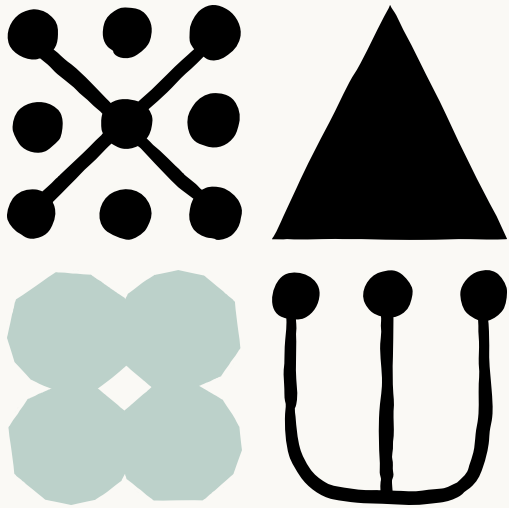


Engineering at Anthropic



Scaling Managed Agents: Decoupling the brain from the hands

Harnesses encode assumptions that go stale as models improve. Managed Agents—our hosted service for long-horizon agent work—is built around interfaces that stay stable as harnesses change.

Published Apr 08, 2026

Get started with Claude Managed Agents by following our [docs](#).

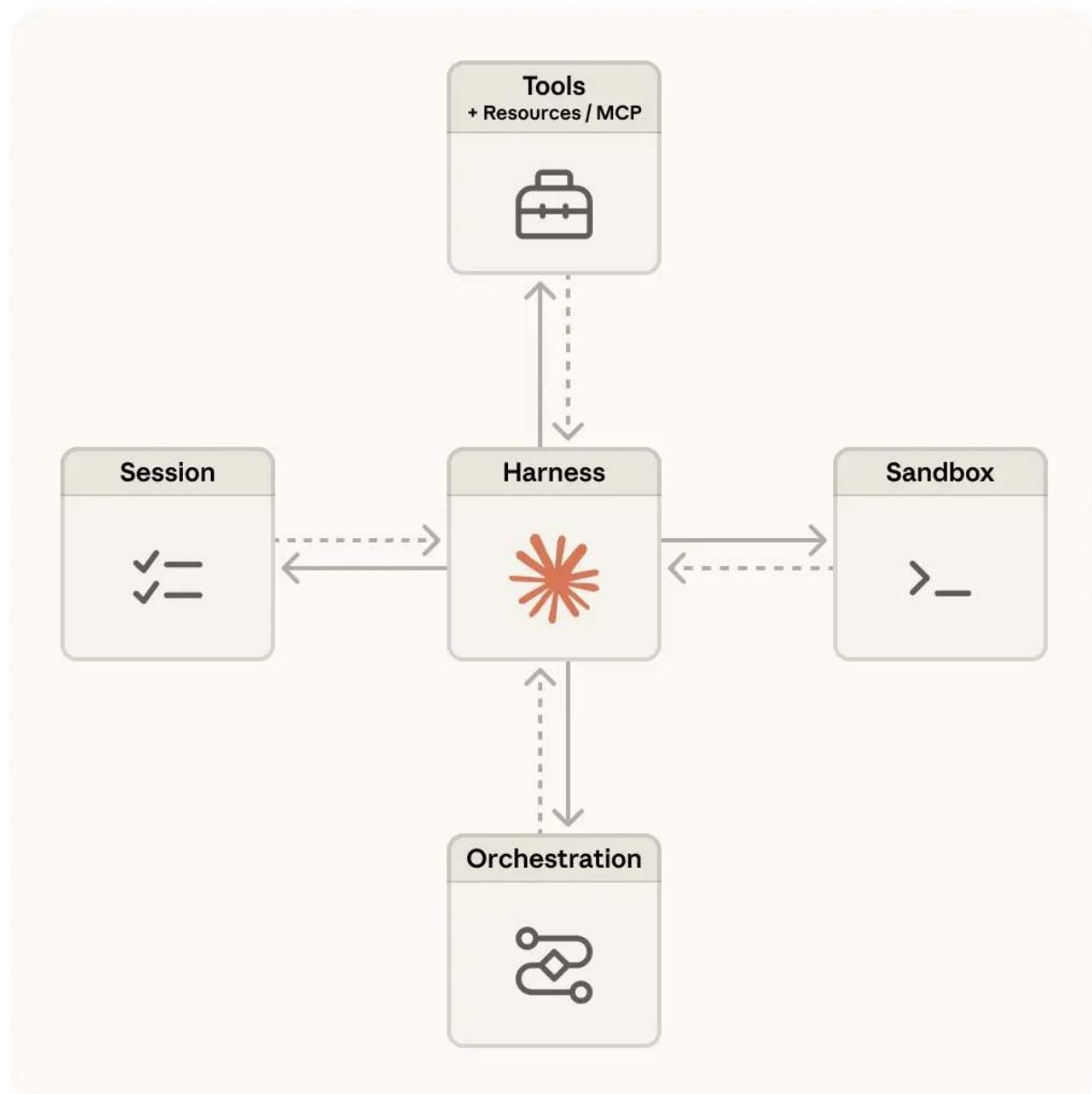
A running topic on the Engineering Blog is how to [build effective agents](#) and [design harnesses](#) for [long-running work](#). A common thread across this work is that harnesses encode assumptions about what Claude can't do on its own. However, those assumptions need to be frequently questioned because they can [go stale](#) as models improve.

