02

効果的なエージェントの作り方

— Building effective agents —

公開日	2024-12-19
原題	Building effective agents
著者	Anthropic Engineering Team
原文	https://www.anthropic.com/engineering/building-effective-agents
翻訳	Claude（機械翻訳／Anthropic）
編集	2026-04-09

効果的なエージェントの作り方

この 1 年間、私たちはあらゆる業界で LLM（大規模言語モデル）エージェントを構築する数十のチームと共に仕事をしてきました。一貫して言えるのは、最も成功している実装は複雑なフレームワークや専門ライブラリを使っていない、ということです。彼らはむしろ、シンプルで組み合わせ可能なパターンでエージェントを作っていました。

本稿では、顧客との協業や自分たち自身のエージェント開発から学んだことを共有し、効果的なエージェントを作るための実践的なアドバイスを開発者に向けて書きます。

エージェントとは何か

「エージェント」という言葉はいくつもの意味で使われます。長期間にわたって自律的に動作し、さまざまなツールを使いながら複雑なタスクをこなす完全自律システムをエージェントと呼ぶお客様もいれば、あらかじめ定義されたワークフローに従う、もう少し決めうちな実装をそう呼ぶ方もいます。Anthropic ではこれら全体を**エージェント型システム**（**agentic systems**）と総称しつつ、アーキテクチャ上は **ワークフロー**（**workflows**）と **エージェント**（**agents**）を明確に区別しています。

- **ワークフロー**とは、LLM やツールが、あらかじめ決められたコードパスに沿ってオーケストレーションされるシステムです。
- 一方で **エージェント**とは、LLM 自身がプロセスやツール利用を動的に指揮し、タスクをどう遂行するかの制御を自ら握るシステムです。

本稿では、これら 2 種類のエージェント型システムを詳しく見ていきます。付録 1「実践におけるエージェント」では、顧客がこの種のシステムから特に大きな価値を得ている 2 つの領域を紹介します。

エージェントを使うべき場面、使うべきでない場面

LLM でアプリケーションを作るときは、まずは最もシンプルな解決策を探し、必要になったときだけ複雑さを追加する、というのが私たちの推奨です。場合によっては「そもそもエージェント型システムを作らない」が正解です。エージェント型システムは多くの場合、タスク性能の向上と引き換えにレイテンシとコストを支払います。このトレードオフが本当に成立するかを、まず検討してください。

もう少し複雑さが正当化される場合、ワークフローは定義が明確なタスクに対して予測可能性と一貫性を提供します。一方、エージェントは柔軟性とモデル主導の意思決定を大規模に必要とする場合に適しています。もっとも、多くのアプリケーションでは、検索（retrieval）と文脈内例示（in-context examples）で単一の LLM 呼

び出しを最適化するだけで十分です。

フレームワークを使うか、どう使うか

エージェント型システムの実装を楽にしてくれるフレームワークは数多くあります。たとえば次のようなものです。

- Claude Agent SDK
- Strands Agents SDK by AWS
- Rivet — ドラッグ＆ドロップの GUI による LLM ワークフロービルダー
- Vellum — 複雑なワークフローを構築・テストできるもう 1 つの GUI ツール

こうしたフレームワークは、LLM の呼び出し、ツールの定義とパース、呼び出しの連鎖といった定型的な低レベル作業を単純化してくれるため、手早く立ち上げるのには向いています。しかし、その反面、基盤となるプロンプトとレスポンスを覆い隠してしまう追加の抽象層を作り出しがちで、デバッグを難しくします。また、もっとシンプルな構成で済むはずの場面でも複雑さを持ち込みたくなる誘惑を生みます。

開発者にはまず、LLM API を直接呼ぶところから始めることを勧めます。多くのパターンはほんの数行のコードで実装できるからです。どうしてもフレームワークを使うなら、その下で何が起きているかを必ず理解しておくこと。ライブラリの内部に対する思い込みは、私たちがよく見かけるお客様のエラー原因の 1 つです。

