KEEPING CONTEXT SMALL

systems lessons for AI agents

Saksham Dhull
AI Agents, WisdomAI



00

INTRODUCTION

I'm Saksham Dhull. I lead the **AI Agents** team at **WisdomAI**, where we build agents for data analysis and business intelligence.

I graduated from IIT Delhi in 2021 and have been in the tech industry for around five years now.



IN THIS SESSION

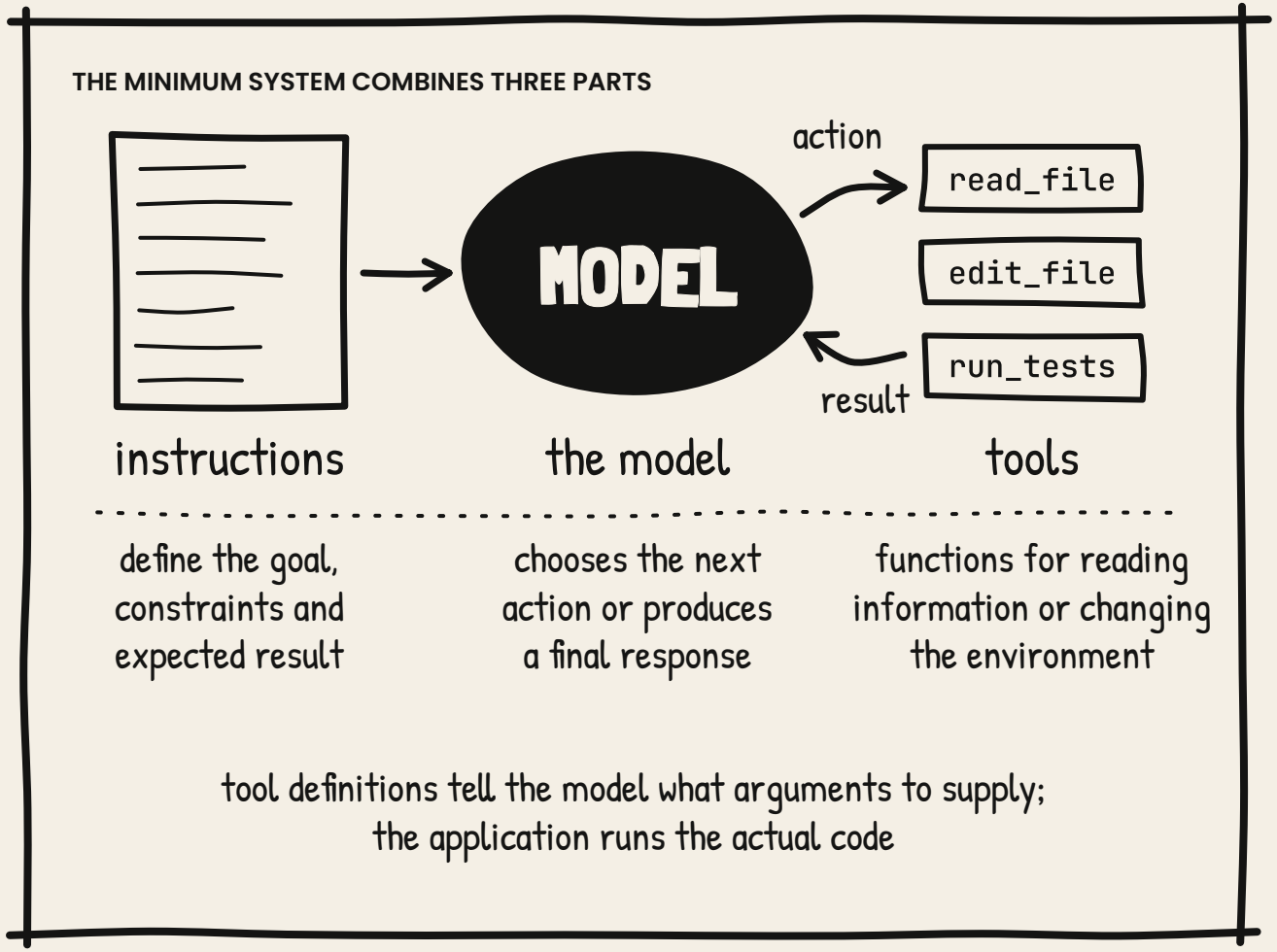
- We will understand the issues faced by agentic systems when the context and capacity of an agent grow.
- How do you maintain the performance while still boosting capacity?
- What knowledge can we borrow from systems software to improve our AI agents?

We'll follow a coding agent as its investigations get longer and its capabilities expand.

WHAT IS AN AGENT?

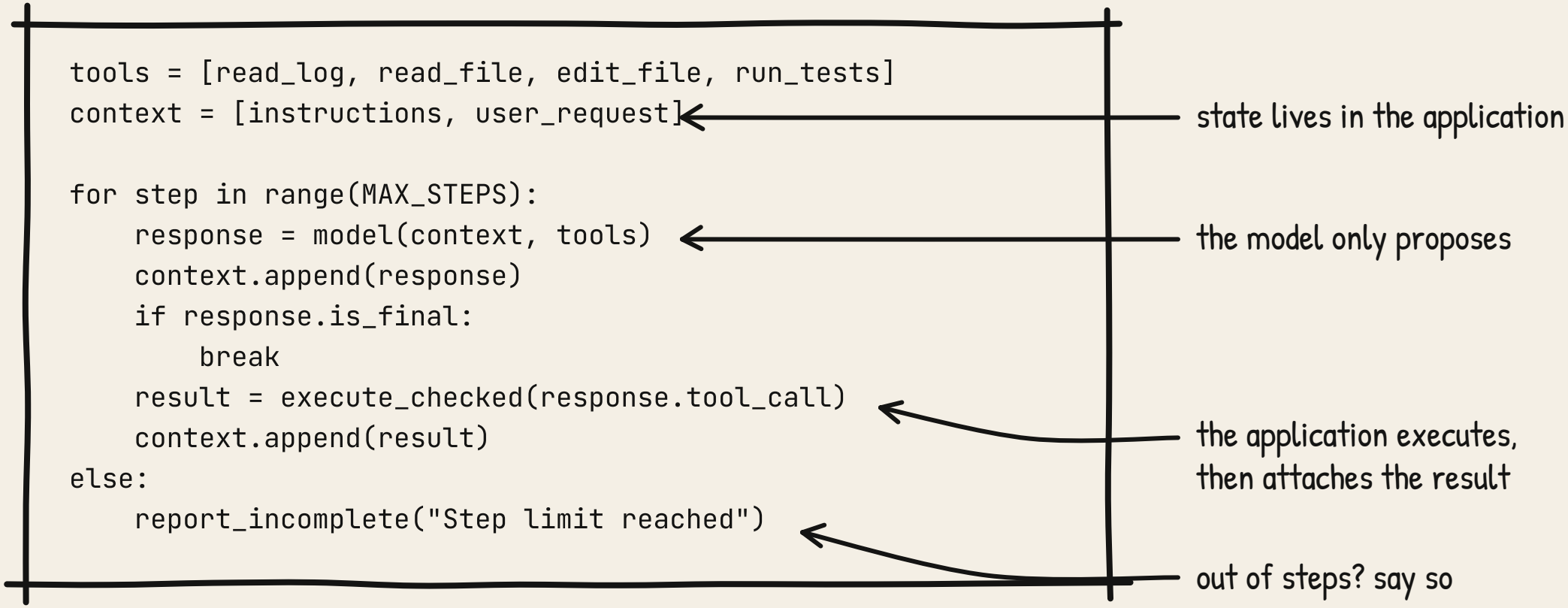
An **agent** is a system in which a large language model chooses an action, receives its result, and uses that feedback to decide what to do next. The model proposes actions; the surrounding application executes them.

This is useful when the next step depends on information discovered during the task. A coding agent may inspect a test failure before deciding which file to read. A fixed workflow already specifies those steps in code; an agent delegates some of those decisions to the model.



THE AGENT LOOP

The application sends the instructions, request and available tool definitions to the model. A model-generated tool call specifies a function name and arguments; it does not execute the function. The application runs the requested function and attaches its result before asking the model for its next decision.



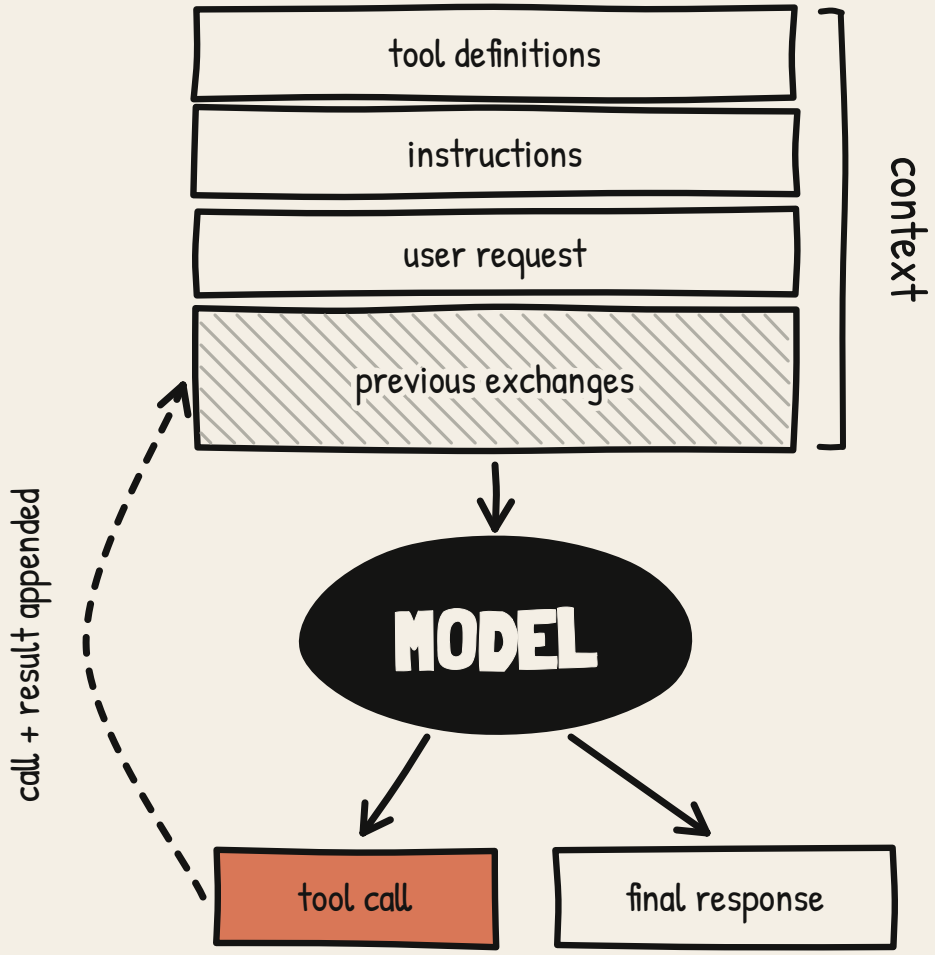
This pseudocode assumes one tool call or a final response per turn.

