

# TraceLab: Characterizing Coding Agent Workloads for LLM Serving

Kan Zhu<sup>1</sup> Mathew Jacob<sup>1</sup> Chenxi Ma<sup>1,2,\*</sup> Yi Pan<sup>3</sup> Stephanie Wang<sup>1</sup>  
Arvind Krishnamurthy<sup>1</sup> Baris Kasikci<sup>1</sup>

<sup>1</sup>University of Washington   <sup>2</sup>Wuhan University of Technology   <sup>3</sup>Shanghai Jiao Tong University

## Abstract

Coding agents are rapidly becoming a major application of agentic LLMs, but serving them efficiently remains challenging. Progress on this challenge requires understanding real workload patterns, yet the data needed for such analysis is largely absent. Existing public traces and benchmarks do not capture real, day-to-day coding-agent usage across multiple agents and model families for serving-system analysis. To help fill this gap, we collect and release a trace of roughly 4,300 coding-agent sessions, containing about 350,000 LLM steps and 430,000 tool calls from our own day-to-day use of Claude Code and Codex. Our analysis shows that coding-agent workloads feature long autonomous loops, long contexts with short outputs, diverse and heavily-tailed tool calls, and high but imperfect prefix cache hit rates. These findings point to concrete opportunities for optimizing serving, including lower-overhead tool calling, append-length-aware prefix, semantic-aware tool-latency prediction, and improved KV-cache management around human-paced gaps. We release the dataset, trace collection pipeline, and analysis code at <https://github.com/uw-syfi/TraceLab.git>; the project website is <https://tracelab.cs.washington.edu>.

## 1 Introduction

Agentic LLMs have grown rapidly, enabling models to handle increasingly complex tasks through reasoning and tool calling [65, 91]. Among these, coding agents are evolving especially fast, as companies release their own coding agents, e.g., Claude Code from Anthropic [4], Codex from OpenAI [55], and Gemini CLI from Google [19]. Open-source solutions such as OpenCode [58] and DeepCode [22], which allow users to run local model deployments, are also gaining popularity.

From a systems perspective, however, serving these coding agents is challenging. Contexts grow over sessions; SLOs are tight; and repeated tool calling places pressure on both the tool-serving infrastructure and the underlying LLM-serving engines. Understanding the characteristics of these workloads is a prerequisite for serving them efficiently.

Existing benchmarks and traces only partially address this. Traces obtained from serving systems like Mooncake [63]

and Splitwise [60] capture real user interactions, but they are not focused on coding agents and lack multi-step tool-call behavior. Capability benchmarks such as Terminal-Bench [51] and SWE-bench [30] measure whether an agent can solve a task. They contain relatively few tasks, each narrowly scoped to a single problem. While they are standard for evaluating model accuracy, they are not designed to capture the diversity of real-world coding-agent usage from a serving system’s perspective.

We aim to help fill this gap. Luckily, coding agents will by default log conversations and tool calls, which serves as a rich source of data for understanding real-world usage. We build a pipeline that extracts the data, normalizes the format, anonymizes it to protect user privacy, and finally runs the analysis to understand the workload characteristics. In this paper, we share initial insights from what is, to our knowledge, the first large-scale cross-provider trace of real coding-agent usage: about 4,300 sessions comprising roughly 350,000 LLM steps and 430,000 tool calls, collected from 43 developers over roughly eight months and spanning both Claude Code and Codex across more than 20 model versions. From our collected traces, we analyze the implications for LLM serving systems.

**At the session level, the workload is largely autonomous and consists of multiple LLM invocations and tool calls..**

To complete a user’s task, the agent will, on average, carry out 8.8 LLM calls and invoke tools 10.8 times. It takes on average 4.3 minutes to complete one request, with long tails whose p90 exceeds 6.4 minutes. For the majority of iterations, the context grows as more user input, tool results, and LLM generations are added. However, two categories of context reduction are observed: a compaction near the context limit and a micro context reduction unique to Codex that occasionally occurs when the user starts their next request.

Due to the multi-step nature of sessions, to avoid repeatedly prefilling the existing conversation history, modern LLM serving systems widely adopt prefix caching [103]. Prefix caching preserves the KV cache for history tokens, so each step only needs to prefill newly appended tokens. Although in API pricing, reading from the prefix cache is roughly 10× cheaper than performing a fresh prefill [5, 56, 57], the accumulated history in one session can become long, and the cache

\*Work done while interning at the University of Washington.

must be read at every step. As a result, we observe that prefix-cache reads dominate the overall API cost, under the pricing snapshot in Table 6.

**Within a step, LLM generation features long context but short output.** Despite our expectation of long generations due to reasoning, the frequently occurring tool calls cut LLM generation into multiple steps, making each step’s LLM invocation have shorter output length than traditional reasoning workloads. The median LLM generation workload is about 119K prefix tokens (prefix cache read of history context), 875 append tokens (fresh new input tokens from user or tool results), and 214 output tokens. Despite the long context, both providers achieve good decode speed. The normalized decode speed has a median of 46.8 normalized tokens per second for Claude and 33.9 for Codex, both with significant variance ( $CV > 50\%$ ). For Codex, the median pure decode speed is 61.3 with an estimated TTFT of 3.1 seconds per step.

**Additionally, tool calls are diverse but heavily tailed for popularity and latency.** While more than 80 distinct tools are observed, the distribution is heavily skewed. The top 3 tools—Bash, Read, and Edit for Claude; `exec_command`, `write_stdin`, and `apply_patch` for Codex—account for more than 80% of all tool calls. The latency of tool calls is also diverse, spanning from milliseconds to hours. Tool calls longer than 1 min are only 4 percent of all tool calls, but they account for 85% of total tool-call time.

**Finally, the prefix cache greatly reduces the cost, but misses are still expensive.** The global prefix cache hit rate is 95.7%, while most misses occur between the last request’s final output and the next request’s user input, as the delay due to human reading, thinking, and typing can frequently exceed the prefix cache eviction time. Overall, cache misses cause 3.8× more tokens to be prefilled than prefills due to truly unique input tokens.

Given these insights, we highlight the following research directions on optimizing LLM serving for coding agents:

1. The overhead of frequent tool-LLM switch motivates denser or fused tool invocations and lower-overhead tool call approval/runtime paths.
2. The mixture of short and long append lengths motivates append-length-aware prefill routing, along with adaptive selection of kernels and serving-engine parallelism.
3. The large latency variation for each tool type suggests that tool-latency prediction for KV cache eviction policy should consider the semantics of the requested operation and recent latency history, not just the tool name.
4. The long prefix motivates sparse attention for reducing the decoding cost.
5. The frequent idle gaps necessitate cheaper KV cache storage, a better compression algorithm, and wise KV-cache eviction, or prefetching policies around those gaps.

Overall, our work provides the first large-scale, cross-provider look into the real-world workload characteristics of coding agents. We release our datasets, trace collection and analysis pipeline, and we are excited to see how the community can use these resources to further understand and optimize for this emerging workload.

## 2 Background and Motivation

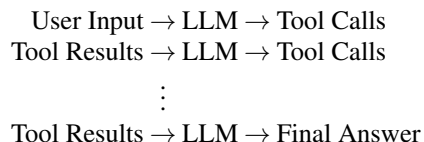
### 2.1 Agentic LLMs and Coding Agents

Agentic LLMs, compared to traditional LLMs, are designed to interact with external tools and environments, perform multi-step reasoning, and thus are more suitable for complex tasks. Coding agents, as a specific type of agentic LLMs, are designed to assist with programming tasks, such as code generation, debugging, and testing. In just a few years, they have become one of the most widely adopted applications of agentic LLMs. The 2025 Stack Overflow Developer Survey reports that 84% of developers now use or plan to use AI tools in their workflow [68], and individual products have reached enormous scale: GitHub Copilot surpassed 20 million users in 2025 [18], Cursor reports more than 7 million monthly active users [6], and Anthropic’s Claude Code reached a \$1 billion annualized revenue run-rate within roughly six months of its public launch [3].

Agentic programming can be organized as a three-level hierarchy consisting of *sessions*, *requests*, and *steps*. A *session* is the highest-level unit of interaction. It maintains the agent’s accumulated context, including the conversation history, the contents of files the agent has read or modified, and the outputs of tool calls. This context persists across user interactions until the session terminates.

Within a session, the user issues one or more *requests*. Each request corresponds to a natural-language task, such as repairing a failing test or adding a command-line option. Requests are processed sequentially, and each request inherits the context accumulated by prior requests in the same session.

A request is typically resolved through an *agentic loop* composed of multiple *steps*. Each step consists of one LLM invocation together with any tool calls it produces. Given the current context, the LLM reasons about the task, may emit intermediate output, and then either invokes one or more tools or returns a final response to the user. When tools are invoked, their results are appended to the context and used as input to the next step. Thus, the first step of a request is *user-initiated*, while subsequent steps are *tool-initiated*. This loop continues until a step returns a final answer, at which point control returns to the user and the next request may begin from the updated session context. Concretely, the agentic loop has the following shape:



## 2.2 Existing Datasets and Benchmarks

Despite the complexity and popularity of coding agents, there is a lack of public datasets that capture their real-world usage and suitable for guiding system optimization.

Widely used workload traces for studying LLM serving systems, such as Mooncake [63], LMSYS-Chat-1M [101], Burst-GPT [77], and Splitwise [60], capture LLM serving traffic at scale, but primarily model traditional usage—human chat, single-turn completion, or short multi-turn interactions—and thus lack the long-horizon, multi-step, tool-calling structure of agentic coding workloads. As a result, they are insufficient for characterizing the workload of coding agents or for evaluating modern serving systems under coding-agent traffic.

Existing benchmarks for coding agents, such as Terminal-Bench [51] and SWE-bench [30], consist of realistic programming tasks but are built for a different goal: evaluating model accuracy on isolated tasks. For example, one Terminal-Bench task asks the agent to “implement an adaptive-rejection sampler,” a well-scoped problem. In practice, however, such a task is only a single *request* among many in the course of building a real project; replaying these benchmarks therefore captures one request in isolation and underestimates the context growth that accumulates over a session. Benchmark tool usage can also deviate from real-world behavior and lacks standardization: some benchmarks restrict the available tools, such as the bash-only setting in SWE-bench, while others leave the permitted toolset unspecified. Because the choice of available tools strongly shapes the tool-call distribution, this can materially change the resulting workload characteristics. Finally, although a few thousand tasks suffice for accuracy evaluation, this scale is too small for modeling the behavior of large serving fleets, particularly those using prefill–decode (PD) or attention–FFN (AF) disaggregation.

In short, neither existing datasets nor benchmarks capture coding agents as *persistent, tool-using systems*—the long sessions, repeated tool calls, accumulated context, and human-paced gaps between requests that ultimately shape serving cost. To close this gap, we collect and analyze a large cross-provider trace of real coding-agent usage spanning many developers, agents, and models over an extended period. Because our dataset is collected from daily research workflows that use coding agents for system building, evaluation, and analysis, it directly captures coding agent behaviors in production-like settings, providing valuable insights for understanding coding-agent workloads and informing the design of future LLM serving systems.

## 3 Trace Format and Data Collection

### 3.1 Raw Agent Logs

By default, both Claude Code and Codex persist a log for each session that records the full interaction between the user and the agent. Although the two systems use different raw schemas, both expose the same logical event types:

- **Metadata:** session-level information such as the model, permission mode, and title.
- **User messages:** user inputs, including the raw input text.
- **Reasoning:** the agent’s internal reasoning steps. The reasoning text is encrypted for both Claude and Codex.
- **Output:** intermediate natural-language output shown to the user, as raw text.
- **Tool calls:** the tools invoked by the agent, with the tool type and raw arguments.
- **Tool results:** the outputs returned by the tools, as raw text.