As just one example, in prior work we found that Claude Sonnet 4.5 would wrap up tasks prematurely as it sensed its context limit approaching—a behavior sometimes called “context anxiety.” We addressed this by adding context resets to the harness. But when we used the same harness on Claude Opus 4.5, we found that the behavior was gone. The resets had become dead weight.

We expect harnesses to continue evolving. So we built Managed Agents: a hosted service in the Claude Platform that runs long-horizon agents on your behalf through a small set of interfaces meant to outlast any particular implementation—including the ones we run today.

Building Managed Agents meant solving an old problem in computing: how to design a system for “programs as yet unthought of.” Decades ago, operating systems solved this problem by virtualizing hardware into abstractions—*process*, *file*—general enough for programs that didn't exist yet. The abstractions outlasted the hardware. The `read()` command is agnostic as to whether it's accessing a disk pack from the 1970s or a modern SSD. The abstractions on top stayed stable while the implementations underneath changed freely.

Managed Agents follow the same pattern. We virtualized the components of an agent: a session (the append-only log of everything that happened), a harness (the loop that calls Claude and routes Claude's tool calls to the relevant infrastructure), and a sandbox (an execution environment where Claude can run code and edit files). This allows the implementation of each to be swapped without disturbing the others. We're opinionated about the shape of these interfaces, not about what runs behind them.



Don't adopt a pet

We started by placing all agent components into a single container, which meant the session, agent harness, and sandbox all shared an environment. There were benefits to this approach, including that file edits are direct syscalls, and there were no service boundaries to design.

But by coupling everything into one container, we ran into an old infrastructure problem: we'd adopted a *pet*. In the pets-vs-cattle analogy, a pet is a named, hand-tended individual you can't afford to lose, while cattle are interchangeable. In our

case, the server became that pet; if a container failed, the session was lost. If a container was unresponsive, we had to nurse it back to health.

Nursing containers meant debugging unresponsive stuck sessions. Our only window in was the WebSocket event stream, but that couldn't tell us *where* failures arose, which meant that a bug in the harness, a packet drop in the event stream, or a container going offline all presented the same. To figure out what went wrong, an engineer had to open a shell inside the container, but because that container often also held user data, that approach essentially meant we lacked the ability to debug.

A second issue was that the harness assumed that whatever Claude worked on lived in the container with it. When customers asked us to connect Claude to their virtual private cloud, they had to either peer their network with ours, or run our harness in their own environment. An assumption baked into the harness became a problem when we wanted to connect it to different infrastructure.

Decouple the brain from the hands

The solution we arrived at was to decouple what we thought of as the “brain” (Claude and its harness) from both the “hands” (sandboxes and tools that perform actions) and the “session” (the log of session events). Each became an interface that made few assumptions about the others, and each could fail or be replaced independently.

The harness leaves the container. Decoupling the brain from the hands meant the harness no longer lived inside the container. It called the container the way it called any other tool: `execute(name, input) → string`. The container became cattle. If the container died, the harness caught the failure as a tool-call error and passed it back to Claude. If Claude decided to retry, a new container could be reinitialized with a standard recipe: `provision({resources})`. We no longer had to nurse failed containers back to health.

Recovering from harness failure. The harness also became cattle. Because the session log sits outside the harness, nothing in the harness needs to survive a crash. When one fails, a new one can be rebooted with `wake(sessionId)`, use `getSession(id)` to get back the event log, and resume from the last event. During the agent loop, the harness writes to the session with `emitEvent(id, event)` in order to keep a durable record of events.

