13

Claude Developer Platform に高度なツール使用機能を導入

— Introducing advanced tool use on the Claude Developer Platform —

公開日	2025-11-24
原題	Introducing advanced tool use on the Claude Developer Platform
著者	Anthropic Engineering Team
原文	https://www.anthropic.com/engineering/advanced-tool-use
翻訳	Claude（機械翻訳／Anthropic）
編集	2026-04-09

Claude Developer Platform に高度なツール使用機能を導入

AI エージェントの未来は、モデルが数百・数千のツールを横断してシームレスに動作する世界です。Git 操作、ファイル操作、パッケージマネージャ、テストフレームワーク、デプロイパイプラインを統合する IDE アシスタント。Slack、GitHub、Google Drive、Jira、社内データベース、そして数十の MCP サーバーを同時に繋ぐ運用コーディネーター。

効果的なエージェント を作るには、すべての定義を前もって文脈に詰め込むことなく、無制限のツールライブラリで働けなくてはなりません。MCP でのコード実行 のブログ記事で議論したように、ツール結果と定義が、エージェントがリクエストを読む前に 5 万トークン以上消費してしまうことがあります。エージェントは必要に応じてツールを発見・ロードし、現在のタスクに関連するものだけを保持すべきです。

エージェントはまた、コードからツールを呼び出す能力も必要です。自然言語でツール呼び出しをすると、各呼び出しが完全な推論パスを必要とし、中間結果が有用かどうかに関係なく文脈に積み上がります。コードはループ、条件分岐、データ変換のようなオーケストレーションロジックに自然に適合します。エージェントは、タスクに応じてコード実行と推論のどちらを使うかを柔軟に選べる必要があります。

エージェントはまた、スキーマ定義だけでなく、例から正しいツールの使い方を学ぶ必要があります。JSON スキーマは構造的に何が有効かを定義できますが、使用パターン(オプションパラメータをいつ含めるべきか、どの組み合わせが意味を持つか、API が期待する慣習は何か)は表現できません。

本日、これを可能にする 3 つの機能をリリースします。

- **Tool Search Tool**: コンテキストウィンドウを消費せずに Claude が検索ツールで数千のツールにアクセスできる機能
- **Programmatic Tool Calling**: コード実行環境でツールを呼び出すことで、モデルのコンテキストウィンドウへの影響を減らす機能
- **Tool Use Examples**: あるツールを効果的に使う方法を示すための普遍的な標準

内部テストでは、これらの機能のおかげで、従来のツール使用パターンでは不可能だったものが作れるようになりました。たとえば **Claude for Excel** は、Programmatic Tool Calling を使って、モデルのコンテキストウィンドウを圧迫することなく数千行のスプレッドシートを読み書きできます。

私たちの経験に基づけば、これらの機能は Claude で作れるものの新しい可能性を開くと信じています。

Tool Search Tool

課題

MCP のツール定義は重要な文脈を提供しますが、接続するサーバーが増えるにつれてトークン使用量は積み上がります。5 サーバー構成の例で考えましょう。

- GitHub: 35 ツール（約 26K トークン）
- Slack: 11 ツール（約 21K トークン）
- Sentry: 5 ツール（約 3K トークン）
- Grafana: 5 ツール（約 3K トークン）
- Splunk: 2 ツール（約 2K トークン）

これで 58 ツールが会話を始める前から約 55K トークンを消費しています。Jira（単独で約 17K トークン）のようなサーバーを追加すれば、すぐに 10 万トークン以上のオーバーヘッドに到達します。Anthropic では、最適化前にツール定義が 134K トークンを消費しているのを見ました。

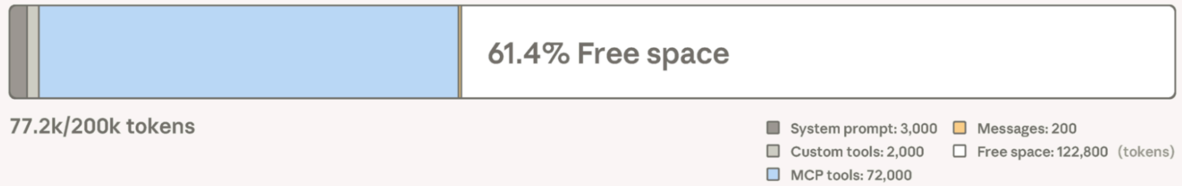
トークンコストだけが問題ではありません。最もよくある失敗は、ツール選択の誤りと誤ったパラメータです——特に `notification-send-user` vs `notification-send-channel` のような似た名前のツールではそうです。

