool-specific behavior to guide serving decisions. **Harnesses** should be co-designed with serving engines to provide hints on tool-call duration.

5.2 Tool Parallelism

Models today can issue one or more tool calls in the same response, or delegate work to subagents.

Tool batching reduces latency. Figure 6a compares the time to complete multiple tool calls when they are issued in one batch versus serially. With serial execution, each additional tool call requires another request (LLM response time and tool execution time) before the next tool call can be issued, so completion time is projected to grow roughly linearly. In contrast, batched completion time remains nearly flat as batch width grows, with P90 latency of 14–28 s across various degrees of parallelism.

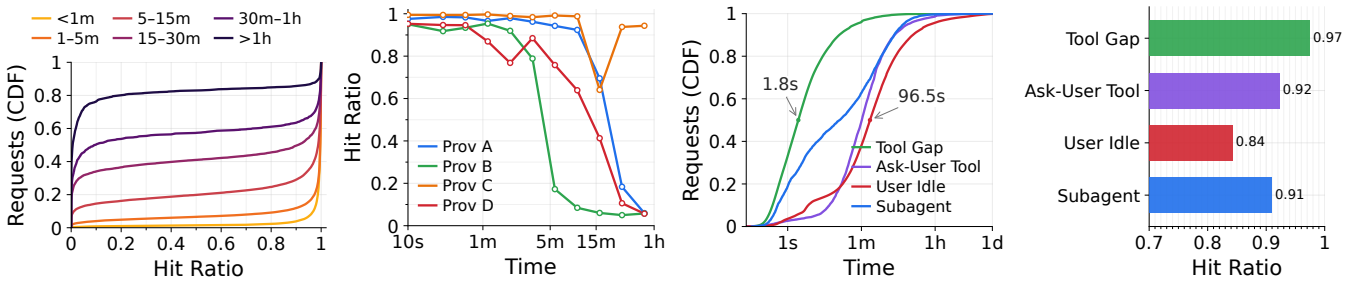
Tool batching is mostly used for read-only native tools. Table 6a shows that 84.5% of requests issue only one tool call. Large batches are rare, as requests issuing batches of four or more tool calls account for only 3% of all tool calling requests. Nevertheless, 67% of sessions use batching at least once, so batching is common across sessions even though most tool calls remain serial. Among batched requests, 71% contain repeated calls to the same tool rather than a mix of different tools. Batching also varies by tool category. Figure 6b and Table 5 show that Subagent, Browser, Read, and MCP are batched most often. In contrast, state-changing native tools such as Edit and Write are rarely batched together. Shell calls also show little batching, as a single invocation can effectively batch by composing multiple commands using pipes.

Subagents are uncommon and typically invoked synchronously. Only 10.4% of sessions use subagents, and most subagent calls spawn a single child rather than multiple children in parallel. Of these calls, 67.0% are synchronous: the parent blocks until the child finishes and receives only the final result, limiting context growth in the parent. The remaining 33.0% are asynchronous, allowing the parent to continue while the child runs, but requiring additional coordination to track and collect its result.

Subagents have a straggler problem. Synchronous subagents are often limited by stragglers. Figure 7a shows that the slowest sibling often takes 50% longer than the median sibling and more than 3× as long as the fastest. Differences in tool-execution time account for most of this imbalance. Figure 7b further shows that role-based fan-outs (e.g., reviewer, editor, and security experts) are more imbalanced than work-based fan-outs (e.g., assigning each subagent one file to work with). Because roles can require substantially different amounts of tool work, they produce greater execution-time imbalance.

Finding 4: Parallel tool calls significantly speed up agent progress; however, most tool calls today are issued serially. Most subagents run synchronously, so variation in their execution times causes severe straggler delays.

Agent harnesses should exploit parallelism more aggressively, both by issuing independent tool calls concurrently and by encouraging the use of the shell to compose commands. They should also schedule subagents intelligently to mitigate stragglers, e.g., by launching heavy subagents first. **Serving systems** and **harnesses** should be co-designed to coordinate subagent execution and model serving to reduce end-to-end latency caused by stragglers.



(a) Hit CDF by gap bin **(b)** Retention by provider **(c)** Gap CDF by cause **(d)** Mean hit ratio by cause

Figure 8: Prefix reuse falls as inter-request gaps grow. (a) Short gaps preserve most of the cached prefix, while longer gaps are associated with more misses. (b) Prefix retention varies across providers, ranging from minutes to beyond an hour. (c) Ordinary tool gaps are typically short, while user idle time, Ask User tools, and subagent execution introduce longer gaps. (d) Cache loss is more common after these longer gaps.

Transition	Steps (%)	Sess. (%)	Prefill (%)
Append	96.00	43.6	37.5
Mutation	**4.00**	**56.4**	**62.5**
Injection	0.96	17.1	7.4
System prompt	0.74	20.3	26.9
Tool definitions	0.40	21.4	13.9
Dropped turns	0.36	6.5	5.3
Compaction	0.04	3.7	0.6
Tool result drop	0.51	5.9	3.2
Other	1.00	14.5	5.2

Table 7: Context mutations by cause under infinite retention. Mutations cause a substantial amount of new prefill.

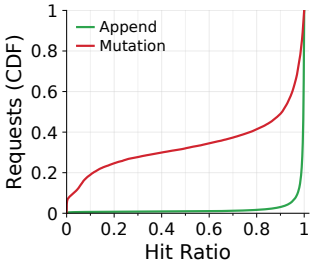


Figure 9: Context mutations break reuse: the hit ratio on the step after a mutation is far below the step after an append.

