# Advanced Computer Networks
# Lecture 1 Internet with Agents

Instructor: 黄倩怡
huangqy89@mail.sysu.edu.cn

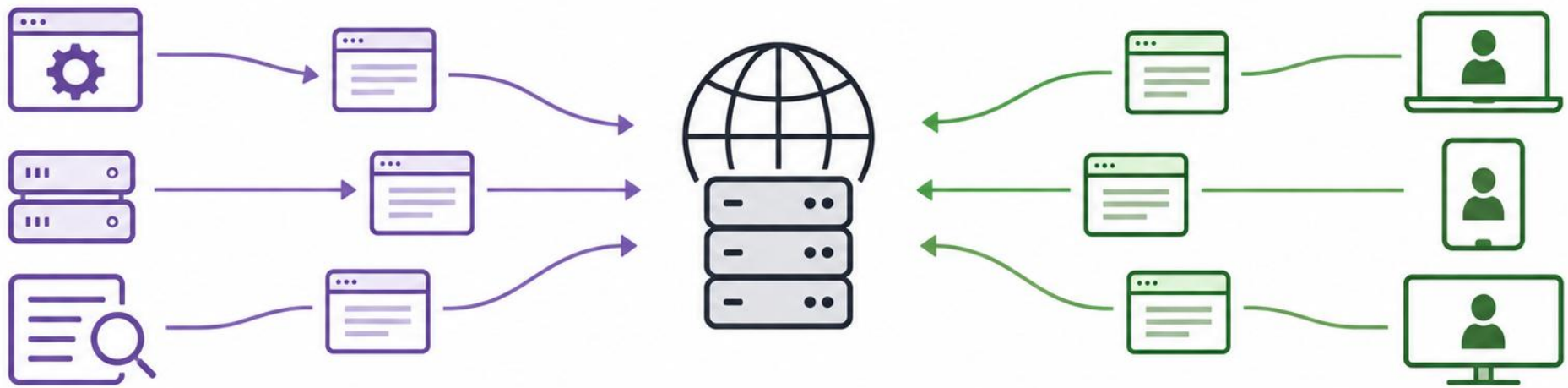
2026.09

# Automated Traffic Accounts for Most Web-Page Requests

Reported Cloudflare Radar snapshot

**57.4%**
**Bots**

**42.6%**
**Humans**

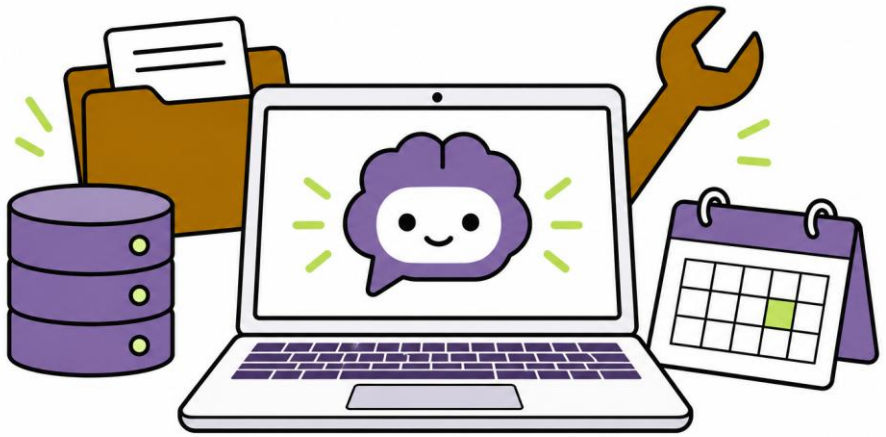


Scope: HTML requests observed by Cloudflare; figures reflect a reported snapshot.
Bots include AI agents, crawlers, scrapers, and other automation.

# TraceLab

Characterizing Coding Agent Workloads for LLM Serving

https://arxiv.org/pdf/2606.30560



## Agents

arXiv:2606.30560

**Advanced Computer Networks**

# Coding Agents Are Taking Off

**84% of developers**

Use or plan to use AI tools in development.

**20M+ Copilot users**

Reported user count in 2025.

**7M+ Cursor users**

Monthly active users.

**$1B Claude Code**

Reported annualized revenue run-rate.

# What Is a Coding Agent?

State persists while the model reads, edits, and executes

**CHAT ASSISTANT**

## Single-turn assistance

- Responds to a prompt
- Usually returns text
- User drives each turn

**CODING AGENT**

## Persistent execution

- Reads and edits files
- Calls tools repeatedly
- Uses results to continue

# Session, Request, and Step

A session contains requests; each request contains multiple steps

**SESSION**

**Persistent context**

Multiple user tasks
Shared conversation
Long-lived history

**REQUEST**

**One user task**

Starts with user input
Includes agent steps
Ends with a response

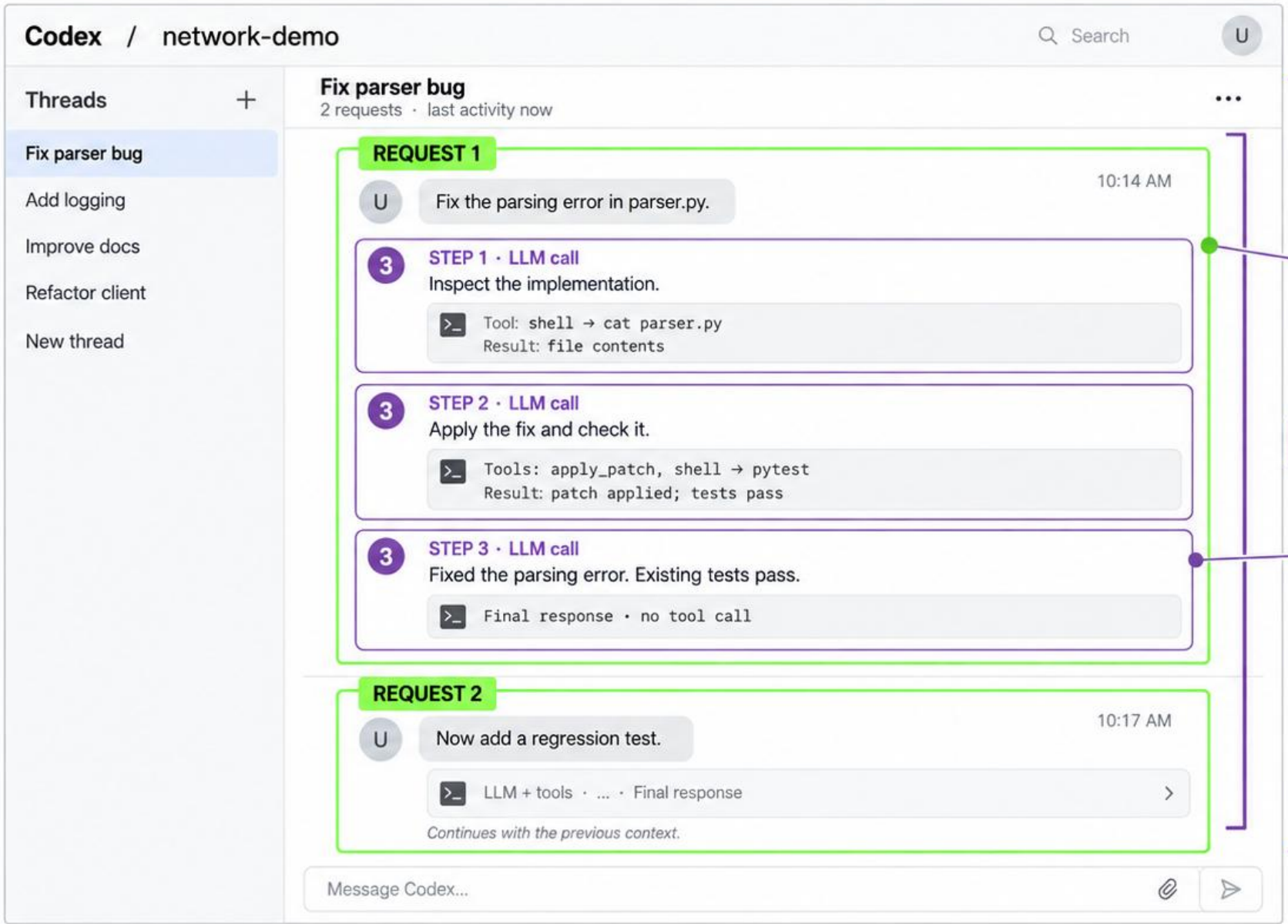
**STEP**

**One LLM invocation**

Model generation
Associated tool calls
Results feed next step

# Session, Request, and Step

## Understanding a coding-agent conversation




SESSION

1 **SESSION**

One conversation with persistent context.
Contains multiple requests.

2 **REQUEST**

One user task, from prompt to final response.
May require multiple steps.

3 **STEP**

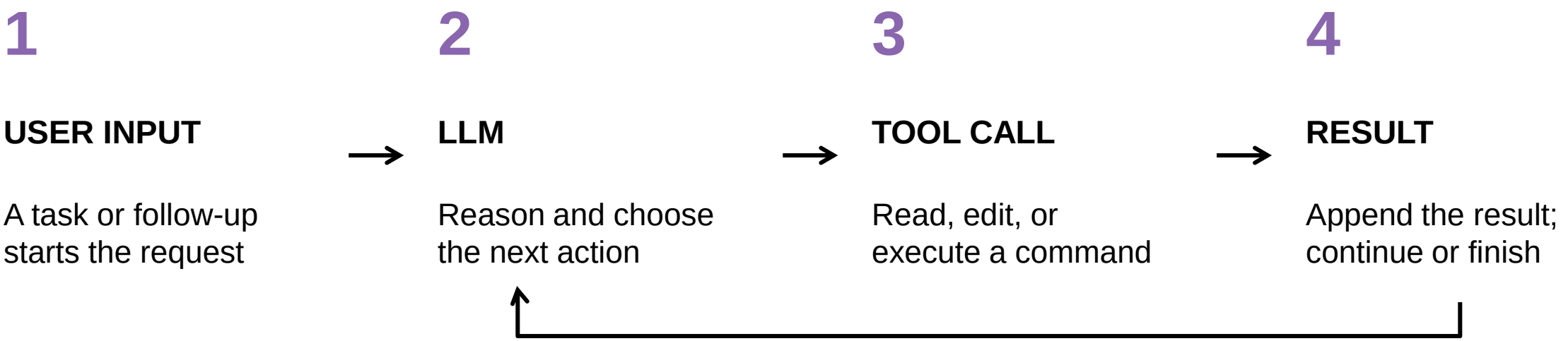
One LLM invocation plus any tool calls it produces.
The final step may use no tools.

Illustrative Codex interface; annotations follow TraceLab §2.

Teaching mockup — not a real screenshot.

# The Agentic Loop

A tool result can trigger another model invocation




**Tool results trigger the next LLM step.**

# Token Accounting

Each model invocation is decomposed into prefix, append, and output

**PREFIX**

**P: reused history**

Previously processed
context retrieved
from the prefix cache

**APPEND**

**A: fresh computation**

New user/tool input
plus cache-miss replay
Total input: I = P + A

**OUTPUT**

**O: generation**

Text and tool calls
Reasoning tokens
when reported

# 4,265 Real Sessions

## 43 developers · 23 model versions · Claude Code and Codex

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 |
|---|---|---|---|
| *Models* | | | |
| 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 |
| *Scope & activity* | | | |
| 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 |
| *Tokens* | | | |
| 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 | — |

1

### 357K steps

LLM invocations across 4,265 sessions.