HOW A CALL WORKS

Before each call, our application assembles the instructions, user request, available tool definitions and previous exchanges. Together, these form the **context**: the information the model can use to decide what to do next.

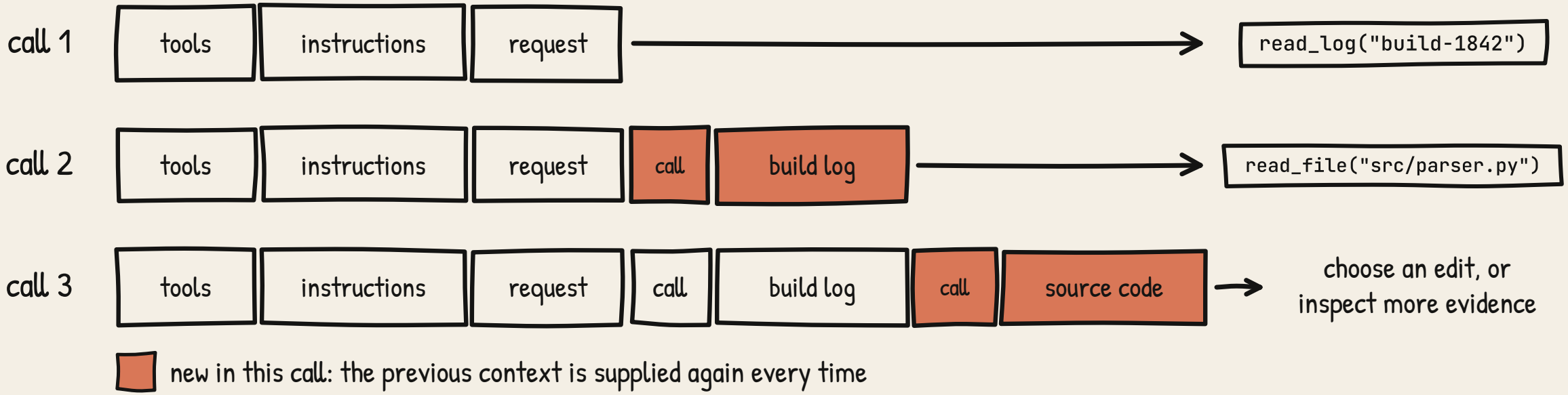
Model calls are stateless: the model does not independently remember previous calls. Our application maintains continuity by supplying earlier actions and observations. Our initial agent includes the complete history; later, context management will decide which parts to retain.

The model then returns either a tool call or a final response. For a tool call, our application executes the function, appends the call and its result to the history, and sends the updated context on the next iteration. **The application maintains the state; the model uses that state to choose the next action.**



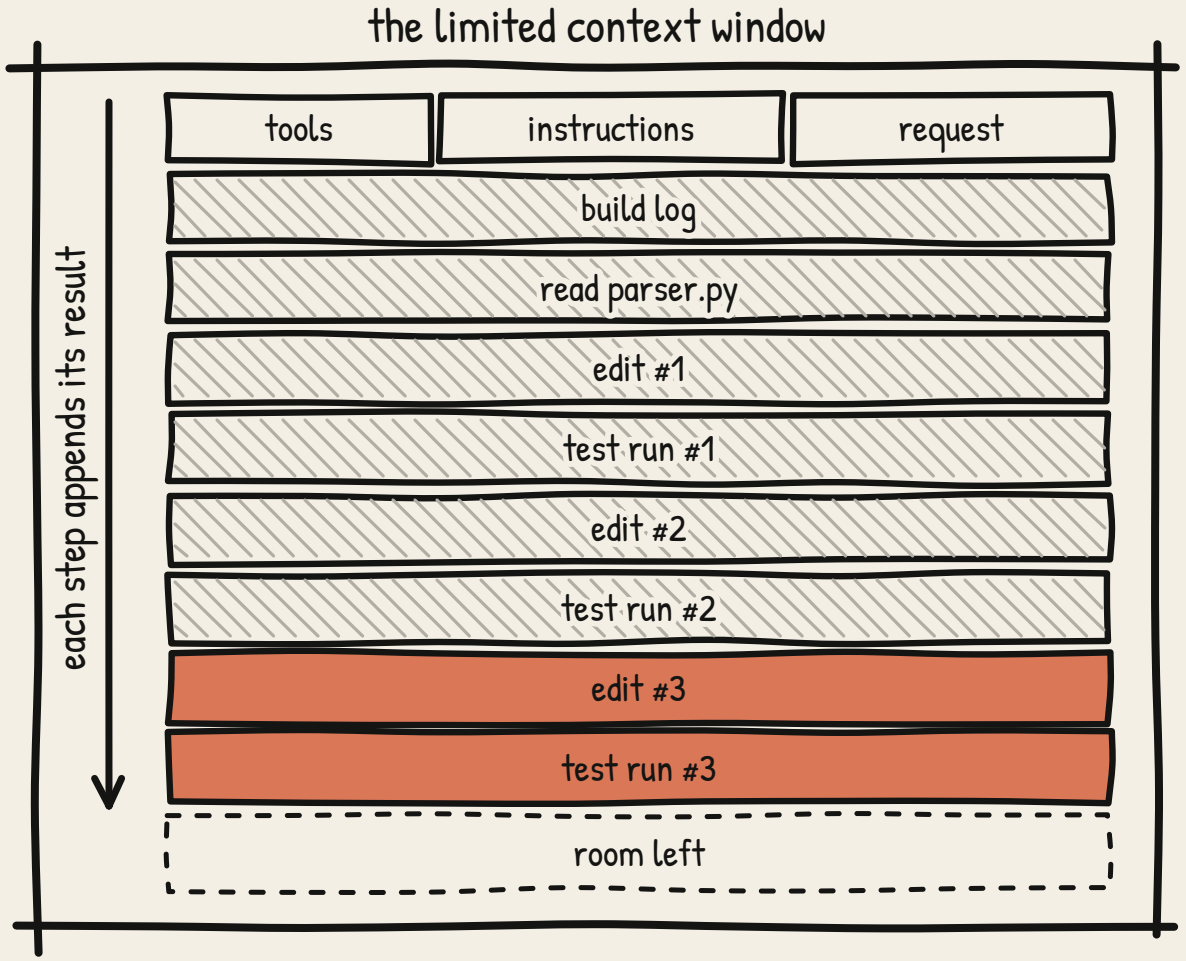
HOW A CALL WORKS

Let's follow a coding agent given the request: "**Investigate this failing build and prepare a fix.**" It can read logs and source files, edit a working copy, and run tests. Success means producing a relevant patch that passes the targeted and related tests without changing the public API or dependencies.



Each new result helps the agent choose its next action. **But our loop also carries every earlier result into subsequent calls.**