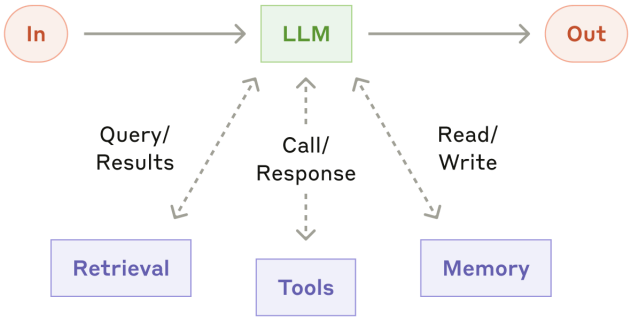
サンプル実装はこちらの cookbook を参照してください。

ビルディングブロック、ワークフロー、エージェント

このセクションでは、本番で実際に見てきたエージェント型システムの一般的なパターンを順に見ていきます。基本となるビルディングブロック、すなわち「拡張された LLM（augmented LLM）」から始め、単純な構成的ワークフロー、そして自律的なエージェントへと段階的に複雑さを増していきます。

ビルディングブロック: 拡張された LLM

エージェント型システムの基本的なビルディングブロックは、検索（retrieval）、ツール、メモリといった拡張を備えた LLM です。現行のモデルはこうした能力を能動的に使いこなすことができます。自ら検索クエリを生成し、適切なツールを選び、何を覚えておくべきかを判断します。



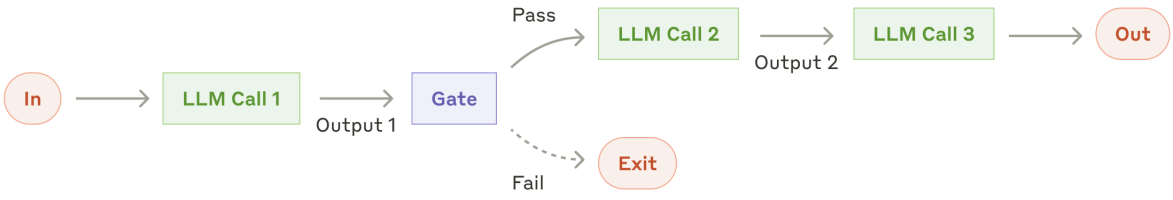
拡張された LLM

実装にあたっては、次の 2 つの観点に特に注目することを勧めます。(1) こうした能力を自分のユースケースに合わせて仕立てること、(2) LLM に対して扱いやすくよくドキュメント化されたインターフェースを提供すること。拡張の実装方法は数多くありますが、1 つのアプローチとしては先頃発表した Model Context Protocol があります。これを使うと、ごくシンプルなクライアント実装でサードパーティ製ツールの成長中のエコシステムに接続できます。

以降の説明では、すべての LLM 呼び出しがこれらの拡張能力にアクセスできる前提で話を進めます。

ワークフロー: プロンプトチェーン(Prompt chaining)

プロンプトチェーンは、タスクを順番に並んだ複数のステップに分解し、各 LLM 呼び出しが前のステップの出力を処理していく方式です。下図の「ゲート(gate)」のように、任意の中間ステップにプログラム的なチェックを入れて、処理が脱線していないかを確認することもできます。



プロンプトチェーンのワークフロー

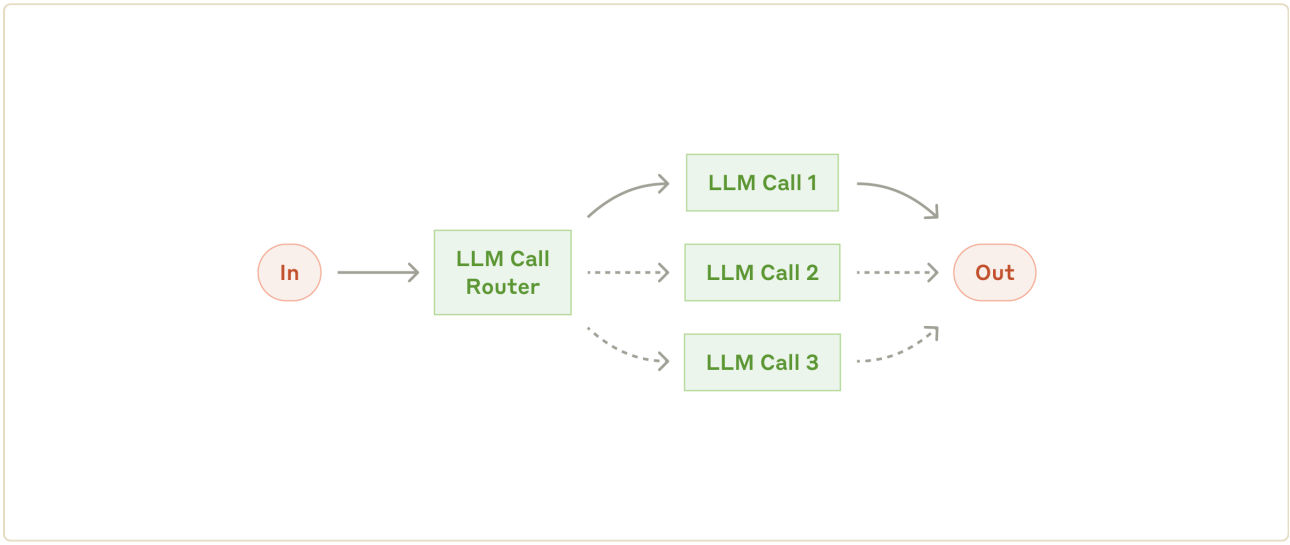
いつ使うか: タスクをきれいに固定のサブタスクへと分解できる状況で理想的です。狙いは、1 回あたりの LLM 呼び出しを易しくすることで、レイテンシと引き換えに精度を上げることにあります。

