08

エージェントのための効果的なツールを、エージェントと一緒に書く

— Writing effective tools for agents — with agents —

公開日	2025-09-11
原題	Writing effective tools for agents — with agents
著者	Anthropic Engineering Team
原文	https://www.anthropic.com/engineering/writing-tools-for-agents
翻訳	Claude（機械翻訳／Anthropic）
編集	2026-04-09

エージェントのための効果的なツールを、エージェントと一緒に書く

Model Context Protocol（MCP） は、実世界のタスクを解くために LLM エージェントに数百ものツールを与えることができます。しかし、そのツールを最大限に有効にするにはどうすればよいのでしょうか？

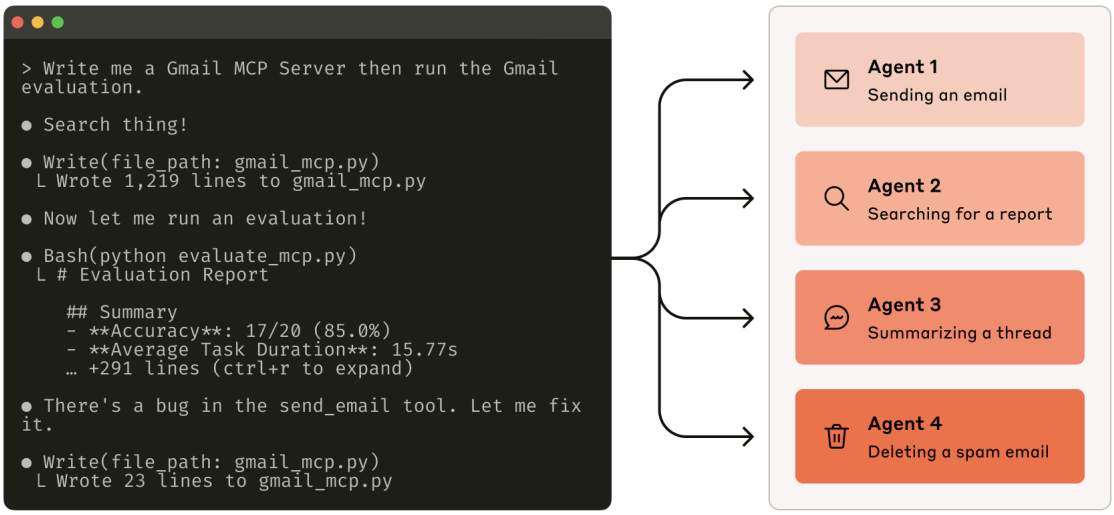
本稿では、さまざまなエージェント型 AI システムで性能を向上させるための、私たちの最も効果的な手法を紹介します[1]。

まず次のことをどう行うかを扱います。

- ツールのプロトタイプを作ってテストする
- ツールの包括的な evaluation をエージェントと一緒に作って走らせる
- Claude Code のようなエージェントと協力して、ツールの性能を自動で高める

そして、高品質なツールを書くために私たちが道中で特定した主要な原則で締めくくります。

- 実装すべき（あるいはすべきでない）ツールを正しく選ぶ
- 機能の境界を明確にするためにツールに名前空間を付ける
- 意味のある文脈をツールからエージェントに返す
- トークン効率のためにツール応答を最適化する
- ツールの説明と仕様をプロンプトエンジニアリングする



eval を構築することで、ツールの性能を体系的に測定できます。その eval に対して Claude Code を使ってツールを自動最適化できます。

ツールとは何か

コンピューティングにおいて、**決定的**（**deterministic**）システムとは、同じ入力に対して毎回同じ出力を生むものを指します。一方 **非決定的**（**non-deterministic**）システム——エージェントのような——は、同じ開始条件でも多様な応答を生成します。

伝統的なソフトウェアを書くとき、私たちは決定的システム同士の契約を結んでいます。たとえば `getWeather("NYC")` という関数呼び出しは、呼ぶたびに必ず同じやり方でニューヨークの天気を取ってきます。