6 Prefix Caching

Prefix caching is a key serving primitive for agentic workloads as it avoids repeated prefill work over long contexts. In this section, we first analyze hit ratios and how reuse changes with idle time in production. We then characterize how context mutations reduce reuse, the tradeoff between storage overhead and retention time for both providers and consumers, and whether session-level signals can help guide storage and retention time.

6.1 Production System

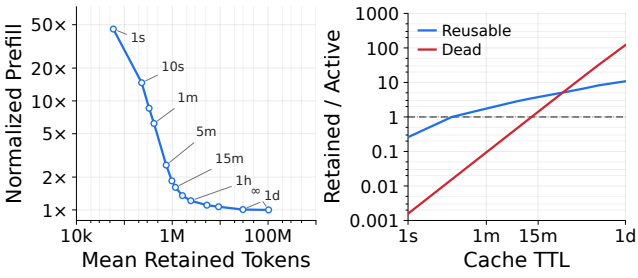
We first examine how long cached prefixes remain available between consecutive LLM requests in production. For each request, we measure the inter-request gap since the preceding request and its hit ratio. We separate inter-request gaps into four categories: *Tool Gap* for ordinary tool execution, *Ask User* for tools that explicitly request user input, *User Idle* for user idle time between turns, and *Subagent* for delegated subagent execution.

Cache retention times vary widely across providers. Figure 8a groups requests by the inter-request gap. Prefix reuse remains high when requests arrive within one minute. The hit ratio declines as the gap grows, to around 58% at 15–30 minutes and only 18% beyond an hour. However, this aggregate hides substantial differences in how long providers retain cached prefixes. Figure 8b compares retention across providers serving the same open-weight models. As the reuse interval increases, hit ratios decline at different rates: Provider D falls below a 50% median hit ratio after 15 minutes and Provider B after 30 minutes, whereas Provider A still maintains a high hit ratio near one hour. The shape of the decline also differs: some providers exhibit sharp drops at particular intervals, which may reflect time-based retention, while others lose reuse more gradually, which may reflect load-dependent eviction. *Provider retention is therefore an important hidden driver of cache misses and user cost*, as the underlying retention mechanism is often opaque to users.

Cache reuse drops most after longer user waits. Figure 8c shows the latency distribution across different sources of inter-request gaps. Most gaps come from routine tool execution, where tool calls account for 92.7% of inter-request gaps and have a median duration of only 1.8 s. In contrast, Ask User tools, subagent execution, and user turns pause the session for substantially longer. Their tails are much longer, where ordinary tool gaps extend to 17.2 s, compared with 6.3 minutes for Ask User calls, 6.2 minutes for subagents, and 17.4 minutes for user turns at P90. Figure 8d shows how these longer pauses translate into cache misses. Ordinary tool calls maintain a high mean hit ratio of 97.5%, while reuse declines across Ask User calls, subagents, and especially user idle periods, reaching only 84% on average. Current retention policies thus already capture most reuse within LLM–tool loops; the remaining challenge is preserving reuse across longer and more variable user gaps.

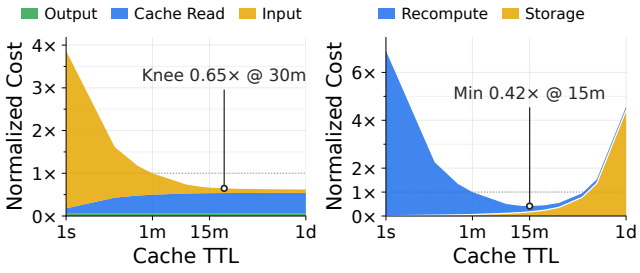
Finding 5: Agentic sessions are bursty, with short inter-request gaps during LLM–tool loops and long gaps when control returns to the user or a subagent, triggering misses.

Serving systems should treat the session as the management unit for KV-cache retention and prioritize retention during active loops. **Harnesses** can expose activity hints to guide retention, such as when a user is unlikely to return for eviction or offloading, or when a session resumes for prefetching or eager prefill. **Users** should measure the hidden retention time when comparing providers.



(a) Prefill vs. storage **(b)** Retained over active tokens

Figure 10: Longer retention reduces prefill but increases storage overhead. (a) New prefill remains 60.9% above the floor at 15 minutes, and about 1% above at one day. (b) Reusable blocks grow slowly with TTL, while dead blocks dominate at long retention and reach about 120× the active blocks at one day.



(a) API cost **(b)** Infrastructure cost

Figure 11: Longer retention reduces API cost by avoiding recomputation. Cost is normalized to a 1-minute TTL. (a) Most savings are captured within 30 minutes, at 0.65×. (b) Infrastructure cost reaches its minimum of 0.42× at 15 minutes, after which the added storage overhead outweighs further recomputation savings.

6.2 Context Mutation

Agent contexts are often assumed to grow monotonically, with each request appending new content to the prior context. However, we find that in practice, agent harnesses frequently modify earlier context in ways that break prefix reuse. We compare consecutive requests within a session and classify a transition as an *append* when the previous context is preserved and new content is added, and as a *mutation* otherwise.

Context mutations are rare per step but common over a session. Most steps simply extend the existing context, but harnesses periodically rewrite, truncate, or compact earlier context. Table 7 shows mutations account for only 4.0% of step transitions, yet are common within a session, with 56.4% of sessions containing at least one mutation, and every cause appears in more than roughly 4% of sessions. They are usually triggered by user input, as user steps account for only 6.4% of all steps but 35.4% of mutated steps, suggesting that harnesses often perform mutation upon a new user turn. Figure 9 shows substantially lower hit ratios after mutations than after append-only steps, as they invalidate a large portion of cached tokens. Table 7 quantifies the resulting new prefill tokens assuming infinite retention, showing that mutations are a major cause of recomputation, contributing 62% of new prefill tokens.