2

### 433K tool calls

Real day-to-day development workflows.

3

### 54.9B input tokens

The trace is observational, not a controlled benchmark.

# Session & Context

Section 4 · Session and Context Management

**01** **How many requests, steps, and tool calls?**

**02** **How does context grow and compact?**

**03** **Which token types dominate estimated cost?**

**04** **Where does session and request time go?**

# Autonomous, Long-Tailed Loops

Section 4 · Most steps are triggered by tools

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

| Metric | Avg | P25 | P50 | P90 | P99 |
|---|---|---|---|---|---|
| *Per session* | | | | | |
| Requests | 9.2 | 1 | 1 | 18 | 137 |
| User-initiated steps† | 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 |
| *Per request* | | | | | |
| 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 |
| *Per step* | | | | | |
| Tool calls | 1.2 | 1 | 1 | 2 | 4 |

1 **9.2 requests**

Average requests per session.

2 **73.6 tool steps**

Average tool-initiated steps per session.

3 **A long tail**

The p99 session has 1,107 tool-initiated steps.

# Context Keeps Growing

Section 4 · New outputs and tool results accumulate · Table 3 excerpt

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 |
|---|---|---|
| *All steps* | | |
| 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 |

1

**99.60% · Claude**

Share of steps with positive context growth.

2

**96.56% · Codex**

Tool results and model output accumulate.

3

**Serving impact**

Each new step usually reads a longer context.

# Compaction Resets Context

Section 4 · Large context reductions occur during the agentic loop

Table 4: Context compactions per session: a near-limit total-input drop (≥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 (≥64k drop) | 324 | 1,306 |
| of which compactions | 284 (87.7%) | 1,235 (94.6%) |
| Sessions with ≥1 compaction | 120 (4.5%) | 292 (18.4%) |
| *Compactions per session* | | |
| Avg (all sessions) | 0.106 | 0.777 |
| Avg (sessions with ≥1) | 2.37 | 4.23 |
| P90 / P99 (sessions with ≥1) | 6 / 12 | 8 / 34 |
| *Trigger* | | |
| User-initiated | 105 (37.0%) | 100 (8.1%) |
| Tool-initiated | 179 (63.0%) | 1,135 (91.9%) |

1

**4.5% · Claude**

Sessions with at least one compaction.

2

**18.4% · Codex**

Compaction is more common in this trace.

3

**Mid-loop resets**

Most compactions occur on tool-initiated steps.

# History Dominates Cost

Section 4 · Estimated cost by token category

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

| Metric | Avg | P50 | P90 | P99 | % cost |
|---|---|---|---|---|---|
| *Per session* | | | | | |
| 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% |
| *Per request* | | | | | |
| 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% |
| *Per step* | | | | | |
| 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% |

1 **59.5% · Prefix**

Repeated reads of cached history dominate cost.

2 **29.2% · Append**

New input and cache-miss replay.

3 **11.2% · Output**

Average session: $9.70; median: $0.61.

# Two Different Timescales

Section 4 · Human gaps dominate sessions; tools dominate requests

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

| Metric | Avg | P50 | P90 | P99 | % time |
|---|---|---|---|---|---|
| *Per session* | | | | | |
| 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% |
| *Per session, human capped (1h)* | | | | | |
| 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% |
| *Per request* | | | | | |
| 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% |
| *Per step* | | | | | |
| LLM generation | 11.5s | 4.9s | 20.2s | 1.3m | 40.7% |
| Tool execution | 16.8s | 0.1s | 10.0s | 3.2m | 59.3% |
| *Per individual latency* | | | | | |
| 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 | |

1

**92.3% · Humans**

Share of session wall-clock time spent waiting.

2

**59.8% · Tools**

Share of request time in tool execution.

3

**41.0% · LLM**

Request-level generation time; median request is 38.3 s.

# LLM Generation

Section 5 · LLM Generation

**01** **How long are the inputs and outputs?**

**02** **Why is append work bimodal?**

**03** **How is prior model output reused?**

**04** **What drives generation latency?**

# Long Inputs, Short Outputs

Section 5 · Median token counts reveal an asymmetric workload

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

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

1

**119K prefix tokens**

Median previously processed context.

2

**875 append tokens**

Median new or replayed input.

3

**214 output tokens**

Median model generation per step.

# Two Kinds of Prefill

Section 5 · Many short appends, but most tokens in rare long appends

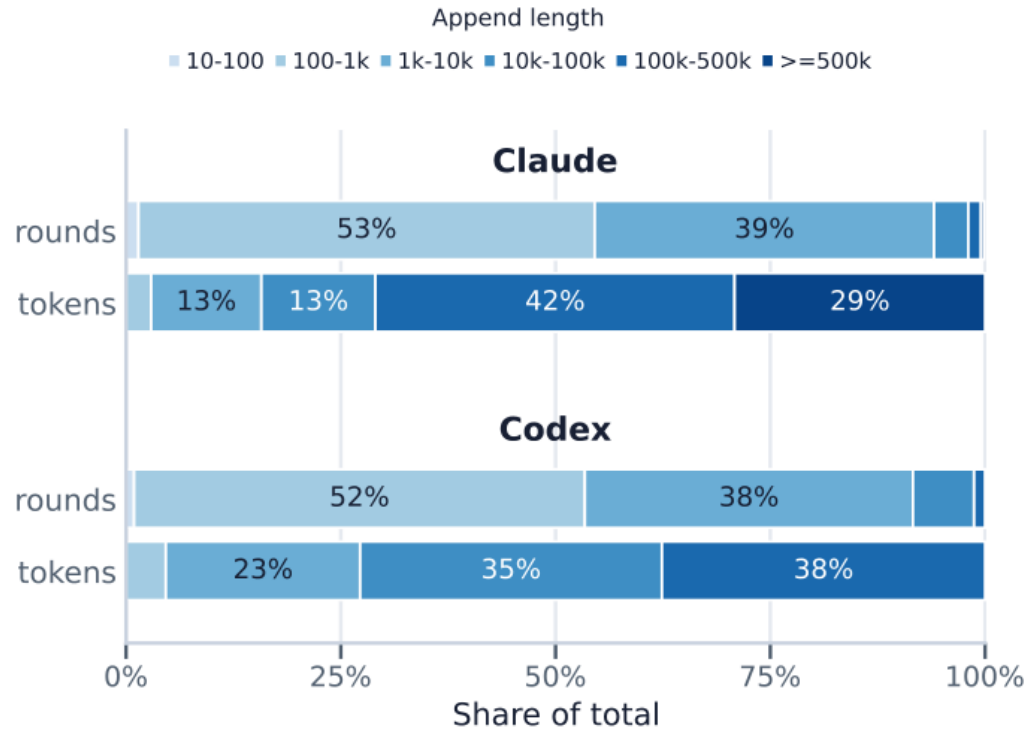



Figure 2: Per-step append length by step count and by total append-token share, split by provider.

**1** **Frequent short steps**

Over 90% of steps append fewer than 10K tokens.

**2** **Rare large prefills**

Over 70% of append tokens come from steps with 10K+ appends.

**3** **Two serving paths**

Optimize short-step latency and large-prefill throughput.

# Short Generations

Section 5 · A request is spread across repeated LLM steps

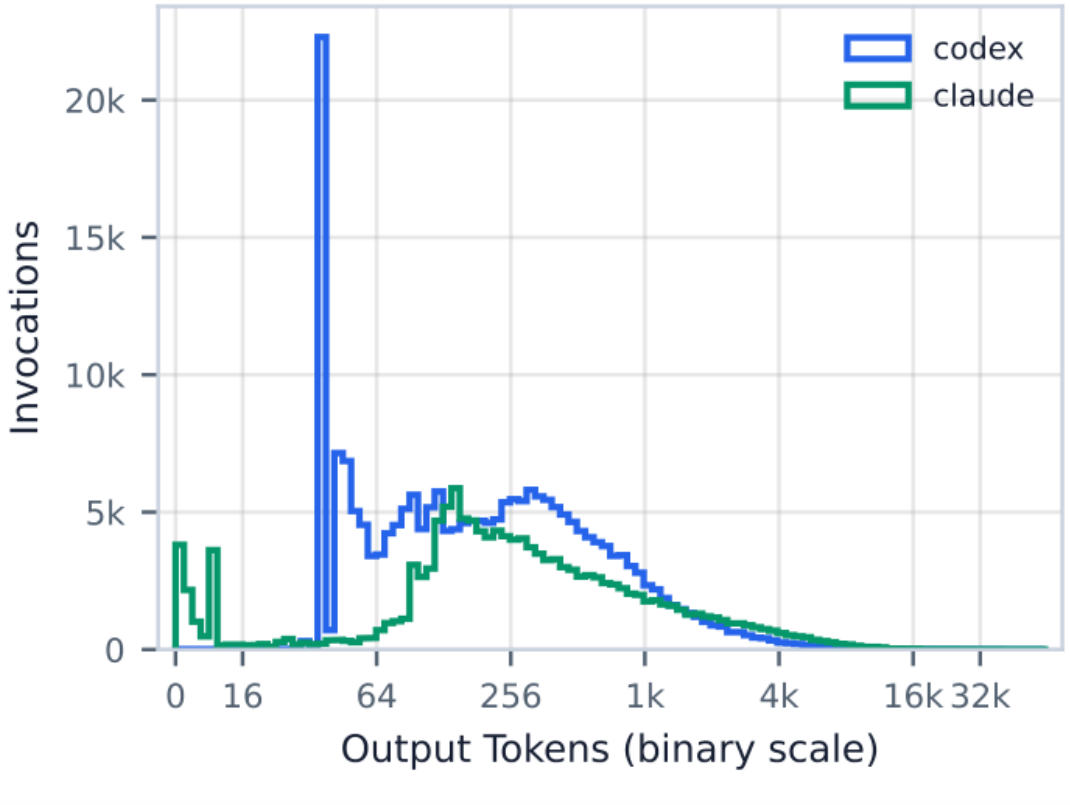



Figure 4: Per-step output-token distribution by provider.

**1**

**252 · Claude**

Median output tokens per step.

**2**

**184 · Codex**

Median output tokens per step.

**3**

**Repeated tool use**

A request spans roughly eight LLM steps.