ツールは、決定的システムと非決定的エージェントの間の契約を反映した新しい種類のソフトウェア です。ユーザーが「今日は傘を持っていくべき？」と尋ねたとき、エージェントは天気ツールを呼ぶかもしれないし、一般知識から答えるかもしれないし、あるいは位置について確認する質問をまずしてくるかもしれません。ときには幻覚を起こしたり、ツールの使い方を把握し損ねたりすることさえあります。

これは、エージェント向けのソフトウェアを書くときには根本的に発想を変えるべきだということを意味します——他の開発者やシステム向けに関数や API を書くようにツールや MCP サーバー を書くのではなく、エージェントのために設計する必要があるのです。

私たちのゴールは、エージェントがさまざまな成功戦略を追えるツールを使い、広範なタスクを効果的に解ける「表面積」を広げることです。幸いなことに、私たちの経験ではエージェントにとって最も「人間工学的」なツールは、人間にとっても驚くほど直感的に把握できるものになります。

ツールの書き方

このセクションでは、エージェントと協力してツールを書き、改善する方法を記します。まずツールのプロトタイプを素早く作ってローカルでテストします。次に、包括的な eval を走らせてその後の変更を測定します。エージェントと並んで働きながら、ツールの評価と改善を繰り返し、実世界のタスクで強い性能をエージェントが出すようになるまで進めます。

プロトタイプを作る

エージェントにとってどのツールが使いやすく、どれが使いにくいかを、実際に触らずに予測するのは難しいです。まずツールのプロトタイプを素早く立ち上げましょう。Claude Code でツールを書くなら（ワンショットでも）、依存するソフトウェアライブラリ、API、SDK（可能なら MCP SDK を含む）のドキュメントを Claude に渡すと助かります。LLM にフレンドリーなドキュメントは、公式サイトのフラットな `llms.txt` ファイルによくあります（私たちの API のもの）。

ツールをローカル MCP サーバー や Desktop Extension（DXT）としてラップすれば、Claude Code や Claude Desktop アプリから接続してテストできます。

ローカル MCP サーバーを Claude Code に接続するには `claude mcp add <name> <command> [args...]` を実行します。

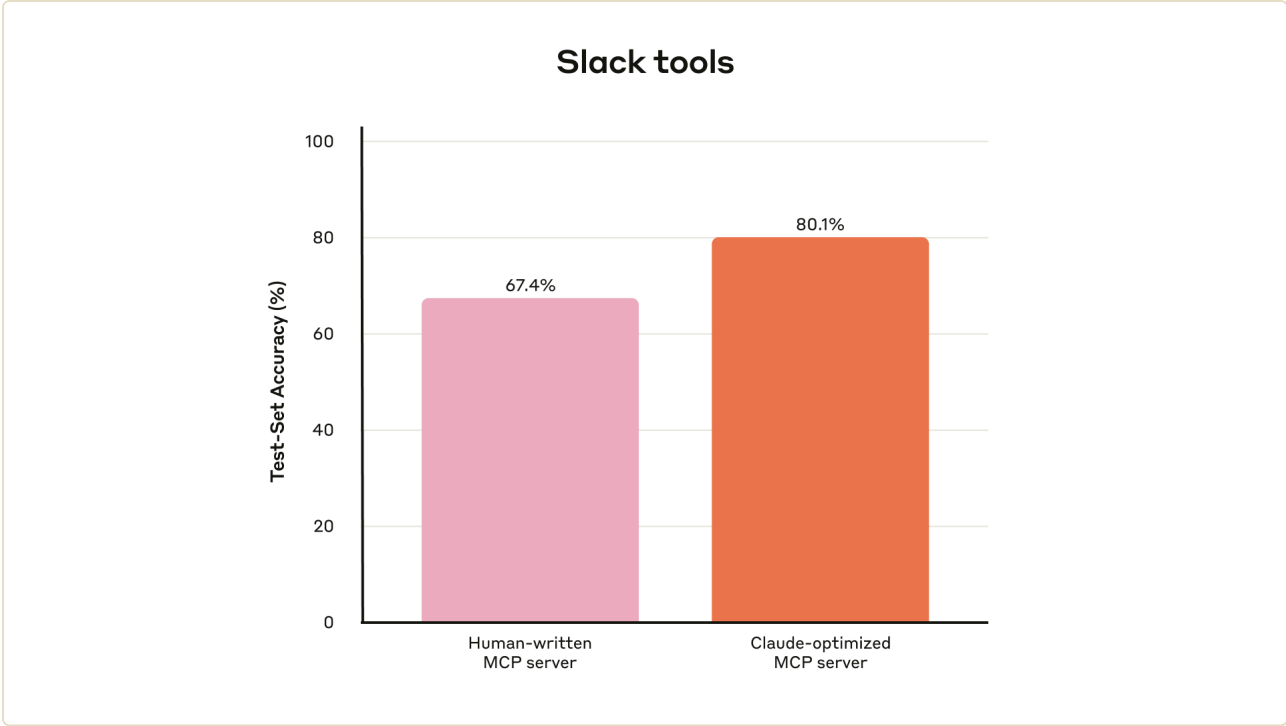
ローカル MCP サーバーまたは DXT を Claude Desktop アプリに接続するには、それぞれ `Settings > Developer` または `Settings > Extensions` に移動します。