Context mutations vary widely in cause and severity. Table 7 groups mutations into six common causes. The most widespread cause is *tool-definition changes*, which appear in 21.4% of sessions and account for 13.9% of fresh prefill. Harnesses may change the set or definitions of tools offered to the model as agent capabilities evolve across turns. Because tool definitions are typically rendered before the system prompt, even a small change invalidates nearly the entire cached prefix. Among the remaining causes, *context injection* occurs most frequently by mutation steps. Harnesses may attach runtime state, such as environment information or instructions, to later user messages, then strip or relocate this information in subsequent turns. Because these changes typically occur late in the prompt, they are frequent but invalidate only a limited portion of the cached prefix.

In contrast, *system-prompt rewrites* occur less frequently per step but are more expensive, appearing in 20.3% of sessions compared with 17.1% for injection. Harnesses may rewrite the system prompt to update permissions, operating modes such as editor, planning, or review mode, or environment information. Because these changes occur near the beginning of the prompt, they invalidate most of the cached prefix. Harnesses also mutate context to manage its size. *History truncation* drops older turns using a rolling context window and appears in 6.5% of sessions, accounting for 5.3% of fresh prefill. *Context compaction* instead replaces older history with a shorter summary; it is less common, appearing in 3.7% of sessions and accounting for only 0.6% of fresh prefill. Finally, *tool-result mutations* arise when harnesses prune old tool outputs, update long-running tool results with progress information, or re-serialize structured outputs such as JSON, causing unnecessary cache misses.

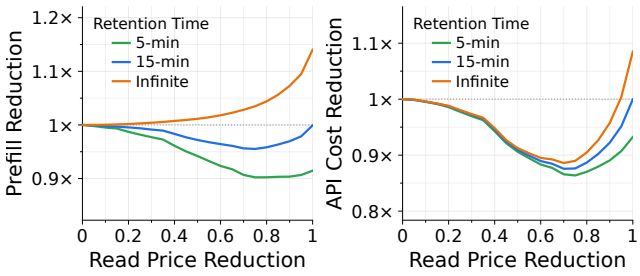
Finding 6: Context mutation is common in agentic sessions, invalidating otherwise reusable prefixes and causing substantial new prefill. Some mutations are intentional, such as environment injection and compaction, while others result from avoidable changes such as re-serialization.

Agent harnesses should preserve stable prefixes by placing volatile state later in the prompt and serializing repeated state deterministically. They should batch mutations into fewer, less frequent updates and quantify their associated cost. **Users** should use these signals to assess how efficiently a harness manages context.

6.3 Prefix Retention Tradeoffs

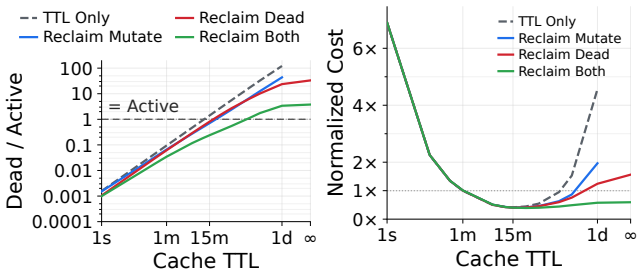
We next ask how long the KV cache should retain prefixes after a request completes. We replay the reconstructed request stream through a simulated prefix cache with varying time-to-live (TTL) values, using hashed 16-token blocks.

One day of retention approaches minimum prefill, but at considerable storage overhead. Figure 10a shows the tradeoff between retention time and new prefill. Short retention



(a) Prefill reduction **(b)** API cost reduction

Figure 12: Lowering cache-read prices to reduce context mutations does not necessarily reduce prefill or API cost as contexts become longer and misses become more expensive. We assume harnesses reduce mutations linearly as cache reads become cheaper, varying retention times and compare against today's repriced token volume.



(a) Dead blocks over active **(b)** Infrastructure cost

Figure 13: Session-level signals reclaim dead blocks invalidated by context mutations or left behind by completed sessions before their TTL expires. Reclaiming both sources helps on both counts in simulation. (a) Dead blocks fall by about 36× at a one-day TTL. (b) Infrastructure cost stays low even at long TTLs.

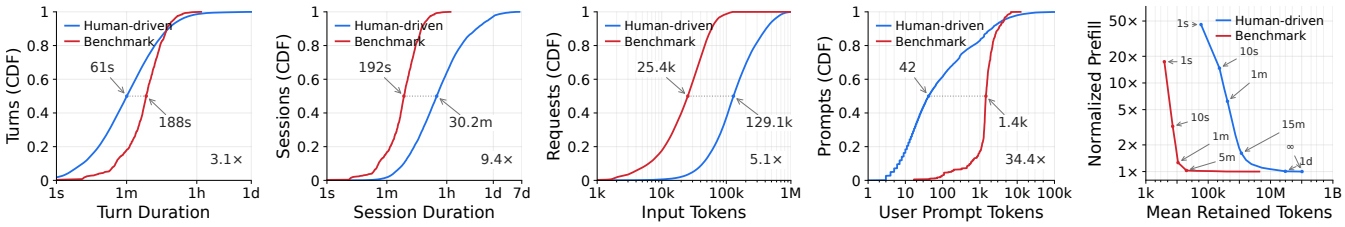
frequently evicts prefixes before reuse, causing substantial recomputation. At 15 minutes, retention captures much of the available reuse, but new prefill remains 60.9% above the infinite-retention floor. Approaching minimum prefill requires much longer retention, with prefill falling to 35% at 30 minutes, 21% at one hour, 7% at six hours, and about 1% at one day. This comes with substantial storage overhead, with the mean storage overhead growing to 2.1× at one hour and 25.6× at one day compared to the 15-minute overhead. Figure 10b shows where this additional storage goes. We classify retained blocks into *reusable* blocks that will be reused and *dead* blocks that will never be reused because the session has ended or the context has changed. At short retention times, both remain modest. As retention increases, however, dead blocks grow much faster than reusable blocks and eventually dominate storage. Thus, naively extending retention increasingly spends storage on unusable blocks, motivating policies that selectively retain reusable prefixes.