プロンプトチェーンが有効な例:

- マーケティング用コピーを生成し、そのあと別の言語に翻訳する。
- まず文書のアウトラインを書き、アウトラインが基準を満たしているかチェックし、その後アウトラインに沿って本文を書く。

ワークフロー: ルーティング（Routing）

ルーティングは、入力を分類して専門化された後続タスクに振り分ける方式です。このワークフローによって関心事を分離でき、各タスクにより特化したプロンプトを構築できます。ルーティングなしに 1 種類の入力に最適化すると、別の入力の性能を損なうことがあります。



ルーティングのワークフロー

いつ使うか: 別々に処理したほうがよい明確なカテゴリが存在し、かつ、LLM あるいは従来型の分類モデル／アルゴリズムで正確に分類できるような、複雑なタスクに向きます。

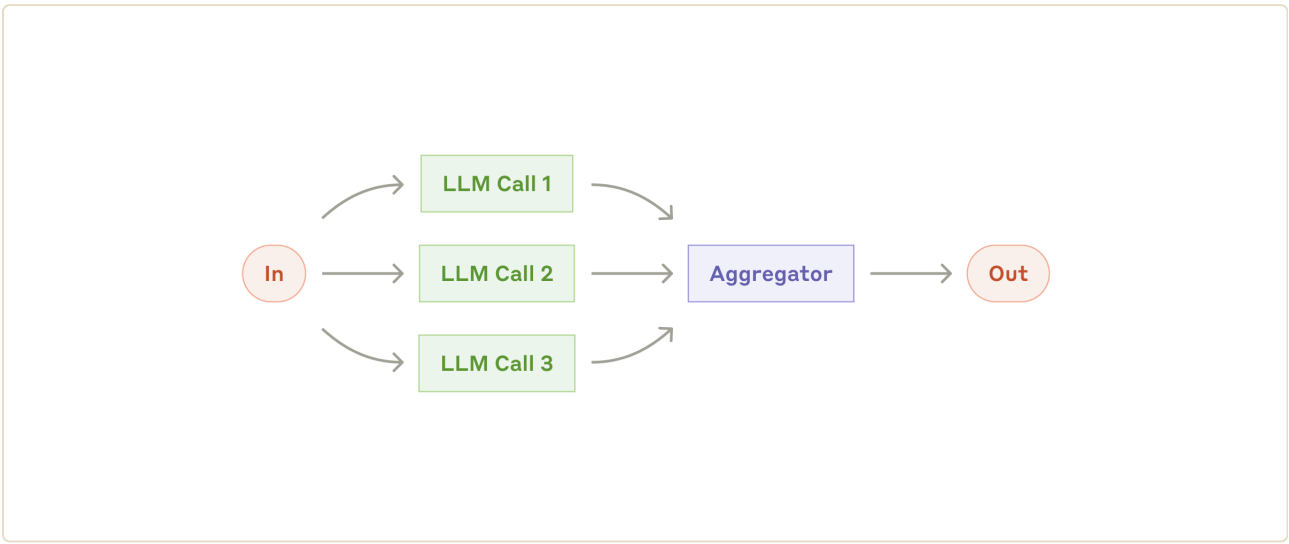
ルーティングが有効な例:

- カスタマーサポートの問い合わせ（一般質問、返金依頼、技術サポート）をそれぞれ異なる下流のプロセス・プロンプト・ツールへ振り分ける。
- 簡単・頻出の質問は Claude Haiku 4.5 のような小型で低コストなモデルに、難しい・非典型的な質問は Claude Sonnet 4.5 のような高性能モデルに振り分け、性能を最適化する。

ワークフロー: 並列化（Parallelization）

LLM は時として、1 つのタスクに対して同時並行に働き、その出力をプログラム的に集約することができます。この「並列化」ワークフローには、大きく 2 つの派生形があります。

- **セクショニング**（Sectioning）: タスクを独立したサブタスクに分割し、並列に実行する。
- **投票**（Voting）: 同じタスクを複数回走らせて多様な出力を得る。



並列化のワークフロー

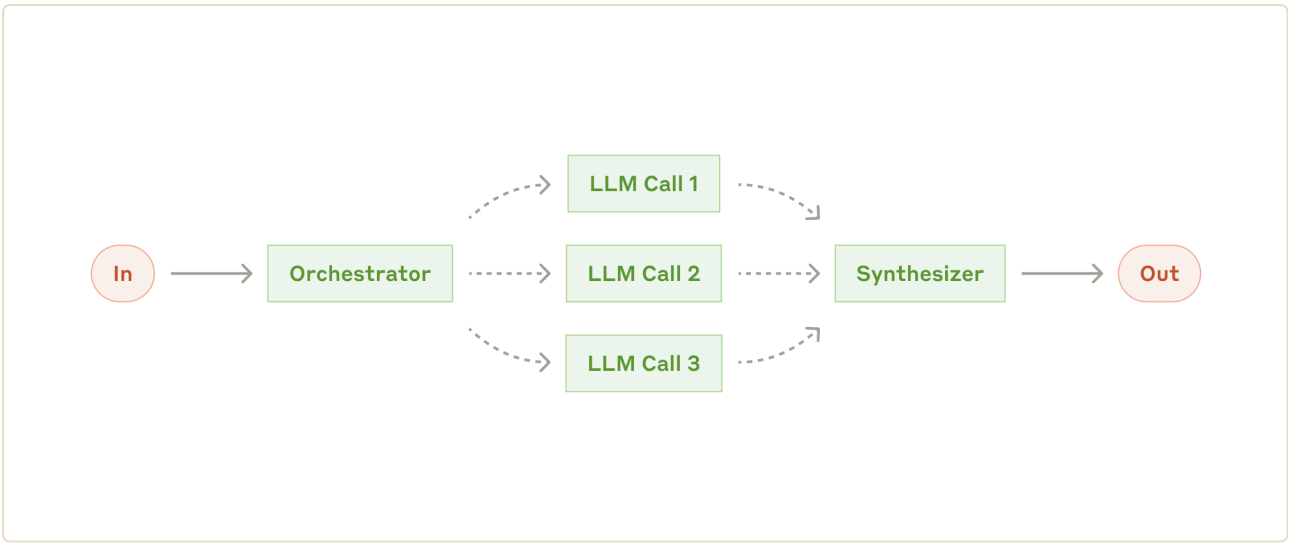
いつ使うか: 分割したサブタスクを並列化すれば速度が稼げる場合、あるいは複数の観点や試行を重ねてより確度の高い結果を得たい場合に有効です。複数の観点を持つ複雑なタスクでは、観点ごとに別個の LLM 呼び出しに処理させた方が、各側面に集中できるため総じて性能が上がる傾向があります。

並列化が有効な例:

- **セクショニング**:
- ガードレールの実装。あるモデルインスタンスがユーザーからのクエリを処理する一方で、別のインスタンスが不適切なコンテンツや要求がないかをスクリーニングする。同じ LLM 呼び出しでガードレールと応答を同時に処理するよりも、こちらの方が性能は良くなる傾向があります。
- LLM の性能を評価する eval の自動化。各 LLM 呼び出しが、与えられたプロンプトに対するモデル性能の異なる側面を評価する。
- **投票**:
- 1 本のコードを脆弱性の観点でレビューする。複数の異なるプロンプトがコードをレビューし、問題があればフラグを立てる。
- 与えられたコンテンツが不適切かどうかを評価する。複数のプロンプトが異なる側面を評価したり、false positive と false negative のバランスを取るために異なる票数しきい値を要求したりする。