Every event is timestamped, and each LLM invocation reports its token usage.

### 3.2 Normalization

While Claude and Codex logs describe similar interactions, they differ in event structure, token accounting, and tool-timing metadata. To facilitate analysis, we normalize both into a unified, step-level schema in which each row corresponds to a single step, i.e., one LLM invocation together with the tool calls it produces.

**Token accounting.** From a serving perspective, we decompose the total input tokens of each step,  $I$ , into two components: prefix tokens  $P$ , the history tokens retrieved from the prefix cache, and append tokens  $A$ , the tokens appended to the context in the current iteration, including user messages, tool results, and tokens introduced by prefix-cache misses. The providers report these components as follows:

$$\begin{aligned} P_{\text{Claude}} &= \text{cache\_read\_input\_tokens}, \\ A_{\text{Claude}} &= \text{input\_tokens} \\ &\quad + \text{cache\_creation\_input\_tokens}, \\ P_{\text{Codex}} &= \text{cached\_input\_tokens}, \\ A_{\text{Codex}} &= \text{input\_tokens\_total} \\ &\quad - \text{cached\_input\_tokens}. \end{aligned}$$

We directly use `output_tokens` ( $O$ ) reported by Claude and Codex for total generation length, which already includes reasoning tokens. Codex additionally reports separated `reasoning_output_tokens`, which we use later to characterize Codex’s generation timing.

Throughout the paper we refer to these three categories as prefix tokens ( $P$ ), append tokens ( $A$ ), and output tokens ( $O$ ). **Timing.** Each step also retains an ordered list of timing events—user messages, tool results, reasoning, output text,

<sup>†</sup>A *user-initiated step* is a step whose first input event is a user message. This need not equal the number of user messages: (i) a message the user sends while the agent is still working is delivered together with the next tool result, so that step is counted as tool-initiated; and (ii) a message that never triggers another model call is not counted as a step at all.

Table 1: Summary of the collected coding-agent trace. Token counts are in billions (B) and millions (M); the price column is an estimated API list-price equivalent.

| Metric                          | Claude   | Codex   | Total             |
|---------------------------------|--|---|-------------------|
| <i>Models</i>                   |  |   |                   |
| Model mix (% within provider)   | opus-4-7 63.1%<br>opus-4-6 12.0%<br>haiku-4-5 7.9%<br>sonnet-4-6 7.8%<br>opus-4-8 7.7%<br>Other 1.5% | gpt-5.5 47.5%<br>gpt-5.4 26.0%<br>gpt-5.3-codex 13.7%<br>gpt-5.2-codex 3.4%<br>Other 9.3% | 23 models         |
| <i>Scope &amp; activity</i>     |  |   |                   |
| Sessions                        | 2,676  | 1,589   | 4,265             |
| Distinct users                  | 37   | 22  | 43                |
| Observation window              | Oct 2025–Jun 2026  | Sep 2025–Jun 2026   | Sep 2025–Jun 2026 |
| LLM steps                       | 140,338  | 216,823   | 357,161           |
| Tool calls                      | 142,388  | 290,122   | 432,510           |
| Est. API price equiv.           | ~\$22.7K   | ~\$17.8K  | ~\$40.4K          |
| <i>Tokens</i>                   |  |   |                   |
| Total input tokens              | 28.47 B  | 26.43 B   | 54.90 B           |
| Append tokens                   | 1.19 B   | 1.15 B  | 2.34 B            |
| Prefix tokens                   | 27.28 B  | 25.29 B   | 52.56 B           |
| Output tokens (incl. reasoning) | 96.9 M   | 90.1 M  | 186.9 M           |
| Reasoning tokens                | — (not reported)   | 36.8 M  | —                 |

Table 2: Per-session, per-request, and per-step count distributions across the coding-agent trace.

| Metric                            | Avg   | P25 | P50 | P90 | P99   |
|-----------------------------------|-------|-----|-----|-----|-------|
| <i>Per session</i>                |       |     |     |     |       |
| Requests                          | 9.2   | 1   | 1   | 18  | 137   |
| User-initiated steps <sup>†</sup> | 8.9   | 1   | 1   | 17  | 129   |
| Tool-initiated steps              | 73.6  | 4   | 15  | 135 | 1,107 |
| Tool calls                        | 101.4 | 8   | 25  | 176 | 1,438 |
| <i>Per request</i>                |       |     |     |     |       |
| User-initiated steps              | 0.9   | 1   | 1   | 1   | 1     |
| Tool-initiated steps              | 7.8   | 0   | 1   | 20  | 86    |
| Tool calls                        | 10.8  | 0   | 2   | 30  | 113   |
| <i>Per step</i>                   |       |     |     |     |       |
| Tool calls                        | 1.2   | 1   | 1   | 2   | 4     |

and tool calls—from which we reconstruct its internal timeline. For tool latency, Codex reports the tool execution time explicitly, whereas for Claude we derive it from the tool’s call and return timestamps.

### 3.3 Anonymization

To protect user privacy, we release only a sanitized trace. We replace all session, tool-call, project, and user identifiers with stable pseudonyms, and drop the raw user messages and tool input/output text, keeping only their character counts along with token usage, timestamps, and pseudonymous user identifiers for analysis. We summarize the resulting trace in Table 1.

## 4 Session and Context Management

In this section, we analyze coding-agent macroscopic metrics, including the agentic loop, the context management, and the cost and wall-clock time each session consumes.

### 4.1 Autonomous Loop

First, we characterize the distribution of the number of requests, tool calls, and steps in Table 2. A session on average has more than 9 requests, with a p99 of 137 requests, which highlights its persistent nature. Most of the steps are tool-initiated, so the loop is mostly autonomous. To solve one request, an agent on average takes around 8 steps with 11 tool calls. Each step on average uses slightly more than 1 tool call, suggesting that some tool calling is parallelized, but this phenomenon is not pervasive.

Table 3: Same-session context change, by step trigger. Each value is the per-step change in total input length (prefix tokens + append tokens) versus the previous step in the session.

| Metric                      | Claude  | Codex   |
|-----------------------------|---------|---------|
| <i>All steps</i>            |         |         |
| Steps                       | 137,629 | 210,221 |
| Positive (growth) %         | 99.60%  | 96.56%  |
| Negative (reduction) %      | 0.39%   | 3.43%   |
| Micro %                     | 0.06%   | 1.08%   |
| Ordinary %                  | 0.09%   | 1.73%   |
| Major reduction %           | 0.24%   | 0.62%   |
| Avg positive growth         | 1,719   | 1,838   |
| <i>User-initiated steps</i> |         |         |
| Steps                       | 16,927  | 17,033  |
| Positive (growth) %         | 98.21%  | 68.63%  |
| Negative (reduction) %      | 1.71%   | 31.32%  |
| Micro %                     | 0.37%   | 12.47%  |
| Ordinary %                  | 0.57%   | 18.09%  |
| Major reduction %           | 0.77%   | 0.76%   |
| Avg positive growth         | 1,527   | 2,742   |
| <i>Tool-initiated steps</i> |         |         |
| Steps                       | 120,702 | 193,188 |
| Positive (growth) %         | 99.79%  | 99.03%  |
| Negative (reduction) %      | 0.21%   | 0.97%   |
| Micro %                     | 0.02%   | 0.07%   |
| Ordinary %                  | 0.03%   | 0.29%   |
| Major reduction %           | 0.16%   | 0.61%   |
| Avg positive growth         | 1,746   | 1,783   |

## 4.2 Context Growth and Compaction

Next we investigate the context-management of coding agents. We show the statistics in Table 3. In most cases, context accumulates and positive growth happens, since new user input, tool results, and output all add to the prior context. However, we do observe negative growth. We categorize the reduction into micro-reduction (0–1024 tokens), ordinary reduction (1k–64k tokens), and major reduction (more than 64k tokens). For Claude, reduction overall is very rare, and mostly major reduction. Codex, however, has more frequent reductions, mostly occurring at user-initiated steps, and most of them are micro to ordinary reductions.

One key type of context reduction is *context compaction*, triggered when the context nears the model’s limit: prior context is summarized and the session starts accumulating again from a very short history. We distinguish it from an arbitrary major reduction with three criteria: the total input must (i) drop by at least 64k tokens in one step, (ii) do so while near the session’s peak context (the pre-drop level is at least 75% of the session’s observed maximum), and (iii) recover

Table 4: Context compactions per session: a near-limit total-input drop ( $\geq 64k$ ) that recovers slowly (no rebound to 75% of the pre-drop level within three steps).

| Metric                              | Claude      | Codex         |
|-------------------------------------|-------------|---------------|
| Sessions                            | 2,676       | 1,589         |
| Major reductions ( $\geq 64k$ drop) | 324         | 1,306         |
| of which compactions                | 284 (87.7%) | 1,235 (94.6%) |
| Sessions with $\geq 1$ compaction   | 120 (4.5%)  | 292 (18.4%)   |
| <i>Compactions per session</i>      |             |               |
| Avg (all sessions)                  | 0.106       | 0.777         |
| Avg (sessions with $\geq 1$ )       | 2.37        | 4.23          |
| P90 / P99 (sessions with $\geq 1$ ) | 6 / 12      | 8 / 34        |
| <i>Trigger</i>                      |             |               |
| User-initiated                      | 105 (37.0%) | 100 (8.1%)    |
| Tool-initiated                      | 179 (63.0%) | 1,135 (91.9%) |

slowly, never rebounding to 75% of the pre-drop level within the next three steps. Table 4 reports the result. Most major reductions are genuine compactions (1,519 of 1,630).

Compaction is not rare but not dominant either: 9.7% of sessions undergo at least one, and those that do average 3.7 with a long tail (p99 of 33). It is overwhelmingly tool-initiated (86.5%, i.e. occurring mid-loop) rather than user-initiated, and far more common in Codex (18.4% of sessions) than Claude (4.5%), consistent with Codex’s shorter context length.

## 4.3 Cost Distribution

We now compute the price of each session, request, and step. We use the providers’ published API pricing for the models [5, 56, 57], summarized in Table 6, and categorize tokens into append tokens, prefix tokens, and output tokens. The result is shown in Table 5.

A session costs \$9.70 on average but only \$0.61 at the median, with a heavy tail (p99 of \$178) driven by a few very long sessions; a request costs \$1.01 on average (median \$0.33) and a step \$0.11.

Prefix tokens dominate spend. Although prefix tokens are billed at roughly one-tenth the fresh-input rate, their volume—the accumulating context replayed on every step—makes them 59.5% of total cost, versus 29.2% for append tokens, including Claude 5-minute cache-write charges, and only 11.2% for output tokens. Output tokens are cheap in aggregate despite their high per-token price simply because each step emits few tokens (median 214). This inverts the usual intuition that generation is the expensive part: for coding agents the cost is overwhelmingly in re-reading context.