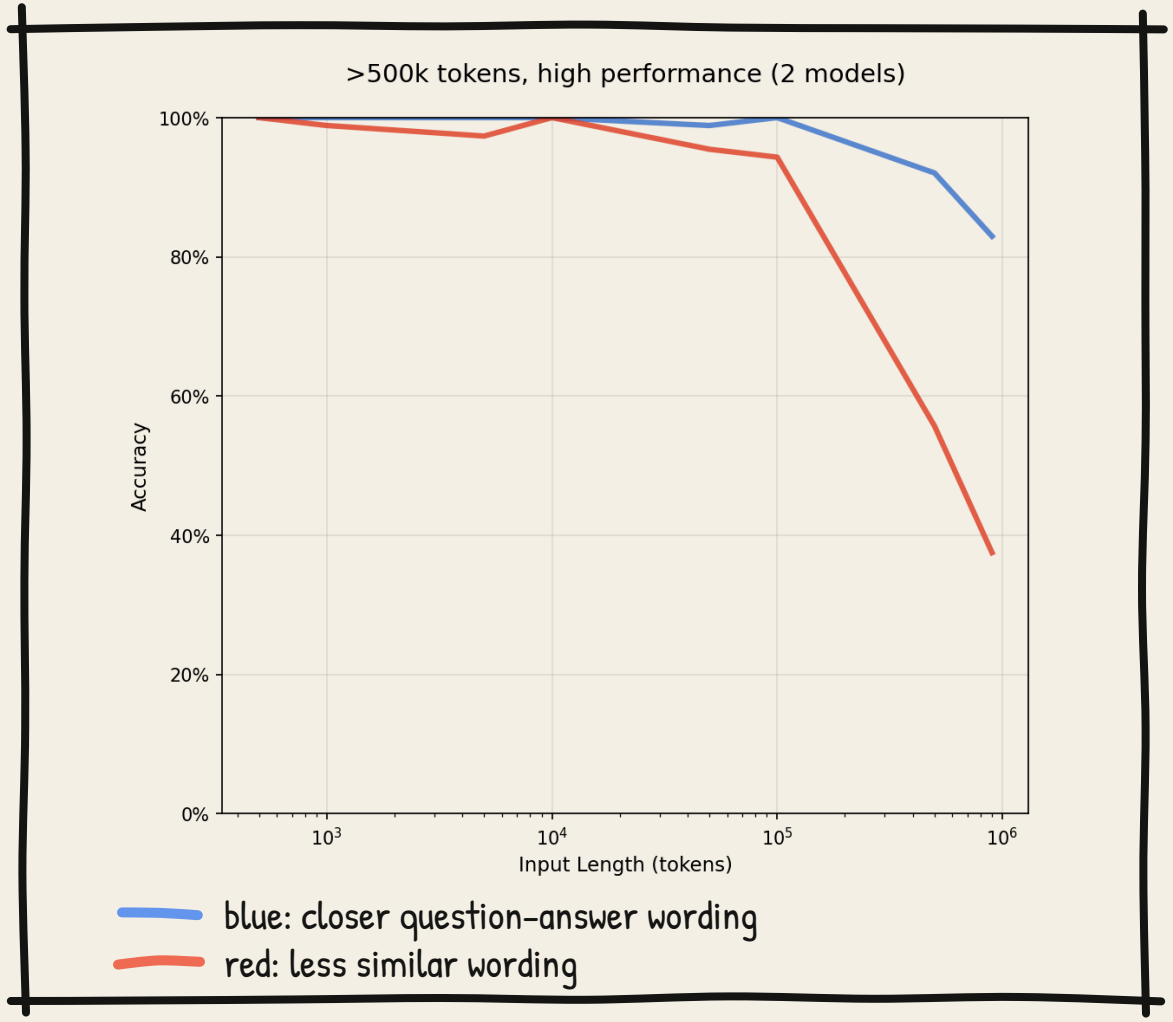
CONTEXT ISSUES IN LONGER TASKS



Each time our coding agent reads a file, makes an edit or runs a test, it adds another result to the conversation. Our loop carries all those results into subsequent calls, so the context keeps growing as the investigation continues. The next decision needs the current state, but the model also receives the entire path taken to reach it.

This creates two problems: **older material consumes the limited context window**, and **a longer input can make the model less accurate at using relevant information**.

CONTEXT ISSUES IN LONGER TASKS



A longer input can make the model less accurate at using relevant information.

Chroma's *Context Rot* study observed declining performance as input length increased while task complexity remained fixed. The extent of that decline varied across models and tasks.

SOURCE: CHROMA, CONTEXT ROT, JULY 2025. ONE OF SIX PANELS: RETRIEVAL ACCURACY VERSUS INPUT LENGTH.

06

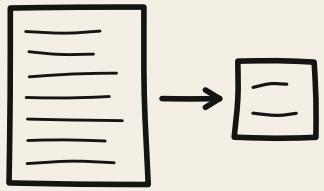
CONTEXT MANAGEMENT

We therefore need a policy for deciding what stays in the next request: keep what remains useful, summarize completed work, and discard irrelevant material. A useful starting policy has three rules:



KEEP

- current requirements
- active plan and progress
- relevant code
- the latest useful test result



SUMMARIZE

- previous investigations
- completed steps



DISCARD

- irrelevant tool outputs
- redundant history
- outdated hypotheses

For our coding agent, we keep the current patch and pending checks, summarize the investigation, and remove older logs from the context. This makes the next request smaller, but some of the removed detail may become useful again.

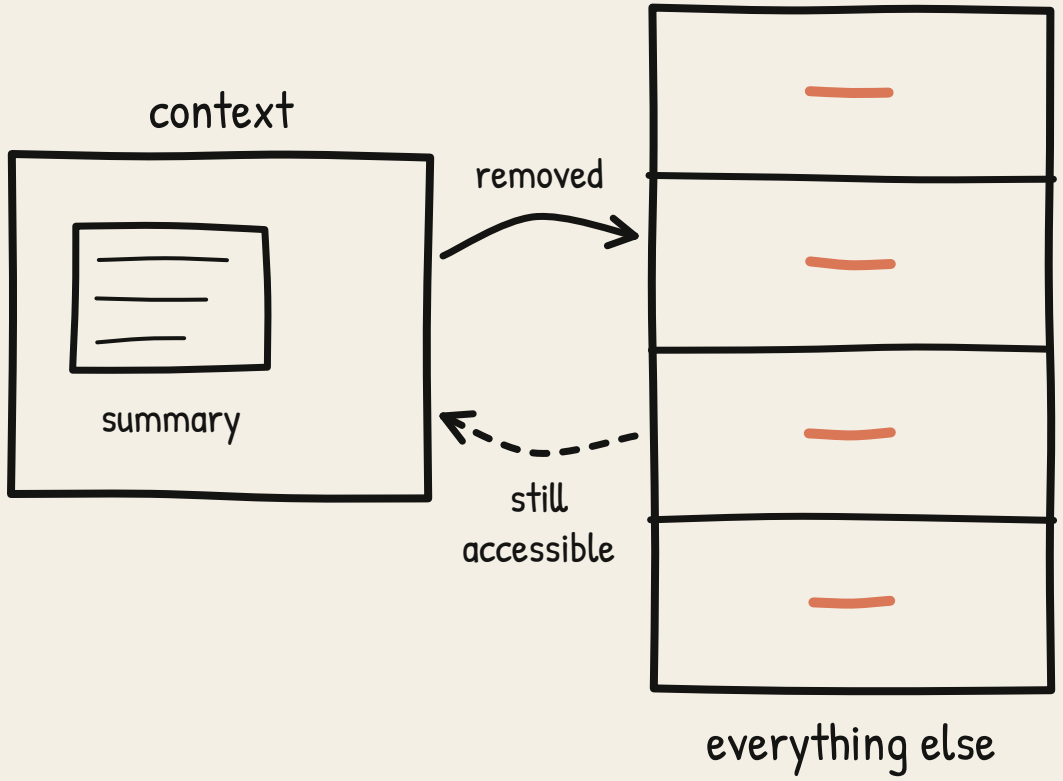
CONTEXT MANAGEMENT

Suppose a later test fails differently. The agent may need the original stack trace to compare the failures instead of just the summary.

We therefore need to separate:

REMOVING INFORMATION FROM CONTEXT ≠ LOSING ACCESS TO IT

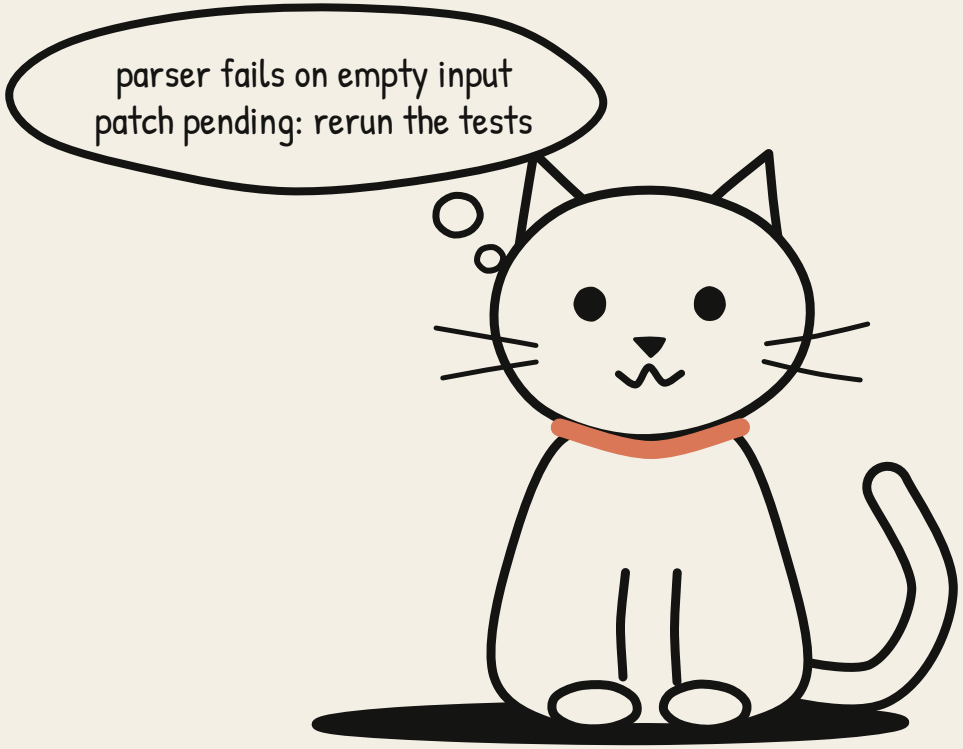
This is where memory comes in.



07

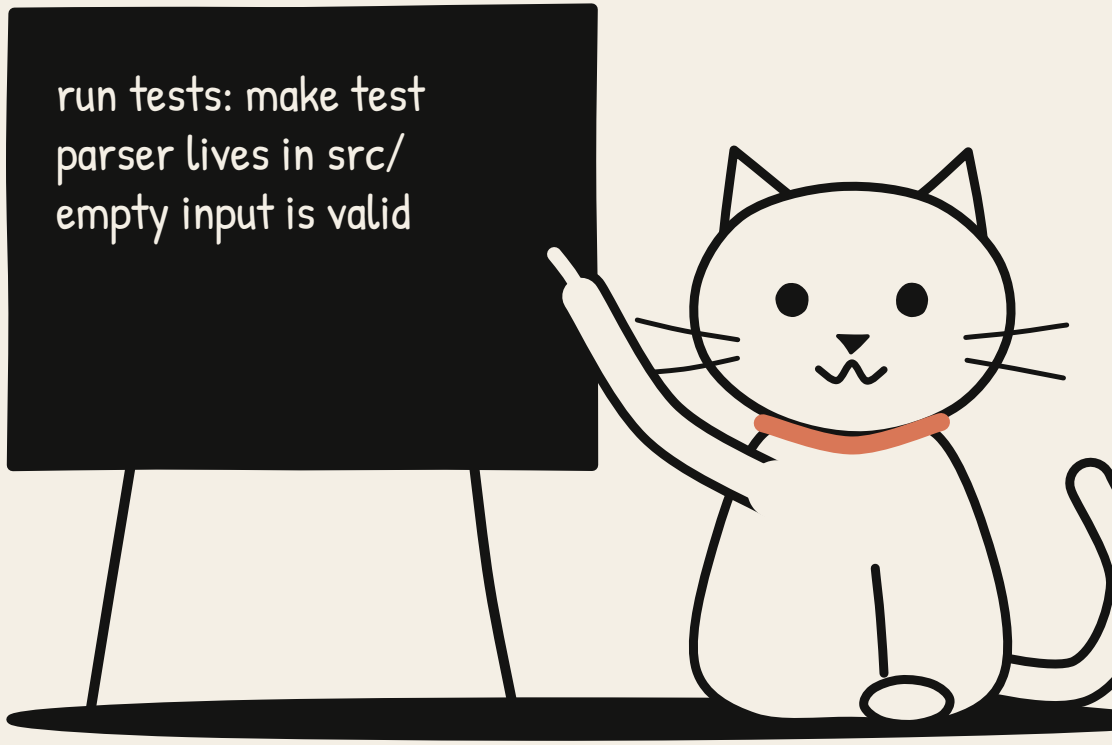
MEMORY

Memory stores information for later use while context management selects what the model needs now. A useful distinction is the scope of that memory:



SHORT-TERM MEMORY

maintains continuity within the current task or conversation: recent observations, findings, changes and pending work

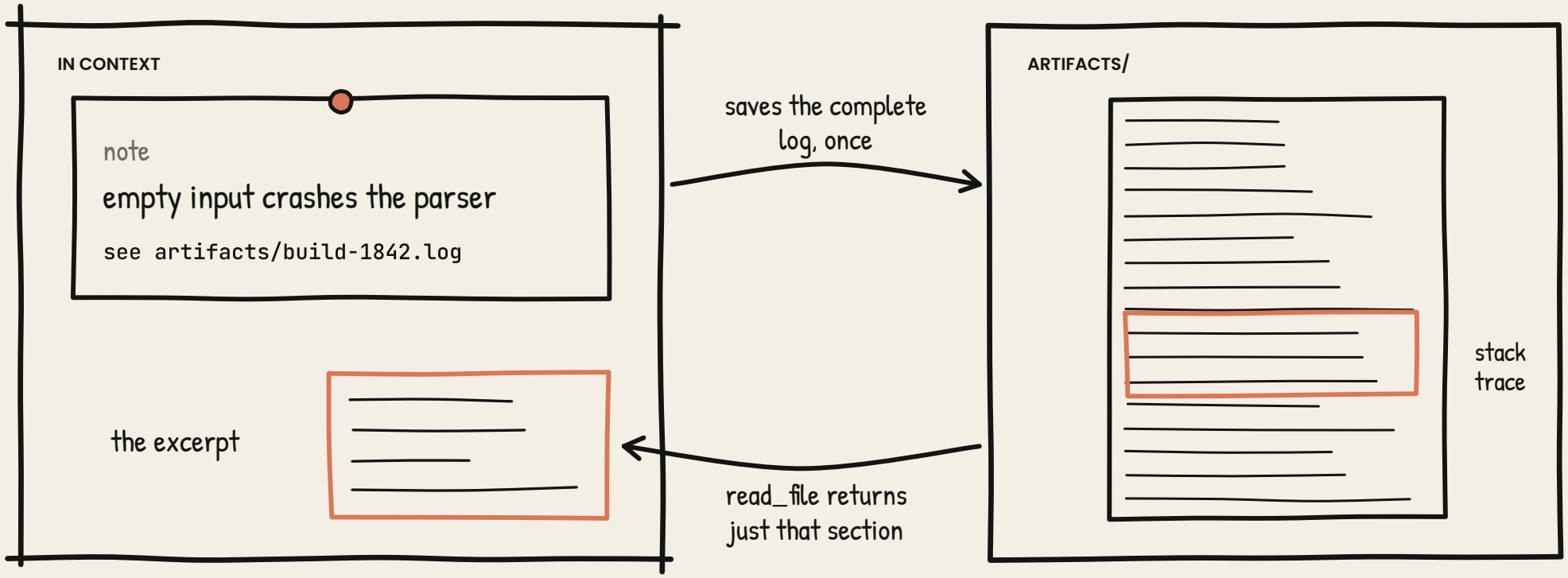


LONG-TERM MEMORY

retains useful information across separate tasks or conversations

MEMORY

For our build investigation, we can implement this with a note and stored artifacts. The application saves the complete log at `artifacts/build-1842.log`, while the note retains the relevant finding and a reference to that file. If the agent later needs the original stack trace, it requests that section through `read_file`. The application returns the excerpt, making it available for the next decision without reloading the entire investigation.



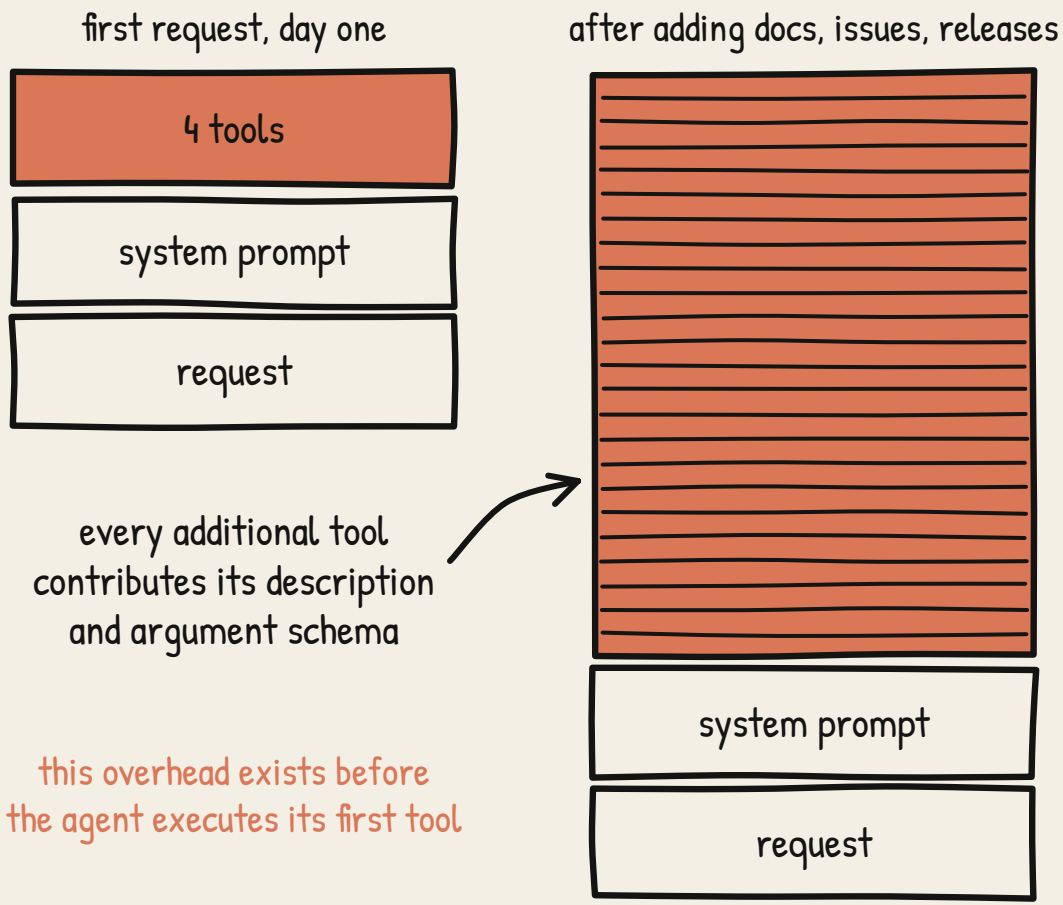
HANDLING MORE CAPABILITIES

So far, we've looked at how context grows as an agent works through a task. But what if the context is already large before the agent takes its first action?

As our coding agent grows, we add capabilities for searching documentation, inspecting issues and checking releases. In our example, the first request already contains **the system prompt, tool definitions and the user's request**. So every additional tool contributes its description and argument schema to that. This overhead exists before the agent executes its first tool. **Trimming conversation history does not reduce the tool definitions loaded upfront.** We need to separate what the system can do from what the model must see right now.

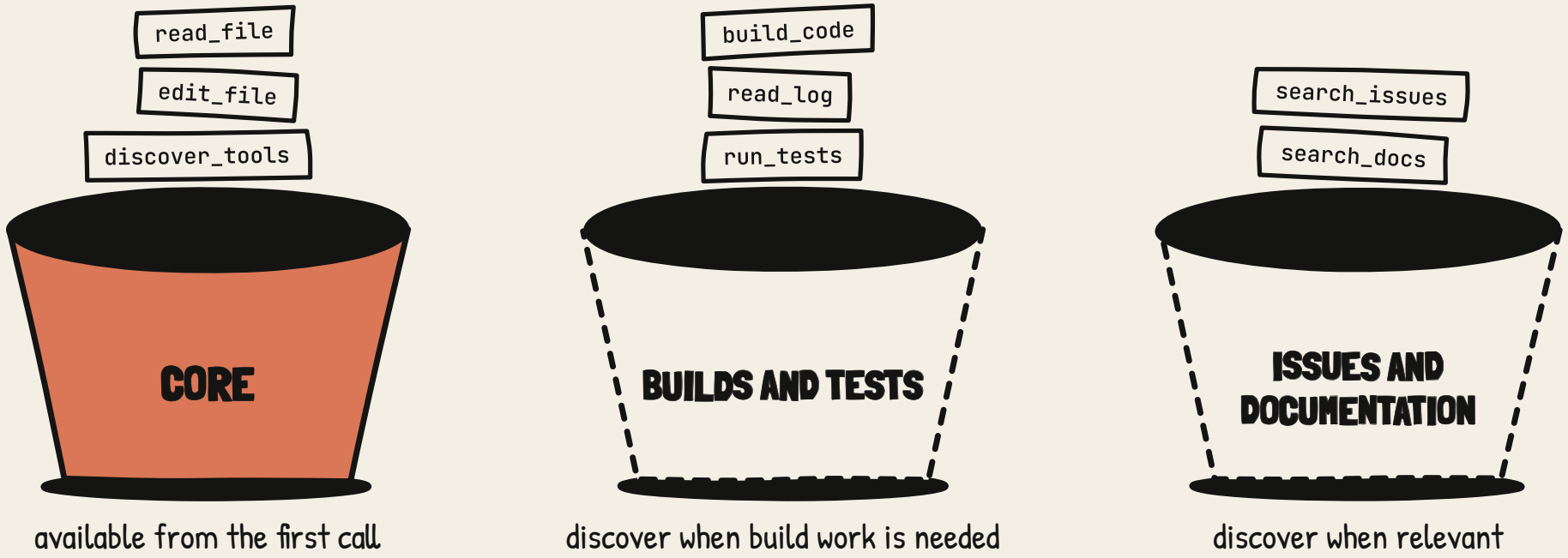
The system prompt defines the agent's role, responsibilities and constraints. We can make those instructions more concise, but we still need to preserve the requirements that govern the run.