# How Outputs Are Reused

Section 5 · Output-cached and output-resend create different work

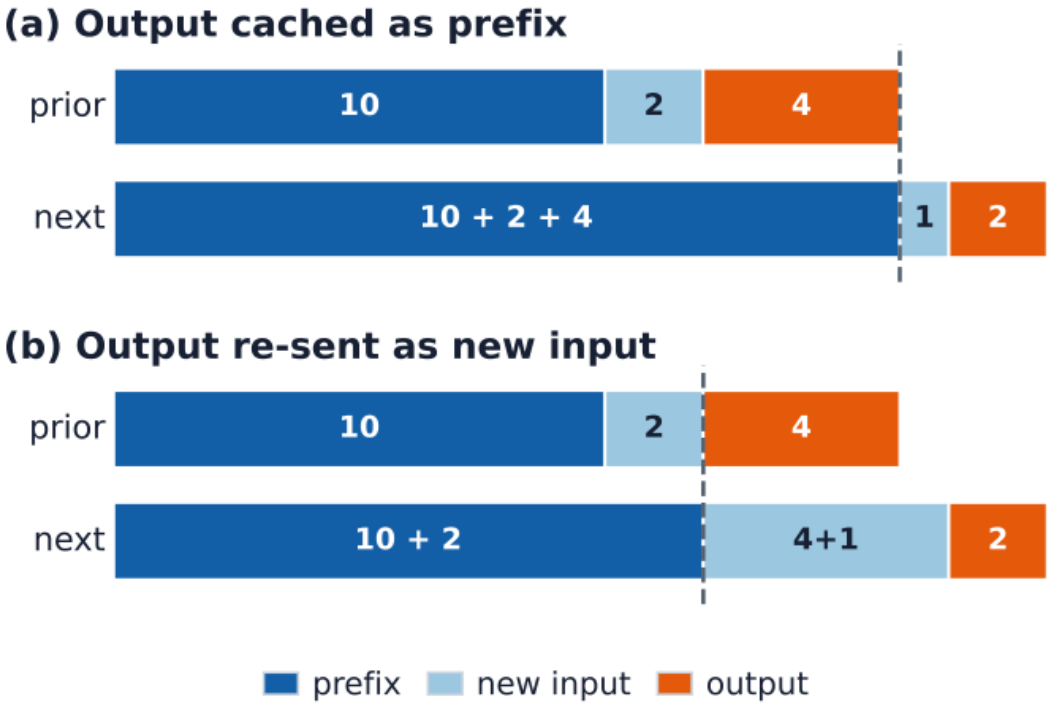



Figure 5: Two ways a prior step's output can be accounted in the next step (schematic; bar lengths illustrative).

**1** **Output-cached**

Prior output joins the next prefix; retain or transfer its KV state.

**2** **Output-resend**

Prior output becomes fresh append input and is re-prefilled.

**3** **Provider differences**

Claude and GPT-5.5 mostly resend; GPT-5.4 mostly caches.

# Variable Generation Speed

Section 5 · Similar context lengths can have very different latency

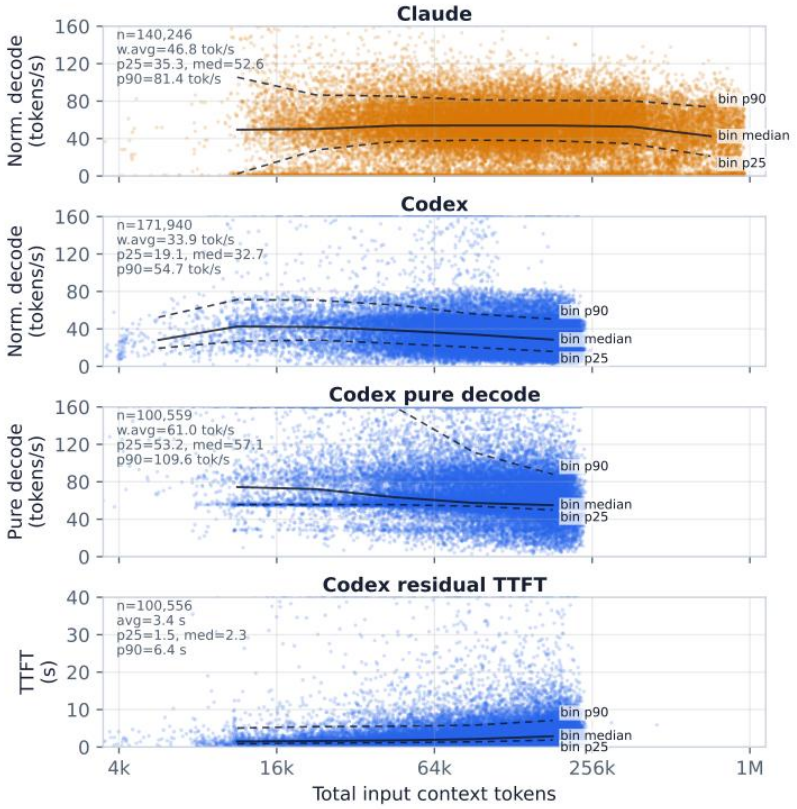



Figure 6: Trace-observed LLM timing versus total input context length.

**1** **46.8 · Claude**

Weighted normalized generation rate, tokens/s.

**2** **33.9 · Codex**

Weighted normalized generation rate, tokens/s.

**3** **3.4 s · Codex**

Average residual time to first token.

# Tool Calls

Section 6 · Tool Calls

**01** **Which tools account for most calls?**

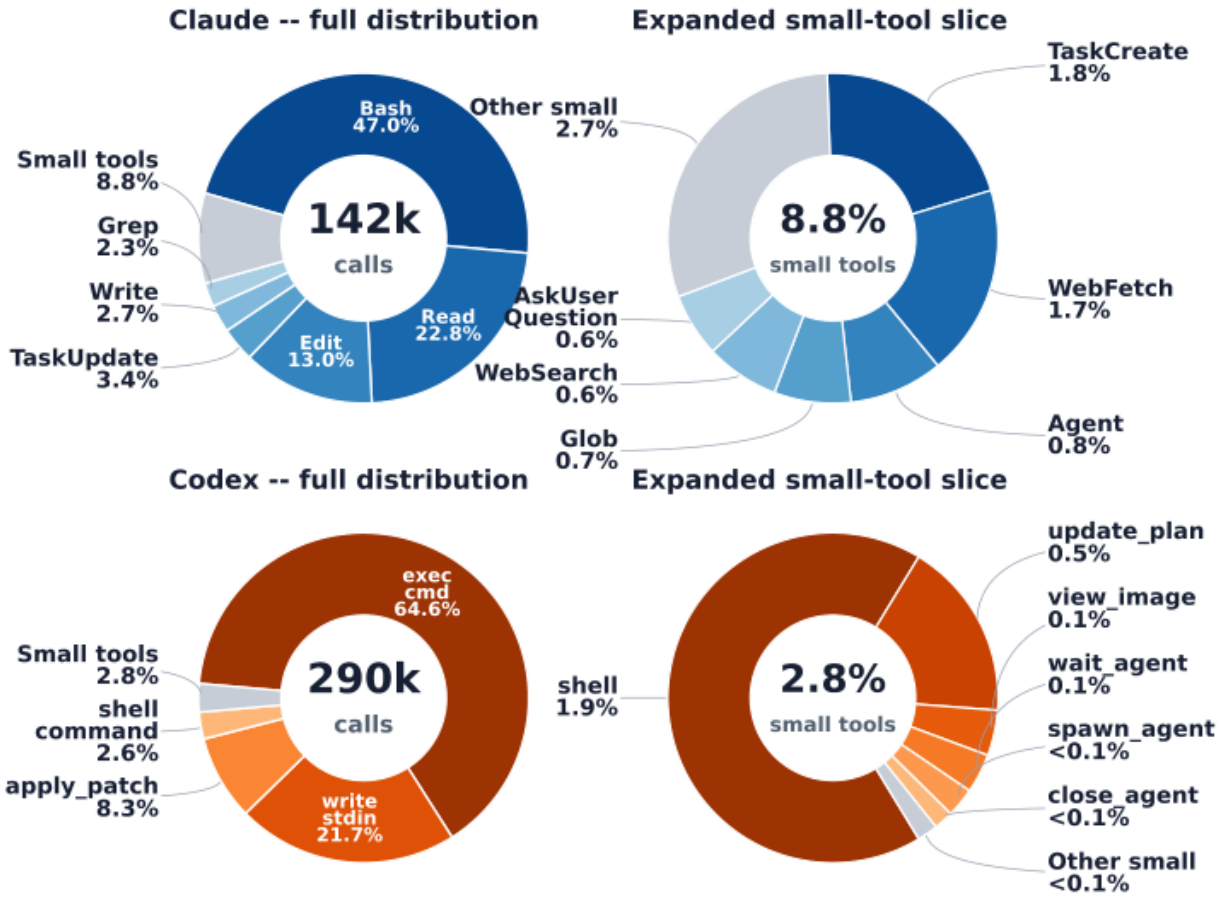
**02** **Which calls dominate execution time?**

**03** **How does tool semantics affect latency?**

**04** **How much time lies outside execution?**

# Commands Dominate

Section 6 · A few tools account for most invocations




**1** **47.0% · Claude**

Bash share of tool invocations.

**2** **64.6% · Codex**

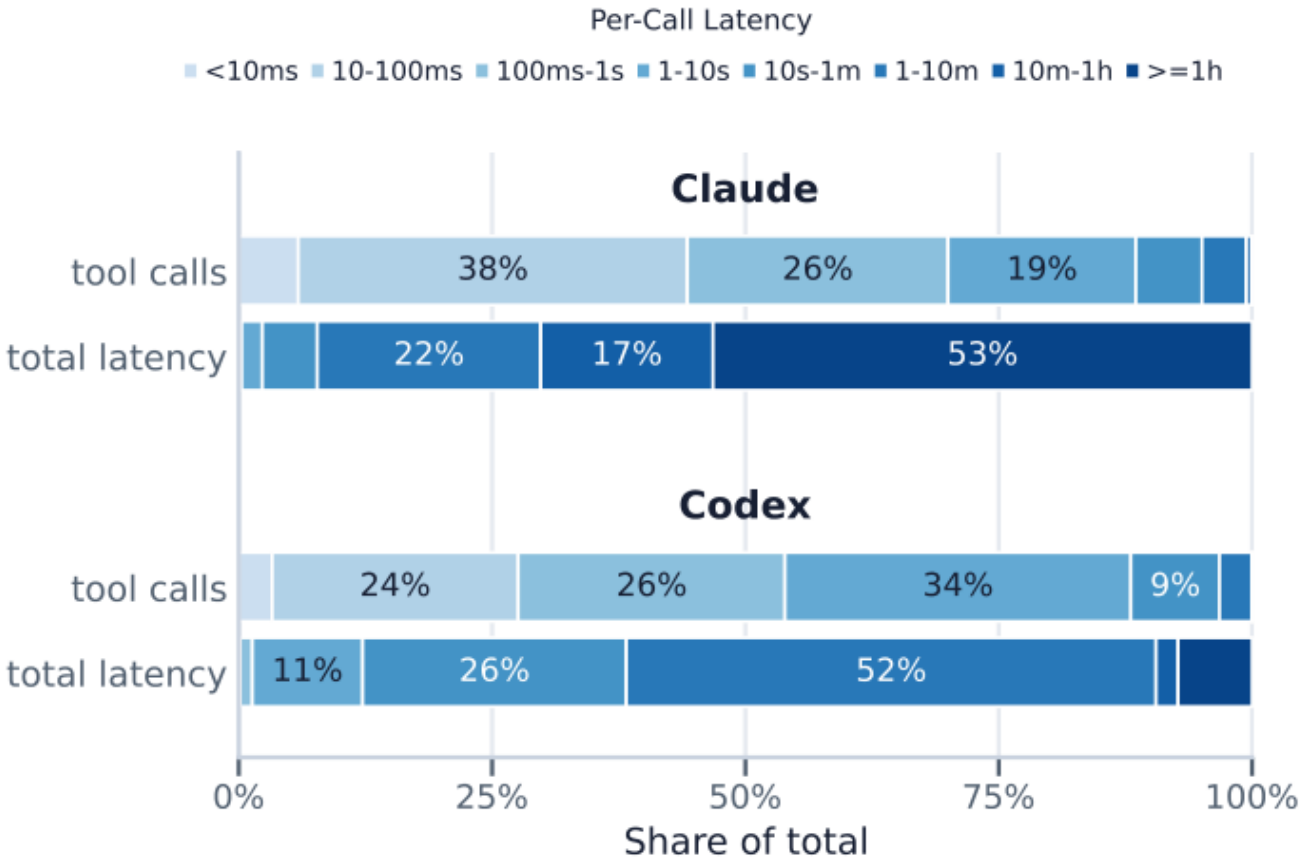
exec_command share of tool invocations.

