21

Managed Agents のスケーリング: 頭と手を切り離す

— Scaling Managed Agents: Decoupling the brain from the hands —

原題	Scaling Managed Agents: Decoupling the brain from the hands
著者	Anthropic Engineering Team
原文	https://www.anthropic.com/engineering/managed-agents
翻訳	Claude（機械翻訳／Anthropic）
編集	2026-04-09

Managed Agents のスケーリング: 頭と手を切り離す

Claude Managed Agents を始めるには ドキュメント に従ってください。

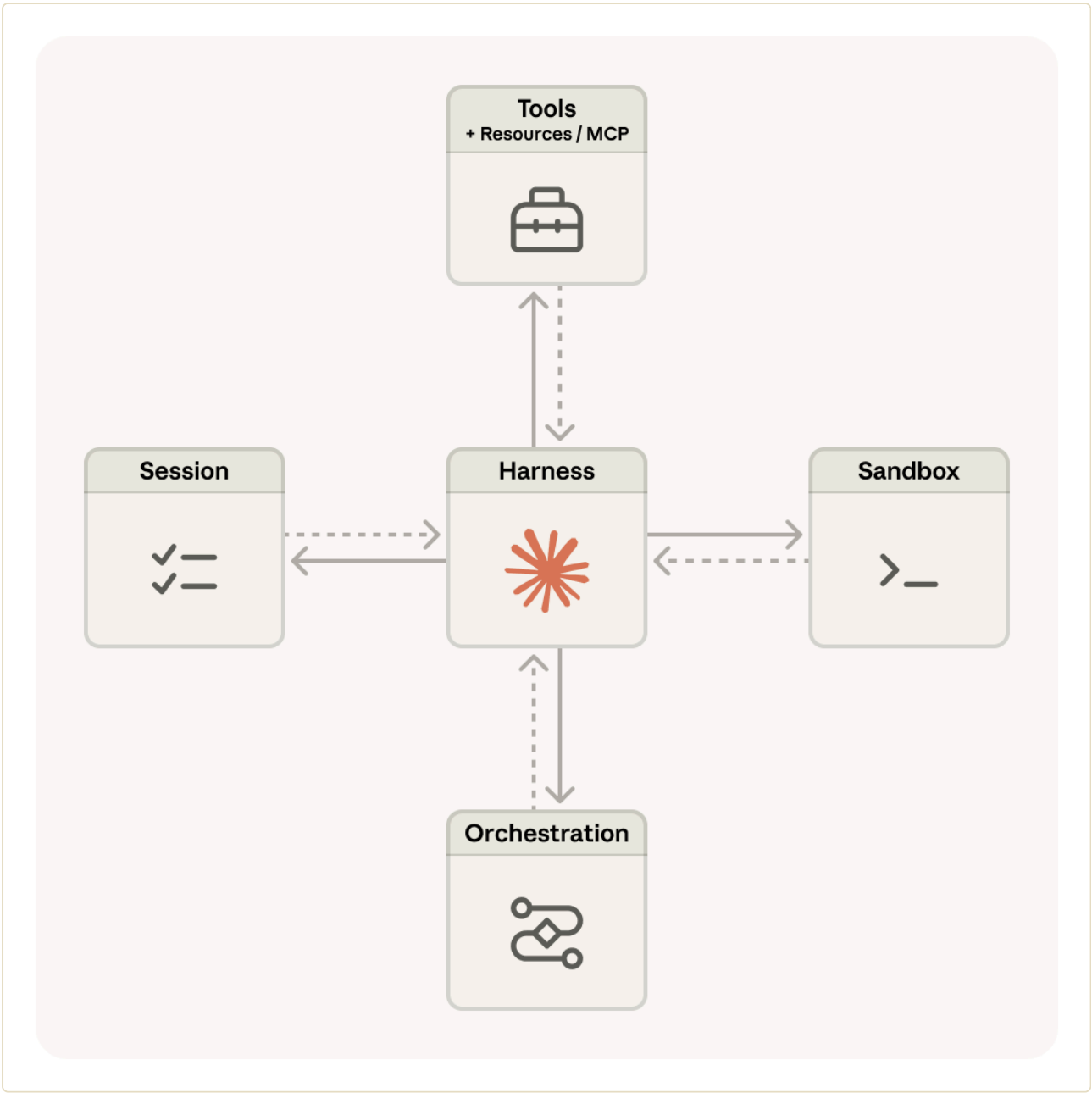
Engineering Blog で続いているトピックは、効果的なエージェントの作り方 と 長時間作業 のための ハーネス設計 です。この作業を貫く共通の糸は、ハーネスが Claude が単独でできないことについての仮定をエンコードしているということです。しかしそれらの仮定は、モデルが改善するにつれ古くなり得る ので、頻繁に疑問視される必要があります。