解決策

すべてのツール定義を前もってロードする代わりに、Tool Search Tool はツールをオンデマンドで発見します。Claude は現在のタスクに実際に必要なツールだけを見ます。

Context Usage: Traditional vs. Tool Search Tool

Traditional approach



Tool search tool

Tool Search Tool は Claude の従来アプローチの 122,800 トークンに対して 191,300 トークンの文脈を保持する。

従来アプローチ:

- すべてのツール定義を前もってロード(50+ の MCP ツールで約 72K トークン)
- 会話履歴とシステムプロンプトが残りのスペースを取り合う
- 総文脈消費: 作業を始める前に約 77K トークン

Tool Search Tool あり:

- Tool Search Tool 自体だけを前もってロード(約 500 トークン)
- 必要に応じてツールをオンデマンド発見(関連 3〜5 ツール、約 3K トークン)
- 総文脈消費: 約 8.7K トークン、コンテキストウィンドウの 95% を温存

これは、完全なツールライブラリへのアクセスを維持しつつ、トークン使用量を 85% 削減することを意味します。大規模ツールライブラリに対する内部 MCP evaluation では、Tool Search Tool を有効にすると精度が大きく向上しました——Opus 4 は 49% から 74% へ、Opus 4.5 は 79.5% から 88.1% へ。

Tool Search Tool の仕組み

Tool Search Tool は、Claude がすべての定義を前もってロードする代わりに、動的にツールを発見できるようにします。すべてのツール定義を API に渡しますが、ツールに `defer_loading: true` を付けてオンデマンドで発見可能にします。遅延ツール(deferred tools)は初期状態では Claude の文脈にロードされません。

Claude は Tool Search Tool 自体と、`defer_loading: false` が付いたツール（最も重要で頻繁に使われるツール）だけを見ます。

Claude が特定の能力を必要とすると、関連ツールを検索します。Tool Search Tool は合致するツールへの参照を返し、それが Claude の文脈で完全な定義に展開されます。

たとえば Claude が GitHub と対話する必要があるなら、「github」を検索し、`github.createPullRequest` と `github.listIssues` だけがロードされます——Slack、Jira、Google Drive からの他の 50+ ツールはロードされません。

こうして Claude は完全なツールライブラリへのアクセスを持ちつつ、実際に必要なツールのトークンコストしか支払いません。

プロンプトキャッシュに関する注記: Tool Search Tool はプロンプトキャッシュを壊しません。遅延ツールは初期プロンプトから完全に除外されるからです。Claude が検索した後にのみ文脈に追加されるので、システムプロンプトと中核ツール定義はキャッシュ可能なままです。

実装例:

```
{
  "tools": [
    // Include a tool search tool (regex, BM25, or custom)
    {"type": "tool_search_tool_regex_20251119", "name": "tool_search_tool_regex"},

    // Mark tools for on-demand discovery
    {
      "name": "github.createPullRequest",
      "description": "Create a pull request",
      "input_schema": {...},
      "defer_loading": true
    }
    // ... hundreds more deferred tools with defer_loading: true
  ]
}
```

MCP サーバーでは、サーバー全体の読み込みを遅延しつつ、特定の高頻度使用ツールだけをロード状態に保てます。

```
{
  "type": "mcp_toolset",
  "mcp_server_name": "google-drive",
  "default_config": {"defer_loading": true}, # defer loading the entire server
  "configs": {
    "search_files": {
      "defer_loading": false
    }  // Keep most used tool loaded
  }
}
```

Claude Developer Platform は、regex ベースと BM25 ベースの検索ツールを標準で提供しますが、埋め込みやその他の戦略を使ったカスタム検索ツールも実装できます。

Tool Search Tool を使うべき場面

他のアーキテクチャ上の決定と同様、Tool Search Tool を有効にするかはトレードオフです。この機能はツール呼び出しの前に検索ステップを追加するので、文脈の節約と精度向上が追加のレイテンシを上回るときに最大の ROI をもたらします。

使うべきとき:

- ツール定義が 10K トークン以上を消費している
- ツール選択の精度に問題がある
- 複数サーバーの MCP ベースシステムを構築している
- 10 以上のツールが利用可能