API cost flattens around 30 minutes, while infrastructure cost is minimized at 15 minutes. We estimate KV cache retention costs from both user API and infrastructure perspectives. For API cost, we assume $10/M uncached input tokens, $1/M cached input tokens, and $50/M output tokens, following Claude Fable 5 pricing [6]. For infrastructure cost, we model recomputation at $1/M tokens and storage at $30/M token-days, based on AWS DRAM pricing and a 320 KB/token KV footprint for a 70B GQA model in FP16. Under these assumptions, Figure 11a shows that API cost drops rapidly as the retention time increases to 15 minutes and then largely flattens. At 30 minutes, the cost is already within 4% of its minimum. Figure 11b shows the corresponding infrastructure tradeoff. However, long retention still provides diminishing returns as increasingly more storage is spent on storing dead blocks.

Lowering cache-read prices to reduce context mutations does not necessarily reduce user cost. Agent harnesses often mutate context to reduce its size and lower cost, but doing so breaks prefix reuse and introduces fresh prefill. Lower cache-read prices may therefore encourage harnesses to retain more context and perform fewer mutations. We model this by linearly reducing context-reducing mutations as cache-read prices fall from $1/M tokens to zero. When a reduction is removed, the prior context is preserved, allowing histories to grow up to the 1M-token context limit. Figure 12a shows the resulting prefill relative to today's harness behavior, with read costs repriced accordingly. With finite retention, however, eliminating reductions leaves longer contexts in the session, making cache misses more expensive and offsetting the benefit of avoiding mutations. As a result, fresh prefill increases as reductions are removed, peaking at about 5% with 15-minute retention and 10% with 5-minute retention. This tradeoff is overcome only when reads become nearly free.

Figure 12b shows a similar result for API cost. Forcing stricter prefix reuse retains more context, which increases the number of cached tokens reread. Even with infinite retention, this additional read volume outweighs the savings from reduced prefill until cache-read prices fall to roughly $0.05/M tokens. With finite retention, miss-induced prefill eliminates the savings entirely, even when cache reads are free. Thus, context reduction serves an important role in bounding both cache-read volume and miss-induced recomputation as context grows. This also implies that stateful LLM APIs [37, 51] should provide context-management mechanisms such as compaction to bound context growth and serving cost.

Session-level lifecycle signals reclaim dead KV blocks and make long retention practical. Longer prefix retention turns KV caching into a memory-management problem: blocks from completed sessions and context mutations remain allocated until their TTL expires, even when they will never be reused. This resembles a memory leak, where harnesses effectively allocate KV blocks without providing an explicit `free()`. We evaluate whether simple lifecycle signals can enable earlier reclamation. Given a `session_id`, a *mutate* signal identifies freed blocks invalidated by a context mutation, while an *end* signal frees all blocks belonging to a completed session. Figure 13a shows that dead blocks ac-



(a) Turn duration **(b)** Session duration **(c)** Input per request **(d)** User prompt size **(e)** Prefill vs. storage

Figure 14: Benchmark replay and production sessions stress serving systems differently. (a) SWE-Bench has longer uninterrupted single-turn execution. (b) Production sessions span multiple turns and last longer overall. (c) Production sessions accumulate larger contexts. (d) SWE-Bench prompts are longer and more uniform, whereas production prompts are shorter and more variable. (e) Five minutes of retention brings SWE-Bench prefill within 3% of its infinite-retention floor; production first reaches this range at one day among the evaluated TTLs.

cumulate rapidly as retention grows. Either signal reduces this accumulation, but combining both reclaims blocks earlier and *reduces accumulated storage by up to* 36× *at a one-day TTL*. Figure 13b shows the corresponding cost benefit: early reclamation keeps infrastructure cost low even with very long retention by preventing dead blocks from accumulating. Thus, lifecycle signals enable longer retention by giving providers hints to keep useful prefixes longer while promptly reclaiming blocks that are no longer reusable.

Finding 7: Longer retention improves prefix reuse, but dead blocks from mutations and completed sessions quickly increase storage overhead. Context reduction remains necessary to control the read and miss costs of long contexts.

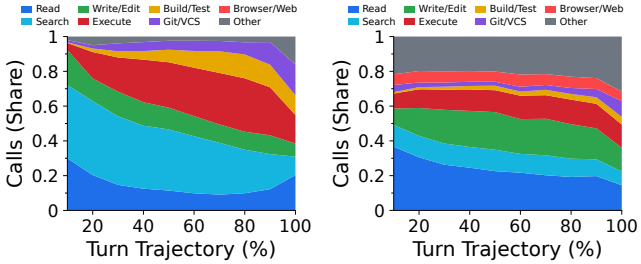
Serving systems and **agent harnesses** should be co-designed to selectively retain useful state and reclaim dead blocks early. Harnesses can expose lifecycle hints such as *mutation* to reclaim invalidated blocks, *pause* to offload state, *resume* to prefetch it, and *end* to release it. **Users** can reduce cost by compacting long sessions or starting a new session after long breaks to avoid expensive misses.