**3** **Concentrated usage**

Top three tools exceed 80% of Claude and 95% of Codex calls.

# Slow Calls Dominate Time

Section 6 · Call frequency and time share tell different stories




**1** **4.9% of calls**

Claude tool calls that last over one minute.

**2** **92% of tool time**

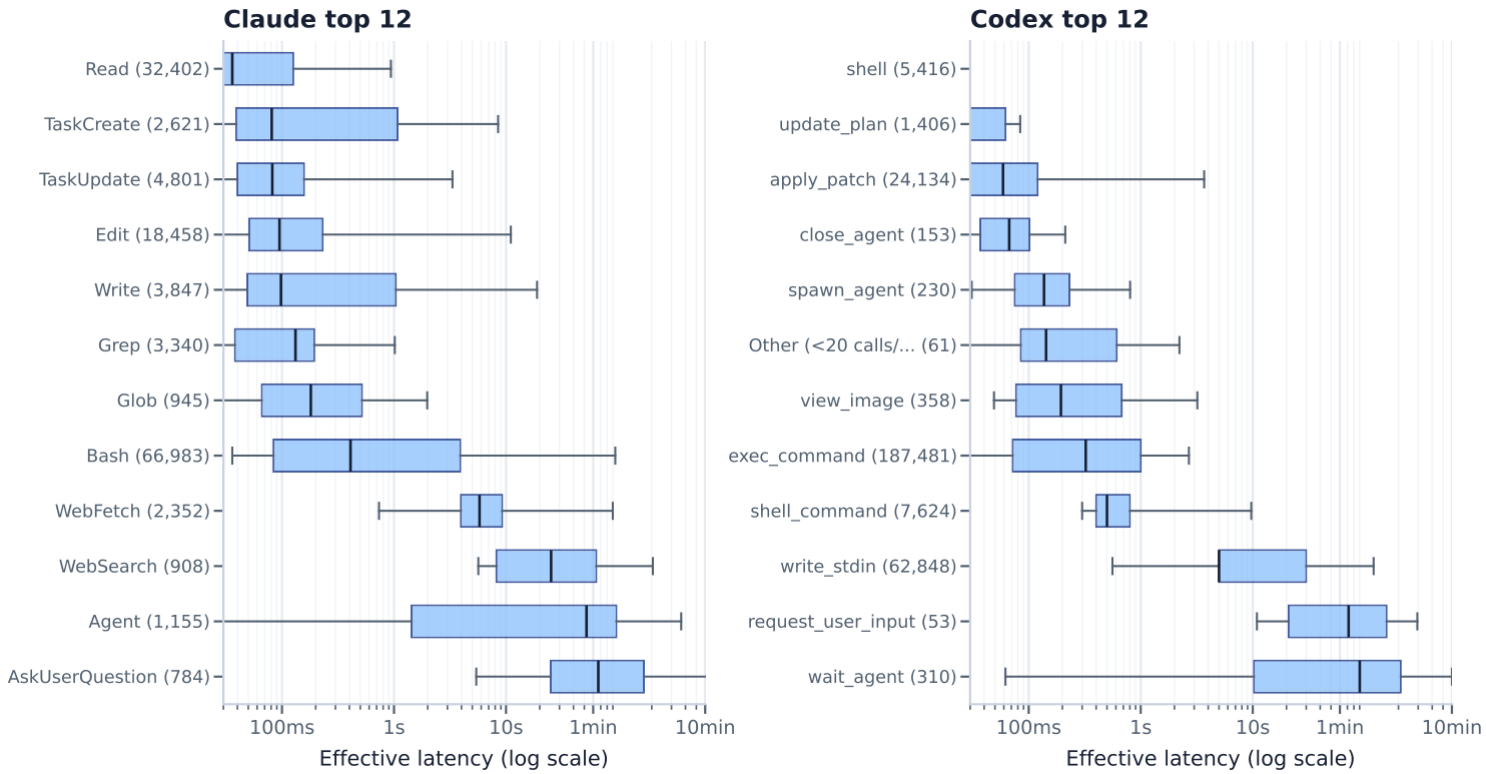
Time consumed by those slow Claude calls.

**3** **Codex: 3.1% → 61%**

A small share of slow calls also dominates Codex tool time.

# Tool Semantics Matter

Section 6 · Latency varies widely even within one tool




**Read/Edit calls are usually short. Commands and interactive tools span seconds to minutes: tool identity alone is insufficient.**

# Overhead Beyond Execution

Section 6 · Codex reports both internal and end-to-end timing

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 |

1

**418.1 hours**

Observed end-to-end tool time.

2

**77.8 hours**

Positive residual outside reported execution.

3

**18.6% overhead**

May include scheduling, approval, startup, and output propagation.

# Prefix Cache

Section 7 · Prefix Cache

**01** **How often does the prefix cache hit?**

**02** **What happens after a human pause?**

**03** **How does retention affect KV storage?**

**04** **How much re-prefill could be avoided?**

# Pauses Reduce Cache Hits

Section 7 · Token-weighted hit rate depends on the step trigger

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% |

1

**95.7% overall**

Token-weighted prefix-cache hit rate.

2

**84.4% · User**

User-initiated steps resume after human pauses.

3

**97.5% · Tool**

Tool-result steps usually resume quickly.

# A Pause Triggers Re-Prefill

Section 7 · One session illustrates the cost of a human pause

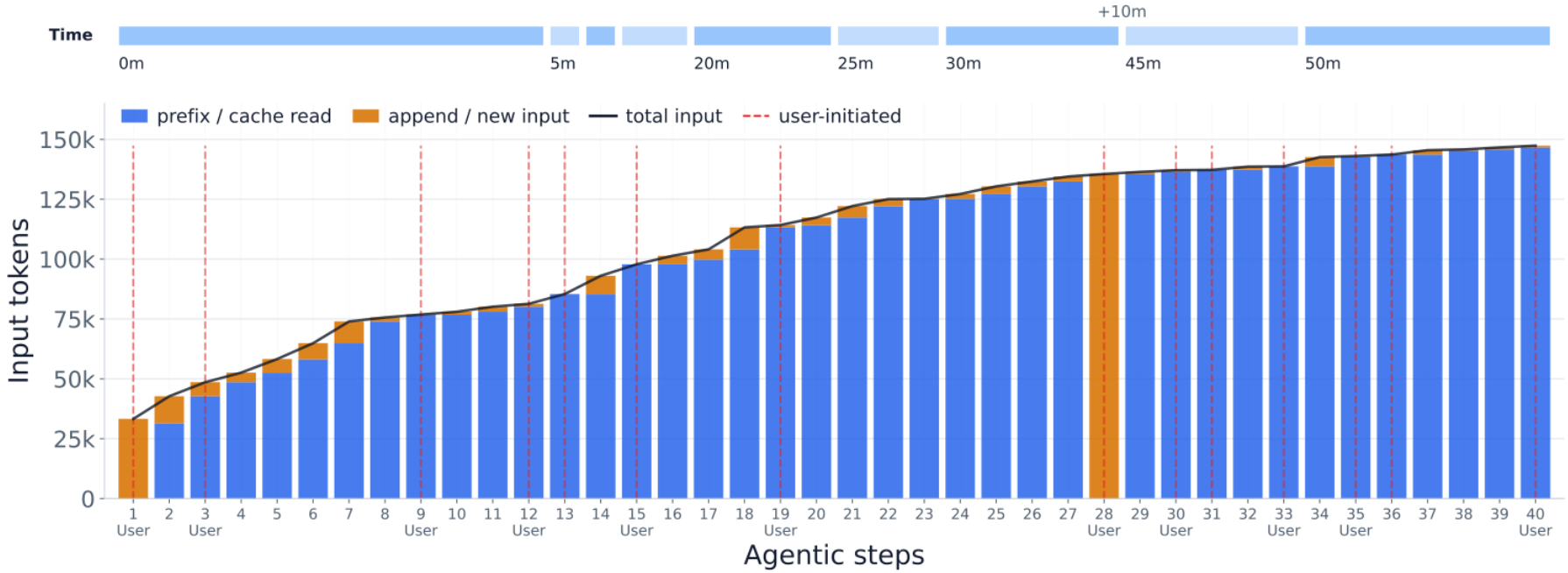



Figure 10: Example session progression over the first 40 agentic steps. Bars decompose each step input into prefix tokens and append tokens; the black line shows total input, red dashed lines mark user-initiated steps, and the top strip shows elapsed time.

**At step 28, about 10 minutes of inactivity causes a cache miss. Previously processed history must be prefilled again.**