効果が薄いとき:

- 小さなツールライブラリ(10 未満)
- すべてのツールが毎セッションで頻繁に使われる
- ツール定義がコンパクト

Programmatic Tool Calling

課題

従来のツール呼び出しは、ワークフローが複雑になるにつれて 2 つの根本的な問題を生みます。

- **中間結果による文脈の汚染**: Claude が 10MB のログファイルをエラーパターン分析するとき、Claude はエラー頻度の要約しか必要としないのに、ファイル全体がコンテキストウィンドウに入ります。複数のテーブルにまたがって顧客データを取ってくるとき、関連度に関係なくすべてのレコードが文脈に蓄積されます。こうした中間結果は膨大なトークン予算を消費し、重要な情報をコンテキストウィンドウから押し出しかねません。
- **推論オーバーヘッドと手動統合**: 各ツール呼び出しはモデル推論パスを要します。結果を受け取った後、Claude はデータを「目視」して関連情報を抽出し、ピース同士の関係を推論し、次に何をするかを決める——すべて自然言語処理で。5 ツールのワークフローは、5 回の推論パスに加え、Claude が各結果をパースし、値を比較し、結論を統合する作業を意味します。これは遅く、エラーも起きやすいです。

解決策

Programmatic Tool Calling は、Claude がツールを個別の API ラウンドトリップではなくコード経由でオーケストレーションできるようにします。Claude がツールを 1 つずつ要求し、各結果が文脈に戻される代わりに、Claude は複数のツールを呼び出し、出力を処理し、実際にコンテキストウィンドウに入る情報を制御するコードを書きます。

Claude はコードを書くのが得意で、オーケストレーションロジックを自然言語のツール呼び出しではなく Python で表現させれば、より信頼できる精密な制御フローを得られます。ループ、条件分岐、データ変換、エラー処理はすべて、Claude の推論の中で暗黙的に行われるのではなくコードで明示的に行われます。

例: 予算コンプライアンスチェック

一般的なビジネスタスクを考えましょう: 「第 3 四半期の出張予算を超えたチームメンバーは誰?」

3 つのツールが利用可能です。

- `get_team_members(department)` - ID とレベル付きのチームメンバーリストを返す
- `get_expenses(user_id, quarter)` - ユーザーの経費明細を返す
- `get_budget_by_level(level)` - 従業員レベルの予算上限を返す