7 Benchmark vs. Production Workloads

Benchmark replay is commonly used to evaluate agent-serving systems, but it differs substantially from human-driven traffic. We compare the two in terms of duration, context growth, tool use, and cacheability, identifying serving behaviors that benchmark replay might miss.

Benchmarks run shorter and miss user idle time. Figures 14a and 14b compare the execution structure of the two workloads. The median SWE-Bench turn lasts 188 s compared with 61 s for human-driven turns, as the benchmark agent receives one issue and works continuously toward completion. Across multiple user turns, however, the median session lasts 30 minutes overall compared with only 3.2 minutes for SWE-Bench. Benchmark replay therefore misses the bursty pattern of LLM–tool execution separated by long user idle periods. Once a SWE-Bench task finishes, there is no need to retain context as the session has ended.

Benchmarks carry shorter contexts and understate production memory pressure. Figure 14d shows that SWE-Bench starts with relatively large user prompts, with a median of 1.4K tokens and little variation across tasks. Prompts in human-driven sessions are typically shorter but have a long tail, as users sometimes provide large inputs such as error logs or code. Human-driven sessions span multiple turns, causing the context to grow substantially over time. Figure 14c shows that the median request in these sessions contains 129K input tokens, compared with 25K for SWE-Bench, making benchmark requests substantially cheaper to retain in memory.

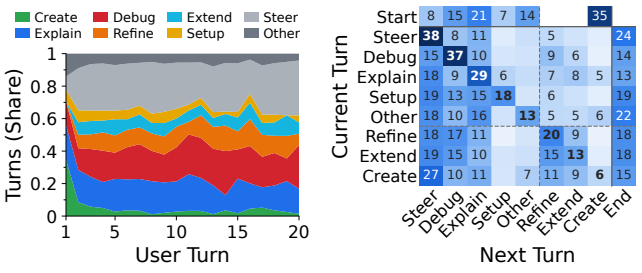


(a) SWE-Bench **(b)** Human-driven

Figure 15: Tool usage evolves differently in benchmark and human-driven turn trajectories. (a) SWE-Bench shifts from Read/Search early to Execute and Build/Test later. (b) The tool mix remains more diverse throughout the turn.

Benchmarks need minutes of retention, while production reuse spans a day. Figure 14e replays both workloads through the TTL simulator. For SWE-Bench, a 1-minute TTL remains 26% above the prefill floor, while 5 minutes captures nearly all reuse. In human-driven sessions, reuse persists across much longer gaps. Even with a 15-minute TTL, prefill remains 61% above the floor, and roughly one day is needed to capture nearly all reuse. Caching is further complicated by context mutations, which can invalidate previously reusable prefixes but are absent from SWE-Bench. Benchmark-only evaluations therefore substantially understate the retention requirements observed in production.

Tool trajectories are more phased in benchmark replay. To compare tool usage between the two workloads, we merge the categories of Table 5 into eight coarser groups by purpose (Read, Search, Write/Edit, Execute, Build/Test, Git/VCS, Browser/Web, and Other), since the SWE-Bench agent is



(a) Intent mix by turn **(b)** Turn-to-turn transitions

Figure 16: User prompts shift from establishing tasks to guiding execution as sessions progress. (a) Opening turns are dominated by feature-creation requests, while later turns contain more steering requests. (b) User intent also exhibits phased transitions.

restricted to performing its work through shell commands. Figure 15 shows that SWE-Bench follows a clear sequence of work across a user turn. Early execution is dominated by repository exploration, with most calls spent on Read and Search. Later, Execute, Build/Test, and Git/VCS become more common as the agent moves from understanding the codebase to implementing and validating a solution. Human-driven trajectories are less structured: Read remains important throughout the turn, although its frequency decreases over time. Write/Edit, Execute, and other operations become more common later in the turn. These agents also use a broader mix of Browser/Web and other tools. These changes are less distinct than in SWE-Bench, suggesting that evaluating tool use only through benchmark replay can underrepresent the diversity of tool usage patterns in production.

Users establish tasks early, then guide execution through persistent interaction phases. We use a large language model to label roughly 10K turns from a random sample of 1.8K sessions, providing several few-shot examples to the model. We classify turns into eight intents: Create starts new work, Extend adds functionality or scope, Refine modifies existing work, Debug addresses failures, Explain asks for understanding, Steer provides nudges such as "continue" or "yes," Setup configures the environment, and Other captures remaining requests. Figure 16a shows how user intent evolves over a session. Opening turns mainly establish what the agent should do, with Create accounting for 35% of first turns. Create then drops sharply, while later turns increasingly guide work already in progress through Steer, Refine, and Extend. Together, these task-continuing intents account for 46% of turns after the first. Figure 16b shows that intents also follow common transition patterns. After a Create request, users often move into follow-up interaction, while Steer, Debug, and Explain often persist across consecutive turns. This locality may help harnesses detect task boundaries. When users shift to unrelated work, harnesses could suggest starting a new session or offloading the task to a subagent, avoiding unnecessary context growth.