ワークフロー: オーケストレーター／ワーカー（Orchestrator-workers）

オーケストレーター／ワーカーのワークフローでは、中央の LLM がタスクを動的に分解し、ワーカー LLM に委任し、結果を統合します。



オーケストレーター／ワーカーのワークフロー

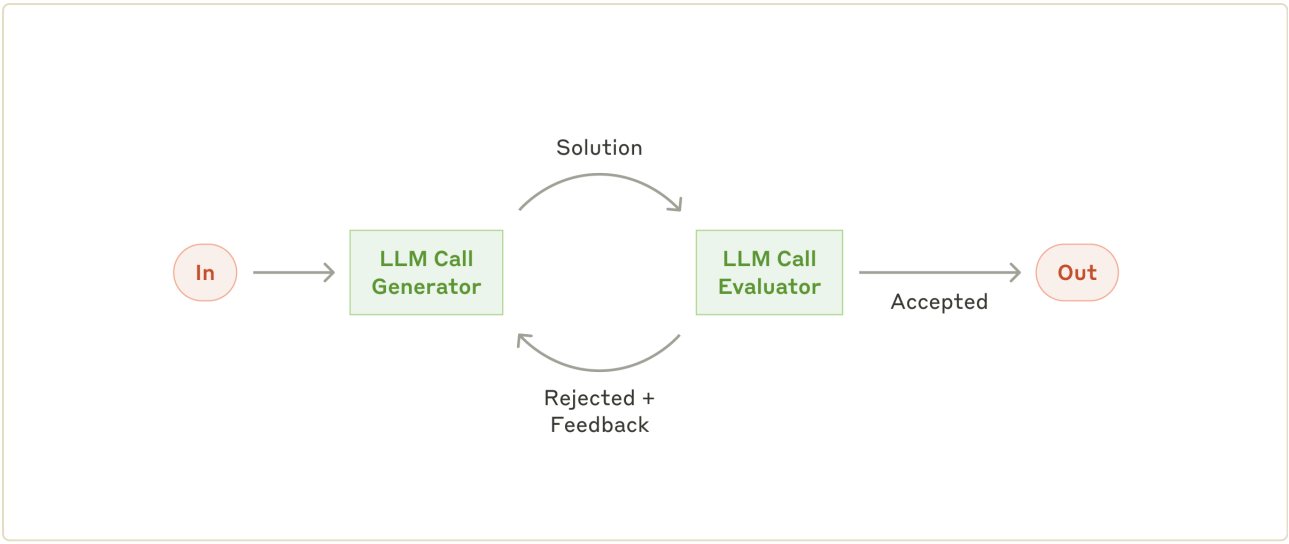
いつ使うか: 必要なサブタスクが事前に予測できない複雑なタスク（たとえばコーディングでは、変更が必要なファイルの本数や各ファイルの変更内容がタスクに依存します）に適しています。トポロジー的には並列化と似ていますが、違いは柔軟性です。サブタスクはあらかじめ定義されているのではなく、入力に応じてオーケストレーターが決めます。

オーケストレーター／ワーカーが有効な例:

- 毎回複数ファイルへの複雑な変更を必要とするコーディング製品。
- 関連情報を得るために複数の情報源から情報を集めて分析する必要がある検索タスク。

ワークフロー: 評価者／最適化者（Evaluator-optimizer）

評価者／最適化者のワークフローでは、ある LLM 呼び出しが応答を生成し、もう 1 つの LLM 呼び出しが評価とフィードバックを返す、というループを回します。



評価者／最適化者のワークフロー

いつ使うか: 評価基準が明確で、反復的な洗練に測定可能な価値がある場合に特に有効です。相性の良さを示すサインは 2 つあります。1 つめは、人間がフィードバックを言語化すると LLM の応答が明らかに改善すること。2 つめは、LLM 自身がそのようなフィードバックを返せること。これは、人間のライターが洗練された文書を仕上げるときの反復プロセスに似ています。