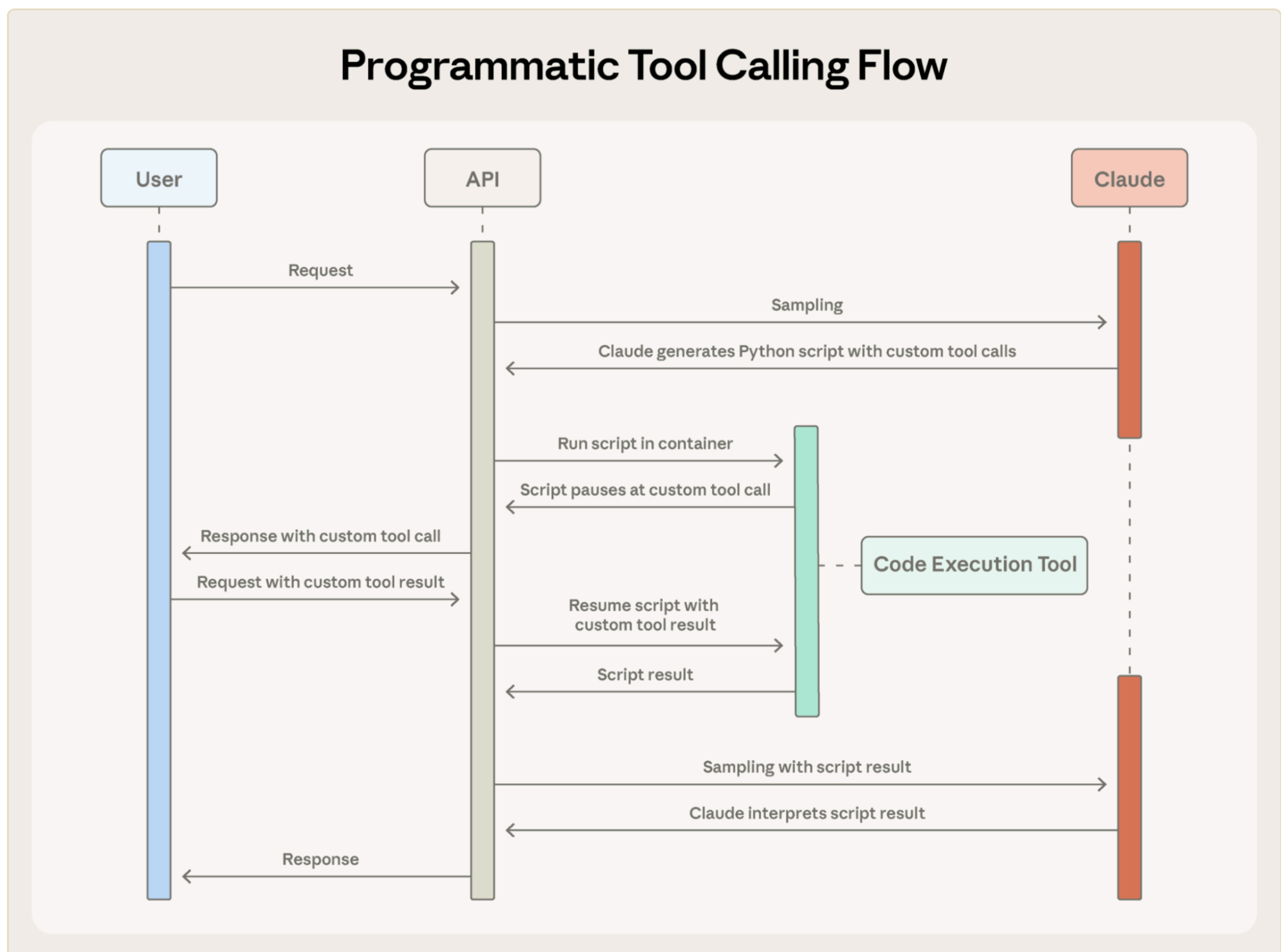
従来アプローチ:

- チームメンバーを取得 → 20 人
- 各人に対して Q3 経費を取得 → 20 回のツール呼び出し、各 50〜100 行の明細（航空券、ホテル、食事、領収書）
- 従業員レベルごとの予算上限を取得
- これすべてが Claude の文脈に入る: 2,000 以上の経費明細（50KB 以上）

- Claude は手で各人の経費を合計し、予算を調べ、経費と予算上限を比較
- モデルとのラウンドトリップが増え、大きな文脈を消費

Programmatic Tool Calling あり:

各ツール結果が Claude に戻される代わりに、Claude はワークフロー全体をオーケストレーションする Python スクリプトを書きます。スクリプトはコード実行ツール（サンドボックス化された環境）で走り、ツールからの結果が必要になったら一時停止します。API 経由でツール結果を返すと、それはモデルに消費されるのではなくスクリプトで処理されます。スクリプトは実行を続け、Claude は最終出力だけを見ます。



Programmatic Tool Calling は、Claude がツールを個別の API ラウンドトリップではなくコード経由でオーケストレーションできるようにし、並列ツール実行を可能にする。

予算コンプライアンスタスクに対する Claude のオーケストレーションコードはこうなります。

```
team = await get_team_members("engineering")

# Fetch budgets for each unique level
levels = list(set(m["level"] for m in team))
budget_results = await asyncio.gather(*[
    get_budget_by_level(level) for level in levels
])

# Create a lookup dictionary: {"junior": budget1, "senior": budget2, ...}
budgets = {level: budget for level, budget in zip(levels, budget_results)}

# Fetch all expenses in parallel
expenses = await asyncio.gather(*[
    get_expenses(m["id"], "Q3") for m in team
])

# Find employees who exceeded their travel budget
exceeded = []
for member, exp in zip(team, expenses):
    budget = budgets[member["level"]]
    total = sum(e["amount"] for e in exp)
    if total > budget["travel_limit"]:
        exceeded.append({
            "name": member["name"],
            "spent": total,
            "limit": budget["travel_limit"]
        })

print(json.dumps(exceeded))
```

Claude の文脈が受け取るのは最終結果だけです: 予算を超えた 2〜3 人。2,000 以上の明細、中間合計、予算ルックアップは Claude の文脈に影響せず、経費の生データ 200KB から結果の 1KB まで消費量が減ります。