# Idle Time and Cache Misses

Section 7 · Longer pauses increase the chance of eviction

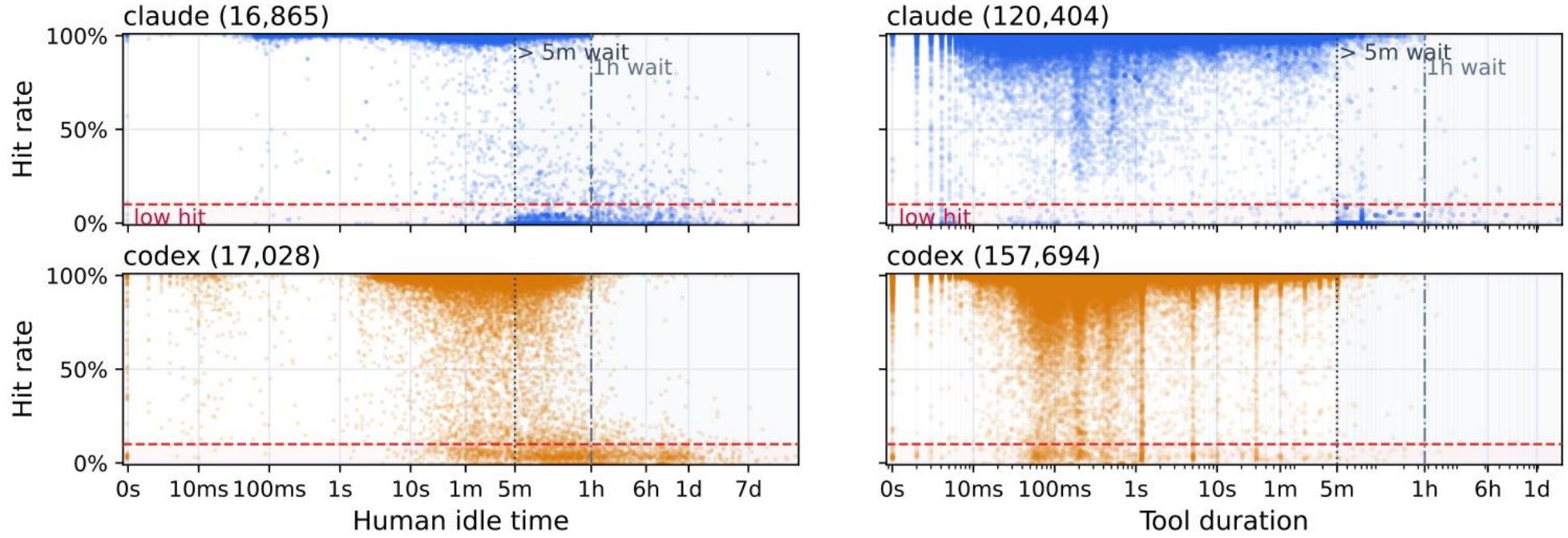
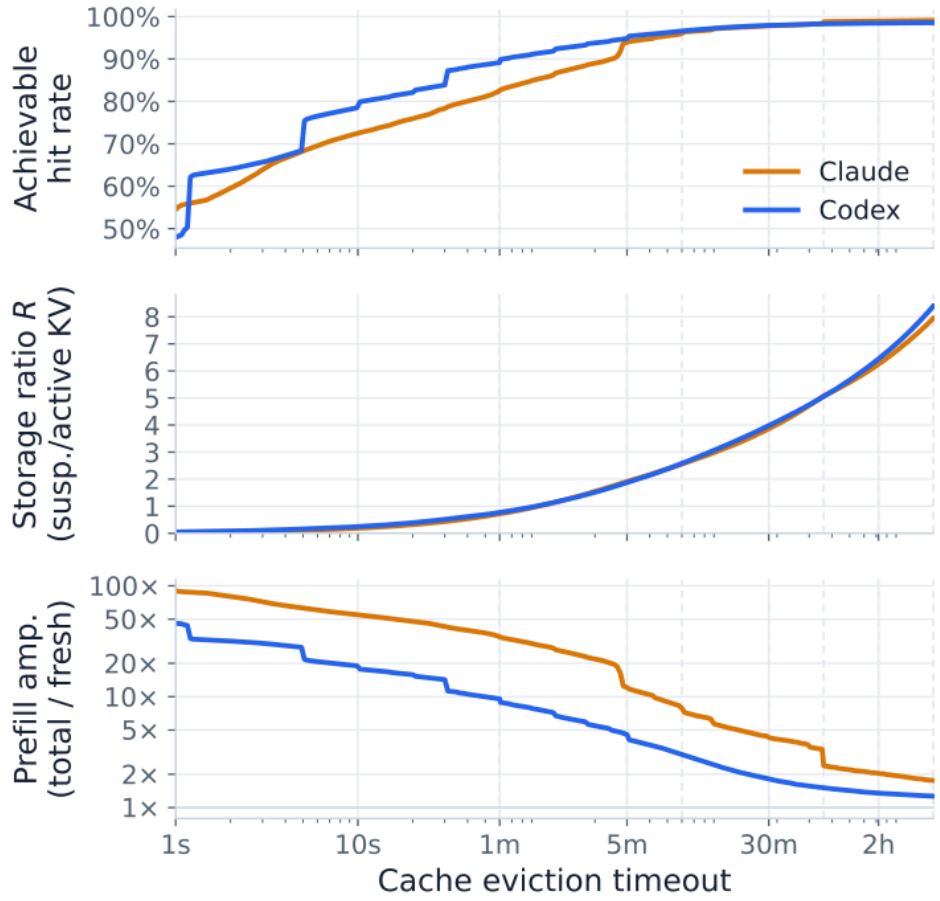



Figure 11: Prefix cache hit rate versus the idle time preceding a step (log $x$-axis), for Claude (top) and Codex (bottom).

**Low-hit steps appear after about five minutes. After roughly one hour, almost all steps miss the prefix cache.**

# Retention Has a Cost

Section 7 · Cache-hit gains must be balanced against KV storage




**1** **1 minute → 1 hour**

Achievable hit rate rises from 85.4% to 98.6%.

**2** **More KV storage**

Suspended-to-active storage ratio grows from 0.74 to 5.07.

**3** **An early gain**

At five minutes: about 94% hits and a storage ratio of 1.9.

# The Value of Retaining KV

Section 7 · Upper-bound savings from preventing cache misses

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 |

1

**45.9% fewer appends**

Upper-bound reduction in append tokens.

2

**12.8% lower cost**

Upper-bound reduction in priced cost.

3

**$5,189 in the trace**

Assumes reusable tokens can be read at the cache rate.

# Two Key Takeaways

Coding agents combine growing context with repeated tool use

**01 LLM CALLS**

## Repeated LLM calls

- One task spans multiple steps
- Context usually grows
- History and results accumulate

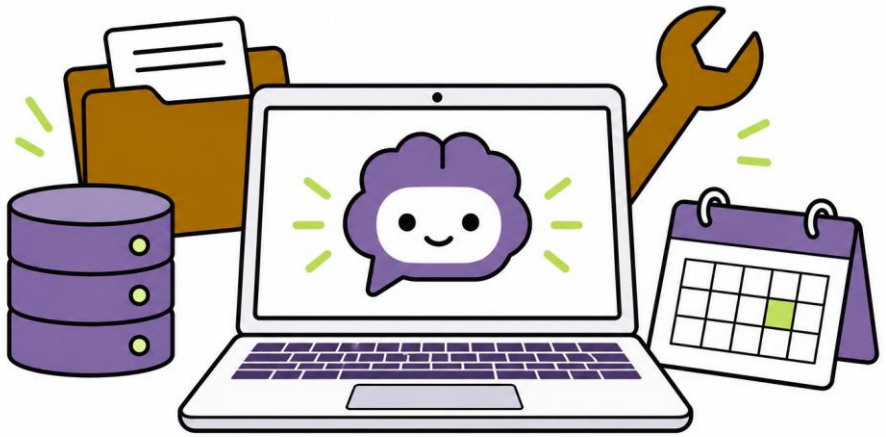
**02 TOOL CALLS**

## Repeated tool calls

- Read files and inspect code
- Edit code and run tests
- Results guide the next step

# Faster Codex

Persistent connections, response IDs, and incremental API processing



API

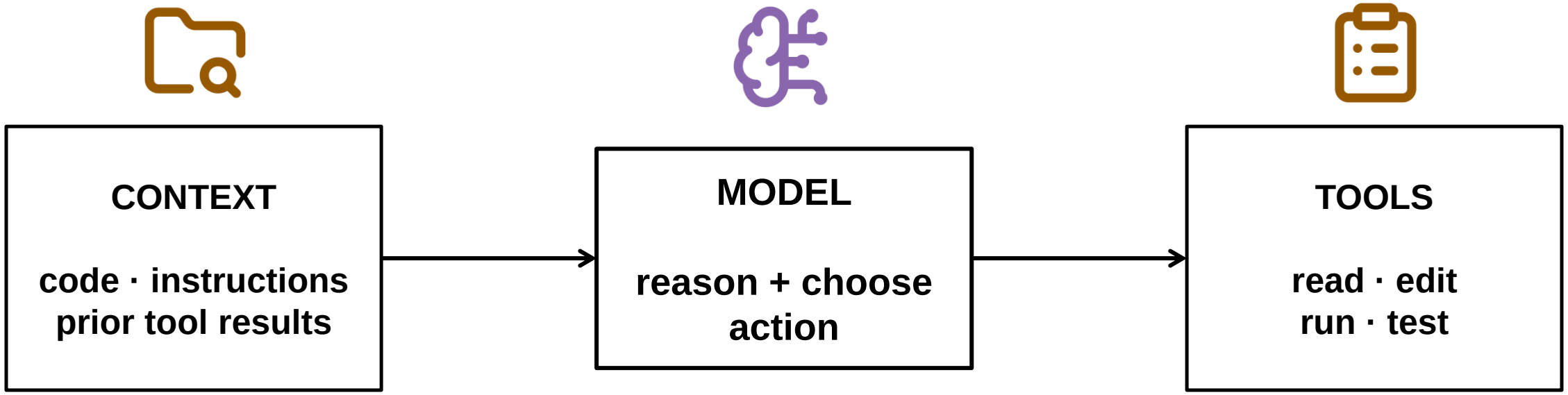
WebSocket mode

**Advanced Computer Networks**

Based on OpenAI Engineering · 22 April 2026

# Why Agent Loops Need a Faster API

Small per-turn delays accumulate across a long coding task




**Repeat with new tool results until the task is complete**

# From Repeated Work to Incremental Work

The optimization target is what the API repeats between turns

**REPEAT**

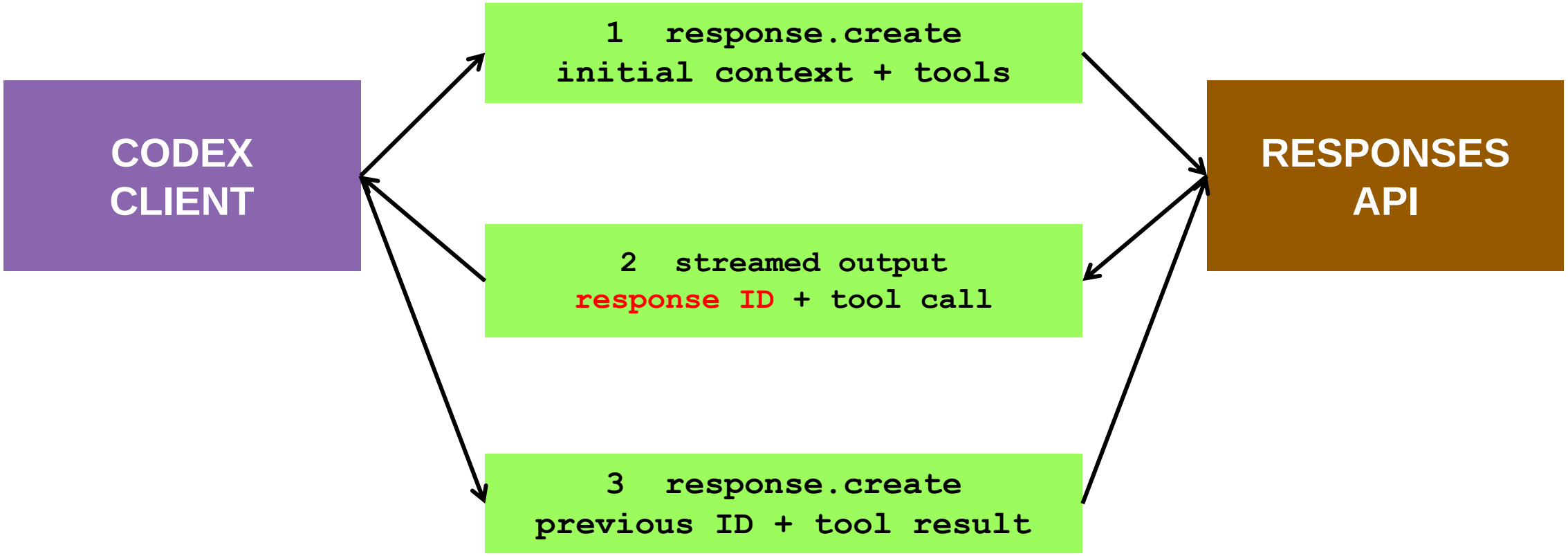
- Resend conversation history
- Rebuild request state
- Reprocess unchanged input
- Repeat setup work
- Wait before inference