## 4.4 Timing Distribution

Finally, we break down wall-clock time in Table 7. We decompose time into three components: *human thinking* (the

Table 5: Per-session, per-request, and per-step cost (USD) by category.

| Metric             | Avg    | P50    | P90    | P99    | % cost |
|--------------------|--------|--------|--------|--------|--------|
| <i>Per session</i> |        |        |        |        |        |
| Total              | \$9.70 | \$0.61 | \$13.4 | \$178  |        |
| Append tokens      | \$2.83 | \$0.24 | \$3.62 | \$46.5 | 29.2%  |
| Prefix tokens      | \$5.77 | \$0.17 | \$7.55 | \$111  | 59.5%  |
| Output tokens      | \$1.09 | \$0.16 | \$1.92 | \$16.9 | 11.2%  |
| <i>Per request</i> |        |        |        |        |        |
| Total              | \$1.01 | \$0.33 | \$2.44 | \$9.33 |        |
| Append tokens      | \$0.29 | \$0.04 | \$0.73 | \$3.91 | 29.2%  |
| Prefix tokens      | \$0.60 | \$0.16 | \$1.35 | \$6.51 | 59.5%  |
| Output tokens      | \$0.11 | \$0.03 | \$0.29 | \$1.08 | 11.2%  |
| <i>Per step</i>    |        |        |        |        |        |
| Total              | \$0.11 | \$0.07 | \$0.20 | \$0.72 |        |
| Append tokens      | \$0.03 | \$0.00 | \$0.03 | \$0.68 | 29.2%  |
| Prefix tokens      | \$0.07 | \$0.05 | \$0.12 | \$0.41 | 59.5%  |
| Output tokens      | \$0.01 | \$0.01 | \$0.03 | \$0.12 | 11.2%  |

Table 6: API pricing snapshot used for cost accounting. Rates are USD per million tokens from provider pricing pages [5, 56, 57]; Claude cache-write rates use the 5-minute prompt-cache tier.

| Provider | Model group  | Input | 5m write | Cache read | Output |
|----------|--------------|-------|----------|------------|--------|
| Claude   | Opus 4.6–4.8 | 5.00  | 6.25     | 0.50       | 25.00  |
| Claude   | Sonnet 4.6   | 3.00  | 3.75     | 0.30       | 15.00  |
| Claude   | Haiku 4.5    | 1.00  | 1.25     | 0.10       | 5.00   |
| Codex    | GPT-5.5      | 5.00  | –        | 0.50       | 30.00  |
| Codex    | GPT-5.4      | 2.50  | –        | 0.25       | 15.00  |

gap from the previous event to the next user message), *LLM generation* (the observed time for the model to generate reasoning, output, and tool inputs), and *tool execution* (effective tool latency). Because human thinking happens *between* requests, request response time contains only generation and tool execution; the final block reports the individual positive human waits, generation spans, and tool latencies that match the corresponding CDF/summary distributions.

From the table, the coding session is mostly idle waiting on the human: human thinking is 92.3% of session wall-clock, surpassing LLM generation (3.3%) and tool execution (4.8%). Most sessions are short—the median session is a single request with no inter-request gap—but a heavy tail of sessions left open for hours or days (session p99 elapsed of ~206h) accumulates most of the idle time. In the individual latency view, positive human-input waits have median 1.4m and p90 20.6m; observed LLM-generation spans have median 5.7s and

Table 7: Per-session, per-request, per-step, and individual latency wall-clock time by category.

| Metric                                | Avg   | P50   | P90   | P99    | % time |
|---------------------------------------|-------|-------|-------|--------|--------|
| <i>Per session</i>                    |       |       |       |        |        |
| Total elapsed                         | 8.2h  | 5.1m  | 5.9h  | 206.5h |        |
| Human thinking                        | 7.6h  | 0.0s  | 4.1h  | 190.0h | 92.3%  |
| LLM generation                        | 16.1m | 2.0m  | 26.7m | 3.9h   | 3.3%   |
| Tool execution                        | 23.5m | 14.9s | 26.1m | 6.9h   | 4.8%   |
| <i>Per session, human capped (1h)</i> |       |       |       |        |        |
| Total                                 | 1.8h  | 5.5m  | 3.4h  | 27.2h  |        |
| Human thinking                        | 1.2h  | 0.0s  | 2.3h  | 18.3h  | 64.3%  |
| LLM generation                        | 16.1m | 2.0m  | 26.7m | 3.9h   | 14.5%  |
| Tool execution                        | 23.5m | 14.9s | 26.1m | 6.9h   | 21.2%  |
| <i>Per request</i>                    |       |       |       |        |        |
| Total (response time)                 | 4.3m  | 38.3s | 6.4m  | 43.9m  |        |
| LLM generation                        | 1.7m  | 28.8s | 3.7m  | 13.6m  | 41.0%  |
| Tool execution                        | 2.5m  | 0.3s  | 2.3m  | 34.4m  | 59.8%  |
| <i>Per step</i>                       |       |       |       |        |        |
| LLM generation                        | 11.5s | 4.9s  | 20.2s | 1.3m   | 40.7%  |
| Tool execution                        | 16.8s | 0.1s  | 10.0s | 3.2m   | 59.3%  |
| <i>Per individual latency</i>         |       |       |       |        |        |
| Human thinking                        | 46.7m | 1.4m  | 20.6m | 13.9h  |        |
| LLM generation                        | 13.2s | 5.7s  | 22.2s | 1.4m   |        |
| Tool execution                        | 18.4s | 0.3s  | 13.6s | 3.6m   |        |

p90 22.2s; and positive tool latencies have median 0.3s and p90 13.6s. When we cap each idle gap at one hour (the *human capped* block), the human share of the now cache-relevant budget falls to 64.3%, with generation and tool execution rising to 14.5% and 21.2%.

Within an individual request, tool execution and generation both contribute to the response time: of the total 2,783 hours of response time, tools account for 59.8% versus 41.0% for generation. An average request takes 4.3 minutes end to end (median 38s, p90 6.4 min).

## 4.5 Takeaways and Systems Opportunities

### Takeaways

- **Autonomous, heavy-tailed sessions.** The loop is largely self-driving—most steps are tool-initiated—and session length is heavy-tailed, with a few very long sessions dominating.
- **Compaction is rare.** Major context reduction is rare; when it happens it is usually a genuine compaction near the context limit, with Codex additionally showing frequent *micro*-reductions that Claude does not.
- **Cost spent on prefix tokens.** Prefix tokens, not genera-

Table 8: Per-step prefix, append, and output token length distribution.

| Tokens               | Avg     | P25    | P50     | P90     | P99     |
|----------------------|---------|--------|---------|---------|---------|
| <i>Prefix tokens</i> |         |        |         |         |         |
| Claude               | 194,361 | 59,949 | 126,180 | 467,082 | 918,111 |
| Codex                | 116,623 | 67,456 | 115,584 | 201,600 | 231,040 |
| <i>Append tokens</i> |         |        |         |         |         |
| Claude               | 8,479   | 384    | 857     | 5,342   | 232,206 |
| Codex                | 5,283   | 406    | 886     | 8,310   | 121,009 |
| <i>Output tokens</i> |         |        |         |         |         |
| Claude               | 690     | 132    | 252     | 1,671   | 6,571   |
| Codex                | 415     | 70     | 184     | 939     | 3,508   |

tion, account for the majority of cost.

- **Human-bound sessions.** Idle time waiting on the human dominates the session end-to-end time, while within a request tool execution and generation contribute comparable time.

### Systems Opportunities

- **Denser tool calling.** Tool calls are frequent, yet most steps issue only one. Encouraging more tool calls per step—or *fusing* them—would amortize the LLM↔tool scheduling overhead and expose more tool-level parallelism.
- **Lower cache costs.** Because cache reads are a leading cost driver, the prior context is worth storing and reloading more efficiently, including cost-effective KV-cache storage hardware and infrastructure, and sparse-attention schemes that shrink the prefix loading cost.

## 5 LLM generation

Next, we focus on the LLM generation and investigate the input, output and timing of the generation.

### 5.1 Input length distribution

First, we investigate the per-step input length, split into prefix tokens (the replayed accumulated context) and append tokens (the freshly added, uncached input). Table 8 shows the distribution for both providers. A median Claude step reads back 126k prefix tokens but appends only 857, while Codex reads 116k and appends 886—roughly two orders of magnitude more prefix tokens than append tokens. Because Claude has a longer context length, its prefix stretches to a p99 of 918k tokens, while Codex saturates near 231k.

In Figure 1, we further analyze the relationship between prefix and append lengths. Most of the data points fall in prefix length 32k-128k and append 256-8k. We can also see two major groups, one group, with short prefix length (<16k)

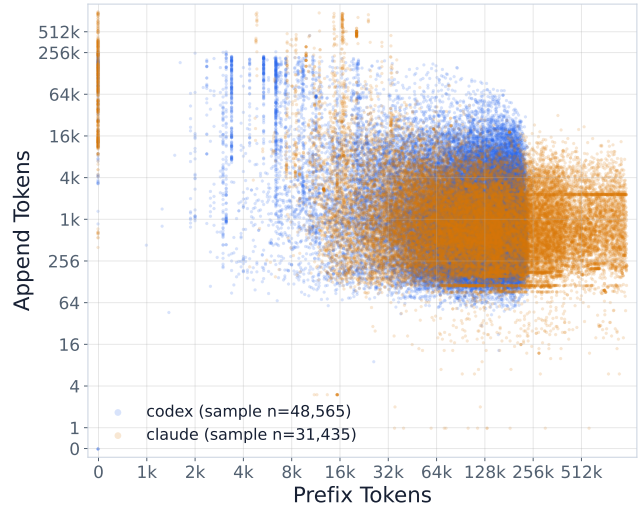


Figure 1: Per-step prefix tokens vs. append tokens.

mainly have relatively large append tokens. They mostly represent prefix cache miss or initial prefills. The other group, with longer prefix, having much shorter append tokens, representing normal context growth due to tool result or user input.

Table 9 quantifies this split. When there are few prefix tokens the append tokens dominate—at a prefix below 1k the median append is 78k tokens for Claude and 124k for Codex—whereas once the prefix grows past 32k the median collapses to well under 1k, as those steps carry only an incremental tool result or user turn.

While long-append steps are rare, they contribute the majority of total prefill workload. Figure 2 shows this effect. For each provider, the upper bar is the share of *steps* and the lower bar the share of total *append tokens*, split by per-step append length. Over 90% of steps append fewer than 1k tokens, yet more than 70% of all append tokens come from the rare steps that append 10k or more.

### 5.2 Output length distribution

In contrast to the large inputs, outputs are short. Table 8 shows a median of only 252 output tokens for Claude and 184 for Codex, with the p90 still under 1.7k. Even at the p99, output tokens stay in the few-thousand-token range, far below the input sizes. This is counterintuitive, by can be explained by the frequent tool calls. As we show in Section 4.1, one full response are effectively cut into average 8 tool call steps, thus the individual output length are short, and sometimes only generates next tool call parameters.

Figure 4 shows the full per-step output distribution: both providers concentrate in the low hundreds of tokens, and Codex additionally has a pronounced spike of very short (~40-token) generations, due to its popular tool call `write_stdin`, that wait for prior command to finish or sending `Ctrl+C` for

Table 9: Append tokens per step, conditioned on the number of prefix tokens, for each provider.

| Prefix        | Steps  | Avg    | P50    | P90    | P99    |
|---------------|--------|--------|--------|--------|--------|
| <i>Claude</i> |        |        |        |        |        |
| <1k           | 2,937  | 136.1K | 78.4K  | 344.3K | 871.9K |
| 1–2k          | 0      | –      | –      | –      | –      |
| 2–4k          | 2      | 2.7K   | 2.7K   | 3.8K   | 4.0K   |
| 4–8k          | 530    | 122.0K | 17.2K  | 385.3K | 881.3K |
| 8–16k         | 4,034  | 38.8K  | 3.5K   | 108.5K | 549.1K |
| 16–32k        | 10,248 | 35.4K  | 1.3K   | 30.7K  | 661.0K |
| 32–64k        | 20,919 | 2.8K   | 951    | 5.3K   | 27.3K  |
| 64–128k       | 33,571 | 1.6K   | 793    | 3.7K   | 12.8K  |
| 128–256k      | 34,840 | 1.4K   | 710    | 3.2K   | 10.1K  |
| >256k         | 33,257 | 1.4K   | 762    | 3.1K   | 8.8K   |
| <i>Codex</i>  |        |        |        |        |        |
| <1k           | 626    | 116.3K | 124.3K | 210.7K | 247.0K |
| 1–2k          | 90     | 22.2K  | 4.3K   | 63.9K  | 192.6K |
| 2–4k          | 2,108  | 56.4K  | 20.8K  | 168.7K | 240.8K |
| 4–8k          | 3,501  | 60.2K  | 25.7K  | 172.4K | 220.7K |
| 8–16k         | 5,503  | 22.8K  | 2.9K   | 84.7K  | 195.5K |
| 16–32k        | 10,470 | 9.6K   | 1.9K   | 18.8K  | 152.2K |
| 32–64k        | 29,925 | 3.7K   | 954    | 8.3K   | 50.4K  |
| 64–128k       | 72,996 | 2.7K   | 796    | 6.1K   | 31.3K  |
| 128–256k      | 91,598 | 2.2K   | 771    | 5.3K   | 21.0K  |
| >256k         | 6      | 750    | 900    | 1.1K   | 1.1K   |

interruption.