ツールは Anthropic API 呼び出しに直接渡してプログラマチックにテストすることもできます。

ツールを自分で試して粗さを見つけてください。ユーザーからフィードバックを集めて、ツールが可能にすべきユースケースとプロンプトの直感を育てましょう。

評価を走らせる

次に、eval を走らせて Claude がツールをどれだけ上手く使えるかを測ります。実世界の利用に根差した eval タスクを多数生成することから始めましょう。結果の分析とツールの改善方法の判断には、エージェントと協力することを勧めます。このプロセスのエンドツーエンドの流れは、ツール評価クックブック を参照してください。



私たちの内部 Slack ツールのホールドアウトテストセットでの性能

評価タスクを生成する

初期プロトタイプがあれば、Claude Code はあなたのツールを素早く探索し、数十のプロンプトと応答ペアを作れます。プロンプトは実世界の利用に触発され、現実的なデータソースとサービス（たとえば内部ナレッジベースやマイクロサービス）に基づいているべきです。ツールに十分な複雑さで負荷をかけない、過度に単純化された／表層的な「サンドボックス」環境は避けることを勧めます。強い eval タスクでは、複数のツール呼び出し——潜在的には数十——が必要になるかもしれません。

強いタスクの例:

- 来週 Jane とミーティングを設定し、最新の Acme Corp プロジェクトについて議論する。前回のプロジェクト計画会議のメモを添付し、会議室を予約してほしい。
- 顧客 ID 9182 が、1 回の購入試行に対して 3 回請求されたと報告してきた。関連するログエントリをすべて見つけ、同じ問題で影響を受けた他の顧客がいないか判定して。
- 顧客 Sarah Chen が解約リクエストを送ってきた。リテンションオファーを用意して。(1) 離脱理由、(2) 最も刺さるリテンションオファー、(3) オファー前に把握しておくべきリスク要因、を判定して。

弱いタスクの例:

- 来週 jane@acme.corp とミーティングを設定。
- 支払いログで `purchase_complete` と `customer_id=9182` を検索。
- 顧客 ID 45892 の解約リクエストを見つけて。

各 eval プロンプトは、検証可能な応答や結果とペアにしてください。検証は、真値とサンプル応答の単純な文字列一致比較のような簡単なものでも、Claude を審査員に据える高度なものでもかまいません。フォーマット、句読点、有効な言い換えの差のような無関係な差で正しい応答を拒絶する、過度に厳格な検証は避けましょう。

プロンプト／応答ペアごとに、エージェントがタスクを解くのに呼ぶと期待されるツールを任意で指定し、各ツールの目的を評価中にエージェントが把握できているかを測ることもできます。ただし、タスクを正しく解く有効な経路は複数あり得るので、戦略への過度な指定や過学習は避けましょう。