一例として、以前の作業で私たちは発見しました——Claude Sonnet 4.5 がコンテキスト上限が近いと感じるとタスクを早めに切り上げる振る舞い、いわゆる「コンテキスト不安（context anxiety）」です。これにはハーネスにコンテキストリセットを追加して対処しました。しかし同じハーネスを Claude Opus 4.5 で使ったら、その振る舞いが消えていることが分かりました。リセットは無用の重荷になっていたのです。

ハーネスは進化し続けると予想しています。そこで Managed Agents を作りました——Claude Platform 上のホステッドサービスで、特定の実装（今私たちが動かしているものを含む）を超えて長持ちする小さなインターフェース群を通じて、長時間エージェントをあなたの代わりに動かします。

Managed Agents を作ることは、コンピューティングの古い問題を解くことを意味しました——「まだ考えられていないプログラム」のためのシステムをどう設計するか。数十年前、オペレーティングシステムはこの問題を、ハードウェアを *プロセス*、*ファイル* のような、まだ存在しないプログラムのために十分汎用的な抽象に仮想化することで解きました。抽象はハードウェアより長生きしました。`read()` コマンドは、1970 年代のディスクパックにアクセスしているのか現代の SSD にアクセスしているのか気にしません。上の抽象は安定したまま、下の実装は自由に変わりました。

Managed Agents は同じパターンに従います。私たちはエージェントのコンポーネントを仮想化しました——セッション（起きたことすべての append-only ログ）、ハーネス（Claude を呼び、Claude のツール呼び出しを関連インフラに振り分けるループ）、サンドボックス（Claude がコードを実行しファイルを編集できる実行環境）。これにより、各々の実装は他を乱さずに交換可能になります。これらのインターフェースの形については意見を持ちますが、その背後で何が走るかについては持ちません。



ペットを飼わないこと

まず、すべてのエージェントコンポーネントを 1 つのコンテナに入れることから始めました。セッション、エージェントハーネス、サンドボックスがすべて環境を共有することを意味しました。このアプローチには利点がありました——ファイル編集は直接 syscall であり、設計すべきサービス境界がありませんでした。

しかしすべてを 1 つのコンテナに結合することで、古いインフラ問題にぶつかりました——私たちは ペット を飼っていたのです。pets-vs-cattle のアナロジーでは、ペットは名前を持ち手厚く世話される個体で失うわけにはいかず、牛は交換可能です。私たちの場合、サーバーがそのペットになりました。コンテナが失敗したらセッションが失われました。コンテナが応答しなくなれば、健康に戻すために看病する必要がありました。

コンテナの看病は、応答しなくなった詰まったセッションのデバッグを意味しました。私たちの唯一の窓はWebSocket イベントストリームでしたが、それは失敗が *どこで* 起きたかを教えてくれませんでした。つまりハーネスのバグ、イベントストリームのパケット落ち、コンテナのオフライン化はすべて同じに見えました。何が悪かったかを理解するには、エンジニアがコンテナ内でシェルを開く必要がありましたが、そのコンテナがしばしばユーザーデータも保持していたため、このアプローチは本質的にデバッグする能力を欠いていることを意味しました。

2 つ目の問題は、ハーネスが Claude が取り組むすべてのものがコンテナに存在すると仮定していたことです。顧客が Claude を自分たちの仮想プライベートクラウドに接続するよう頼んだとき、彼らはネットワークを私たちのものとピアリングするか、自分の環境でハーネスを走らせるかしかありませんでした。ハーネスに焼き付けられた仮定が、異なるインフラに接続したいときに問題になりました。