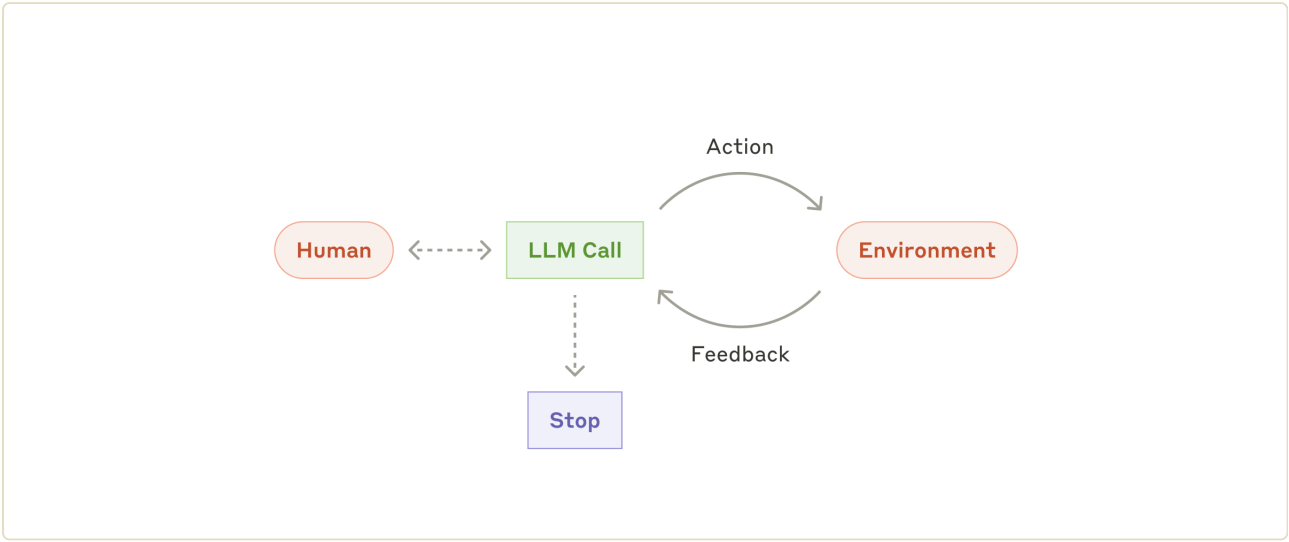
評価者／最適化者が有効な例:

- 文学翻訳。翻訳を担当する LLM が最初は拾いきれない微妙なニュアンスを、評価側の LLM が有用な批評として指摘できる場合。
- 包括的な情報を集めるために検索と分析を複数ラウンド必要とする、複雑な検索タスク。評価者がさらに検索を続けるべきかどうかを判断します。

エージェント

エージェントは、LLM が鍵となる能力——複雑な入力の理解、推論と計画、ツールの信頼できる使用、そしてエラーからの回復——において成熟してきたことを背景に、本番環境でも登場し始めています。エージェントは、人間ユーザーからの指示、あるいは対話的な議論から仕事を始めます。タスクが明確になると、エージェントは計画を立てて自律的に動き出し、必要に応じて人間にさらなる情報や判断を求めにきます。実行中はステップごとに、ツール呼び出しの結果やコード実行結果のような「グラウンドトゥルース」を環境から得ることで、自分の進捗を評価することが重要です。エージェントはチェックポイントや障害に遭遇したタイミングで、人間のフィードバックを求めて立ち止まることもできます。タスクは完了をもって終了することが多いですが、制御を保つために停止条件（たとえば最大反復回数）を組み込むのも一般的です。

エージェントは洗練されたタスクを扱えますが、その実装自体はしばしば素直です。要するに、環境からのフィードバックに応じてツールを使う LLM をループで回しているだけです。そのため、ツール群とそのドキュメントを明快で思慮深く設計することが決定的に重要になります。ツール開発のベストプラクティスについては付録 2「ツールをプロンプトエンジニアリングする」で掘り下げます。