**REUSE**

- Send a reference + delta
- Recover cached state
- Process the new input
- Keep useful setup results
- Start the next turn sooner

# One Connection, Multiple Turns

A persistent WebSocket carries requests and streamed responses




**Run the tool locally, then reuse the socket**

# Continue with a Response ID

A follow-up request points to existing state and supplies the delta

```
{
  "type": "response.create",
  "model": "gpt-5.4",
  "previous_response_id": "resp_123",
  "input": [{
    "type": "function_call_output",
    "call_id": "call_abc",
    "output": "Tests passed"
  }]
}
```

**1** **CREATE**

Use the familiar Responses request

**2** **REFERENCE**

Point to the prior response state

**3** **APPEND**

Send only the new tool result

# What State Does the API Reuse?

The response ID is a handle to state already available on the server

**TOOLS**

**Definitions**

Schemas and namespaces needed for the next model request

**HISTORY**

**Response state**

Previous response plus prior input and output items

**TOKENS**

**Rendered input**

Reusable token data that can be extended with new input

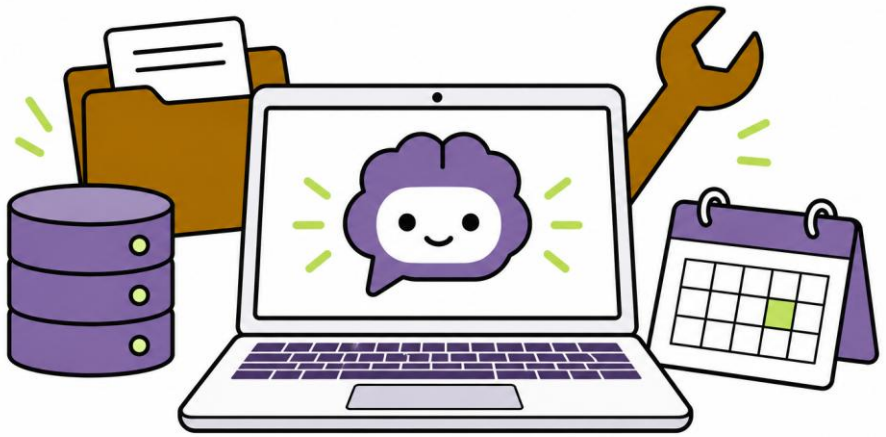
# Reported Workflow Improvements

Examples cited by OpenAI Engineering · 22 April 2026

- Codex quickly ramped the majority of their traffic onto WebSockets. Codex users running the latest models such as GPT-5.3-Codex, GPT-5.4, and beyond all benefit from WebSocket mode's speed up.
- Vercel integrated WebSocket mode into the AI SDK and saw latency decrease by up to 40%.
- Cline's multi-file workflows are 39% faster.
- OpenAI models in Cursor became up to 30% faster.

# MCP Explained

Connecting AI applications to tools, data, and workflows



MCP

**Advanced Computer Networks**

Official documentation walkthrough

2026-07-28

# MCP Roadmap

Four questions: motivation, architecture, protocol, and workflow

**01** **Why do agents need a standard interface?**

**02** **Who participates, and what can a Server expose?**

**03** **How are messages represented and transported?**

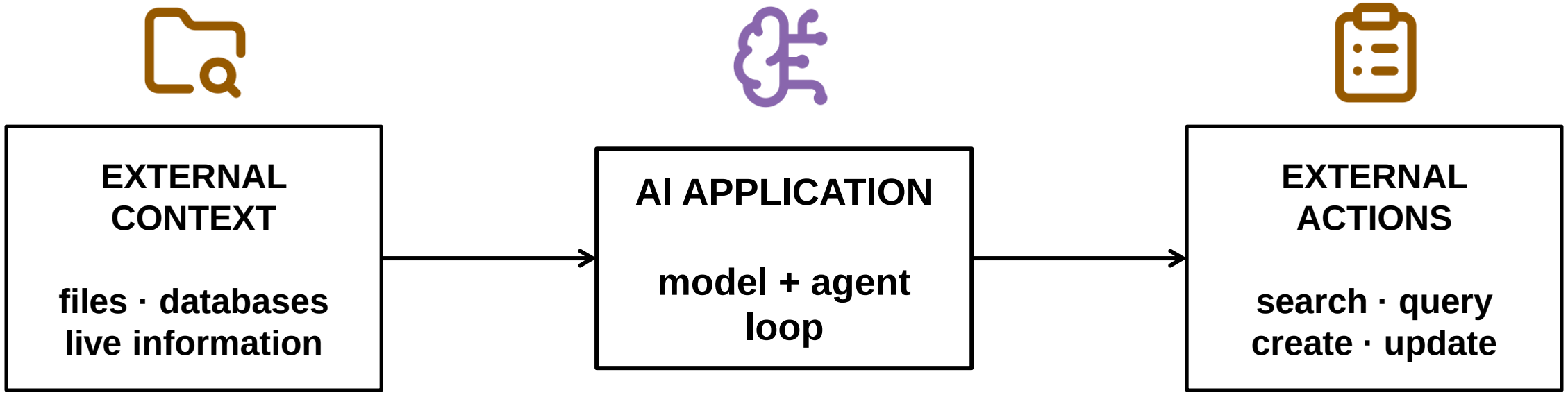
**04** **How does a complete tool interaction work?**

# Why MCP?

Part 1 · From repeated tool use to a common interface

# Why MCP?

AI applications need external context and actions

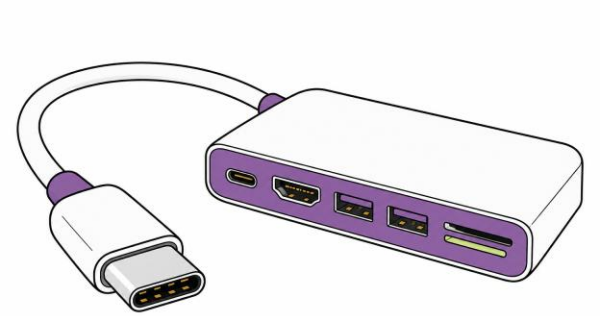



**The application needs a reliable interface for both context and action.**

# What is MCP?

A standard connection between AI applications and external systems

**“An open-source standard for connecting AI applications to external systems.”**

*“A USB-C port for AI applications”*

# What Can MCP Do?

Official examples combine context and action

**Personal assistant**

Calendar + Notion context

**Coding agent**

Figma design → web app

**Enterprise chatbot**

Query multiple databases

**Design agent**

Blender model → 3D printer

# Roles and Capabilities

Part 2 · Who connects, and what does a Server expose?

# How MCP Works

## Client-server architecture




[2]

# How MCP Works

## Client-server architecture




[2]

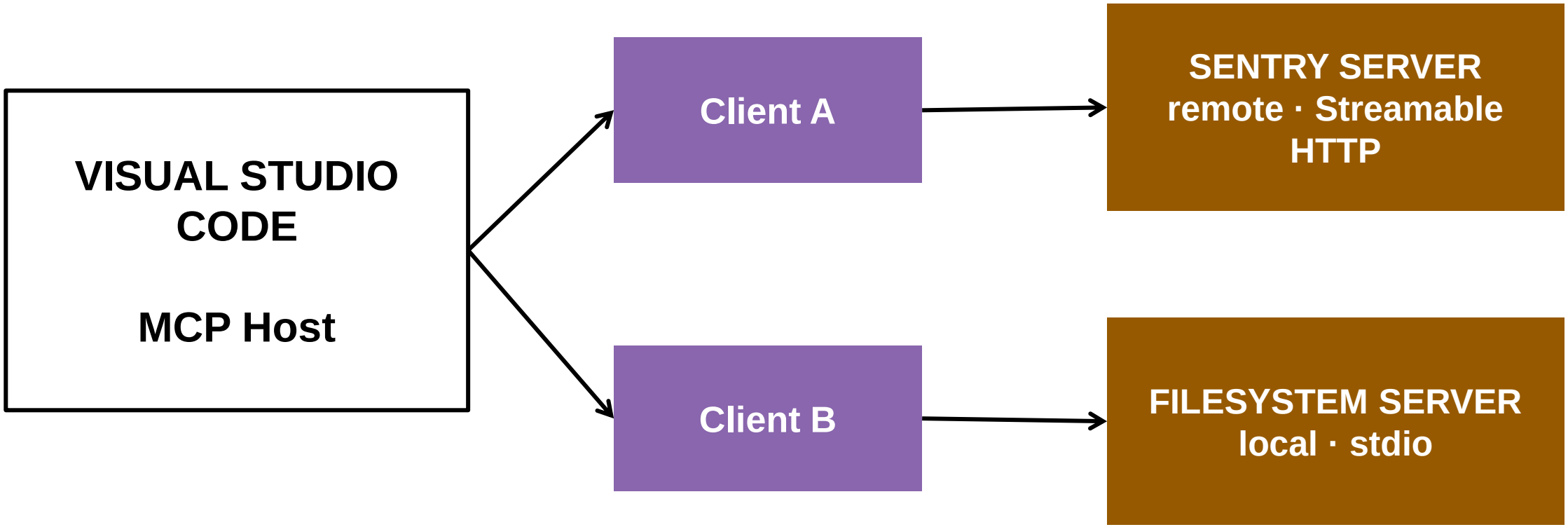
# MCP Architecture

Host, Client, and Server have distinct responsibilities


**Host creates one Client for each Server it connects to.**

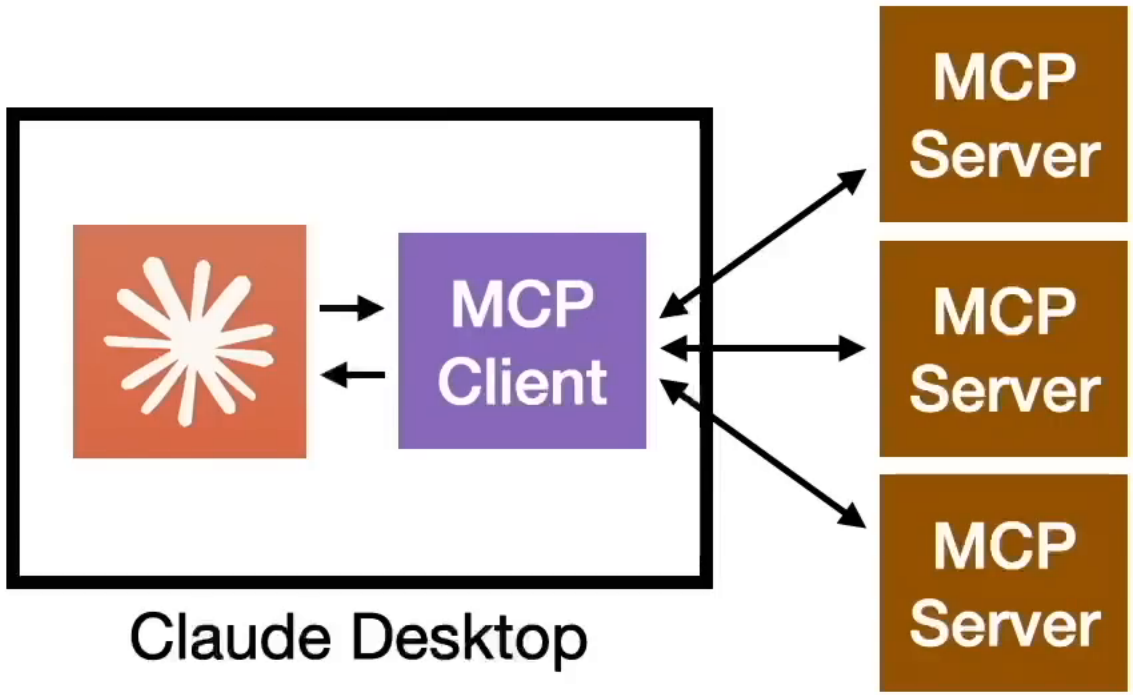
# Example: VS Code

One Host creates a dedicated Client for each Server



# MCP Client

Built into AI apps and send requests to MCP servers




**Client Responsibilities**

- 🔍 Discover server capabilities
- 💿 Receive data from servers
- 🪛 Manage LLM tool execution