### 5.3 Output token attribution

Next, we investigate how a prior step’s output tokens are accounted for in the next step’s prompt. If the serving system saves the newly produced KV-cache entries during generation, then the next step can reuse the prior step’s whole prompt composition: prefix tokens, append tokens, and output tokens. We call this case *output-cached*. However, we also observe cases where the next step’s prefix excludes the prior output and instead re-sends it as part of the next step’s append tokens. We call this case *output-resend*.

Figure 5 shows the two accounting schemes schematically. In Figure 5(a), the prior step has 10 units of prefix, 2 units of append, and 4 units of output; all 16 units become the next step’s prefix, followed by that step’s own append and output. In Figure 5(b), only the prior prefix and append are cached ( $10 + 2 = 12$ ). The prior output is re-sent as part of the next step’s append; with one additional unit of fresh input, the next append becomes  $4 + 1 = 5$ .

We distinguish the two cases using their key invariants. Under output-cached, the next step’s prefix gain—the next prefix minus the prior prefix and append—should track the prior output. Under output-resend, the next prefix should remain

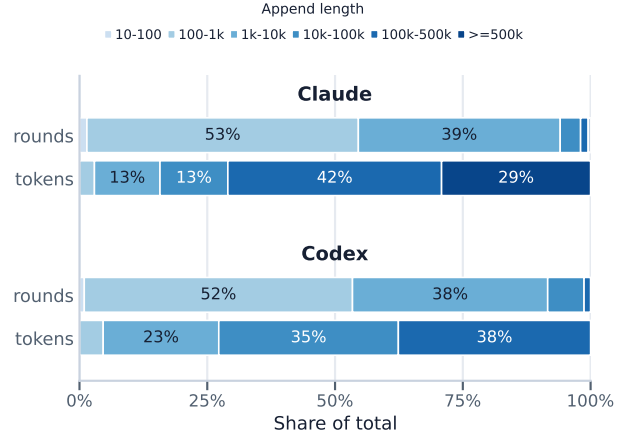


Figure 2: Per-step append length by step count and by total append-token share, split by provider.

close to the sum of the prior prefix and append, while the next step’s append should include the prior output. Figure 3 applies this test separately within each model and focuses on prior outputs of at least 2k tokens, where the signal is large enough to be visible. The figure indicates that Claude primarily uses output-resend. Codex, however, changes behavior across versions: gpt-5.4 is mostly output-cached, while gpt-5.5 is mostly output-resend.

We hypothesize that this difference reflects KV-cache pool management choices in PD-disaggregated serving. The output-cached policy requires transferring the KV entries produced during decode back to the shared KV-storage backend. The next prefill instance can then load those entries, extend the cache for the next prompt, write the updated KV entries back to storage, and hand them off to the decode instance. This path is more complex, but it avoids re-prefilling the prior output. The output-resend policy, in contrast, only requires prefill instances to write to the shared KV-storage backend. Decode instances are read-only with respect to shared KV state: they fetch the prefix KV needed for generation, but discard the KV entries for newly generated output tokens.

### 5.4 Timing

We next examine trace-observed LLM generation latency. Let  $t_{\text{input}}$  be the latest input-event timestamp (`user_message` or `tool_result`) before the first model output, and let  $t_{\text{last}}$  be the last model-output timestamp (`reasoning`, `text`, or `tool_call`). For a step with provider-reported output tokens  $O$ , we define normalized decode speed as

$$s_{\text{norm}} = \frac{O}{t_{\text{last}} - t_{\text{input}}}.$$

This is an end-to-end trace speed: it includes TTFT, reasoning, visible output emission, and trace logging effects.

For Codex steps with exact reasoning-token accounting,

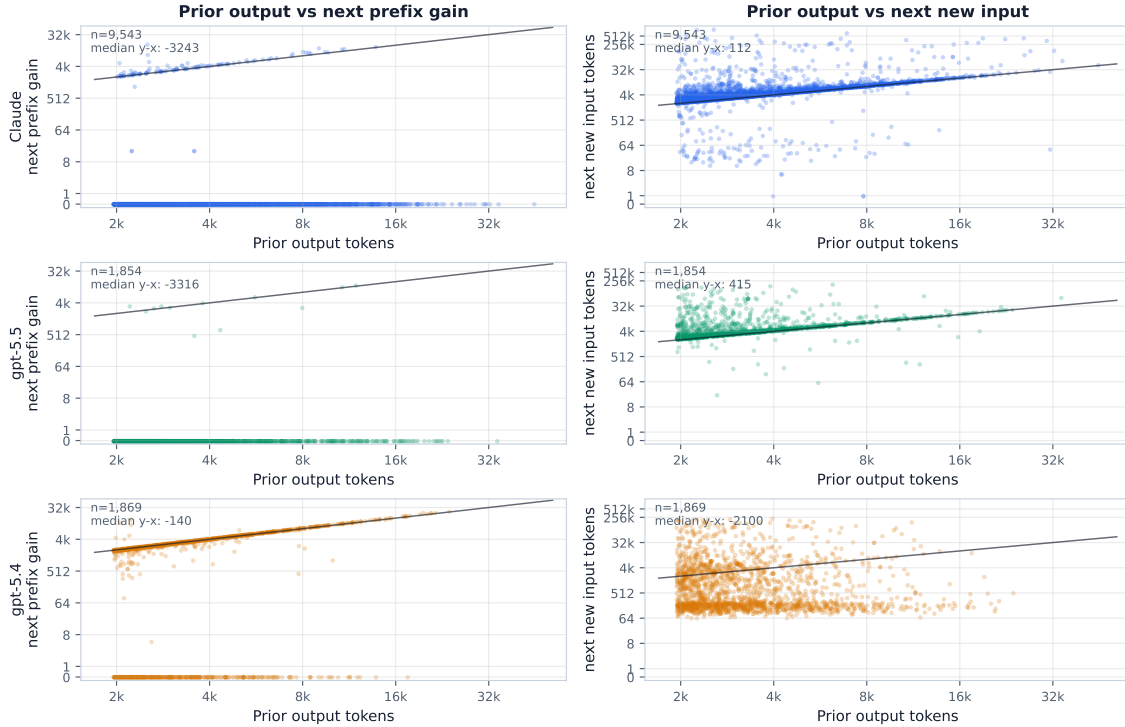


Figure 3: Merged output-attribution evidence by model for previous outputs of at least 2k tokens. Left: prior output versus next-step prefix gain; right: prior output versus next-step append tokens.

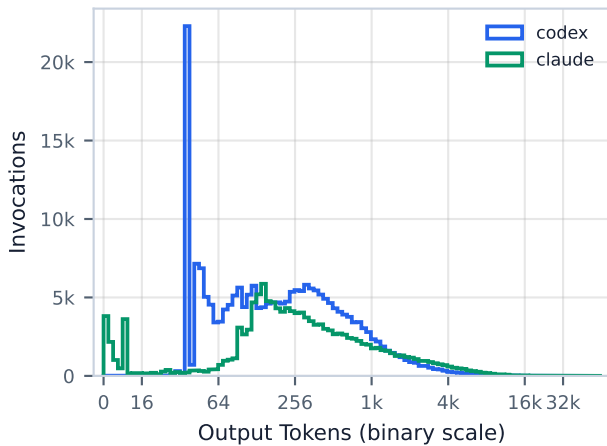


Figure 4: Per-step output-token distribution by provider.

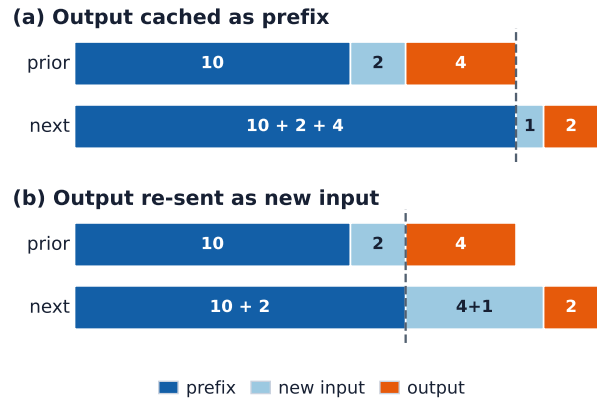


Figure 5: Two ways a prior step's output can be accounted in the next step (schematic; bar lengths illustrative).

We then estimate pure decode speed and residual TTFT as

$$s_{\text{pure\_decode}} = \frac{O_{\text{visible}}}{t_{\text{visible}} - t_{\text{reason}}},$$

$$\hat{\ell}_{\text{pure\_decode}} = \frac{\sum (t_{\text{visible}} - t_{\text{reason}})}{\sum O_{\text{visible}}},$$

$$\widehat{\text{TTFT}} = (t_{\text{reason}} - t_{\text{input}}) - O_{\text{reason}} \hat{\ell}_{\text{pure\_decode}}.$$

Figure 6 plots these timing metrics against the total input

let  $O_{\text{reason}}$  be reasoning tokens,  $O_{\text{visible}} = O - O_{\text{reason}}$  be non-reasoning output tokens,  $t_{\text{reason}}$  be the reasoning timestamp, and  $t_{\text{visible}}$  be the last non-reasoning model-output timestamp.

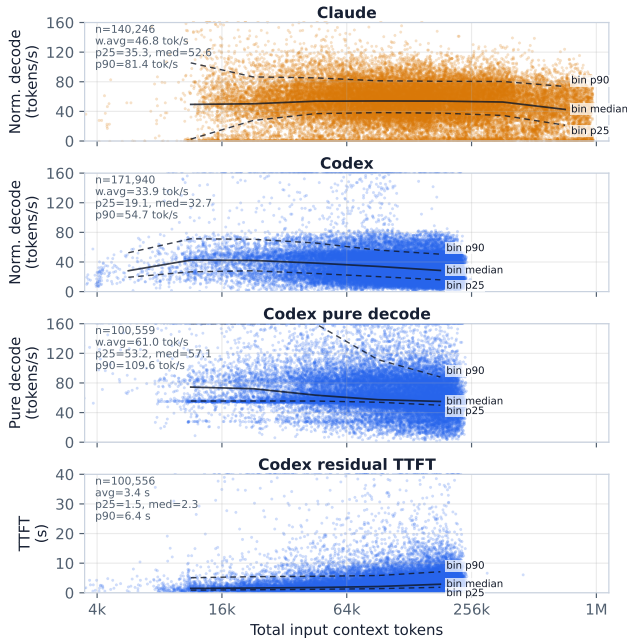


Figure 6: Trace-observed LLM timing versus total input context length.

context length. The median step is 46.8 and 33.9 normalized tokens/s for Claude and Codex, respectively, but the per-step variance is large at every context length. Claude’s binned median stays near 50–54 tokens/s through most of its range and drops only at the longest contexts, reaching roughly 43 tokens/s around 740k input tokens. Codex shows a clearer context-length trend: its binned median falls from about 43 tokens/s around 12k–23k input tokens to about 29 tokens/s near 185k. In the Codex-only panels, median pure decode speed falls from about 74 tokens/s around 12k input tokens to about 55 tokens/s near 185k, while median residual TTFT increases from about 1.5 s to 2.9 s over the same range. Thus, longer context is associated with slower observed generation, especially for Codex, but context length alone does not explain the wide spread; scheduling, model version, output shape, and backend state likely also contribute.