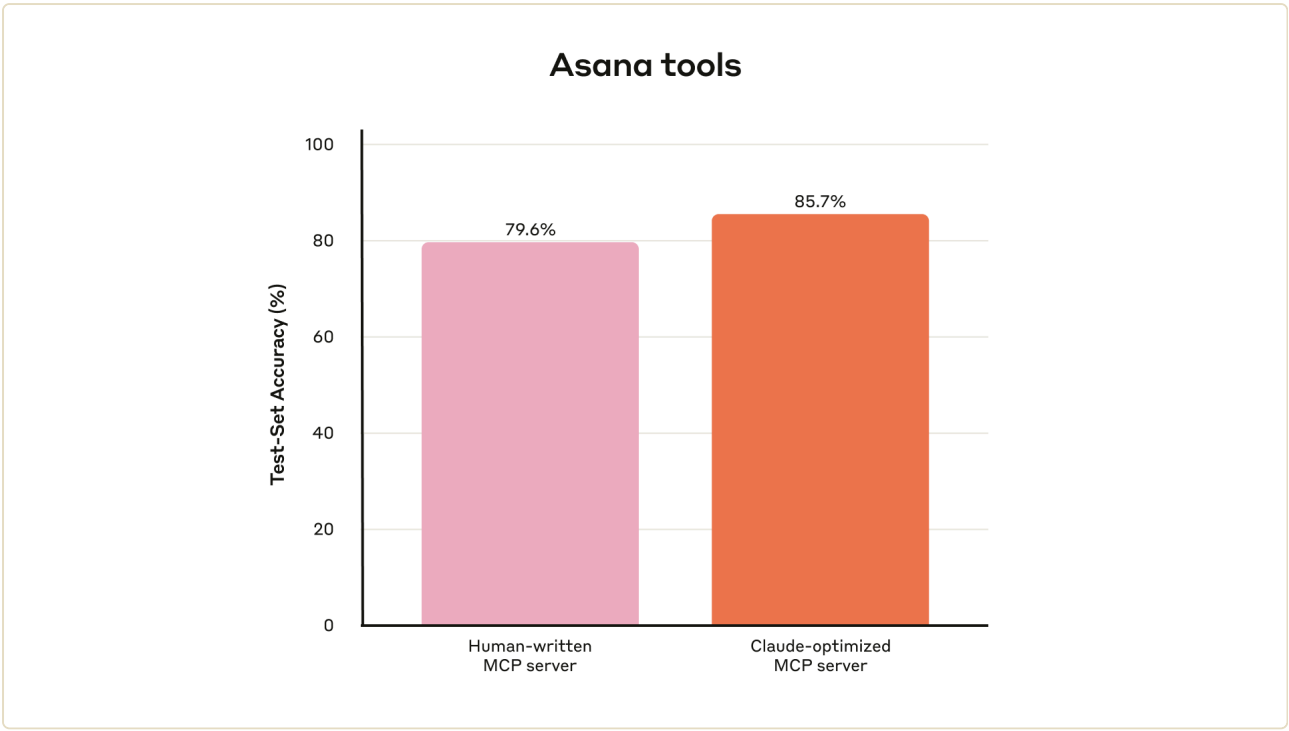
評価を実行する

eval は LLM API を直接呼ぶプログラム的な方法で走らせることを勧めます。シンプルなエージェントループ（LLM API 呼び出しとツール呼び出しを交互に繰り返す `while` ループ）を使い、各 eval タスクに 1 ループを割り当てます。各 eval エージェントには、単一のタスクプロンプトとあなたのツール群を与えます。

eval エージェントのシステムプロンプトには、（検証用の）構造化された応答ブロックだけでなく、推論とフィードバックのブロックも出力させるよう指示することを勧めます。ツール呼び出し・応答ブロックの *前に* これらを出力させると、思考連鎖（CoT）の振る舞いを引き出して LLM の実効知能が上がる可能性があります。

Claude で eval を走らせるなら、interleaved thinking（インターリーブ思考） を有効にすれば同等の機能がすぐに使えます。これは、エージェントが特定のツールを呼ぶ／呼ばない理由を探り、ツールの説明と仕様における改善点を具体的に浮かび上がらせるのに役立ちます。

トップレベルの正確性に加え、個々のツール呼び出しとタスクの総実行時間、ツール呼び出し総数、総トークン消費量、ツールエラーといった指標も集めることを勧めます。ツール呼び出しを追跡すると、エージェントがよく辿るワークフローが見え、ツール統合の機会がわかることがあります。