頭と手を切り離す

辿り着いた解は、「頭（brain）」（Claude とそのハーネス）を「手（hands）」（アクションを実行するサンドボックスとツール）と「セッション」（セッションイベントのログ）の両方から切り離すことでした。それぞれがインターフェースとなり、互いについてほとんど仮定をせず、独立して失敗したり置き換えられたりできます。

ハーネスはコンテナを去る。頭と手を切り離すことは、ハーネスがもはやコンテナ内で生きないことを意味しました。ハーネスは他のツールを呼ぶのと同じようにコンテナを呼びます——`execute(name, input) → string`。コンテナは牛になりました。コンテナが死ねば、ハーネスは失敗をツール呼び出しエラーとして捕まえ、Claude に返します。Claude が再試行を決めれば、新しいコンテナを標準レシピ `provision({resources})` で再初期化できます。失敗したコンテナを健康に戻すために看病する必要がもうなくなったのです。

ハーネス失敗からの回復。ハーネス自体も牛になりました。セッションログがハーネスの外に位置するので、ハーネス内の何もクラッシュを生き延びる必要がありません。1 つが失敗すると、新しいものを `wake(sessionId)` で再起動し、`getSession(id)` でイベントログを取り戻し、最後のイベントから再開できます。エージェントループ中、ハーネスは `emitEvent(id, event)` でセッションに書き込み、イベントの持続的記録を保ちます。

Component	Interface (pseudocode)	Satisfied by

Session	`getSession(session_id) -> (Session, Event[])`
	
	`getEvents(session_id) -> PendingEvent[] // not yet processed`
	
	`emitEvent(id, event)`
		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	`wake(session_id) -> void`	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	`yield Effect<T> -> EffectResult<T>`	Any loop that yields effects and appends progress to the Session.
Sandbox	`provision({resources})` `execute(name, input) -> String`	Any executor that can be configured once and called many times as a tool— a local process, a remote container, etc.
Resources	`[{source_ref, mount_path}]`	Any durable store the container can fetch from by reference— Filestore, GCS, a git remote, S3.
Tools	`{name, description, input_schema}`	Any capability describable as a name and an input shape— MCP server, custom tool, etc.

セキュリティ境界。結合された設計では、Claude が生成した信頼できないコードは資格情報と同じコンテナで走っていました——そのためプロンプトインジェクションは Claude に自分の環境を読むよう説得するだけで済みました。攻撃者がそれらのトークンを手にすれば、新鮮で制限のないセッションを spawn し、作業をそれに委譲できます。狭いスコープ付けは明白な緩和ですが、これは Claude が限定されたトークンでできないことについての仮定をエンコードしており——Claude はますます賢くなっています。構造的な修正は、トークンが Claude の生成コードが走るサンドボックスから決して到達できないようにすることでした。

これを保証するために 2 つのパターンを使いました。認証はリソースにバンドルするか、サンドボックス外の vault に保持できます。Git の場合、各リポジトリのアクセストークンを使ってサンドボックス初期化中にリポをクローンし、ローカル git remote に配線します。Git `push` と `pull` はサンドボックス内から動き、エージェントはトークン自体を一切扱いません。カスタムツールについては MCP をサポートし、OAuth トークンをセキュ

ア vault に保存します。Claude は MCP ツールを専用プロキシ経由で呼びます。このプロキシはセッションに関連付けられたトークンを受け取ります。プロキシはその後 vault から対応する資格情報を取り、外部サービスへ呼び出しを行います。ハーネスは資格情報を一切意識しません。

セッションは Claude のコンテキストウィンドウではない

長時間タスクはしばしば Claude のコンテキストウィンドウの長さを超え、これに対処する標準的な方法はすべて、何を保つかの不可逆な決定を伴います。こうした手法は以前のコンテキストエンジニアリング の作業で探索してきました。たとえば圧縮は Claude にコンテキストウィンドウのサマリーを保存させ、メモリツールは Claude にコンテキストをファイルに書かせて、セッション間の学習を可能にします。これはコンテキストトリミング——古いツール結果や思考ブロックのようなトークンを選択的に取り除く——と組み合わせられます。