## 5.5 Takeaways and Systems Opportunities

### Takeaways

- **Long context, short output.** A typical step reads on the order of 100k (mostly cached) input tokens but emits only hundreds of output tokens.
- **Bimodal input structure.** Steps with short prefixes tend to carry large append tokens, whereas when the prefix is long, the append tokens are usually incremental.
- **Mixed output attribution.** Claude and GPT-5.5 primarily use resend output as next round input, whereas GPT-5.4 primarily treats output as part of next round’s prefix.

- **Noisy decoding speed.** Longer contexts are associated with slower observed generation, especially for Codex, but the variance remains large even at similar context lengths.

### Systems Opportunities

- **Append-aware prefill optimization.** The bimodal input structure suggests two different optimization targets. Rare long-append steps should be optimized for prefill throughput, while the common short-append steps may need another path for latency optimizations.
- **Output-cache policy design.** The coexistence of output-cached and output-resend behavior exposes a tradeoff between transferring decode-produced KV entries back to shared storage and re-prefilling prior outputs in the next step. Serving systems should choose this policy based on output length, cache pressure, and KV cache infra.
- **Dedicated TTFT optimization.** The significant residual TTFT for Codex suggests that better autoscaling, request routing, and decode admission control could meaningfully reduce end-to-end latency, especially for this multi-step coding-agent workflows.

## 6 Tool Calls

In this section, we analyze coding agents’ tool-calling behavior, including tool call counts, latency, and human interactions.

### 6.1 Tool Call Count Distribution

We first investigate tool call counts. In the trace, we observe 54 different tools for Claude and 31 for Codex. Tool popularity is shown in Figure 7. The most common tool is command execution for both models, followed by file operations such as Read and Edit. For Claude, the top three tools account for over 80% of calls, while for Codex they account for 95%.

### 6.2 Tool Call Latency Distribution

Next, we analyze tool call latency. Figure 9 shows the latency distribution for the top 12 most common tools for both Claude and Codex. Overall, average tool-call latency is 16.8s, but latency depends heavily on tool type. For example, file operations such as Read and Edit are mostly short, on the order of milliseconds to seconds, while Agent and AskUserQuestion can take minutes to finish. Even within each tool type, however, latency varies significantly. For example, Claude’s Bash tool ranges from milliseconds to minutes, although its median is below one second.

Tool-call latency has a strong long tail. Figure 8 compares, for each latency bin, its share of tool calls with its share of total tool-call time. For Claude, calls under 1 s account for 70% of calls but less than 1% of total tool time, while calls longer than 1 min are only 4.9% of calls but contribute 92% of total tool time. Codex is less extreme but still dominated

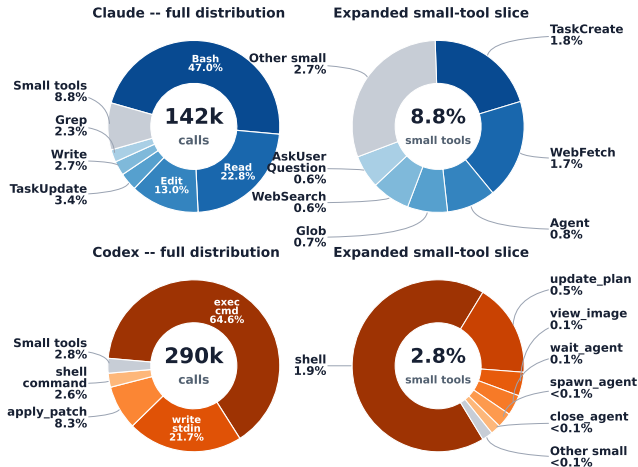


Figure 7: Tool call count distribution for Claude and Codex. Command execution dominates both, followed by file operations such as read and edit.

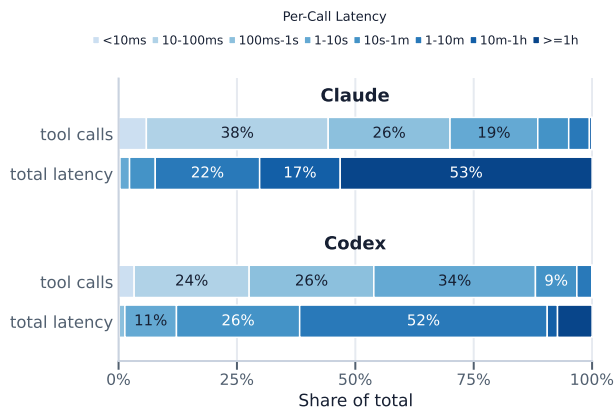


Figure 8: Tool-call latency bins by call share and by total-latency share, split by provider.

by slow calls: calls under 10s account for 88% of calls but only 12% of time, while calls longer than 1 min are 3.1% of calls and 61% of time.

### 6.3 Tool Call Overhead

Codex exposes two timing views for most tool calls: the trace-observed end-to-end span from tool emission to tool result, and the runner-reported internal execution time. For each call we defining overhead as  $R = \max(0, T_{e2e} - T_{int})$ . This captures time around a tool call that is not reported as internal execution, which can include permission approval, runtime scheduling, shell startup, output propagation, client-side book-keeping, etc.

Table 10 reports this overhead for the 253,391 Codex tool calls with valid end-to-end and internal timings, covering 87.3% of Codex tool calls. These calls account for 418.1h end-

Table 10: Codex end-to-end and internal tool latency, with positive residual statistics.

| Tool          | Calls | E2E    | Int.   | Res.  | Avg   | P50/90/99       |
|---------------|-------|--------|--------|-------|-------|-----------------|
| All timed     | 253k  | 418.1h | 341.5h | 77.8h | 1.11s | 0.13/0.24/10.0s |
| exec_command  | 184k  | 97.8h  | 33.8h  | 64.1h | 1.26s | 0.16/0.27/8.6s  |
| write_stdin   | 61k   | 314.6h | 303.7h | 11.7h | 0.69s | 0.01/0.10/14.9s |
| shell_command | 6.1k  | 5.3h   | 3.8h   | 1.8h  | 1.08s | 0.04/0.16/4.0s  |
| apply_patch   | 2.5k  | 0.5h   | 0.3h   | 0.21h | 0.30s | 0.01/0.41/3.9s  |

to-end and 341.5h of runner-reported internal execution, leaving 77.8h of residual (18.6%). The residual is dominated by exec\_command, which contributes 64.1h. The average residual is 1.11s, and the p50/p90 remain small (0.13s/0.24s), but the p99 reaches 10.0s. Thus, aggregate overhead comes from many small gaps plus a long tail. Our hypothesis is that bash commands more likely require human or auto approval, which enlarges the p99 latency. Claude does not expose comparable internal timing coverage, so we do not make the same quantitative decomposition for Claude.

## 6.4 Takeaways and System Opportunities

### Takeaways

- **Heavily skewed tool call counts and latency.** A small number of tools dominate call counts, and a small number of slow calls dominate total tool-call time.
- **Per-type latency variation** Tool type can guide latency prediction, but there is still significant variation within each tool type.
- **Considerable overhead** For Codex, observed end-to-end tool latency can substantially exceed runner-reported internal execution time, indicating non-execution overheads.

### System Opportunities

- **Semantic-based latency prediction** When predicting tool-call latency for prefix cache management, tool type alone is insufficient. The large variation within each tool type suggests using semantic information about the requested operation, together with recent latency history. [74]
- **Low overhead tool calling** Developing low-latency auto-approval and reducing tool-call framework overhead can reduce tool call overhead, which is a non-negligible part of tool call latency and overall session latency.

## 7 Prefix Cache

In this section, we examine a key component of LLM serving for coding agents: the prefix cache. We analyze its hit rate and how it relates to idle time. Then we characterize the gap

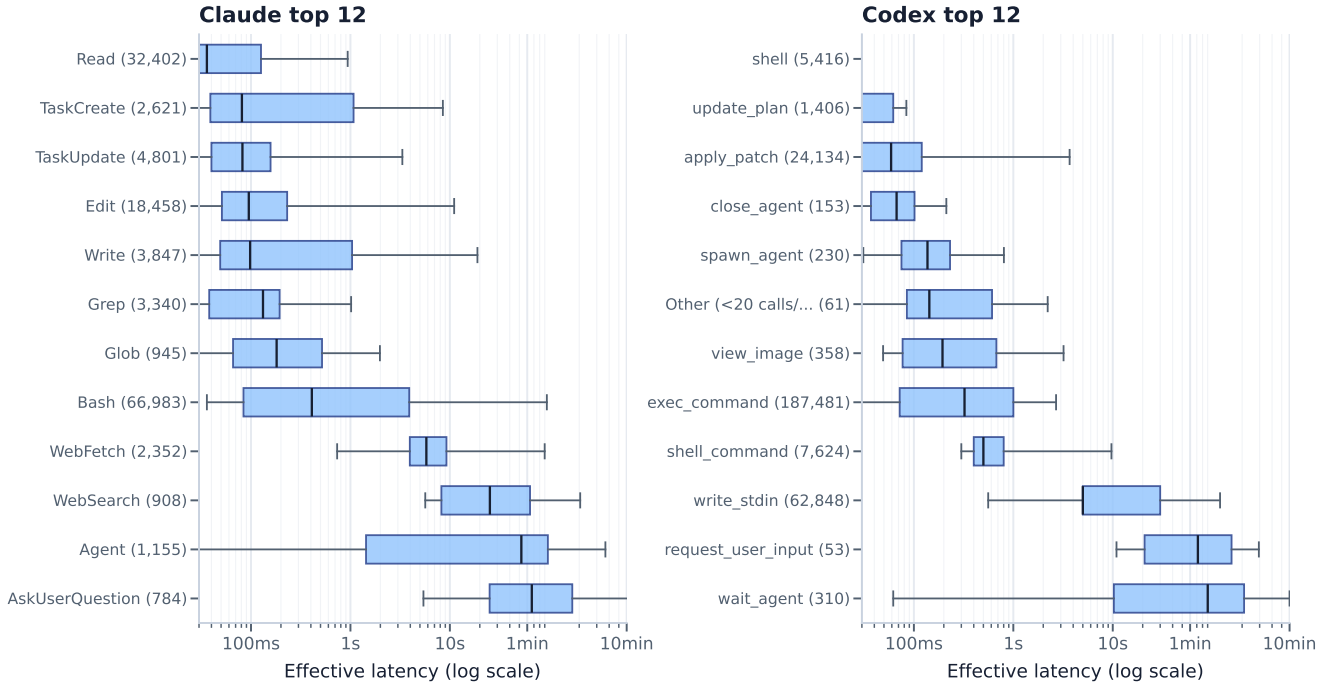


Figure 9: Effective tool-call latency for the top 12 tools by call count in each provider. Boxes show the interquartile range, center lines show medians, and whiskers show p5–p95.

Table 11: Token-weighted prefix cache hit rate by provider and step trigger.

| Metric                           | Claude | Codex | Total |
|----------------------------------|--------|-------|-------|
| Prefix cache-hit rate            | 95.8%  | 95.7% | 95.7% |
| Prefix hit rate (user-initiated) | 86.9%  | 78.2% | 84.4% |
| Prefix hit rate (tool-result)    | 97.9%  | 97.2% | 97.5% |

to the optimal, the trade-off between KV storage space and the eviction time of cached prefixes, and the consumer-visible cost of human thinking time.