**Typically don't need to built this**

# Local and Remote Servers

Deployment location changes how messages are transported

**LOCAL SERVER**

**Usually launched by the Host**

- same machine
- stdio transport
- typically one Client

**REMOTE SERVER**

**Runs as a network service**

- server platform
- Streamable HTTP
- typically many Clients

# Three Server Primitives

Tools perform actions; Resources supply data; Prompts guide use

**TOOLS**

**Executable functions**

file operations
API calls
database queries

**RESOURCES**

**Addressable data**

file contents
database records
API responses

**PROMPTS**

**Reusable templates**

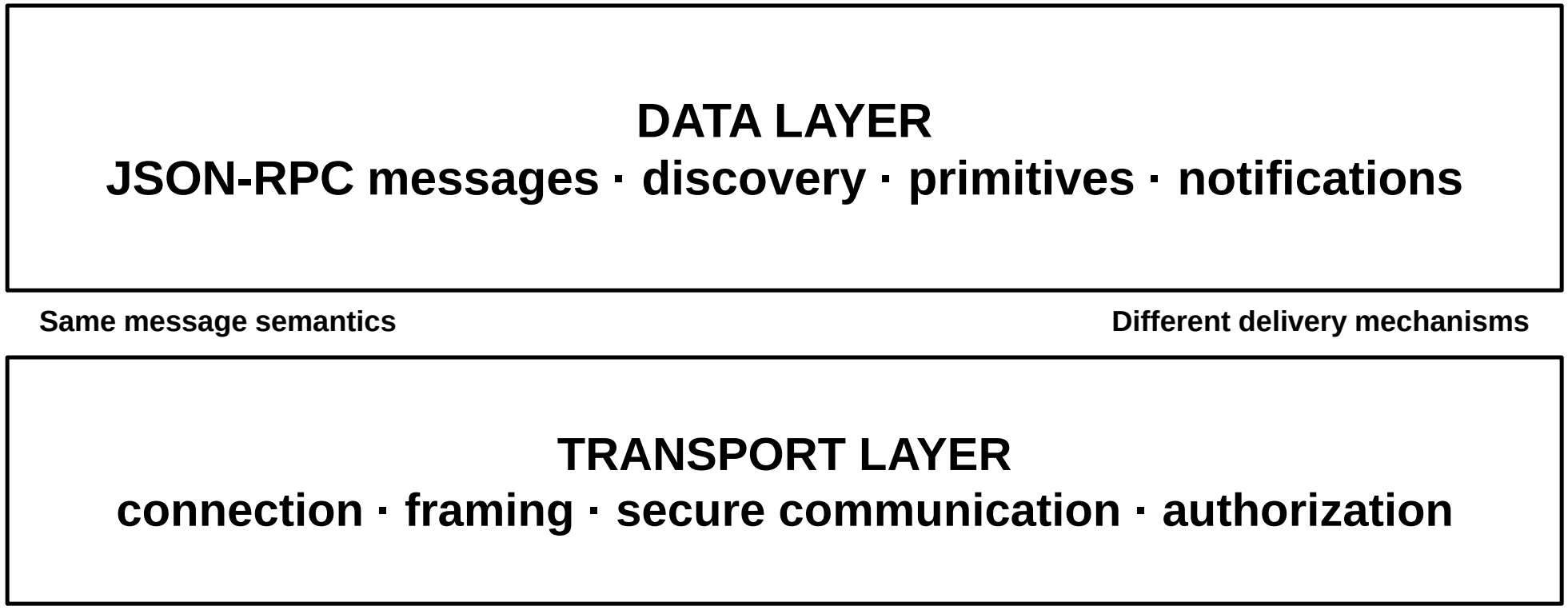
system prompts
few-shot examples
workflow scaffolds

# Messages and Transport

Part 3 · From interfaces to messages and connections

# Two Protocol Layers

MCP separates message semantics from message delivery



# Transport Mechanisms

Local processes use stdio; remote Servers use Streamable HTTP

## stdio

**Direct communication between local processes**

- stdin / stdout framing
- no network overhead
- Host often launches Server

## Streamable HTTP

**Remote communication using HTTP POST**

- optional SSE streaming
- HTTP authentication
- OAuth recommended

# Stateless Requests

Each request carries its protocol version and capabilities

```
"_meta": {
  "io.modelcontextprotocol/protocolVersion":
    "2026-07-28",
  "io.modelcontextprotocol/clientInfo": {
    "name": "example-client",
    "version": "1.0.0"
  },

"io.modelcontextprotocol/clientCapabilities":
{
    "elicitation": {}
  }
}
```

**WHY THIS MATTERS**

- Server does not rely on connection state
- Version travels with the request
- Capabilities are explicit
- Client identity aids compatibility

# JSON-RPC Messages

Requests, responses, and notifications use different patterns

| REQUEST | RESPONSE | NOTIFICATION |
| --- | --- | --- |
| **has id** | **same id** | **no id** |
| **expects response** | **result or error** | **no response** |

**Every method keeps the familiar JSON-RPC 2.0 envelope.**

# An End-to-End MCP Interaction

Part 4 · Put the roles, capabilities, and protocol together

# The Official Walkthrough

Discover capabilities, list tools, call a tool, and subscribe

**1**

**DISCOVER**

server identity, versions, capabilities

→

**2**

**LIST TOOLS**

names, descriptions, input schemas

→

**3**

**CALL TOOL**

typed arguments and structured result

→

**4**

**SUBSCRIBE**

changes trigger a catalog refresh

**We now read the wire messages in this order.**

# 1. Discover Capabilities

server/discover identifies supported capabilities

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "server/discover",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion":
        "2026-07-28",

"io.modelcontextprotocol/clientCapabilities":
        { "elicitation": {} }
    }
  }
}
```

**1** **Method**

server/discover is implemented by every Server.

**2** **Version**

The Client declares the protocol version it is speaking.

**3** **Capabilities**

The Client says it can support elicitation.

# Discovery Response

The Server declares compatibility, capabilities, and cache policy

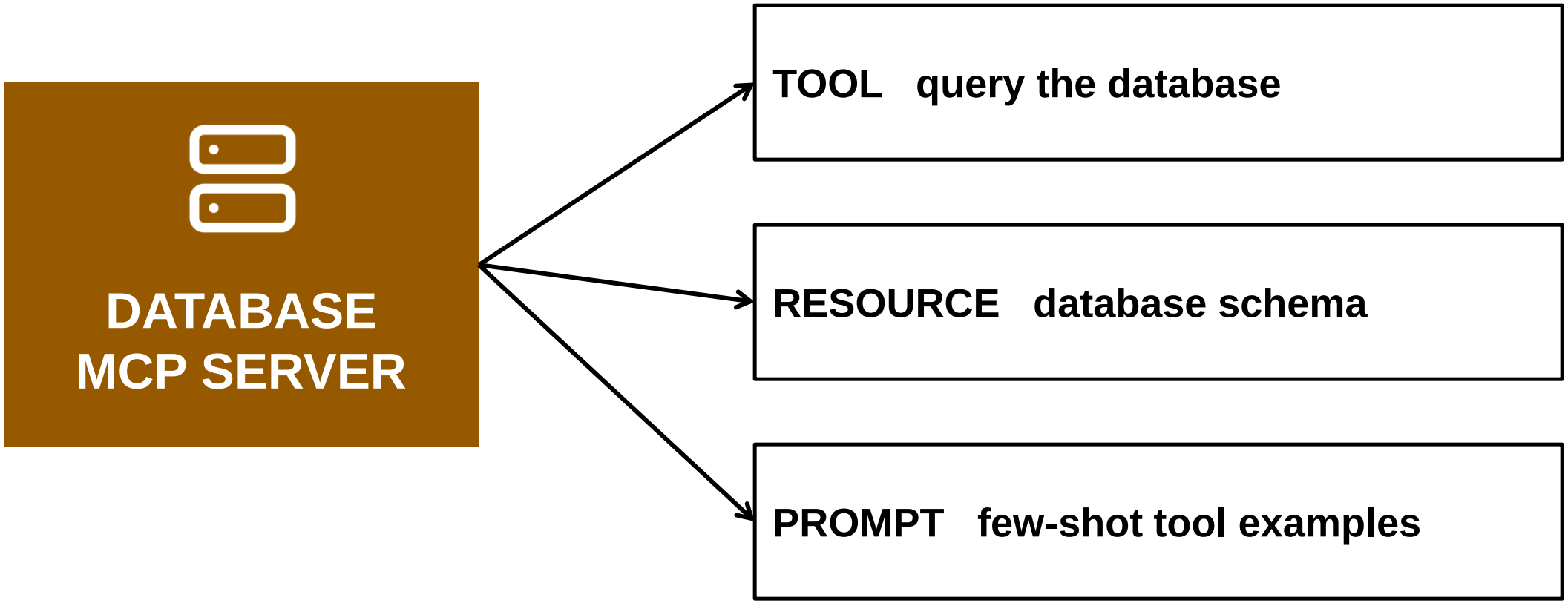
```
{
  "supportedVersions": ["2026-07-28"],
  "capabilities": {
    "tools": { "listChanged": true },
    "resources": {}
  },
  "_meta": {
    "io.modelcontextprotocol/serverInfo": {
      "name": "example-server",
      "version": "1.0.0"
    }
  },
  "ttlMs": 3600000,
  "cacheScope": "public"
}
```

**READ THE RESPONSE**

- Mutually supported version
- Tools + Resources available
- Tool-list changes supported
- Server identity and version
- Cacheable for one hour

# Example: Database Server

One Server can expose Tools, Resources, and Prompts



# 2. Discover Tools

tools/list describes the tools available to the Host

```
{
  "tools": [
    {
      "name": "calculator_arithmetic",
      "title": "Calculator"
    },
    {
      "name": "weather_current",
      "title": "Weather Information",
      "inputSchema": {
        "required": ["location"]
      }
    }
  ],
  "ttlMs": 300000
}
```

**WHAT THE HOST LEARNS**

- Unique tool names
- Human-readable titles
- Detailed descriptions
- JSON input schemas
- Freshness and cache scope

# The Tool Contract

Names, descriptions, and schemas guide tool selection and use

**`weather_current`**

**Get current weather information for any location worldwide**

```
location   string   required
units      metric | imperial | kelvin
```

**`name`**

**routes the call**

**`description`**

**helps the model select**

**`inputSchema`**

**documents + validates**

# Python: Build the Tool Registry

Official Python pseudocode · tools/list

```
# Pseudo-code using MCP Python SDK patterns
available_tools = []
for client in app.mcp_clients():
    tools_response = await client.list_tools()
    available_tools.extend(tools_response.tools)
conversation.register_available_tools(available_tools)
```

**1** **List**

list_tools() maps to a tools/list request.

**2** **Combine**

Collect tool metadata from connected Servers.

**3** **Expose to the LLM**

The app registers these tool definitions for the conversation.

# 3. Call a Tool

tools/call sends the tool name and schema-shaped arguments

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tools/call",
  "params": {
    "name": "weather_current",
    "arguments": {
      "location": "San Francisco",
      "units": "imperial"
    },
    "_meta": {
      "io.modelcontextprotocol/protocolVersion":
        "2026-07-28"
    }
  }
}
```

**1** **Exact name**

Must match a name returned by tools/list.