私たちの内部 Asana ツールのホールドアウトテストセットでの性能

結果を分析する

エージェントは、矛盾したツールの説明、非効率的なツール実装、混乱を招くツールスキーマまで、あらゆる問題を見つけ出しフィードバックを返してくれる心強いパートナーです。ただし、エージェントがフィードバックや応答で *言及しないこと* の方が、言及することより重要な場合が多いことを忘れないでください。LLM は必ずしも本当の意味を言葉にしない のです。

エージェントがつまずいたり混乱したりする箇所を観察しましょう。eval エージェントの推論やフィードバック（CoT）を読み通して粗さを特定します。生のトランスクリプト（ツール呼び出しと応答を含む）を見直し、エージェントの CoT に明示的に書かれていない振る舞いも捕まえましょう。行間を読みましょう——eval エージェントは必ずしも正解や正しい戦略を知っているわけではないのです。

ツール呼び出しの指標を分析しましょう。冗長なツール呼び出しが多いのはページネーションやトークン上限のパラメータの見直しが必要かもしれないというサインであり、無効なパラメータによるツールエラーが多いのは説明や例の改善が必要かもしれません。Claude の Web 検索ツール をローンチしたとき、Claude がツールの `query` パラメータに無意味に `2025` を付けて検索結果にバイアスをかけ性能を下げていることを特定しました（ツール説明を改善して正しい方向に誘導しました）。

エージェントと協力する

エージェントに結果を分析させ、ツールを改善させることさえできます。eval エージェントのトランスクリプトを結合して Claude Code に貼り付けるだけです。Claude はトランスクリプトの分析と、たくさんのツールを一度にリファクタリングするのが得意です——たとえば、新しい変更時にツールの実装と説明が一貫したままになるようにします。

実際、本稿のアドバイスのほとんどは、内部ツール実装を Claude Code で繰り返し最適化するなかで得られたものです。私たちの eval は内部ワークスペースの上に作られ、実プロジェクト、ドキュメント、メッセージを含む内部ワークフローの複雑さを反映しています。

「訓練用」eval に過学習しないよう、ホールドアウトテストセットに頼りました。これらのテストセットは、研究者が手で書いたツール実装や Claude 自身が生成したツール実装を超えて、さらなる性能改善を引き出せることを示しました。

次のセクションでは、このプロセスから学んだことを共有します。

効果的なツールを書くための原則

このセクションでは、学んだことをいくつかの指針に蒸留します。

エージェントに合ったツールを選ぶ

ツールを増やせば結果が良くなるとは限りません。よくあるミスは、既存のソフトウェア機能や API エンドポイントを単に包むだけのツールを作ることです——そのツールがエージェント向きかどうかを問わずに。これは、エージェントが従来ソフトウェアとは異なる「アフォーダンス」を持つからです。ツールで取り得るアクションの感じ取り方が違うのです。

LLM エージェントは「コンテキスト」に限りがあります（一度に処理できる情報量に上限があります）。一方、コンピュータのメモリは安価かつ潤沢です。アドレス帳から連絡先を探すタスクを考えてみてください。従来のソフトウェアは、連絡先のリストを効率的に保存・処理し、1 つずつチェックして次に進めます。

しかし、LLM エージェントがすべての連絡先を返すツールを使って、トークン単位で全員分を読まされるとしたら、関係のない情報に限られたコンテキストを無駄遣いしています（アドレス帳から連絡先を探すのに、すべてのページを上から下まで読む——ブルートフォース検索——のを想像してみてください）。エージェントにとっても人間にとっても、より良い自然なアプローチは、関連ページにまずジャンプする（アルファベット順に見つける）ことです。

私たちが勧めるのは、eval タスクに合わせた、特定の高インパクトワークフローを狙う思慮深いツールを少数作り、そこからスケールすることです。アドレス帳の例なら、`list_contacts` ではなく `search_contacts` や `message_contact` の実装を選ぶかもしれません。

ツールは機能を統合し、内部で *複数の* 個別の操作（あるいは API 呼び出し）を扱うことができます。たとえば、ツール応答を関連メタデータで豊かにしたり、頻繁にチェーンされる多段階タスクを 1 つのツール呼び出しで扱ったりできます。