効率の向上は実質的です。

- **トークン節約**: 中間結果を Claude の文脈から外すことで、PTC はトークン消費量を劇的に削減。平均使用量は 43,588 から 27,297 トークンに下がり、複雑な調査タスクで 37% の削減。
- **レイテンシ低下**: 各 API ラウンドトリップはモデル推論（数百ミリ秒〜数秒）を必要とします。Claude が 20+ のツール呼び出しを 1 コードブロックでオーケストレーションするなら、19+ の推論パスがなくなります。API は毎回モデルに戻ることなくツール実行を扱います。
- **精度向上**: 明示的なオーケストレーションロジックを書くことで、Claude は自然言語で複数のツール結果をやりくりするよりミスが減ります。内部の知識取得は 25.6% から 28.5% に、GIA ベンチマーク は 46.5% から 51.2% に改善。

本番ワークフローは、乱雑なデータ、条件付きロジック、スケールを必要とする操作を伴います。Programmatic Tool Calling は、Claude が生データ処理ではなく実行可能な結果にフォーカスを保ちつつ、その複雑さをプログラム的に扱えるようにします。

Programmatic Tool Calling の仕組み

1. ツールをコードから呼び出し可能にマーク

`code_execution` を tools に追加し、プログラム的実行にオプトインするツールに `allowed_callers` を設定します。

```
{
  "tools": [
    {
      "type": "code_execution_20250825",
      "name": "code_execution"
    },
    {
      "name": "get_team_members",
      "description": "Get all members of a department...",
      "input_schema": {...},
      "allowed_callers": ["code_execution_20250825"] # opt-in to programmatic tool c
alling
    },
    {
      "name": "get_expenses",
      ...
    },
    {
      "name": "get_budget_by_level",
      ...
    }
  ]
}
```

API はこれらのツール定義を Claude が呼べる Python 関数に変換します。

2. Claude がオーケストレーションコードを書く

ツールを 1 つずつ要求する代わりに、Claude は Python コードを生成します。

```
{
  "type": "server_tool_use",
  "id": "srvtoolu_abc",
  "name": "code_execution",
  "input": {
    "code": "team = get_team_members('engineering')\n..." # the code example above
  }
}
```

3. ツールは Claude の文脈を経由せずに実行

コードが `get_expenses()` を呼ぶと、`caller` フィールド付きのツールリクエストを受け取ります。

```
{
  "type": "tool_use",
  "id": "toolu_xyz",
  "name": "get_expenses",
  "input": {"user_id": "emp_123", "quarter": "Q3"},
  "caller": {
    "type": "code_execution_20250825",
    "tool_id": "srvtoolu_abc"
  }
}
```

あなたが結果を提供し、それは Claude の文脈ではなくコード実行環境で処理されます。この要求／応答サイクルはコード内の各ツール呼び出しごとに繰り返されます。

4. 最終出力だけが文脈に入る

コードの実行が終わると、コードの結果だけが Claude に返されます。

```
{
  "type": "code_execution_tool_result",
  "tool_use_id": "srvtoolu_abc",
  "content": {
    "stdout": "[{\"name\": \"Alice\", \"spent\": 12500, \"limit\": 10000}...]"
  }
}
```

これが Claude の見るすべてで、途中で処理された 2,000+ の経費明細は見ません。

Programmatic Tool Calling を使うべき場面

Programmatic Tool Calling はワークフローにコード実行ステップを追加します。この追加オーバーヘッドは、トークン節約、レイテンシ改善、精度向上が実質的なときに報われます。

最も有益な場面:

- 集約やサマリーだけが必要な大規模データセットを処理する
- 3 つ以上の依存したツール呼び出しを含む多段階ワークフロー
- Claude が見る前にツール結果をフィルタ、ソート、変換する
- 中間データが Claude の推論に影響すべきでないタスク
- 多数のアイテムにまたがる並列操作（たとえば 50 のエンドポイントを確認）

効果が薄い場面:

- 単純な単一ツール呼び出し
- Claude がすべての中間結果を見て推論すべきタスク
- 小さな応答の素早いルックアップ

Tool Use Examples

課題

JSON スキーマは構造（型、必須フィールド、許容される enum）を定義するのが得意ですが、使用パターン（オプションパラメータをいつ含めるか、どの組み合わせが意味を持つか、API が期待する慣習）は表現できません。