Finding 8: Evaluating serving systems only on benchmarks can understate the storage and retention requirements observed in production. Human-driven sessions last longer, carry larger contexts, contain idle gaps, and exhibit more diverse tool and user interaction patterns.

Serving systems should evaluate on real workloads, or augment benchmarks with an orchestrator that answers clarification questions, writes short prompts to steer the session, and produces idle gaps.

8 Related Work

LLM and agentic serving systems. Prior work improves LLM serving through batching, memory management, prefill–decode disaggregation, placement, and prefix caching. Orca, vLLM, Sarathi-Serve, and vAttention [2, 21, 42, 68] optimize request execution and memory use, while Splitwise, DistServe, LoongServe, and Llumnix [41, 47, 56, 76] place and migrate work across serving resources. Recent systems target agentic workloads more directly. Parrot, Agentix, Autellix, and Ayo [26, 31, 32, 48] exploit agent structure for scheduling, while InferCept, Conveyor, Continuum, and MORI [1, 23, 57, 62] manage KV cache across tool-call gaps. KVFlow, SAGA, and Murakkab [9, 13, 40] further use workflow structure for KV-cache management, scheduling, and orchestration. A related body of work improves prefix reuse directly. SGLang and Preble [45, 74] schedule around shared prefixes, while Pensieve, MemServe, and LMCache [14, 29, 52, 69] retain or offload KV state across requests. Prompt Cache, CachedAttention, and HCache [10–12] support reusable prompt state, while CacheGen, Cake, and Strata [17, 30, 61] reduce KV movement cost. CacheBlend and EPIC [15, 65] extend reuse beyond exact prefixes, while HotPrefix, Marconi, Learned Prefix Caching, and BatchLLM [24, 39, 63, 75] improve cache admission, eviction, or scheduling. H2O, StreamingLLM, and SnapKV [25, 59, 71] reduce KV-cache footprint through eviction or compression, while Quest and InfiniGen [22, 49] selectively load needed state. As more serving systems target agentic workloads, representative evaluation becomes increasingly important. Our study characterizes production agent workloads and releases the underlying trace to support more realistic benchmarks and system evaluation.

Workload analysis. Existing production LLM traces characterize request arrivals, token lengths, load variation, and prefix reuse across chat, API, and code-completion workloads but provide limited visibility into agent behavior [35, 41, 43, 44, 46, 53, 58]. Agent workloads are commonly studied through benchmarks including SWE-Bench and Terminal-Bench [16, 33], GAIA and τ-bench [34, 66], and AgentBench, WebArena, and OSWorld [28, 60, 77]. More recent work examines production coding-agent sessions. Agentic Coding in the Wild, TraceLab, CacheWise, and SWE-chat [7, 27, 50, 78] analyze client-side logs or shared sessions from Copilot, Claude Code, and Codex. However, these traces cover only a narrow

set of agent harnesses and mostly expose timing information, limiting broader workload analysis. Our work provides a broader provider-side characterization and releases a richer trace with tool- and block-level information, supporting a wider range of downstream agentic workload analyses and characterization.

9 Limitations and Future Work

Our trace provides a provider-side view of production coding-agent workloads, which introduces several limitations. Because we collected data at the inference service, we do not have access to precise client-side measurements of tool execution latency, and tool latency is also affected by users' devices. Most coding agents also do not expose explicit session identifiers, so we must reconstruct sessions. As a result, the sessions we reconstruct may differ from those defined by individual clients. *FreeInference* does not constrain agent usage patterns and behavior, resulting in a heterogeneous workload whose characteristics may not fully generalize to services with more controlled or restricted workloads.

10 Conclusion

We present a provider-side characterization of production coding-agent workloads from *FreeInference*. Our analysis shows that agentic workloads are structured around long-lived sessions, where growing context is repeatedly reused across bursts of LLM–tool execution but disrupted by user interaction and context mutation. These session dynamics directly affect serving decisions such as prefix retention, yet are not well represented in existing general LLM workload datasets and benchmarks. We plan to release the trace to support further research on realistic agentic workloads and serving systems.

References

[1] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. InferCept: Efficient Intercept Support for Augmented Large Language Model Inference. In *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*, volume 235 of *Proceedings of Machine Learning Research*, pages 81–95. PMLR, 2024.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, Santa Clara, CA, jul 2024. USENIX Association.

[3] Anthropic. Building effective agents. `https://www.anthropic.com/engineering/building-effective-agents`, December 2024. Anthropic Engineering Blog. Published December 19, 2024; accessed 2026-09-17.

[4] Anthropic. Effective harnesses for long-running agents. `https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents`, November 2025. Anthropic Engineering Blog. Published November 26, 2025; accessed 2026-09-17.

[5] Anthropic. Claude Code. `https://code.claude.com/docs/en/overview`, 2026. Accessed: 2026-09-10.

[6] Anthropic. Pricing. `https://platform.claude.com/docs/en/about-claude/pricing`, 2026. Claude Developer Platform documentation. Accessed: 2026-09-18.

[7] Joachim Baumann, Vishakh Padmakumar, Xiang Li, John Yang, Diyi Yang, and Sanmi Koyejo. SWE-chat: Coding Agent Interactions From Real Users in the Wild. arXiv preprint arXiv:2604.20779, 2026.

[8] Chaokun Chang, Yukun Zhou, Kaihua Fu, Dakai An, Tianyu Feng, Hanfeng Lu, Sheng Yao, Pu Guo, Yinghao Yu, Yizhou Shan, Bo Li, Binhang Yuan, and Wei Wang. From LLM Inference to Agentic Workloads: Characterization and Implications for Serving Systems. arXiv preprint arXiv:2608.15127, 2026.