例:

- `list_users`、`list_events`、`create_event` を実装する代わりに、空き時間を見つけてイベントを予定する `schedule_event` を実装する。
- `read_logs` を実装する代わりに、関連するログ行と周辺文脈だけを返す `search_logs` を実装する。
- `get_customer_by_id`、`list_transactions`、`list_notes` を実装する代わりに、顧客の最近かつ関連情報を一度にまとめる `get_customer_context` を実装する。

作る各ツールには、明確で区別可能な目的を持たせましょう。ツールは、同じ基盤リソースへのアクセスを与えられた人間が行うのとほとんど同じやり方でエージェントがタスクを分割・解決できるようにし、中間出力に使われたはずのコンテキストを節約すべきです。

ツールが多すぎたり機能が重なっていたりすると、エージェントが効率的な戦略を追うのを阻害することもあります。何を作る／作らないかを注意深く選定することは、大きなリターンをもたらします。

ツールに名前空間を付ける

AI エージェントは、数十の MCP サーバーと数百もの異なるツール——他の開発者が作ったものも含めて——にアクセスできるようになるかもしれません。ツールの機能が重なったり目的が曖昧だったりすると、エージェントはどれを使うべきか混乱します。

名前空間（関連ツールを共通の接頭辞でグループ化）は、多数のツール間の境界を画定するのに役立ちます。MCP クライアントにはデフォルトでこれを行うものもあります。たとえば、サービスごと（`asana_search`、`jira_search`）やリソースごと（`asana_projects_search`、`asana_users_search`）に名前空間を付けると、エージェントが正しいタイミングで正しいツールを選ぶ助けになります。

接頭辞ベースと接尾辞ベースの名前空間の選択は、私たちのツール利用 eval に無視できない影響を与えることが分かっています。効果は LLM ごとに変わるので、自分の eval に基づいて命名スキームを選ぶことを勧めます。

エージェントは、間違ったツールを呼んだり、正しいツールを間違ったパラメータで呼んだり、呼ぶツールが少なすぎたり、ツール応答を誤処理したりする可能性があります。タスクの自然な分割を反映した名前のツールを選んで実装することで、同時にエージェントのコンテキストに読み込まれるツールと説明の数を減らし、エージェント側のコンテキストで行う計算をツール呼び出し側へオフロードできます。これは、エージェントが間違いを犯す全体的なリスクを下げます。

意味のある文脈をツールから返す

同じ脈絡で、ツール実装は高信号な情報だけをエージェントに返すよう注意すべきです。柔軟性よりも文脈関連性を優先し、低レベルの技術的識別子（例: `uuid`、`256px_image_url`、`mime_type`）は避けましょう。`name`、`image_url`、`file_type` のようなフィールドは、エージェントの下流アクションや応答を直接導きやすいです。

エージェントは、暗号のような識別子よりも、自然言語の名前・用語・識別子をずっとうまく扱う傾向があります。任意の英数字 UUID を、より意味論的に有意味で解釈可能な言語（あるいは 0 始まりの ID スキーム）に解決するだけで、幻覚を減らして Claude の検索タスクの精度が大幅に改善することが分かっています。

場合によっては、エージェントは下流のツール呼び出しをトリガするためだけにでも、自然言語と技術識別子の両方の出力を操る柔軟性を必要とします（たとえば `search_user(name='jane')` → `send_message(id=12345)`）。ツールにシンプルな `response_format` enum パラメータを露出させ、エージェントがツールを `"concise"` か `"detailed"` 応答のどちらにするか制御できるようにすれば、両方を可能にできます（以下の画像）。

さらに柔軟性が欲しければフォーマットを増やせます——GraphQL のように、受け取りたい情報片を正確に選べるようにするのです。以下は、ツール応答の冗長度を制御する ResponseFormat enum の例です。

```
enum ResponseFormat {
   DETAILED = "detailed",
   CONCISE = "concise"
}
```