Tools are different. Our agent may support many operations, but does it need every tool for every task? **Can we load tool definitions when they become relevant, just as we retrieve other information when needed?**



LAZY LOADING WITH TOOL SPACES

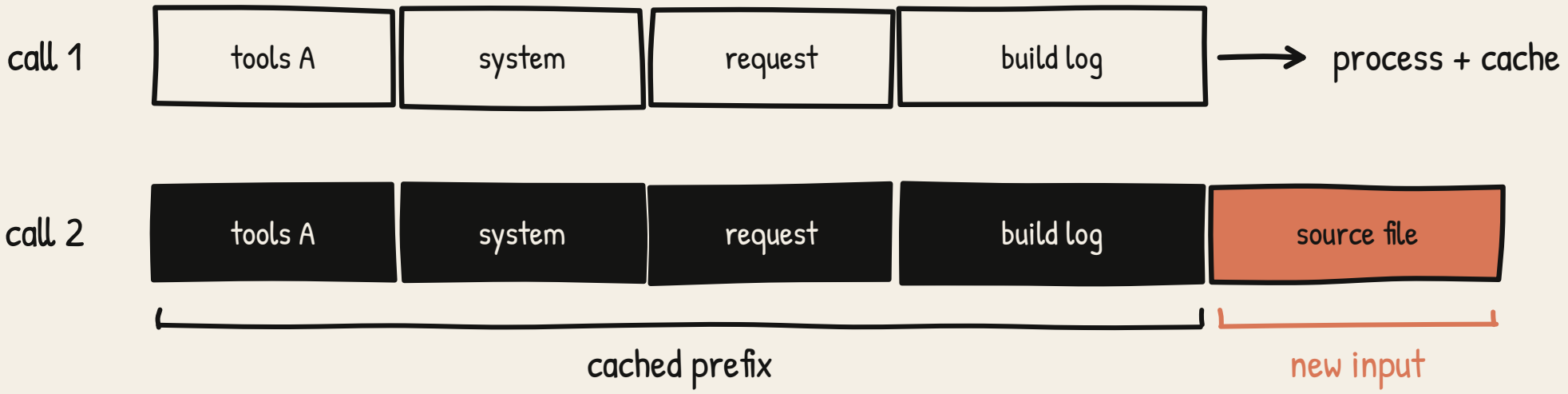
Instead of loading every tool upfront, we define tool spaces. Tool spaces group relevant tools into buckets. The agent then starts with a small set of essential tools and a way to discover additional capabilities (load a tool space). When it needs a capability, it loads the relevant bucket. This is **lazy loading**: making tools available without including all of them in the initial context.



Lazy loading trades a smaller initial context for discovery work, which can add model calls and latency. But the cost is not limited to that lookup: how we add the newly discovered tools can also affect how much processing the next request reuses.

PROMPT CACHING AND TOOL SPACES

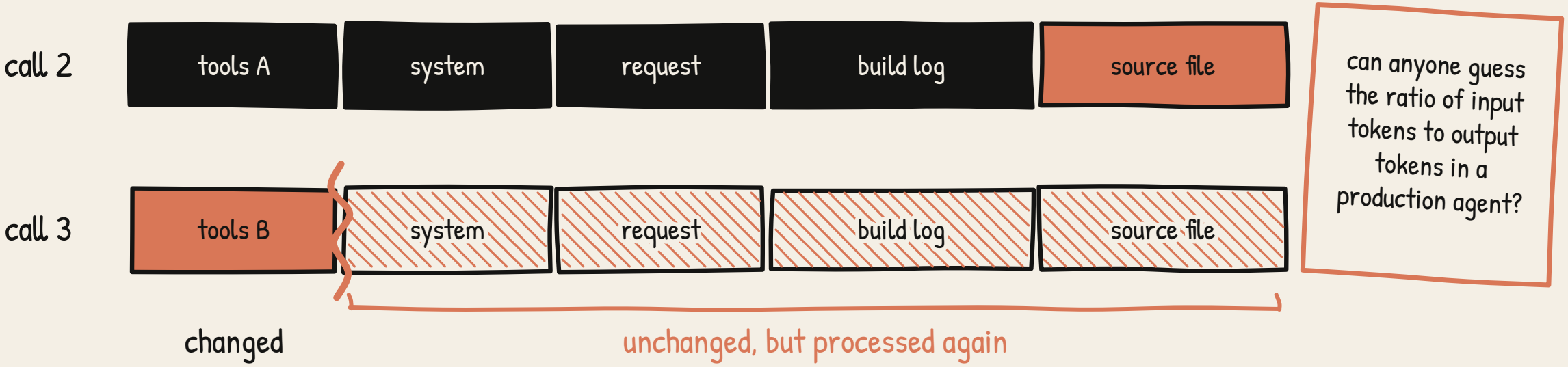
Across consecutive calls, our agent repeatedly supplies the same instructions, task and earlier observations. **Prompt caching** reuses processing for an unchanged beginning of that input, the *prefix*, when a matching cache entry is available. In Claude's documented layout, the order is **tool definitions → system prompt → conversation**.



The second call reuses the shared beginning and processes the new source-file input. It still generates a new response; cached tokens still occupy context. **Caching reduces repeated input processing, not the context size or the work of executing tools.**

PROMPT CACHING AND TOOL SPACES

If lazy loading replaces the initial catalogue, changing tools A to tools B, the earlier full prefix stops matching. We can lose reuse of the system prompt and conversation even though those parts have not changed.



1/10

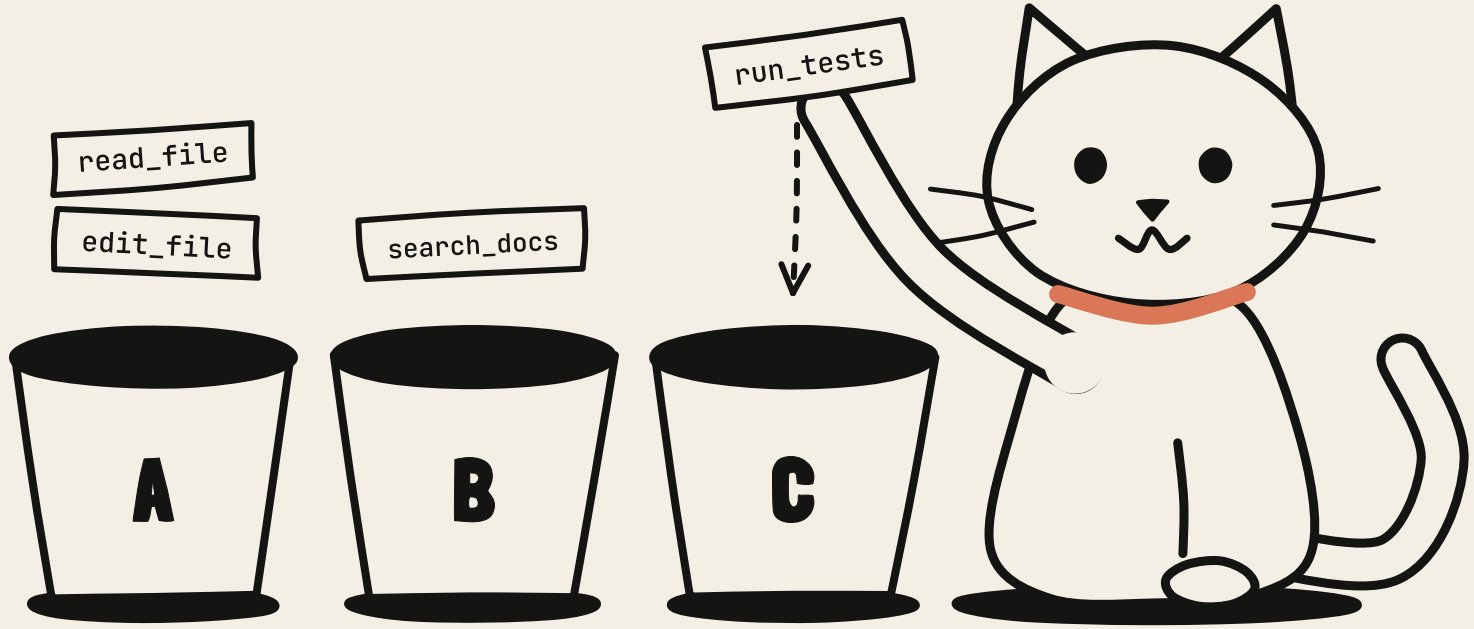
For frontier models, cache reads cost 1/10th of the standard input-token rate. Losing a hit increases the cost of the affected input.

So, how should we group and load tools so our cost and latency don't take the capacity hit?

GROUPING BY CATEGORY

This is the simplest starting point. We classify and group the tools by their responsibilities, so every user or account gets the same predefined groups and loading policy.

But this categorization depends on what the tools do, not how an account actually uses them. An account that uses your agent heavily on some tools from each of the buckets ends up loading all of them.