しかし文脈を選択的に保持・破棄する不可逆な決定は、失敗につながります。将来のターンがどのトークンを必要とするかを知るのは難しいです。メッセージが圧縮ステップで変換されると、ハーネスは圧縮されたメッセージを Claude のコンテキストウィンドウから取り除き、これらは保存されていれば復元可能です。以前の研究はこれに対処する方法を探っています——コンテキストウィンドウの *外* に存在するオブジェクトとしてコンテキストを保存するのです。たとえば、コンテキストは LLM がそれをフィルタしたりスライスしたりするコードを書いてプログラム的にアクセスする REPL 内のオブジェクトであり得ます。

Managed Agents では、セッションがこの同じ利点を提供し、Claude のコンテキストウィンドウの外に存在するコンテキストオブジェクトとして機能します。しかしサンドボックスや REPL 内に保存されるのではなく、コンテキストはセッションログに持続的に保存されます。インターフェース `getEvents()` は、イベントストリームの位置スライスを選択することで頭に文脈を詢問させます。インターフェースは柔軟に使え、頭が最後に読むのを止めた場所から再開したり、特定の瞬間の前の数イベントを巻き戻して前後を見たり、特定のアクションの前のコンテキストを再読したりできます。

取得したイベントは、Claude のコンテキストウィンドウに渡される前にハーネス内で変換することもできます。これらの変換はハーネスがエンコードするもの何でも可能で、プロンプトキャッシュヒット率を高めるコンテキスト整理やコンテキストエンジニアリングを含みます。セッションでの復元可能なコンテキストストレージと、

ハーネスでの任意のコンテキスト管理という 2 つの関心事を分離したのは、将来のモデルでどのような具体的なコンテキストエンジニアリングが必要になるかを予測できないからです。インターフェースは、コンテキスト管理をハーネスに押し込み、セッションが持続的で詢問可能であることだけを保証します。

多くの頭、多くの手

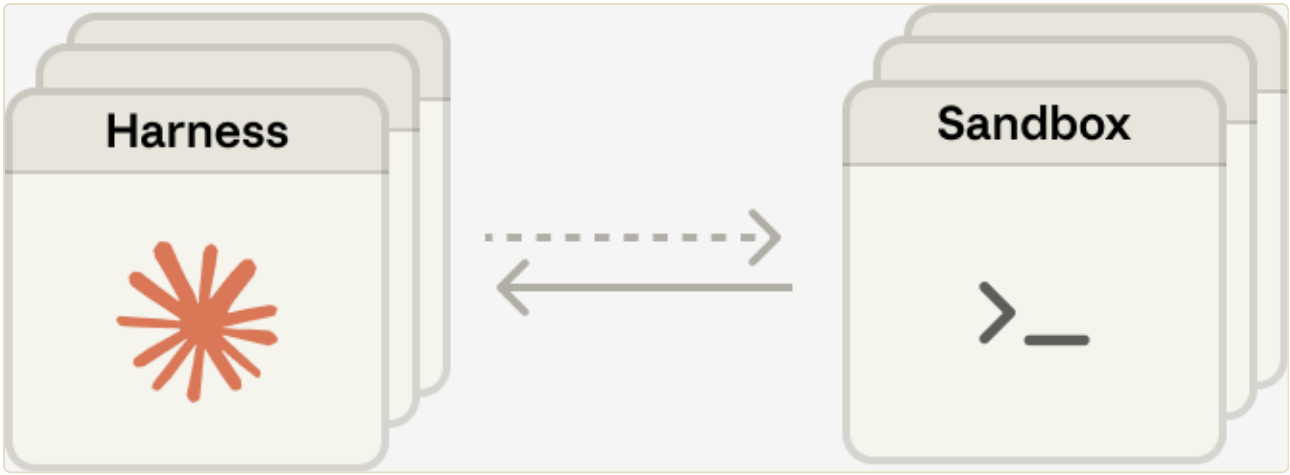
多くの頭。頭と手を切り離すことは、初期の顧客からの苦情の 1 つを解決しました。チームが Claude を自分の VPC 内のリソースに対して動かしたいと思ったとき、唯一の道はネットワークを私たちのものとピアリングすることでした——ハーネスを保持するコンテナがすべてのリソースがその横に位置すると仮定していたからです。ハーネスがもはやコンテナ内にないと、その仮定はなくなりました。同じ変更は性能的な見返りもありました。最初に頭をコンテナに入れたとき、多くの頭には多くのコンテナが必要でした。各頭について、コンテナがプロビジョンされるまで推論は起きませんでした。すべてのセッションが前もってコンテナセットアップの全コストを支払いました。すべてのセッションは、決してサンドボックスに触れないものでさえ、リポをクローンし、プロセスを起動し、私たちのサーバーから保留イベントを取得しなければなりませんでした。