### 7.1 Per-Step Prefix Cache Hit Rate

To begin with, we break down the cache hit rate by step type. Table 11 shows that prefix caching is consistently high overall: both Claude and Codex serve about 96% of prompt tokens from the prefix cache. The main misses are user-initiated steps, where human thinking time can make the idle time long enough to trigger eviction; tool-result steps remain near-perfect because they usually resume shortly.

In Figure 10, we further illustrate the prefix cache hit rate over an example session. Most steps are initiated close together in time, so the prefix cache retains the context. At step 28, a cache miss occurs due to 10 minutes of human inactivity, causing a large fresh prefill.

### 7.2 Prefix Cache Eviction Time

Next, we dive into the prefix cache eviction time. We measure the idle time between steps and plot the corresponding prefix cache hit rate in Figure 11. For both Claude and Codex, long gaps are significantly more common for user-initiated steps than for tool-result steps. When the gap is larger than 5 minutes, low-hit-rate steps begin to appear, and after 1 hour, almost all steps miss the cache. Tool gaps, especially for Codex, rarely exceed 5 minutes, since Codex moves long-running tasks into the background.

### 7.3 Gap to Optimal and Redundant Prefill

While the overall cache hit ratio is high, there is still a gap to optimal. Specifically, the upper bound of the prefix cache hit rate is determined by the user prompts and tool results, since these are truly fresh tokens the system has never seen before. To measure them precisely, we take the total context growth and subtract the output tokens. Table 12 reports these fresh tokens against the total append tokens. Across all steps, fresh tokens are only 19.0% of append tokens (12.3% for Claude, 25.8% for Codex), meaning the remaining ~81% of prefill can, in principle, be served from cache, marking the gap to optimal. The split is sharply step-dependent: user-initiated steps are almost entirely re-sent context (fresh is just 1.7% of their append tokens for Claude and 4.5% for Codex), due to their lower cache hit rate, whereas tool-result steps carry more of the genuinely new content (27.1% and 40.5%,

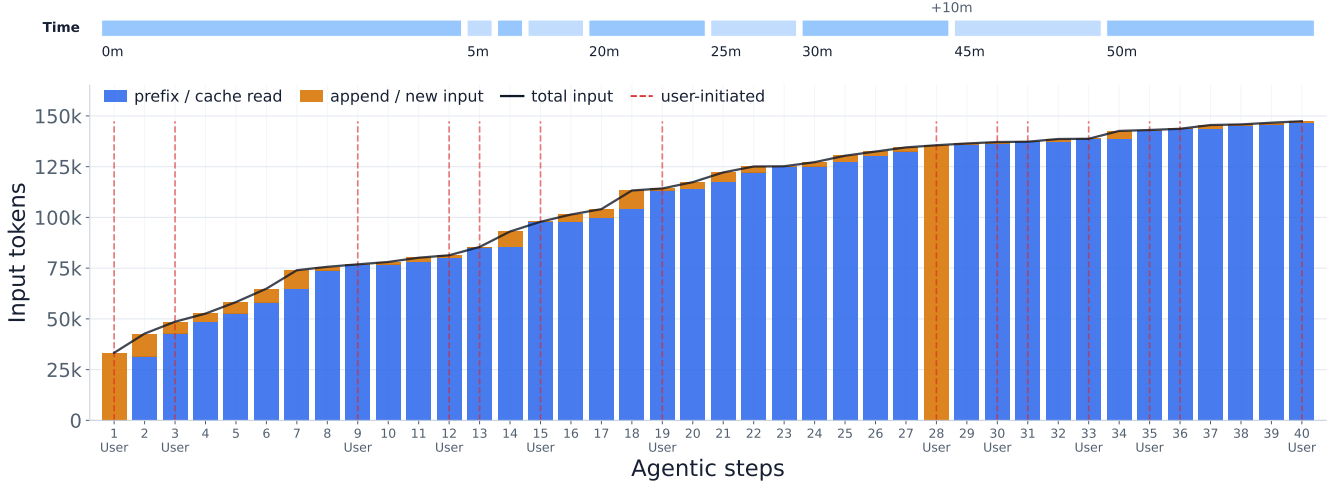


Figure 10: Example session progression over the first 40 agentic steps. Bars decompose each step input into prefix tokens and append tokens; the black line shows total input, red dashed lines mark user-initiated steps, and the top strip shows elapsed time.

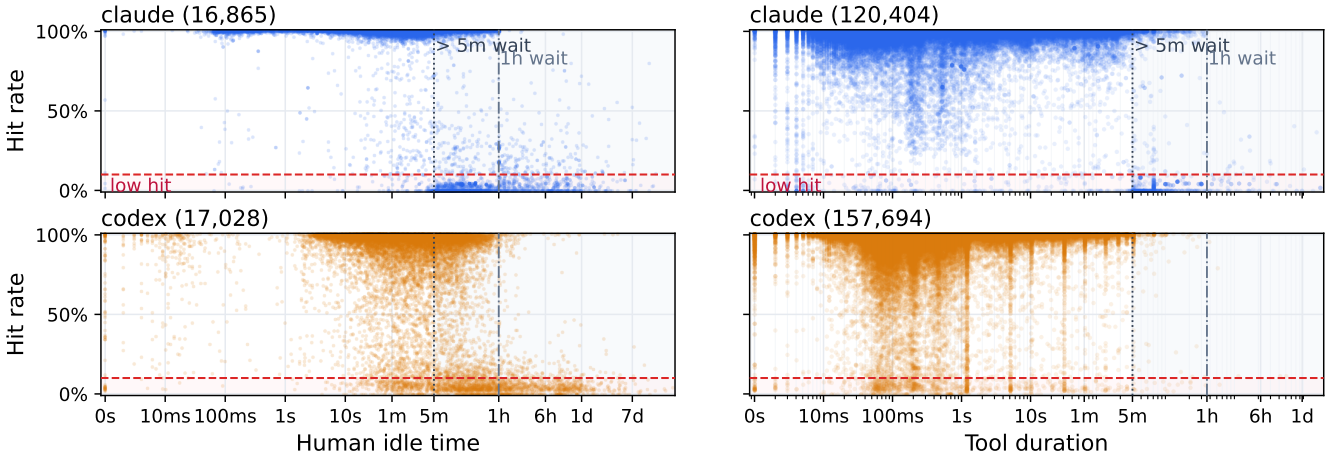


Figure 11: Prefix cache hit rate versus the idle time preceding a step (log  $x$ -axis), for Claude (top) and Codex (bottom).

respectively).

We further compute the *prefill amplification factor*, the ratio of total prefilled tokens to the irreducible fresh tokens—equivalently  $1/(\text{fresh \% of append})$ —i.e., how many times more tokens are prefilled than a perfect, eviction-free cache would require. The deployed cache amplifies prefill by  $5.3\times$  overall ( $8.1\times$  for Claude,  $3.9\times$  for Codex), flagging an optimization opportunity.

#### 7.4 Storage Trade-off

We next examine the trade-off between the prefix cache storage space and the hit rate. Based on the observed human thinking time, tool latency, and LLM generation time, we vary the cache eviction time and compute the corresponding hit rate and storage space. With a global request batch  $B$ , the fraction of request that is actively decoding is approximately

$B \cdot T_{\text{generation}} / (\hat{T}_{\text{human}} + \hat{T}_{\text{tool}} + T_{\text{generation}})$ , where  $T_{\text{generation}}$  is the average LLM generation time, and  $\hat{T}_{\text{human}}$  and  $\hat{T}_{\text{tool}}$  are the average human thinking time and tool latency, capped by the eviction time. This indicates that the ratio of the suspended requests’ KV cache storage to that of the active decoding requests is approximately

$$R = \frac{\hat{T}_{\text{human}} + \hat{T}_{\text{tool}}}{T_{\text{generation}}}.$$

We sweep the eviction time and derive the corresponding achievable hit rate and prefill amplification under an idealized rule: a step is a complete cache miss when its preceding idle gap exceeds the eviction time, and a full hit otherwise. Figure 12 plots all three quantities against a shared eviction-timeout axis. Raising the timeout from 1 min to 1 h lifts the achievable hit rate from 85.4% to 98.6%, but increases the

Table 12: Fresh prefill versus total append tokens, by provider and step trigger. *Fresh* tokens are the per-step context growth minus the prior step’s output tokens; *prefill amplification* is append tokens/fresh.

| Metric                | Claude   | Codex    | Total    |
|-----------------------|----------|----------|----------|
| <i>Overall</i>        |          |          |          |
| Total append tokens   | 1,154.0M | 1,115.6M | 2,269.6M |
| Total fresh tokens    | 142.1M   | 288.1M   | 430.2M   |
| Fresh % of append     | 12.3%    | 25.8%    | 19.0%    |
| Prefill amplification | 8.1×     | 3.9×     | 5.3×     |
| <i>User-initiated</i> |          |          |          |
| Total append tokens   | 672.1M   | 454.7M   | 1,126.8M |
| Total fresh tokens    | 11.4M    | 20.4M    | 31.8M    |
| Fresh % of append     | 1.7%     | 4.5%     | 2.8%     |
| Prefill amplification | 58.8×    | 22.3×    | 35.4×    |
| <i>Tool-result</i>    |          |          |          |
| Total append tokens   | 481.8M   | 661.0M   | 1,142.8M |
| Total fresh tokens    | 130.7M   | 267.7M   | 398.4M   |
| Fresh % of append     | 27.1%    | 40.5%    | 34.9%    |
| Prefill amplification | 3.7×     | 2.5×     | 2.9×     |

storage ratio from  $R = 0.74$  to  $R = 5.07$  ( $\sim 7\times$  more suspended KV). Most of the gain is cheap: by 5 min the hit rate is already  $\sim 94\%$  at  $R \approx 1.9$ , and the remaining push to 1 h costs  $\sim 2.7\times$  more storage for only  $\sim 4$  more points of hit rate.

The prefill amplification panel (bottom) tells a similar story. An idealized cache that never evicts would prefill only fresh tokens, so its amplification *floors at*  $1\times$ ; tightening the eviction time re-prefills evicted context and inflates it (merged  $18.9\times$  at 1 min,  $7.4\times$  at 5 min,  $1.8\times$  at 1 h).

### 7.5 Cost of Human Thinking Time

The eviction sweep above is a system/operator view: longer retention improves hit rate but consumes more KV storage. We now analyze the same prefix cache behavior from a consumer perspective. For a user, “thinking” between requests can become a direct cost: if the session prefix expires during the pause, the next user-initiated step pays the fresh-input price to prefill context that was already present before the pause.