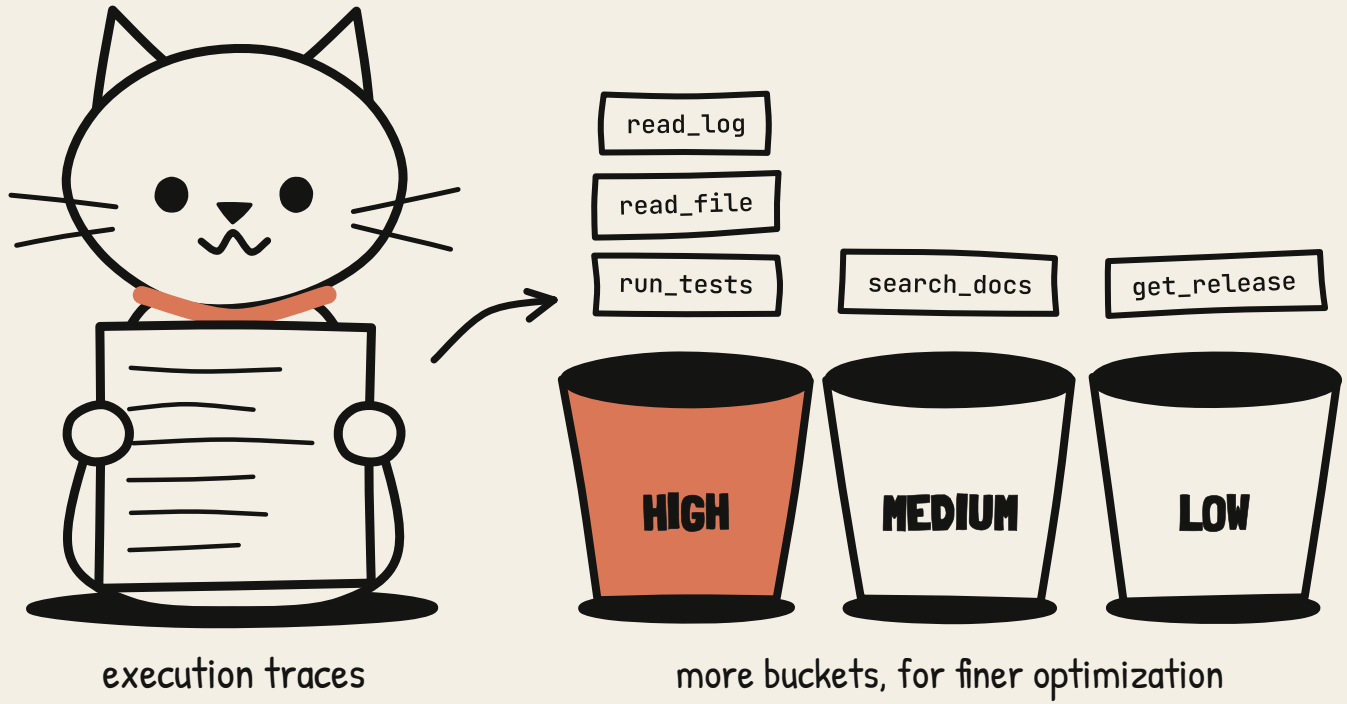
GROUPING BY USAGE

This starts with execution traces. We examine how often each tool is needed across requests and group tools by their loading frequency: keeping frequently used tools available upfront and deferring less-used tools.

- **High usage:** Load these tools at the start because most requests are likely to need them.
- **Medium usage:** Load a relevant bundle when the request reaches that kind of work.
- **Low usage:** Keep these tools discoverable and load their definitions only when needed.

For a build-heavy workload, `read_log`, `read_file` and `run_tests` might belong in the high-usage tier, while release-information tools remain deferred.

Global usage patterns can differ drastically from individual account or user patterns; optimizing for global usage could surge cost and latency in some accounts.



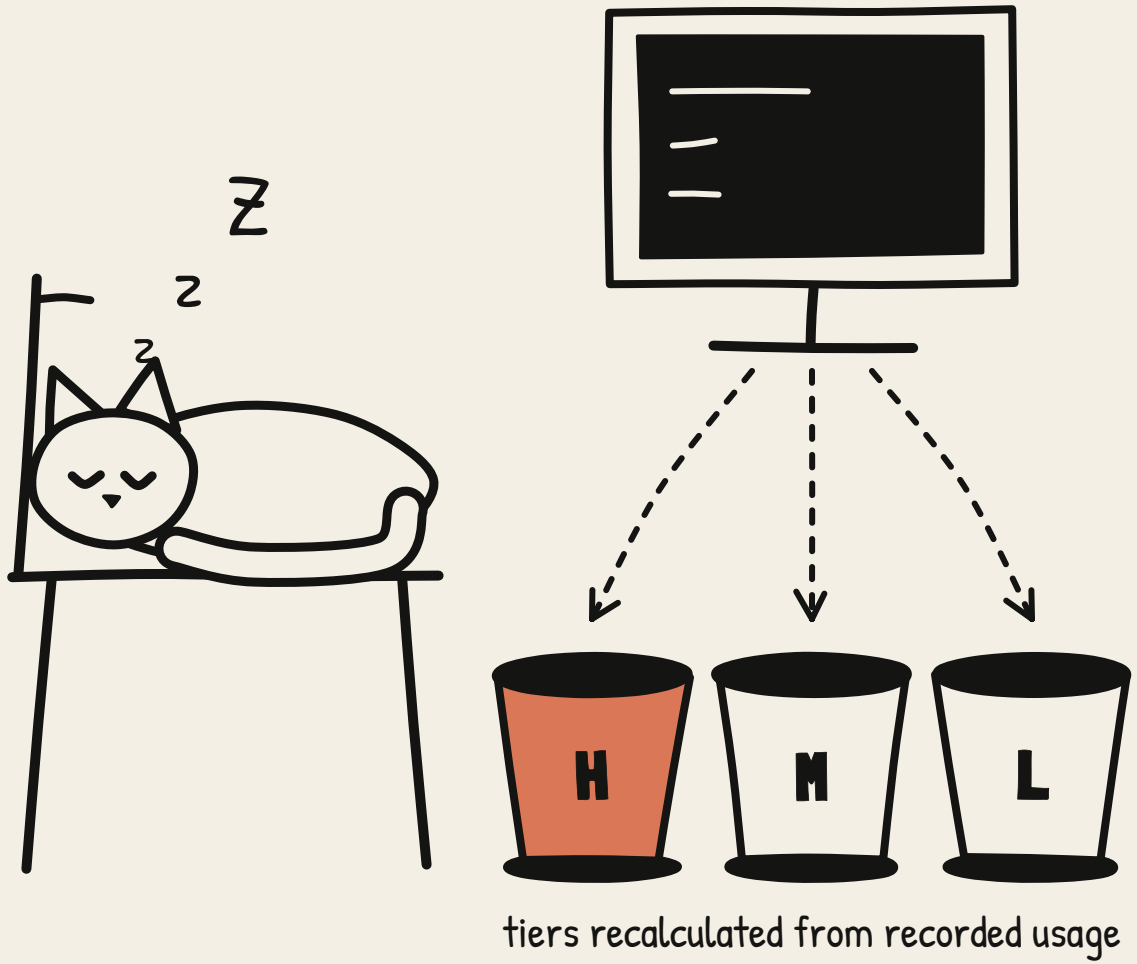
ADAPTIVE USAGE-BASED LOADING

We can automate the same process instead of treating the buckets as permanent configuration.

The application records tool usage against an account, user or namespace and periodically recalculates its high-, medium- and low-usage tiers. At the start of a new run, it retrieves the relevant profile and loads that profile's high-usage tools. The remaining tools stay available through discovery.

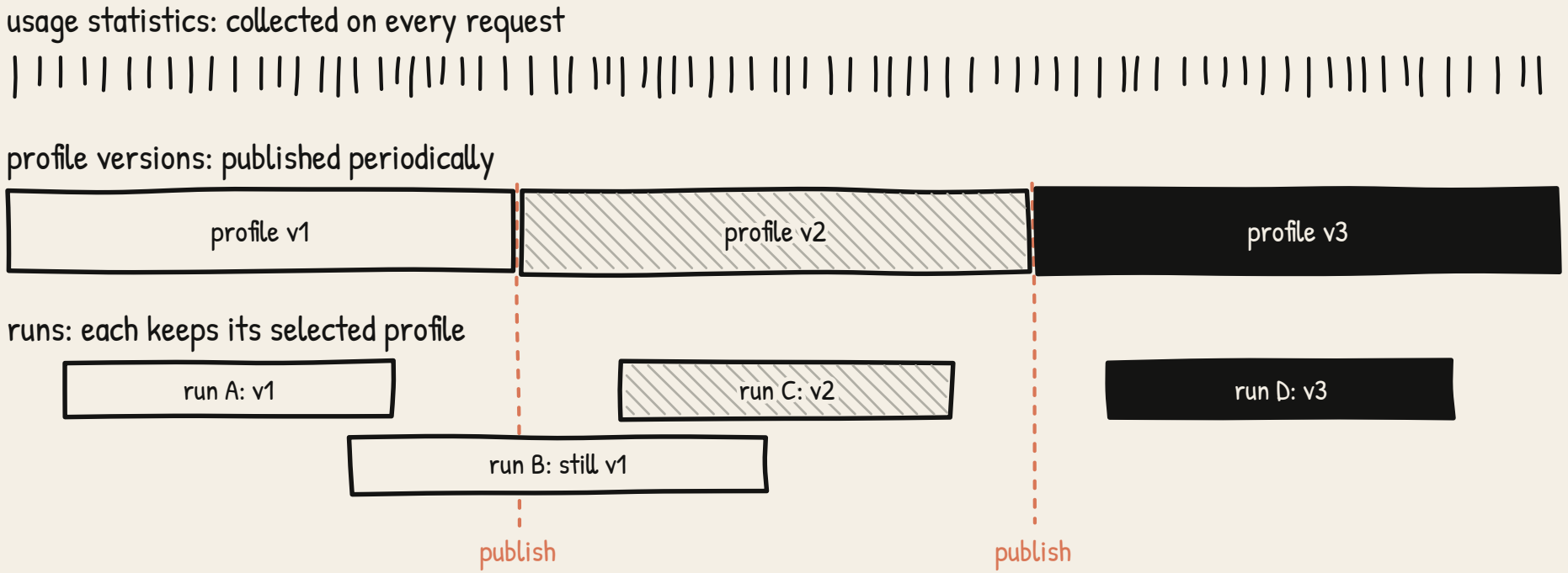
For example, `run_tests` could be loaded upfront for an account that frequently investigates builds, but remain deferred for an account that mostly searches project documentation. If an account's workload changes, its loading profile can change with it.