詳細なツール応答の例（206 トークン）:

```
● I'll search slack for recent bug reports and use the detailed format to see which channel IDs and threads to
investigate further.

● slack - search (MCP)(query: "bug", sort: "timestamp", sortDir: "desc", limit: 100, responseFormat:
"detailed")
 L Search results for: "bug"

    ═══ Result 1 of 89 ═══
    Channel: #dev (C1234567890)
    From: @jane.doe (U123456789)
    Time: 2024-01-15 10:30:45 UTC
    TS: 1705316445.123456
    Text: Found a critical bug in the login flow.

    ═══ Result 2 of 89 ═══
    Channel: DM with @john.smith
    From: @john.smith (U987654321)
    Time: 2024-01-14 15:22:18 UTC
    TS: 1705247738.234567
    Text: The bug report for issue #123 is ready for review
    Files: bug-report-123.pdf
    …
```

簡潔なツール応答の例(72 トークン):

```
● I'll search slack for recent bug reports and use the concise format to read as many messages as possible.

● slack - search (MCP)(query: "bug", sort: "timestamp", sortDir: "desc", limit: 100, responseFormat: "concise")
 L Search: "bug" (89 results)

    1. #dev - @jane.doe: Found a critical bug in the login flow. [Jan 15]
    2. DM - @john.smith: The bug report for issue #123 is ready for review [Jan 14]
    …
```

Slack のスレッドとスレッド返信は、スレッド返信を取得するのに必要な一意の `thread_ts` で識別されます。`thread_ts` や他の ID(`channel_id`、`user_id`)は、`"detailed"` ツール応答から取り出して、これらが必要な後続ツール呼び出しを可能にできます。`"concise"` ツール応答はスレッド内容だけを返し、ID は除外します。この例では、`"concise"` でトークンを約 1/3 に抑えています。

ツール応答の構造自体——XML、JSON、Markdown など——も eval 性能に影響を与え得ます。万能解はありません。これは LLM が次トークン予測で学習されていて、訓練データと一致するフォーマットでより良い性能を出す傾向があるからです。最適な応答構造はタスクとエージェントごとに大きく異なります。自分の eval に基づいて最良の応答構造を選ぶことを勧めます。

トークン効率のためにツール応答を最適化する

文脈の質を最適化することは重要です。しかし、ツール応答でエージェントに返す文脈の *量* を最適化することも重要です。

コンテキストを大量に使いかねないツール応答には、ページネーション、範囲選択、フィルタリング、切り詰めを、妥当なデフォルト値と組み合わせて実装することを勧めます。Claude Code では、ツール応答をデフォルトで 25,000 トークンに制限しています。エージェントの実効コンテキスト長は時とともに伸びると予想していますが、コンテキスト効率の良いツールの必要性は残り続けるでしょう。

応答を切り詰める場合は、エージェントを助ける指示で導きましょう。知識検索タスクでは、1 回の広い検索ではなく複数の小さく的を絞った検索をするような、よりトークン効率の良い戦略をエージェントに直接勧めることもできます。同様に、ツール呼び出しがエラー（入力バリデーション時など）を出したら、不透明なエラーコードやトレースバックではなく、具体的で実行可能な改善点を明確に伝えるようにエラー応答をプロンプトエンジニアリングできます。

切り詰められたツール応答の例:

```
● I'll find all of your transactions on Stripe and provide a summary for you.

● stripe - transactions_search (MCP)(limit: 5000, responseFormat: "concise")
  L ## Transaction Search Results

    Found **2,847 transactions** matching your query.

    The results are truncated. Showing first 3 results:

    | Date | Description | Amount | Category |
    |------|-------------|--------|----------|
    | 2024-01-15 | Payment from Acme Corp | +$5,200.00 | Revenue |
    | 2024-01-14 | Payment from TechStart | +$3,100.00 | Revenue |
    | 2024-01-13 | Payment from Cloud Co. | +$3,100.00 | Revenue |

    **Summary of all 2,847 results:**
    - Total Revenue: $458,291.00
    - Date Range: Jan 1 - Jan 15, 2024

    ## To refine these results, you can:

    - **Search for specific vendors**: Use `transactions_search(payee: "Acme Corp")` to see only Acme Corp
charges
    - **Filter by amount range**: Use `transactions_search(minAmount: 1000, maxAmount: 5000)`
    - **Get next page**: Use `transactions_search(query: <query>, page: 2)`
```