We estimate an upper bound on this consumer-side cost by asking how much append prefill could be avoided if user-initiated steps retained their prefix cache across human thinking time. For each user-initiated step  $S$  with a predecessor step  $P$  in the same session, we keep the total input length unchanged but cap the append tokens at the net context growth,  $\max(0, L_S - L_P)$ , where  $L$  is prefix tokens plus append tokens. If the observed append tokens is already smaller than this value, we leave it unchanged. Tool-result steps and session-first steps are also unchanged. This assumes shifted tokens

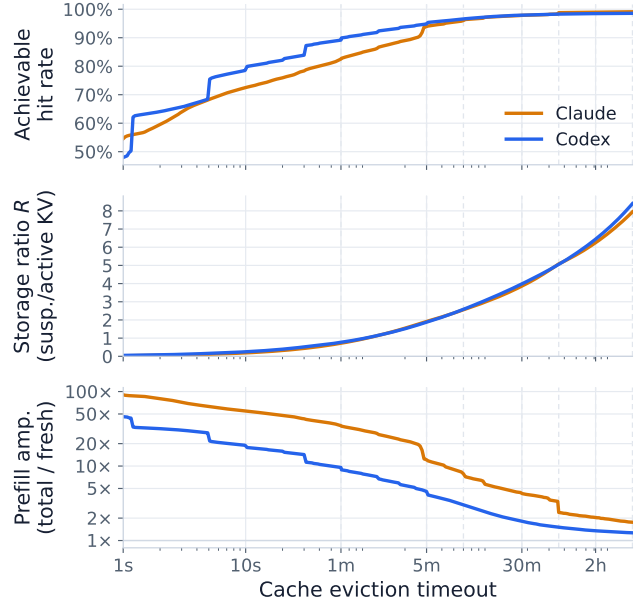


Figure 12: Prefix cache eviction trade-off versus the eviction timeout (shared log  $x$ -axis), for Claude and Codex: achievable hit rate (top), storage ratio  $R$  of suspended to active KV (middle), and prefill amplification (bottom).

Table 13: Upper-bound append-token and cost savings from eliminating user-thinking-induced prefix cache misses.

| Metric                      | Claude          | Codex           | Total           |
|-----------------------------|-----------------|-----------------|-----------------|
| User steps w/ predecessor   | 16,927          | 17,033          | 33,960          |
| Observed append             | 1.19 B          | 1.15 B          | 2.34 B          |
| Append after retained cache | 541.9 M         | 721.7 M         | 1.26 B          |
| Append reduction            | 648.1 M (54.5%) | 423.9 M (37.0%) | 1.07 B (45.9%)  |
| Observed total cost         | \$22,654        | \$17,777        | \$40,431        |
| Cost after retained cache   | \$18,973        | \$16,269        | \$35,242        |
| Cost reduction              | \$3,680 (16.2%) | \$1,508 (8.5%)  | \$5,189 (12.8%) |
| Avg saved / reduced step    | \$0.263         | \$0.116         | \$0.192         |

can be served from cache at the cache-read rate, so the resulting savings are an upper bound rather than an achievable policy guarantee. Table 13 reports the resulting token and cost reductions.

Table 13 shows that this upper-bound estimate reduces append prefill by 1.07 B tokens overall, or 45.9% of observed append tokens. At current list prices, this corresponds to a final cost reduction of \$5,189 (12.8%) over priced rounds. The reduction is larger for Claude in both token and dollar terms due to its longer average context length.

### 7.6 Takeaways and System Opportunities

#### Takeaways

- **User-initiated steps miss the cache.** Overall prefix cache hit rate is high, but user-initiated steps are more likely to miss the cache due to longer idle time.

- **Idle time driven misses.** Cache misses mainly increase when idle time is larger than 5 min and almost never hit when past 1 h.
- **Large gap to optimal.** Only 19% of append tokens are fresh, indicating a large gap to optimal and significant redundant prefill, and a trade-off exists between hit rate and storage space.
- **Human thinking time is costly.** In our retained-cache upper-bound estimate, cache misses after human thinking time account for 45.9% of append tokens and 12.8% of total priced cost.

### System Opportunities

- **Step type driven eviction.** Choosing the right eviction time depends on the ratio of cache storage cost to prefill cost. Advances in KV compression and reductions in storage cost can help reduce the prefill workload.
- **Prediction based prefetching.** Gap-duration prediction and KV cache prefetching can evict the cache more aggressively while still providing similar TTFT for the user.
- **Keeping the cache alive.** Agent harnesses can periodically refresh the cache during long human-thinking gaps to avoid eviction and reduce prefill cost, or reduce the frequency of user interventions through better automation.

## 8 Related Work

### 8.1 Real-world coding-agent traces and developer usage

Some existing work studies coding agents and AI-assisted programming in real use. CacheWise [74] is a companion systems study from the same broad research direction: it uses CATraces, a real Claude Code trace, to characterize closed-loop coding-agent requests and to design a vLLM KV-cache layer with prefix-aware scheduling and predictive eviction from tool-call metadata. TraceLab builds on this line but analyzes a much broader cross-provider trace spanning Claude Code and Codex with several orders of magnitude larger token counts. SWE-chat [9] collects thousands of real coding-agent sessions from open-source developers and analyzes behavioral outcomes such as vibe-coding, code survival, and security. AgentPack [109] and The Rise of AI Teammates in Software Engineering [38] collect large numbers of agent-authored code edits or pull requests from GitHub. These works analyze code-change outcomes but not the interaction timeline or serving system that produced them. Programming by Chat [73], Reading Between the Lines [54], the GitHub Copilot productivity study [110], the professional-developer qualitative study [27], and the Cursor adoption study [20] similarly characterize how developers use AI coding tools, ranging from IDE chat logs and surveys to repository-level

productivity and technical-debt outcomes. TraceLab is mainly system-focused, investigating the system implications of agentic workloads.

### 8.2 Production LLM serving traces and conversation datasets

Large-scale LLM serving traces have driven many top-tier systems papers, but their workloads are mostly chat, API, or code-completion traffic. Mooncake [63] releases a replayed production trace with arrival times, token counts, and remapped block hashes for the Kimi serving stack. Splitwise [60] characterizes Azure “coding” and “conversation” traces, where the coding trace is closer to Copilot-style completion than multi-step agent execution. BurstGPT [77], DynamoLLM [69], ServeGen [82], KVCache Cache in the Wild [75], POLCA [61], and BatchLLM [104] study burstiness, energy, request generation, multi-turn shared-prefix reuse, power oversubscription, and global prefix sharing in production or production-like LLM traffic. A second family of datasets, including LMSYS-Chat 1M [101], WildChat [100], MT-Bench [102], Chatbot Arena [13], OpenAssistant [34], WildBench [41], and Clio [71], captures real conversational use or derives benchmarks from user conversations. These traces are useful for trace-driven serving research, but they do not expose the session/request/step structure of coding agents: long accumulated context, frequent tool-triggered model invocations, tool latency, and human-paced gaps between requests.

TraceLab fills this missing workload category by measuring real coding-agent traffic at the systems detail needed for serving decisions.

### 8.3 Coding-agent benchmarks and tool-use evaluation

Software-engineering benchmarks evaluate whether agents can solve tasks. SWE-bench [30] and its variants, including SWE-bench Multimodal [88], Multi-SWE-bench [96], SWE-Gym [59], and SWE-Lancer [53], use realistic issue-resolution tasks. Terminal-Bench [51] evaluates agents in containerized terminal environments. LiveCodeBench [28], BigCodeBench [108], HumanEval [12], MBPP [7], DS-1000 [35], APPS [21], RepoBench [43], ClassEval [15], CrossCodeEval [14], Commit0 [99], MLE-bench [11], and Prompting LLMs to Tackle the Full Software Development Lifecycle [36] cover competitive programming, library use, data science, repository context, cross-file completion, whole-library generation, ML engineering, and full software development workflows. Agent frameworks and tool-use benchmarks provide the execution substrate for these tasks: SWE-agent [87] and OpenHands [76] implement tool-using software-engineering agents; Agentless [81] and AutoCodeRover [97] explore more structured repair pipelines; ReAct [91] and Toolformer [65] define influential reasoning-and-action and tool-learning paradigms; tau-bench [90], tau2-bench [8], ToolLLM [64], Gorilla [62], AgentBench [44], WebArena [106], GAIA [52], and OSWorld [86] test general tool

use, web interaction, computer use, and multi-step assistance.

These benchmarks and frameworks are essential for capability evaluation, but they are intentionally task-centric: a benchmark instance is usually a single isolated request, often with constrained tooling and limited session history. TraceLab instead measures whole day-to-day sessions, capturing accumulated context, human delays, repeated tool calls, cache retention, and serving cost that benchmark replay cannot faithfully represent.

#### 8.4 LLM serving systems and agent-aware scheduling

Modern LLM serving systems optimize batching, memory management, disaggregation, and scheduling. Orca [93] introduced iteration-level batching for variable-length generation. vLLM/PagedAttention [33] manages KV memory in non-contiguous pages and enables prefix sharing. SarathiServe [2], DistServe [105], Splitwise [60], Inference without Interference [25], and DeepSpeed-FastGen [23] target the pre-fill/decode throughput-latency trade-off through chunked pre-fills, phase splitting, or disaggregated execution. FlexGen [66] studies throughput-oriented offloading, AlpaServe [40] investigates statistical multiplexing, Fast Distributed Inference Serving [79] performs token-level preemptive scheduling, Llumnix [70] enables live migration, LoongServe [78] proposes elastic sequence parallelism, and NanoFlow [107] presents Intra-device parallelism, these work greatly enhanced the throughput and SLO attainment for large deployments. Agent- and program-aware systems extend this direction: SGLang [103] support RadixAttention for prefix reuse, Parrot [42] schedules application dataflow via semantic variables, Autellix [50] treats agent executions as dynamic programs or DAGs, and VibeServe [32] uses agents to tailor serving stacks for specific use cases.

These systems provide the mechanisms to serve coding-agent traffic, and several already recognize that agent programs are not independent stateless requests. TraceLab provides a standard benchmark to guide the evolve of serving engines for real coding agents.

#### 8.5 Prefix-cache and KV-state management

KV reuse is central to efficient multi-turn and agentic serving. SGLang’s RadixAttention [103], Mooncake [63], Preble [67], Pensieve [94], ChunkAttention [92], Hydragen [31], Prompt Cache [17], CacheBlend [89], CacheGen [46], EPIC [26], and LMCACHE [45] study different ways to reuse, place, retrieve, compress, or recompute KV state across requests and prompt fragments. CachedAttention/AttentionStore [16] and InferCept [1] are especially close to interactive or augmented LLM settings: they consider saving session KV across idle periods or choosing among GPU retention, CPU swapping, and re-computation when generation pauses for tools or humans. DualPath [80] observes that agentic workloads can shift the bottleneck to loading large KV state from external storage, while Continuum [37] selects a TTL for finished-request KV

using empirical tool-call distributions and scheduling costs.

TraceLab pinpoints the noisy tool latency and human thinking time of the coding trace, which requires further research on how to manage the prefix cache.

#### 8.6 Long-context attention and KV compression

Another line of work reduces the cost of long contexts by changing attention or compressing KV state. StreamingLLM [85], InfLLM [83], Native Sparse Attention [95], MoBA [49], MInference [29], and Quest [72] reduce attention cost through locality and block-level or page-level importance estimation and sparse selection. H2O [98], Scissorhands [47], SnapKV [39], PyramidKV [10], DuoAttention [84], KIVI [48], and KVQuant [24] reduce KV footprint through heavy-hitter retention, prompt-token selection, layer/head-aware budgets, quantization, or mixed full/streaming-head caches.

These techniques are often evaluated on long documents or long-context benchmarks, but TraceLab demonstrate that the same pressure appears in coding agents because sessions replay very large prefixes while appending only small tool results or user turns.

### 9 Limitations and Future Work

Our study has several limitations that also point to future work. First, our trace is drawn from our own day-to-day use of coding agents. Although it spans multiple developers, agents, and model versions, the results may not fully generalize to other organizations, development workflows, or agent domains. Second, our analysis is limited to externally visible logs, including prompts, model responses, tool calls, timestamps, and token metadata. Without internal telemetry from service providers, we can identify workload patterns but can only infer the mechanisms behind them, such as scheduling policies, cache behavior, or backend routing. Third, agent workloads are expanding beyond coding to broader computer-use and automation tasks. Future trace studies should extend this methodology to other agent domains and compare how their context growth, tool use, latency, and cache behavior differ from coding-agent workloads.