Component	Interface (pseudocode)	Satisfied by
Session	<pre>getSession(session_id) -> (Session, Event[]) getEvents(session_id) -> PendingEvent[] // not yet processed emitEvent(id, event)</pre>	Any append-only log that can be consumed in order from any event point and accepts idempotent appends — Postgres, SQLite, in-memory array, etc.
Orchestration	<pre>wake(session_id) -> void</pre>	Any scheduler that can call a function with an ID and retry on failure — a cron job, a queue consumer, a while-loop, etc.
Harness	<pre>yield Effect<T> -> EffectResult<T></pre>	Any loop that yields effects and appends progress to the Session.
Sandbox	<pre>provision({resources}) execute(name, input) -> String</pre>	Any executor that can be configured once and called many times as a tool— a local process, a remote container, etc.
Resources	<pre>[{source_ref, mount_path}]</pre>	Any durable store the container can fetch from by reference— Filestore, GCS, a git remote, S3.
Tools	<pre>{name, description, input_schema}</pre>	Any capability describable as a name and an input shape— MCP server, custom tool, etc.

The security boundary. In the coupled design, any untrusted code that Claude generated was run in the same container as credentials—so a prompt injection only had to convince Claude to read its own environment. Once an attacker has those tokens, they can spawn fresh, unrestricted sessions and delegate work to them. Narrow scoping is an obvious mitigation, but this encodes an assumption about what Claude can't do with a limited token—and Claude is getting increasingly smart. The structural fix was to make sure the tokens are never reachable from the sandbox where Claude's generated code runs.

We used two patterns to ensure this. Auth can be bundled with a resource or held in a vault outside the sandbox. For Git, we use each repository's access token to clone the repo during sandbox initialization and wire it into the local git remote. Git `push` and `pull` work from inside the sandbox without the agent ever handling the token itself. For custom tools, we support MCP and store OAuth tokens in a secure vault. Claude calls MCP tools via a dedicated proxy; this proxy takes in a token associated with the session. The proxy can then fetch the corresponding credentials from the vault and make the call to the external service. The harness is never made aware of any credentials.

The session is not Claude's context window

Long-horizon tasks often exceed the length of Claude's context window, and the standard ways to address this all involve irreversible decisions about what to keep. We've explored these techniques in [prior work](#) on context engineering. For example, compaction lets Claude save a summary of its context window and the memory tool lets Claude write context to files, enabling learning across sessions. This can be paired with context trimming, which selectively removes tokens such as old tool results or thinking blocks.

But irreversible decisions to selectively retain or discard context can lead to failures. It is difficult to know which tokens the future turns will need. If messages are transformed by a compaction step, the harness removes compacted messages from Claude's context window, and these are recoverable only if they are stored. [Prior work](#) has explored ways to address this by storing context as an object that lives *outside* the context window. For example, context can be an object in a REPL that the LLM programmatically accesses by writing code to filter or slice it.



In Managed Agents, the session provides this same benefit, serving as a context object that lives outside Claude's context window. But rather than be stored within the sandbox or REPL, context is durably stored in the session log. The interface,

`getEvents()`, allows the brain to interrogate context by selecting positional slices of the event stream. The interface can be used flexibly, allowing the brain to pick up from wherever it last stopped reading, rewinding a few events before a specific moment to see the lead up, or rereading context before a specific action.

Any fetched events can also be transformed in the harness before being passed to Claude's context window. These transformations can be whatever the harness encodes, including context organization to achieve a high prompt cache hit rate and context engineering. We separated the concerns of recoverable context storage in the session and arbitrary context management in the harness because we can't predict what specific context engineering will be required in future models. The interfaces push that context management into the harness, and only guarantee that the session is durable and available for interrogation.

Many brains, many hands

Many brains. Decoupling the brain from the hands solved one of our earliest customer complaints. When teams wanted Claude to work against resources in their own VPC, the only path was to peer their network with ours, because the container holding the harness assumed every resource sat next to it. Once the harness was no longer in the container, that assumption went away. The same change had a performance payoff. When we initially put the brain in a container, it meant that many brains required as many containers. For each brain, no inference could happen until that container was provisioned; every session paid the full container setup cost up front. Every session, even ones that would never touch the sandbox, had to clone the repo, boot the process, fetch pending events from our servers.

That dead time is expressed in time-to-first-token (TTFT), which measures how long a session waits between accepting work and producing its first response token. TTFT is the latency the user most acutely *feels*.