**2** **Arguments**

Follow the published input schema.

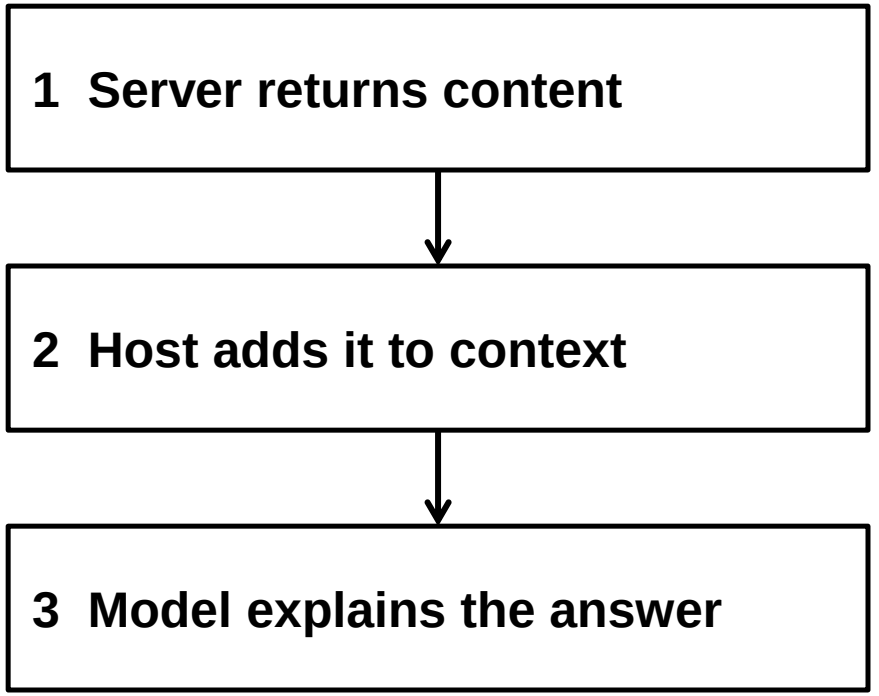
**3** **Request id**

Correlates this call with its response.

# Tool Results

The Server returns content; the Host passes it to the model

```
{
  "result": {
    "resultType": "complete",
    "content": [
      {
        "type": "text",
        "text": "San Francisco: 68°F
..."
      }
    ]
  }
}
```




**Responses may contain text, images, resources, or other structured content.**

# Python: Execute a Tool Call

Official Python pseudocode · tools/call

```
# Pseudo-code for AI application tool execution
async def handle_tool_call(conversation, tool_name, arguments):
    client = app.find_mcp_client_for_tool(tool_name)
    result = await client.call_tool(tool_name, arguments)
    conversation.add_tool_result(result.content)
```

**1** **Route**

The app finds the Client for the selected tool.

**2** **Execute**

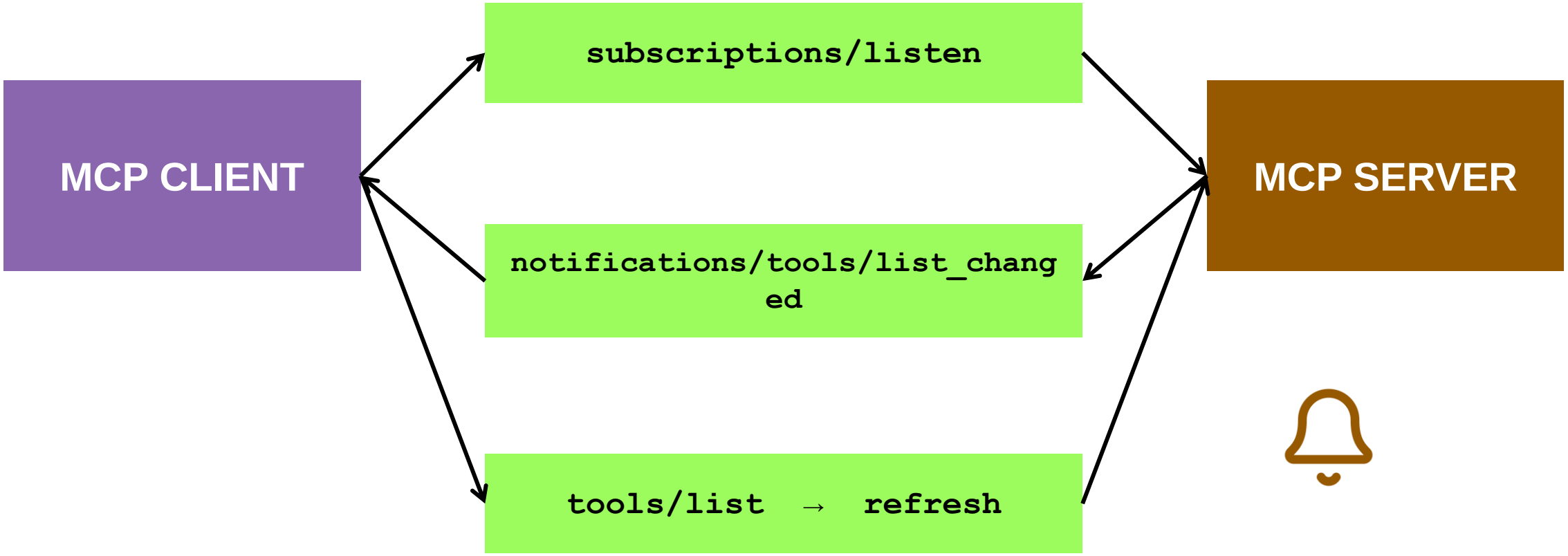
call_tool() sends the name and arguments.

**3** **Return context**

Add result.content to the conversation for the LLM.

# 4. Subscribe to Changes

Notifications tell the Client when a catalog needs refreshing




**No id in a notification → no response is expected**

# Python: Refresh Available Tools

Official Python pseudocode · subscriptions/listen

```
# Pseudo-code for AI application notification handling
async def follow_tool_changes(client):
    async with client.listen(tools_list_changed=True) as sub:
        async for _event in sub:
            tools_response = await client.list_tools()
            app.update_available_tools(client, tools_response.tools)
            if app.conversation.is_active():
                app.conversation.notify_llm_of_new_capabilities()
```

**1** **Subscribe**

listen() requests tool-list change events.

**2** **Refresh**

After an event, list_tools() fetches current definitions.

**3** **Update context**

Update the app registry and notify an active conversation.

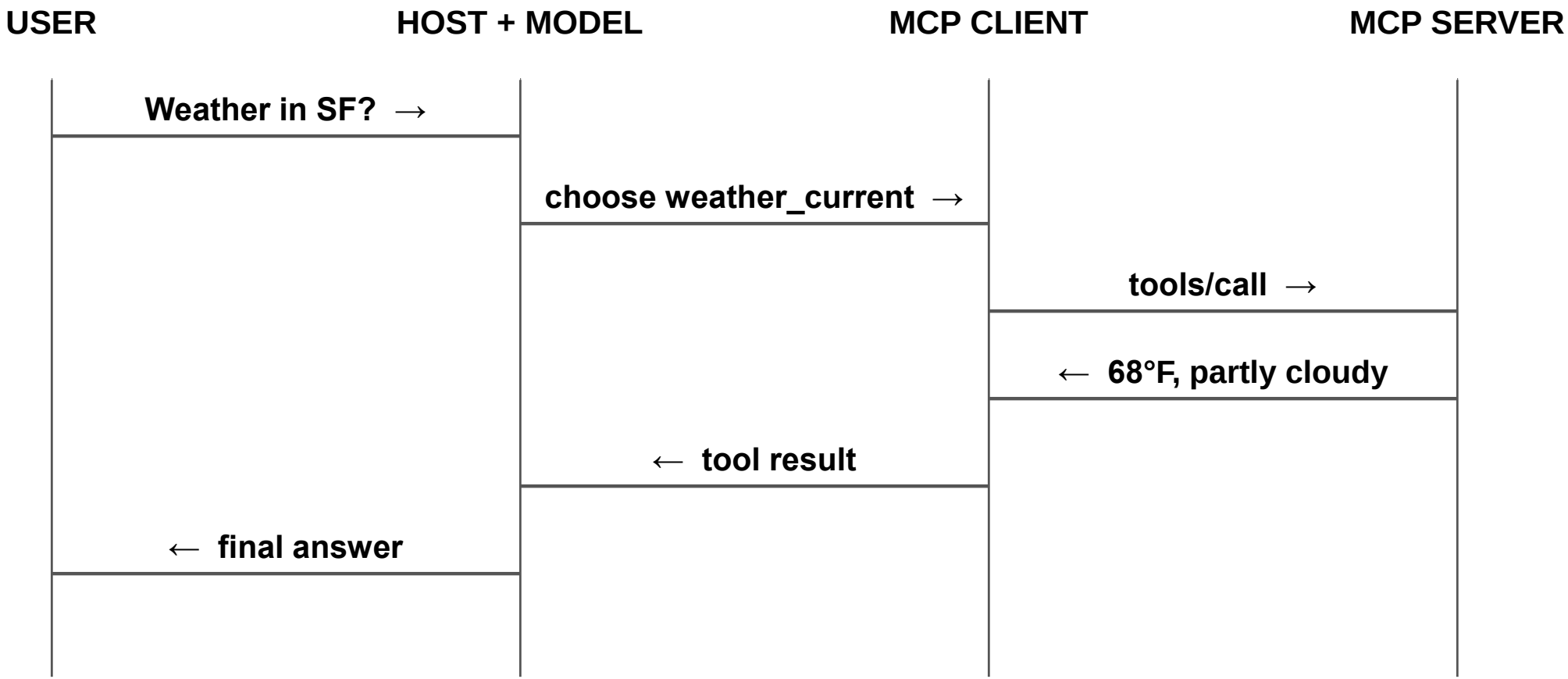
# Requesting User Input

Elicitation asks the user for missing information


**The Host controls how the request is shown and what the user approves.**

# Example: Weather Query

Follow the request from user intent to the final answer



# Classroom Exercise

Design an MCP Server for campus network diagnosis

**Scenario**

**A student reports: “The course website is unreachable.”**

**TOOLS**

**Which actions?**
**ping_host · dns_lookup · check_http**

**RESOURCES**

**Which context?**
**network map · outage bulletin**

**PROMPT**

**Which reusable workflow?**
**diagnose_connectivity**