New accounts can start with a global default until there is enough activity to justify a separate profile. More granular profiles are worth testing when their usage patterns differ meaningfully.



ADAPTIVE USAGE-BASED LOADING

The distinction is between **continuously observing usage** and **continuously changing the prompt**. We can collect statistics on every request while publishing new profile versions periodically. Each active run keeps its selected profile and tool ordering stable; subsequent runs use the updated version.



The aim is to preload what that workload usually needs, leave uncommon capabilities discoverable, and adapt without constantly rebuilding the starting context. Whether this reduces cost must be checked against task success, discovery calls and actual cache reuse.

TRADE-OFFS BETWEEN DIFFERENT TOOL LOADING SCENARIOS

Category, usage and adaptive grouping decide **which tools belong together**. We also need to decide **how those groups enter the context**. The same grouping can have different costs depending on whether loading it changes the initial catalogue or preserves the existing prefix.

LOADING DESIGN	CONTEXT TRADE-OFF	CACHE TRADE-OFF
Keep the full catalogue	Unused definitions remain loaded	Stable initial prefix
Rewrite the catalogue every turn	Fewer definitions per turn	Earlier cached content may no longer match
Stable core plus lazy loading	Small starting set; loading adds a step	Can preserve the original prefix but invalidates on loading

Frontier models are now adding capabilities to improve tool use with prompt caching, with features like `cache-endpoint` and `defer-loading`.

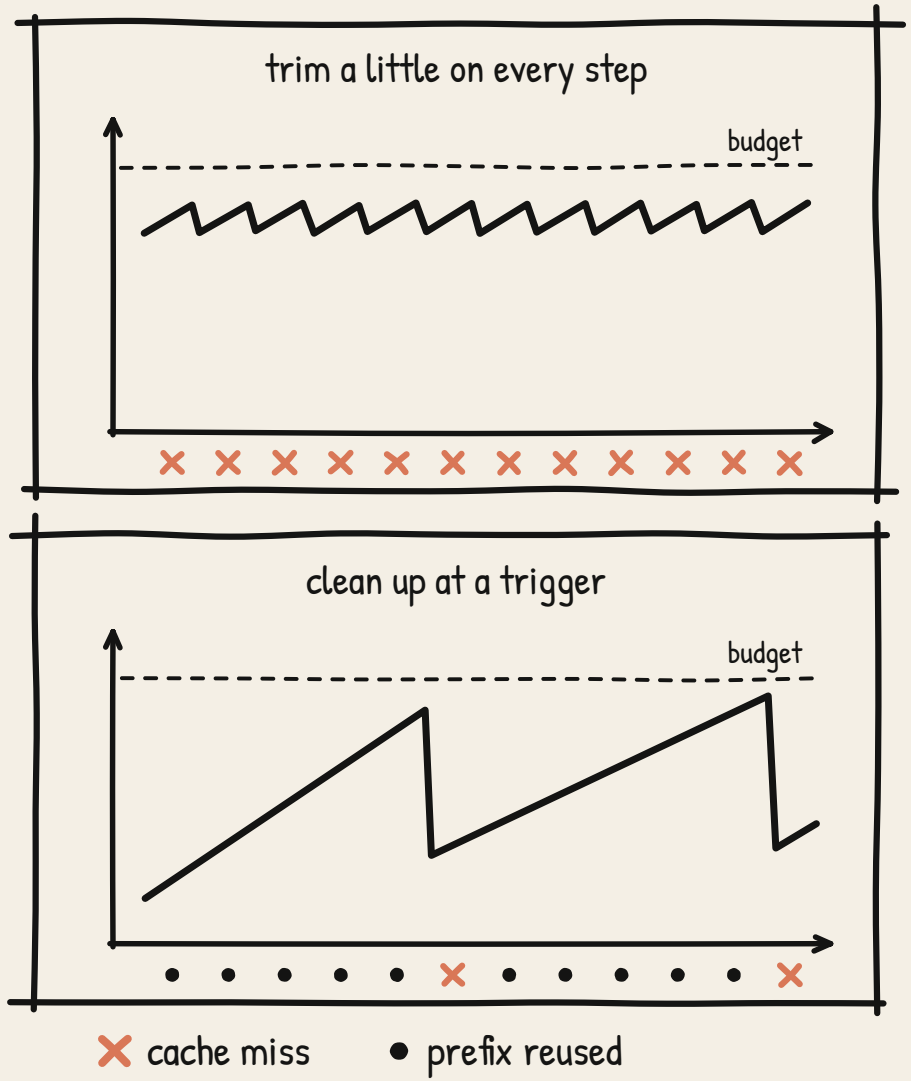
PROMPT CACHING AND CONTEXT MANAGEMENT

Changing tool definitions is not the only way to lose cache reuse. Summarizing or removing earlier messages also changes the input from that point onward. If we repeatedly rewrite the history, we repeatedly process revised input instead of reusing the existing prefix.

We should therefore consider **when to change the context and what to preserve**, rather than trying to make every request smaller.

- **Use a clear trigger:** Trigger a cleanup when approaching a chosen context budget or when accumulated, outdated information is interfering with the task. Leave headroom for new tool results and the model's response rather than waiting until the window is full.
- **Make fewer, meaningful changes.** Removing a small amount of history repeatedly can keep disrupting cache reuse for little benefit. When cleanup is needed, remove enough unnecessary material to create useful space, then let subsequent calls reuse the revised history.

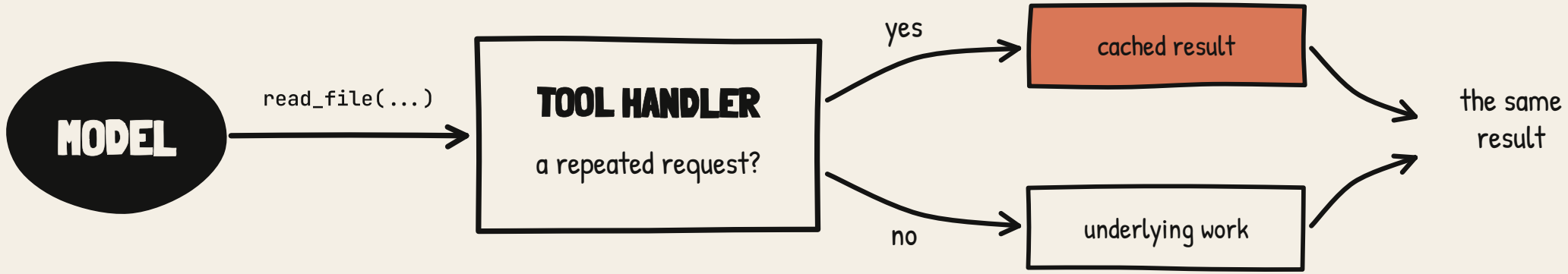
We should be selective about what enters context on every step, but we do not need to rewrite the accumulated history on every step.



CACHING TOOL RESULTS

Our agent can now retrieve details that are no longer in context. But retrieval is not free: it may request the same unchanged file or build log again. Can the application serve that result without repeating the underlying work?

We can add a **cache inside the tool handler**, but each tool needs a way to identify repeated requests and decide when an earlier result is no longer valid. For example, we could use these two designs:



FILE-READING TOOL

For `read_file("src/parser.py")`, cache the contents using the repository, file path and current file version. Repeated reads of that version reuse the result. An edit changes the version, so the next read fetches the updated contents.

DATABASE-QUERY TOOL

For a read-only query such as “count the failed tests for build-1842,” cache the result by query and parameters within the same database and permission scope. Invalidate it when the relevant test records change.

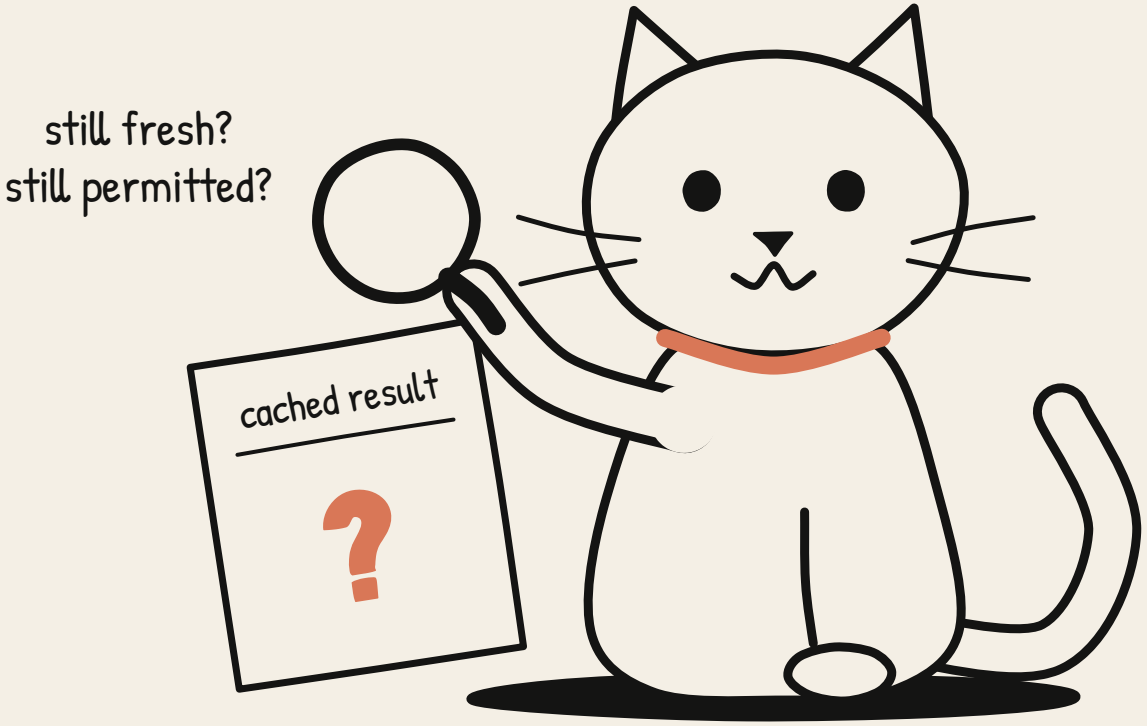
CACHING TOOL RESULTS

The model doesn't know whether we serve the result by actual execution or a cache, so the trouble of making sure our caches are valid falls on us.