### 10 Conclusion

In this paper, we present a trace-based analysis of coding-agent workloads. Notably, we find that these workloads feature long autonomous loops, long contexts paired with short outputs, strongly long-tailed tool calls, and high yet imperfect prefix-cache hit rates. Taken together, these observations point to serving opportunities, including denser or lower-overhead tool calling, append-length-specific prefill optimization, and better KV-cache eviction or prefetching around human-paced gaps. We hope this work provides a useful reference for future system research on coding agents and, more broadly, on LLM-based agents, and we look forward to community efforts that extend this trace-based analysis methodology to larger-scale

and more diverse agent workloads.

## References

- [1] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiyang Zhang. Incept: Efficient intercept support for augmented large language model inference, 2024.
- [2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve, 2024.
- [3] Anthropic. Anthropic acquires bun as claude code reaches \$1b milestone. <https://www.anthropic.com/news/anthropic-acquires-bun-as-claude-code-reaches-usd1b-milestone>, 2025. Reached \$1B annualized run-rate revenue within ~6 months of public launch; accessed 2026-06-14.
- [4] Anthropic. Claude code. <https://www.anthropic.com/claude-code>, 2025. Terminal-based coding agent; accessed 2026-06-21.
- [5] Anthropic. Claude API pricing. <https://platform.claude.com/docs/en/about-claude/pricing>, 2026. Accessed 2026-06-25.
- [6] Anysphere. Cursor: The AI code editor. [https://en.wikipedia.org/wiki/Cursor\\_\(company\)](https://en.wikipedia.org/wiki/Cursor_(company)), 2025. Reports 7M+ monthly active users; accessed 2026-06-14.
- [7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.
- [8] Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan.  $\tau^2$ -bench: Evaluating conversational agents in a dual-control environment, 2025.
- [9] Joachim Baumann, Vishakh Padmakumar, Xiang Li, John Yang, Diyi Yang, and Sanmi Koyejo. Swe-chat: Coding agent interactions from real users in the wild, 2026.
- [10] Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, and Wen Xiao. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling, 2025.
- [11] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. Mle-bench: Evaluating machine learning agents on machine learning engineering, 2025.
- [12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebguss, Alex Nichol, Alex Paino, Nikolos Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.
- [13] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. Chatbot arena: An open platform for evaluating llms by human preference, 2024.
- [14] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion, 2023.
- [15] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation, 2023.
- [16] 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, 2024.
- [17] In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference, 2024.
- [18] GitHub. GitHub copilot surpasses 20 million users. <https://dataconomy.com/2025/07/31/github-copilot-now-has-over-20-million-users/>,

2025. Figure reported in Microsoft Q4 FY25 earnings; accessed 2026-06-14.
- [19] Google. Gemini CLI: An open-source ai agent that brings the power of gemini directly into your terminal. <https://github.com/google-gemini/gemini-cli>, 2025. Accessed 2026-06-21.
- [20] Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. Speed at the cost of quality: How cursor ai increases short-term velocity and long-term complexity in open-source projects, 2026.
- [21] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps, 2021.
- [22] HKUDS. Deepcode: Open agentic coding. <https://github.com/HKUDS/DeepCode>, 2025. Accessed 2026-06-21.
- [23] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. DeepSpeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference, 2024.
- [24] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization, 2025.
- [25] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. Inference without interference: Disaggregate llm inference for mixed downstream workloads, 2024.
- [26] 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, 2025.
- [27] Ruanqianqian Huang, Avery Reyna, Sorin Lerner, Haijun Xia, and Brian Hempel. Professional software developers don't vibe, they control: Ai agent use for coding in 2025, 2025.
- [28] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024.
- [29] Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention, 2024.
- [30] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024.
- [31] Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-throughput llm inference with shared prefixes, 2024.
- [32] Keisuke Kamahori, Shihang Li, Simon Peter, and Baris Kasicki. Vibeserve: Can ai agents build bespoke llm serving systems?, 2026.
- [33] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023.
- [34] Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi-Rui Tam, Keith Stevens, Abdullah Barhoum, Nguyen Minh Duc, Oliver Stanley, Richárd Nagyfi, Shahul ES, Sameer Suri, David Glushkov, Arnav Dantuluri, Andrew Maguire, Christoph Schuhmann, Huu Nguyen, and Alexander Mattick. Openassistant conversations – democratizing large language model alignment, 2023.
- [35] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation, 2022.
- [36] Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, Zhiyin Yu, He Du, Ping Yang, Dahua Lin, Chao Peng, and Kai Chen. Prompting large language models to tackle the full software development lifecycle: A case study, 2024.
- [37] Hanchen Li, Runyuan He, Qiuyang Mang, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. Continuum: Efficient and robust multi-turn llm agent scheduling with kv cache time-to-live, 2026.

- [38] Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. The rise of ai teammates in software engineering (se) 3.0: How autonomous coding agents are reshaping software engineering, 2025.
- [39] 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, 2024.
- [40] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving, 2023.
- [41] Bill Yuchen Lin, Yuntian Deng, Khyathi Chandu, Faeze Brahman, Abhilasha Ravichander, Valentina Pyatkin, Nouha Dziri, Ronan Le Bras, and Yejin Choi. Wildbench: Benchmarking llms with challenging tasks from real users in the wild, 2024.
- [42] 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, 2024.
- [43] Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems, 2023.
- [44] 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, 2025.
- [45] Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, and Junchen Jiang. Lm-cache: An efficient kv cache layer for enterprise-scale llm inference, 2025.
- [46] 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, 2024.
- [47] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time, 2023.
- [48] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache, 2024.
- [49] Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, Chao Hong, Shaowei Liu, Weiran He, Enming Yuan, Yuzhi Wang, Zhiqi Huang, Huan Yuan, Suting Xu, Xinran Xu, Guokun Lai, Yanru Chen, Huabin Zheng, Junjie Yan, Jianlin Su, Yuxin Wu, Neo Y. Zhang, Zhilin Yang, Xinyu Zhou, Mingxing Zhang, and Jiezhong Qiu. Moba: Mixture of block attention for long-context llms, 2025.
- [50] 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, 2025.
- [51] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jena Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muenighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raouf, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026.
- [52] Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants, 2023.
- [53] Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. Swe-lancer: Can frontier llms earn \$1 million from real-world freelance software engineering?, 2025.

- [54] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. Reading between the lines: Modeling user behavior and costs in ai-assisted programming, 2024.
- [55] OpenAI. Codex CLI: A lightweight coding agent that runs in your terminal. <https://github.com/openai/codex>, 2025. Accessed 2026-06-21.
- [56] OpenAI. API pricing. <https://openai.com/api/pricing/>, 2026. Accessed 2026-06-25.
- [57] OpenAI. Codex pricing. <https://developers.openai.com/codex/pricing>, 2026. Accessed 2026-06-25.
- [58] OpenCode. Opencode: The open source AI coding agent. <https://opencode.ai/>, 2025. Accessed 2026-06-21.
- [59] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym, 2025.
- [60] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting, 2024.
- [61] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Briji Warrier, Nithish Mahalingam, and Ricardo Bianchini. Polca: Power oversubscription in llm cloud providers, 2023.
- [62] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis, 2023.
- [63] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kv-cache-centric disaggregated architecture for llm serving, 2025.
- [64] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023.
- [65] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023.
- [66] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.
- [67] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiyang Zhang. Preble: Efficient distributed prompt scheduling for llm serving, 2024.
- [68] Stack Overflow. 2025 stack overflow developer survey: Ai. <https://survey.stackoverflow.co/2025/ai/>, 2025. Reports 84% of developers use or plan to use AI tools; accessed 2026-06-14.
- [69] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. Dynamollm: Designing llm inference clusters for performance and energy efficiency, 2024.
- [70] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving, 2024.
- [71] Alex Tamkin, Miles McCain, Kunal Handa, Esin Durmus, Liane Lovitt, Ankur Rathi, Saffron Huang, Alfred Mountfield, Jerry Hong, Stuart Ritchie, Michael Stern, Brian Clarke, Landon Goldberg, Theodore R. Sumers, Jared Mueller, William McEachen, Wes Mitchell, Shan Carter, Jack Clark, Jared Kaplan, and Deep Ganguli. Clio: Privacy-preserving insights into real-world ai use, 2024.
- [72] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference, 2024.
- [73] Ningzhi Tang, Chaoran Chen, Zihan Fang, Gelei Xu, Maria Dhakal, Yiyu Shi, Collin McMillan, Yu Huang, and Toby Jia-Jun Li. Programming by chat: A large-scale behavioral analysis of 11,579 real-world ai-assisted ide sessions, 2026.
- [74] Shubham Tiwari, Tapan Chugh, Nash Rickert, Simon Peter, Ratul Mahajan, and Haiying Shen. Cache-wise: Understanding workloads and optimizing kv-cache management for efficiently serving llm coding agents, 2026.
- [75] 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 kv-cache cache at a large cloud provider, 2026.

- [76] 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, Yanjun 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, 2025.
- [77] 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, 2025.
- [78] 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, 2024.
- [79] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models, 2024.
- [80] Yongtong Wu, Shaoyuan Chen, Yinmin Zhong, Rilun Huang, Yixuan Tan, Wentao Zhang, Liyue Zhang, Shangyan Zhou, Yuxuan Liu, Shunfeng Zhou, Mingxing Zhang, Xin Jin, and Panpan Huang. Dualpath: Breaking the storage bandwidth bottleneck in agentic llm inference, 2026.
- [81] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents, 2024.
- [82] Yuxing Xiang, Xue Li, Kun Qian, Wenyan Yu, Ennan Zhai, and Xin Jin. Servegen: Workload characterization and generation of large language model serving in production, 2026.
- [83] Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. Inllm: Training-free long-context extrapolation for llms with an efficient context memory, 2024.
- [84] Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. Duoattention: Efficient long-context llm inference with retrieval and streaming heads, 2024.
- [85] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks, 2024.
- [86] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Oworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024.
- [87] 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, 2024.
- [88] John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. Swe-bench multimodal: Do ai systems generalize to visual software domains?, 2024.
- [89] 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, 2025.
- [90] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan.  $\tau$ -bench: A benchmark for tool-agent-user interaction in real-world domains, 2024.
- [91] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.
- [92] Lu Ye, Ze Tao, Yong Huang, and Yang Li. Chunkattention: Efficient self-attention with prefix-aware kv cache and two-phase partition, 2024.
- [93] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 521–538, Carlsbad, CA, July 2022. USENIX Association.
- [94] Lingfan Yu, Jinkun Lin, and Jinyang Li. Stateful large language model serving with pensieve, 2024.
- [95] Jinyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Y. X. Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention, 2025.

- [96] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-swebench: A multilingual benchmark for issue resolving, 2025.
- [97] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement, 2024.
- [98] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuan-dong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H<sub>2</sub>o: Heavy-hitter oracle for efficient generative inference of large language models, 2023.
- [99] Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch, 2024.
- [100] Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. Wildchat: 1m chatgpt interaction logs in the wild, 2024.
- [101] 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, 2024.
- [102] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.
- [103] 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, 2024.
- [104] 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, 2026.
- [105] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Dist-serve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.
- [106] 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, 2024.
- [107] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: Towards optimal large language model serving throughput, 2025.
- [108] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widayarsi, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions, 2025.
- [109] Yangtian Zi, Zixuan Wu, Aleksander Boruch-Gruszecki, Jonathan Bell, and Arjun Guha. Agentpack: A dataset of code changes, co-authored by agents and humans, 2026.
- [110] Albert Ziegler, Eirini Kalliamvakou, Shawn Simister, Ganesh Sittampalam, Alice Li, Andrew Rice, Devon Rifkin, and Edward Aftandilian. Productivity assessment of neural code completion, 2022.