そのデッドタイムは time-to-first-token（TTFT）——セッションが作業を受け入れてから最初の応答トークンを生成するまでの待ち時間——で表現されます。TTFT はユーザーが最も鋭く *感じる* レイテンシです。

頭と手を切り離すことは、必要な場合のみハーネスがツール呼び出し（`execute(name, input) → string`）経由でコンテナをプロビジョンすることを意味します。コンテナを直ぐ必要としないセッションは、それを待ちません。オーケストレーション層がセッションログから保留イベントを引き出すとすぐに推論を開始できます。このアーキテクチャで、p50 TTFT は約 60% 下がり、p95 は 90% 超下がりました。多くの頭にスケールすることは単に多くのステートレスハーネスを開始し、必要な場合のみ手に繋ぐことを意味するだけになりました。

多くの手。各頭を多くの手に繋ぐ能力も欲しかったです。実践的には、これは Claude が多くの実行環境について推論し、作業をどこに送るかを決めなければならないことを意味します——単一シェルで動くよりも難しい認知タスクです。以前のモデルがこれを扱えなかったので、頭を単一のコンテナに入れることから始めました。知性がスケールするにつれ、単一コンテナは代わりに制限になりました——そのコンテナが失敗すると、頭が及んでいたすべての手のステートを失ったのです。

頭と手を切り離すことで、各手はツール `execute(name, input) → string` になります——名前と入力が入り、文字列が返ります。そのインターフェースは任意のカスタムツール、任意の MCP サーバー、私たち自身のツールをサポートします。ハーネスはサンドボックスがコンテナか、電話か、Pokémon エミュレータかを知りません。そして、手が頭に結合されていないので、頭は手を互いに渡せます。



結論

私たちが直面した課題は古いものです——「まだ考えられていないプログラム」のためのシステムをどう設計するか。オペレーティングシステムは、ハードウェアをまだ存在しないプログラムのために十分汎用的な抽象に仮想化することで数十年持ちこたえてきました。Managed Agents では、将来のハーネス、サンドボックス、または Claude 周辺の他のコンポーネントに対応するシステムを設計することを目指しました。

Managed Agents は同じ精神のメタハーネスで、将来 Claude が必要とする *具体的な* ハーネスについては意見を持ちません。むしろ、多くの異なるハーネスを可能にする汎用インターフェースを持つシステムです。たとえば Claude Code はタスク横断で広く使う優れたハーネスです。タスク特化型エージェントハーネスが狭いドメインで優れることも示してきました。Managed Agents はこれらのどれにも対応でき、Claude の知性と共に時を経ます。

メタハーネス設計は、Claude 周辺のインターフェースについて意見を持つことを意味します——Claude はステート（セッション）を操作し計算を行う（サンドボックス）能力を必要とすると予想します。Claude は多くの頭と多くの手にスケールする能力も必要とすると予想します。これらが長い時間軸にわたって信頼でき安全に走るようインターフェースを設計しました。しかし、Claude が必要とする頭や手の数や場所について仮定はしません。

謝辞

執筆は Lance Martin、Gabe Cemaj、Michael Cohen。貢献への特別な感謝を Agents API チームと Jake Eaton に。