役に立たないエラー応答の例:

```
● Sure. I'll fetch John's contact information from his profile.

● asana - user_info (MCP)(userId: "john.doe@acme.corp")
 L {
     "error": {
       "code": "RESOURCE_NOT_FOUND",
       "status": 422,
       "message": "Invalid value",
       "details": {
         "field": "userId",
         "value": "john.doe@acme.corp",
         "type": "invalid_value"
       }
     }
   }
```

役に立つエラー応答の例:

```
● Sure. I'll fetch John's contact information from his profile.

● asana - user_info (MCP)(userId: "john.doe@acme.corp")
 L Tool Response:

    # Resource Not Found: Invalid `userId`

    ## Error Summary
    Your request to `/api/user/info` failed because the `userId` `john.doe@acme.corp` does not exist or is in
the wrong format.

    ## Valid User IDs
    Examples:
    - `1928298149291729`
    - `9381719375914731`

    ## Resolving a User ID
    - Call user_search()
```

ツール切り詰めとエラー応答は、エージェントをよりトークン効率の良いツール利用（フィルタやページネーションの使用）に誘導したり、正しくフォーマットされたツール入力の例を示したりできます。

ツール説明をプロンプトエンジニアリングする

ここで、ツールを改善する最も効果的な手法の 1 つに辿り着きます——ツールの説明と仕様のプロンプトエンジニアリングです。これらはエージェントのコンテキストにロードされるため、効果的なツール呼び出し行動へエージェントを集合的に誘導できます。

ツール説明と仕様を書くときは、チームの新人にツールを説明するときのことを考えてみましょう。暗黙的に持ってくるかもしれない文脈——特殊なクエリ形式、ニッチな用語の定義、基盤リソース間の関係——を明示化しましょう。期待される入力と出力を（厳格なデータモデルで強制しつつ）明確に記述してあいまいさを避けましょう。特に、入力パラメータは明確に命名すべきです——`user` という名前のパラメータではなく `user_id` にしてみましょう。

eval があれば、プロンプトエンジニアリングの影響をより確信を持って測定できます。ツール説明の小さな調整でも劇的な改善が得られます。Claude Sonnet 3.5 は、ツール説明を精密に調整した後に SWE-bench Verified 評価で最先端の性能を達成し、エラー率を劇的に下げてタスク完了を改善しました。

ツール定義のその他のベストプラクティスは、開発者ガイド にあります。Claude 向けのツールを作っているなら、ツールが Claude のシステムプロンプト に動的にロードされる仕組みについて読むことも勧めます。最後に、MCP サーバー用のツールを書いているなら、ツール注釈（tool annotations） でどのツールがオープンワールドアクセスや破壊的変更を必要とするかを開示できます。

今後に向けて

エージェント向けの効果的なツールを作るには、ソフトウェア開発のやり方を、予測可能な決定論的パターンから非決定論的パターンへと再方向付けする必要があります。

本稿で述べた反復的・evaluation 駆動のプロセスを通じて、ツールを成功させる一貫したパターンが特定できました——効果的なツールは、意図的かつ明確に定義され、エージェントの文脈を賢く使い、多様なワークフローで組み合わせられ、実世界のタスクを直感的に解決できるようにします。

将来、エージェントが世界と対話する具体的なメカニズム——MCP プロトコルのアップデートから基盤 LLM 自体のアップグレードまで——は進化していくと予想しています。エージェントのツールを改善する体系的・evaluation 駆動のアプローチによって、エージェントがより有能になるにつれ、それが使うツールも共に進化することを保証できます。

謝辞

執筆は Ken Aizawa、Research（Barry Zhang、Zachary Witten、Daniel Jiang、Sami Al-Sheikh、Matt Bell、Maggie Vo）、MCP（Theodora Chu、John Welsh、David Soria Parra、Adam Jones）、Product Engineering（Santiago Seira）、Marketing（Molly Vorwerck）、Design（Drew Roper）、Applied AI（Christian Ryan、Alexander Bricken）からの貴重な貢献とともに。

---

[1] 基盤 LLM 自体の訓練を超えた範囲で。