[9] Gohar Irfan Chaudhry, Esha Choukse, Haoran Qiu, Íñigo Goiri, Rodrigo Fonseca, Adam Belay, and Ricardo Bianchini. Murakkab: Resource-Efficient agentic workflow

orchestration in cloud platforms. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pages 567–587, Seattle, WA, July 2026. USENIX Association.

[10] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-Efficient large language model serving for multi-turn conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 111–126, Santa Clara, CA, July 2024. USENIX Association.

[11] Shiwei Gao, Youmin Chen, and Jiwu Shu. Fast state restoration in LLM serving with HCache. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, pages 128–143. Association for Computing Machinery, 2025.

[12] In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. In *Proceedings of Machine Learning and Systems*, volume 6, 2024.

[13] Dongxin Guo, Jikun Wu, and Siu-Ming Yiu. SAGA: Workflow-atomic scheduling for AI agent inference on GPU clusters. arXiv preprint arXiv:2605.00528, 2026.

[14] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. MemServe: Context Caching for Disaggregated LLM Serving with Elastic Memory Pool. arXiv preprint arXiv:2406.17565, 2024.

[15] Junhao Hu, Wenrui Huang, Weidong Wang, Haoyi Wang, Tiancheng Hu, Qin Zhang, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. EPIC: Efficient position-independent caching for serving large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 24391–24402. PMLR, 2025.

[16] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[17] Shuowei Jin, Xueshen Liu, Qingzhao Zhang, and Zhuoqing Mao. Compute or load KV cache? why not both? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 28031–28043. PMLR, 2025.

[18] Kilo Code. Kilo Code. `https://kilo.ai/`, 2026. Accessed: 2026-09-10.

[19] Donghwan Kim, Prakhar Singh, Younghoon Min, Jongryool Kim, Jongse Park, and Kiwan Maeng. Characterization of Multi-Model Agentic AI Systems on General Tasks via Trace-Driven Simulation. arXiv preprint arXiv:2606.01725, 2026.

[20] Leonid Kondrashov, Hongrui Liu, JooYoung Park, Boxi Zhou, Zonghao Liu, Chengzhi Lu, Riccardo Mancini, Esha Choukse, Haris Javaid, German Sviridov, Tao Peng, Chen Zhao, Anastasia Avdeeva, Aleksei Gusev, Marios Kogias, Luo Mai, and Dmitrii Ustiugov. Rethinking AI Cloud Infrastructure for Agentic Serving Systems with the Aries Experimentation Framework. arXiv preprint arXiv:2607.29069, 2026.

[21] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP'23, pages 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[22] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 155–172. USENIX Association, jul 2024.

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

[24] Yuhang Li, Rong Gu, Chengying Huan, Zhibin Wang, Renjie Yao, Chen Tian, and Guihai Chen. HotPrefix: Hotness-aware KV cache scheduling for efficient prefix sharing in LLM inference systems. *Proceedings of the ACM on Management of Data*, 3(4):250:1–250:27, 2025.

[25] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. SnapKV: LLM Knows What You are Looking for Before Generation. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024.

[26] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient Serving of LLM-based Applications with

Semantic Variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 929–945, 2024.

[27] Banruo Liu, Haoran Qiu, Íñigo Goiri, Rodrigo Fonseca, Ricardo Bianchini, and Esha Choukse. Agentic Coding in the Wild: Characterizing GitHub Copilot Traces at Production Scale. arXiv preprint arXiv:2608.00101, 2026.

[28] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as Agents. In *The Twelfth International Conference on Learning Representations (ICLR 2024)*. OpenReview.net, 2024.

[29] Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, and Junchen Jiang. LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference. arXiv preprint arXiv:2510.09665, 2025.

[30] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. CacheGen: KV cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 38–56. Association for Computing Machinery, 2024.

[31] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E Gonzalez, et al. Agentix: An efficient serving engine for LLM agents as general programs. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pages 2443–2459, 2026.

[32] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. Autellix: An Efficient Serving Engine for LLM Agents as General Programs. arXiv preprint arXiv:2502.13965, 2025.

[33] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, et al. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. arXiv preprint arXiv:2601.11868, 2026.

[34] Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: a benchmark for general AI assistants. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[35] William Nixon, Jon Durbin, Florian Standhartinger, Haryadi S. Gunawi, and Juncheng Yang. A Year in LLM Serving: Workload Evolution, Caching and Load-Balancing. arXiv preprint arXiv:2608.13573, 2026.

[36] OpenAI. Codex CLI. `https://github.com/openai/codex`, 2026. Accessed: 2026-09-10.

[37] OpenAI. Responses api. `https://developers.openai.com/api/reference/responses`, 2026. Accessed: 2026-09-17.

[38] OpenCode. OpenCode: The open source AI coding agent. `https://opencode.ai/`, 2026. Accessed: 2026-09-10.

[39] Rui Pan, Zhuang Wang, Zhen Jia, Can Karakus, Luca Zancato, Tri Dao, Yida Wang, and Ravi Netravali. Marconi: Prefix caching for the era of hybrid LLMs. In *Proceedings of Machine Learning and Systems*, volume 7, 2025.

[40] Zaifeng Pan, Ajjkumar Patel, Yipeng Shen, Zhengding Hu, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. KVFlow: Efficient prefix caching for accelerating LLM-based multi-agent workflows. In *Advances in Neural Information Processing Systems*, volume 38, pages 139912–139931, 2025.