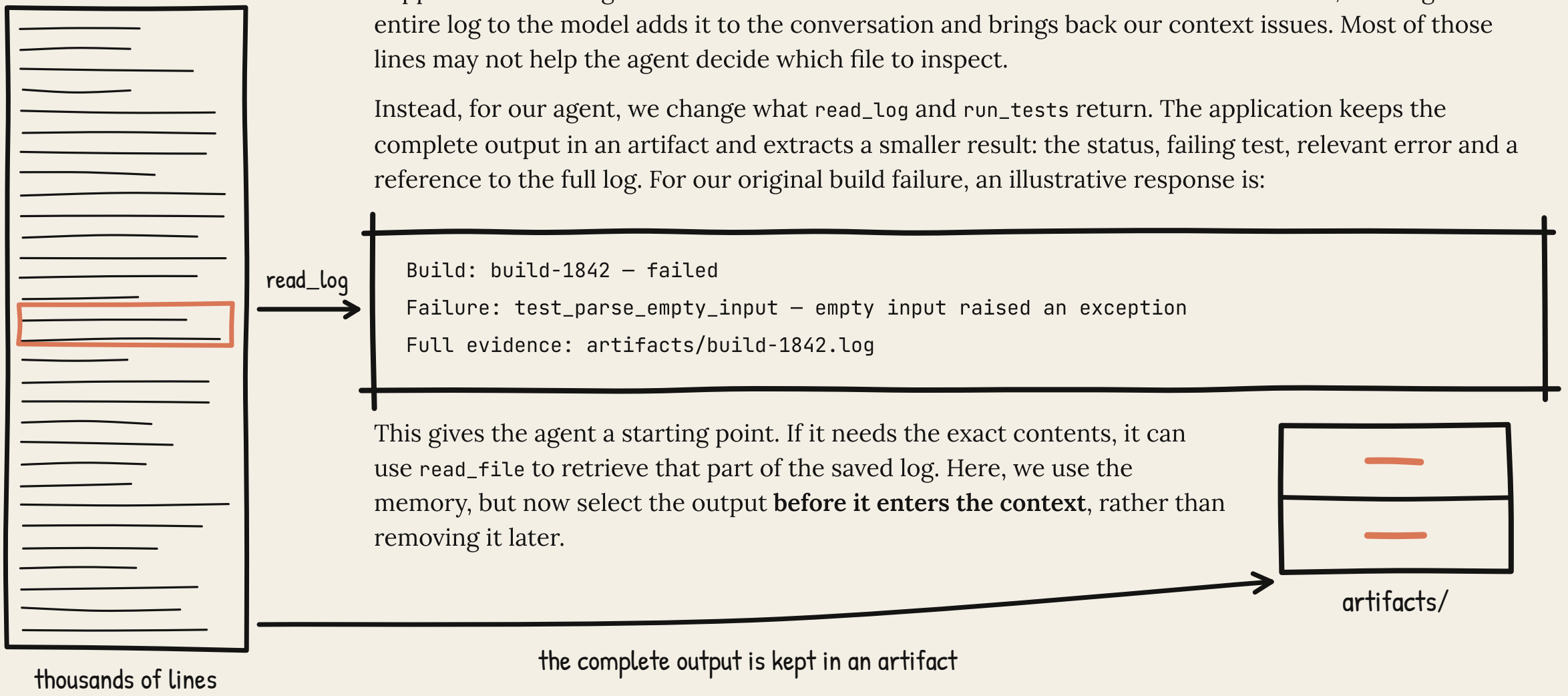
Instead of starting with outputs that require strict cache invalidation rules, start with results whose validity is easy to establish, such as an immutable completed log or a versioned file.

Read-only operations can still return stale data, so the application must check freshness and permissions before reusing a result.

This saves compute and latency, but returning a large cached result can still fill the context.



KEEP THE OUTPUTS CRISP

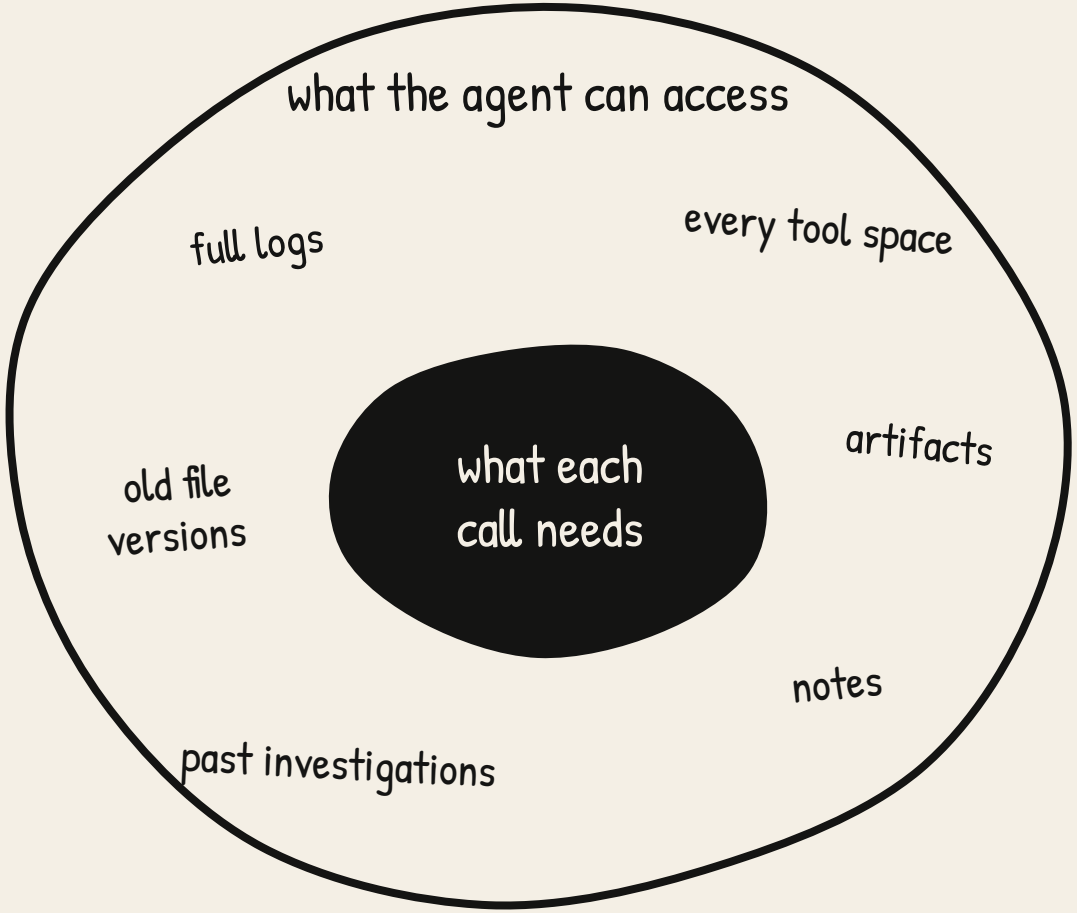


Suppose the build log contains thousands of lines. Even when we fetch it from a cache, sending the entire log to the model adds it to the conversation and brings back our context issues. Most of those lines may not help the agent decide which file to inspect.

Instead, for our agent, we change what `read_log` and `run_tests` return. The application keeps the complete output in an artifact and extracts a smaller result: the status, failing test, relevant error and a reference to the full log. For our original build failure, an illustrative response is:

This gives the agent a starting point. If it needs the exact contents, it can use `read_file` to retrieve that part of the saved log. Here, we use the memory, but now select the output **before it enters the context**, rather than removing it later.

KEEPING CONTEXT SMALL



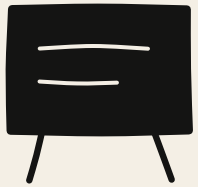
We began with a model choosing actions in a loop. Each action added to its history, and each new capability added tool definitions. As the agent took on more work, it carried more information into every call.

- The change was to separate **what the agent can access** from **what each call needs**. For our build investigation, full logs remain retrievable in memory, while the current request carries the relevant failure, current patch and pending checks.
- Tools follow the same principle. The assistant can support repository, documentation and release operations without loading every definition for a build investigation. We are reducing what stays active, not removing access to the rest.
- Caching complements this design by reusing unchanged input processing and still-valid tool results. It makes repeated work cheaper without changing what the next decision actually needs.

The system can retain extensive knowledge and capabilities without putting all of them into every model call. **Keeping context small means controlling the working set.**

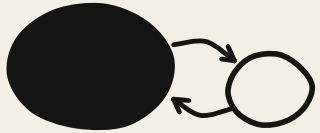
FURTHER EXPLORATION

So far, we have focused on one agent completing one task. We can extend the same coding agent to explore three related problems:



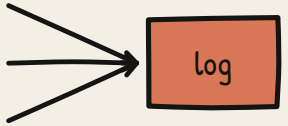
MEMORY ACROSS TASKS

Save verified repository knowledge, such as how to run its tests, so later investigations do not have to rediscover it. Retrieve only what the new task needs, and check whether project changes have made a stored fact outdated.



SEPARATE CONTEXTS

A documentation search may generate far more context than the main investigation needs. Give a helper agent that focused question and have it return the relevant findings and source references. Compare the context saved with the extra calls and coordination.



PERFORMANCE UNDER LOAD

Several agents may request the same log before any one of them has populated the cache. Test whether identical reads within the same permission scope can share one pending fetch, and limit how many tool operations run at once. Compare behaviour with empty and populated caches.

Choose one extension and compare it with the original agent on the same tasks. Check correctness first, then total time, cost and repeated work. Start with the original documentation and example code; use execution traces to understand why the result changed.

THANK YOU

questions?



Saksham Dhull
AI Agents, WisdomAI