サポートチケット API を考えましょう。

```json
{
  "name": "create_ticket",
  "input_schema": {
    "properties": {
      "title": {"type": "string"},
      "priority": {"enum": ["low", "medium", "high", "critical"]},
      "labels": {"type": "array", "items": {"type": "string"}},
      "reporter": {
        "type": "object",
        "properties": {
          "id": {"type": "string"},
          "name": {"type": "string"},
          "contact": {
            "type": "object",
            "properties": {
              "email": {"type": "string"},
              "phone": {"type": "string"}
            }
          }
        }
      },
      "due_date": {"type": "string"},
      "escalation": {
        "type": "object",
        "properties": {
          "level": {"type": "integer"},
          "notify_manager": {"type": "boolean"},
          "sla_hours": {"type": "integer"}
        }
      }
    },
    "required": ["title"]
  }
}
```

スキーマは何が有効かを定義しますが、重要な疑問に答えていません。

- **フォーマットの曖昧さ**: `due_date` は「2024-11-06」「Nov 6, 2024」「2024-11-06T00:00:00Z」のどれを使うべき?
- **ID 慣習**: `reporter.id` は UUID、「USR-12345」、あるいは単に「12345」?
- **ネスト構造の使い方**: Claude はいつ `reporter.contact` を埋めるべき?
- **パラメータの相関**: `escalation.level` と `escalation.sla_hours` は優先度とどう関係する?

こうした曖昧さは不正なツール呼び出しと一貫性のないパラメータ使用につながります。

解決策

Tool Use Examples は、ツール定義の中にサンプルのツール呼び出しを直接提供できるようにします。スキーマだけに頼る代わりに、具体的な使用パターンを Claude に示します。

```
{
    "name": "create_ticket",
    "input_schema": { /* same schema as above */ },
    "input_examples": [
      {
        "title": "Login page returns 500 error",
        "priority": "critical",
        "labels": ["bug", "authentication", "production"],
        "reporter": {
          "id": "USR-12345",
          "name": "Jane Smith",
          "contact": {
            "email": "jane@acme.com",
            "phone": "+1-555-0123"
          }
        },
        "due_date": "2024-11-06",
        "escalation": {
          "level": 2,
          "notify_manager": true,
          "sla_hours": 4
        }
      },
      {
        "title": "Add dark mode support",
        "labels": ["feature-request", "ui"],
        "reporter": {
          "id": "USR-67890",
          "name": "Alex Chen"
        }
      },
      {
        "title": "Update API documentation"
      }
    ]
  }
```

これら 3 つの例から Claude は学びます。

- **フォーマット慣習**: 日付は YYYY-MM-DD、ユーザー ID は USR-XXXXX、ラベルは kebab-case
- **ネスト構造パターン**: ネストされた contact オブジェクトを持つ reporter オブジェクトの構築方法

- **オプションパラメータの相関**: 重大バグには完全な連絡先情報 + タイトな SLA のエスカレーションがある、機能リクエストには reporter があるが contact / escalation は無い、内部タスクには title だけ

内部テストでは、tool use examples は複雑なパラメータ処理の精度を 72% から 90% に改善しました。

Tool Use Examples を使うべき場面

Tool Use Examples はツール定義にトークンを追加するので、精度向上が追加コストを上回るときに最大の価値があります。

最も有益な場面:

- 有効な JSON が正しい使い方を意味しない複雑なネスト構造
- 多くのオプションパラメータと包含パターンが重要なツール
- スキーマに捕捉されないドメイン特化の慣習を持つ API
- 例によってどちらを使うべきかが明確になる類似ツール(例: `create_ticket` vs `create_incident`)

効果が薄い場面:

- 使い方が明白な単一パラメータの単純ツール
- Claude が既に理解している URL やメールのような標準フォーマット
- JSON スキーマ制約でよりよく扱えるバリデーション

ベストプラクティス

現実の行動を取るエージェントを作るには、スケール、複雑性、精度を同時に扱わなければなりません。この 3 つの機能は、ツール使用ワークフローのそれぞれ異なるボトルネックを解決するために協調して働きます。効果的に組み合わせる方法を紹介します。