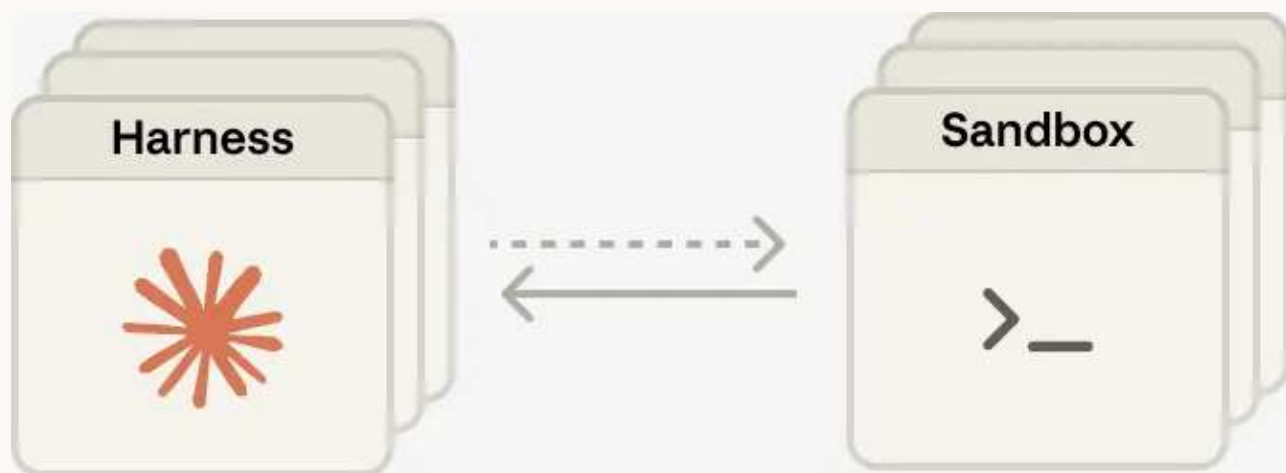
Decoupling the brain from the hands means that containers are provisioned by the brain via a tool call `(execute(name, input) → string)` only if they are needed. So a session that didn't need a container right away didn't wait for one. Inference could start as soon as the orchestration layer pulled pending events from the session

log. Using this architecture, our p50 TTFT dropped roughly 60% and p95 dropped over 90%. Scaling to many brains just meant starting many stateless harnesses, and connecting them to hands only if needed.

Many hands. We also wanted the ability to connect each brain to many hands. In practice, this means Claude must reason about many execution environments and decide where to send work—a harder cognitive task than operating in a single shell. We started with the brain in a single container because earlier models weren't capable of this. As intelligence scaled, the single container became the limitation instead: when that container failed, we lost state for every hand that the brain was reaching into.

Decoupling the brain from the hands makes each hand a tool,

`execute(name, input) → string`: a name and input go in, and a string is returned. That interface supports any custom tool, any MCP server, and our own tools. The harness doesn't know whether the sandbox is a container, a phone, or a Pokémon emulator. And because no hand is coupled to any brain, brains can pass hands to one another.



Conclusion

The challenge we faced is an old one: how to design a system for “programs as yet unthought of.” Operating systems have lasted decades by virtualizing the hardware into abstractions general enough for programs that didn't exist yet. With Managed Agents, we aimed to design a system that accommodates future harnesses, sandboxes, or other components around Claude.

Managed Agents is a meta-harness in the same spirit, unopinionated about the *specific* harness that Claude will need in the future. Rather, it is a system with general interfaces that allow many different harnesses. For example, Claude Code is an excellent harness that we use widely across tasks. We’ve also shown that task-specific agent harnesses excel in narrow domains. Managed Agents can accommodate any of these, matching Claude’s intelligence over time.

Meta-harness design means being opinionated about the interfaces around Claude: we expect that Claude will need the ability to manipulate state (the session) and perform computation (the sandbox). We also expect that Claude will require the ability to scale to many brains and many hands. We designed the interfaces so that these can be run reliably and securely over long time horizons. But we make no assumptions about the number or location of brains or hands that Claude will need.

Acknowledgements

Written by Lance Martin, Gabe Cemaj, and Michael Cohen. Thanks to Nodir Turakulov and Jeremy Fox for helpful conversations on these topics. Special thanks to the Agents API team and Jake Eaton for their contributions.

Get the developer newsletter

Product updates, how-tos, community spotlights, and more.
Delivered monthly to your inbox.



Please provide your email address if you'd like to receive our monthly developer newsletter. You can unsubscribe at any time.