[41] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.

[42] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*. ACM, mar 2025.

[43] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, Santa Clara, CA, feb 2025. USENIX Association.

[44] Haoran Qiu, Anish Biswas, Zihan Zhao, Jayashree Mohan, Alind Khare, Esha Choukse, Íñigo Goiri, Zeyu Zhang, Haiying Shen, Chetan Bansal, et al. Modserve: Modality-and stage-aware resource disaggregation for scalable multimodal model serving. In *Proceedings of the 2025 ACM Symposium on Cloud Computing*, pages 817–830, 2025.

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

[46] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1348–1362. IEEE, mar 2025.

[47] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 173–191. USENIX Association, jul 2024.

[48] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. Towards End-to-End Optimization of LLM-based Applications with Ayo. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*. ACM, mar 2025.

[49] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. QUEST: Query-Aware Sparsity for Efficient Long-Context LLM Inference. In *Forty-first International Conference on Machine Learning (ICML 2024)*. OpenReview.net, 2024.

[50] Shubham Tiwari, Tapan Chugh, Nash Rickert, Simon Peter, Ratul Mahajan, and Haiying Shen. CacheWise: Understanding Workloads and Optimizing KVCache Management for Efficiently Serving LLM Coding Agents. arXiv preprint arXiv:2606.16824, 2026.

[51] vLLM Project. Agentic api: Stateful api logic for agentic applications using vllm. `https://github.com/vllm-project/agentic-api`, 2026. Accessed: 2026-09-17.

[52] vLLM Project. Automatic prefix caching. `https://docs.vllm.ai/en/latest/design/prefix_caching/`, 2026. vLLM design documentation. Accessed: 2026-09-17.

[53] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. KVCache Cache in the Wild: Characterizing and Optimizing KVCache Cache at a Large Cloud Provider. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 465–482, Boston, MA, jul 2025. USENIX Association.

[54] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *The Thirteenth International Conference on Learning Representations (ICLR 2025)*. OpenReview.net, 2025.

[55] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. BurstGPT: A Real-world Workload Dataset to Optimize LLM Serving Systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD '25)*, pages 5831–5841. ACM, aug 2025.

[56] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. ACM, nov 2024.

[57] Tian Xia, Hanchen Li, Zhifei Li, Xiaokun Chen, Hao Kang, Yifan Qiao, Yi Xu, and Ion Stoica. Idleness is Relative: Exploiting Tool-Call Idle Windows for Offloading in Agentic Systems with MORI. arXiv preprint arXiv:2606.00866, 2026.

[58] Yuxing Xiang, Xue Li, Kun Qian, Yan Zhang, Wenyuan Yu, Ennan Zhai, Xin Jin, and Jingren Zhou. ServeGen: Workload Characterization and Generation of Large Language Model Serving in Production. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pages 1845–1859, Renton, WA, may 2026. USENIX Association.

[59] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient Streaming Language Models with Attention Sinks. In *The Twelfth International Conference on Learning Representations (ICLR 2024)*. OpenReview.net, 2024.

[60] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhou-

jun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024.

[61] Zhiqiang Xie, Ziyi Xu, Mark Zhao, Yuwei An, Vikram Sharma Mailthody, Scott Mahlke, Michael Garland, and Christos Kozyrakis. Strata: Hierarchical context caching for long context language model serving. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pages 1–16, Seattle, WA, July 2026. USENIX Association.

[62] Yechen Xu, Xinhao Kong, Tingjun Chen, and Danyang Zhuo. Conveyor: Efficient Tool-aware LLM Serving with Tool Partial Execution. arXiv preprint arXiv:2406.00059, 2024.

[63] Dongsheng Yang, Austin Li, Kai Li, and Wyatt Lloyd. Learned prefix caching for efficient LLM inference. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

[64] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024.

[65] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, pages 94–109. Association for Computing Machinery, 2025.

[66] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations*, volume 2025, pages 9965–10017, 2025.

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

[68] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, OSDI22, pages 521–538, 2022.

[69] Lingfan Yu, Jinkun Lin, and Jinyang Li. Stateful Large Language Model Serving with Pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, pages 144–158, New York, NY, USA, 2025. Association for Computing Machinery.

[70] Yichao Yuan, Ankita Nayak, Souvik Kundu, and Nishil Talati. Agentic AI Workload Characteristics. arXiv preprint arXiv:2605.26297, 2026.

[71] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, 2023.

[72] Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. WildChat: 1M ChatGPT Interaction Logs in the Wild. In *The Twelfth International Conference on Learning Representations (ICLR 2024)*. OpenReview.net, 2024.

[73] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. LMSYS-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset. In *The Twelfth International Conference on Learning Representations (ICLR 2024)*. OpenReview.net, 2024.

[74] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, pages 62557–62583, 2024.

[75] Zhen Zheng, Xin Ji, Taosong Fang, Fanghao Zhou, Chuanjie Liu, and Gang Peng. BatchLLM: Optimizing large batched LLM inference with global prefix sharing and throughput-oriented token batching. In *Proceedings of Machine Learning and Systems*, volume 8, 2026.

[76] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, Santa Clara, CA, jul 2024. USENIX Association.

[77] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou,

Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *The Twelfth International Conference on Learning Representations (ICLR 2024)*. OpenReview.net, 2024.

[78] Kan Zhu, Mathew Jacob, Chenxi Ma, Yi Pan, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. TraceLab: Characterizing Coding Agent Workloads for LLM Serving. arXiv preprint arXiv:2606.30560, 2026.