戦略的に機能を重ねる

すべてのエージェントが与えられたタスクで 3 機能すべてを使う必要はありません。最大のボトルネックから始めましょう。

- ツール定義による文脈の肥大 → Tool Search Tool
- 大きな中間結果による文脈汚染 → Programmatic Tool Calling
- パラメータエラーと不正な呼び出し → Tool Use Examples

このフォーカスしたアプローチは、エージェントの性能を制限している具体的な制約に対処できます。事前に複雑さを足す必要はありません。

次に必要に応じて追加機能を重ねます。これらは補完的です——Tool Search Tool が正しいツールを見つけ、Programmatic Tool Calling が効率的な実行を保証し、Tool Use Examples が正しい呼び出しを保証します。

より良い発見のための Tool Search Tool のセットアップ

ツール検索は名前と説明に対してマッチングするので、明確で説明的な定義が発見精度を改善します。

```
// 良い例
{
    "name": "search_customer_orders",
    "description": "Search for customer orders by date range, status, or total amoun
t. Returns order details including items, shipping, and payment info."
}

// 悪い例
{
    "name": "query_db_orders",
    "description": "Execute order query"
}
```

Claude が何が利用可能かを知るためのシステムプロンプトガイダンスを追加します。

```
You have access to tools for Slack messaging, Google Drive file management,
Jira ticket tracking, and GitHub repository operations. Use the tool search
to find specific capabilities.
```

最も使う 3〜5 のツールを常にロードしたままにし、残りは遅延させます。これは一般操作の即時アクセスとそれ以外のオンデマンド発見のバランスを取ります。

正しい実行のための Programmatic Tool Calling のセットアップ

Claude はツール出力をパースするコードを書くので、返却フォーマットを明確に文書化してください。これは Claude が正しいパースロジックを書くのに役立ちます。

```
{
    "name": "get_orders",
    "description": "Retrieve orders for a customer.
Returns:
    List of order objects, each containing:
    - id (str): Order identifier
    - total (float): Order total in USD
    - status (str): One of 'pending', 'shipped', 'delivered'
    - items (list): Array of {sku, quantity, price}
    - created_at (str): ISO 8601 timestamp"
}
```

プログラム的オーケストレーションから恩恵を受けるオプトインツールの例:

- 並列実行可能なツール(独立した操作)
- 再試行が安全な操作(冪等)

パラメータ精度のための Tool Use Examples のセットアップ

振る舞いの明確化のために例を作りましょう。

- 現実的なデータを使う(実在の都市名、妥当な価格。「string」や「value」ではなく)
- 最小限、部分、完全な指定パターンの多様性を示す
- 簡潔に保つ: ツールあたり 1〜5 例
- 曖昧さにフォーカス(正しい使い方がスキーマから自明でない場所にだけ例を追加)

始め方

これらの機能はベータで利用可能です。有効にするには、ベータヘッダーを追加し、必要なツールを含めます。

```
client.beta.messages.create(
    betas=["advanced-tool-use-2025-11-20"],
    model="claude-sonnet-4-5-20250929",
    max_tokens=4096,
    tools=[
        {"type": "tool_search_tool_regex_20251119", "name": "tool_search_tool_rege
x"},
        {"type": "code_execution_20250825", "name": "code_execution"},
        # Your tools with defer_loading, allowed_callers, and input_examples
    ]
)
```

詳細な API ドキュメントと SDK の例は以下を参照してください。

- Tool Search Tool の ドキュメント と クックブック
- Programmatic Tool Calling の ドキュメント と クックブック
- Tool Use Examples の ドキュメント

これらの機能は、ツール使用を単純な関数呼び出しから知的オーケストレーションへと進めます。エージェントが数十のツールと大きなデータセットにまたがる複雑なワークフローに取り組むにつれ、動的発見、効率的な実行、信頼できる呼び出しが基盤となります。

あなたが何を作るのか、楽しみにしています。

謝辞

執筆は Bin Wu、貢献者は Adam Jones、Artur Renault、Henry Tay、Jake Noble、Noah Picard、Sam Jiang、Claude Developer Platform チーム。本作業は Chris Gorgolewski、Daniel Jiang、Jeremy Fox、Mike Lambert の基礎研究の上に成り立っています。Joel Pobar の LLMVM、Cloudflare の Code Mode、Code Execution as MCP を含め、AI エコシステム全体からインスピレーションも得ています。Andy Schumeister、Hamish Kerr、Keir Bradwell、Matt Bleifer、Molly Vorwerck には特別に感謝します。