自律エージェント

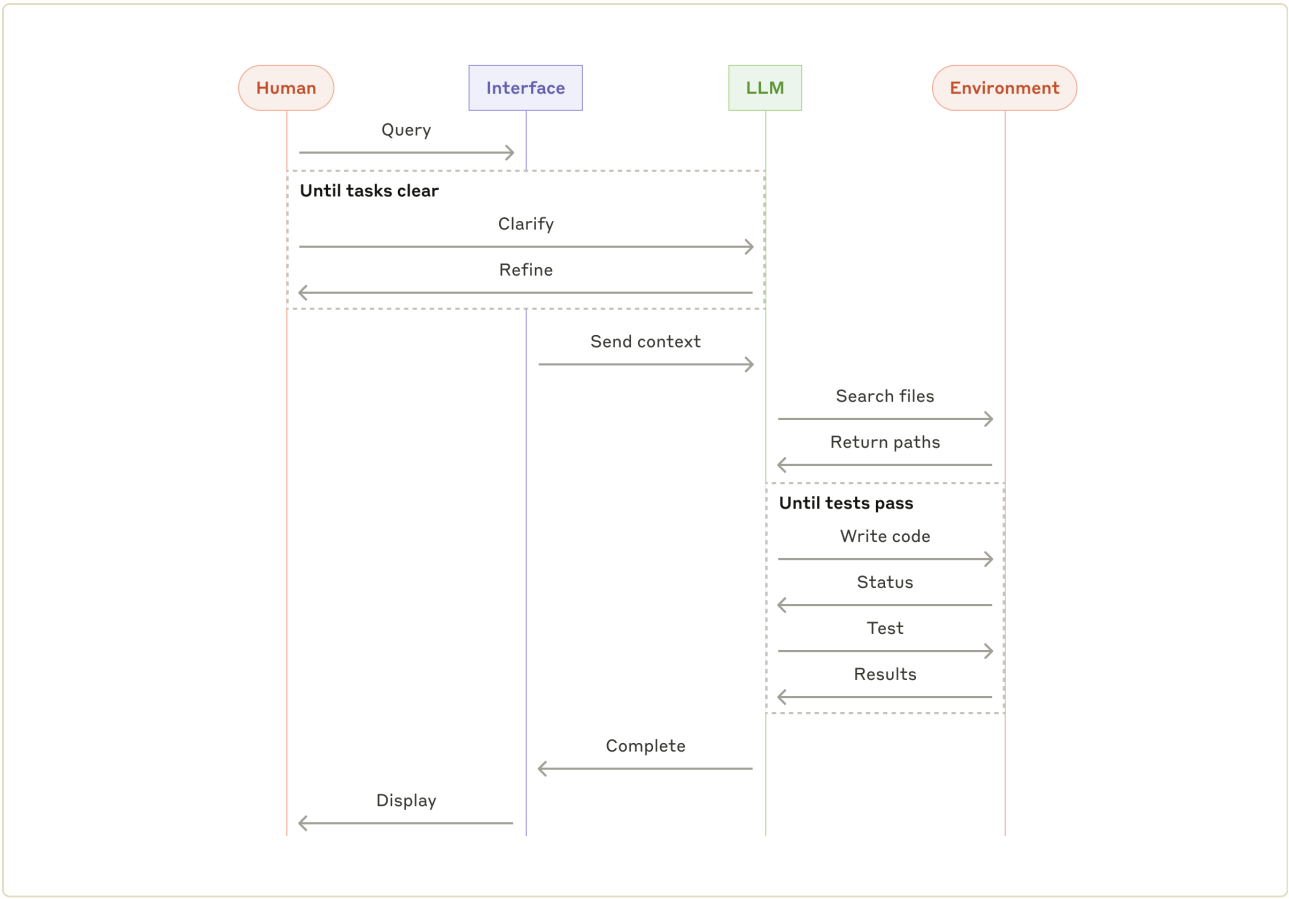
エージェントをいつ使うか: 必要なステップ数の予測が難しい／不可能で、固定のパスをハードコードできないようなオープンエンドな問題に向きます。LLM は多ターンにわたって動作する可能性があり、その意思決定をある程度信頼する必要があります。エージェントの自律性は、信頼できる環境下でタスクをスケールさせるのに最適です。

自律性が高いぶんコストは上がり、エラーが蓄積するリスクもあります。適切なガードレールを設けたうえで、サンドボックス環境での十分なテストを勧めます。

エージェントが有効な例:

以下は私たち自身の実装からの例です。

- SWE-bench タスクを解くコーディングエージェント。タスク記述に基づき多数のファイルを編集します。
- 「コンピュータ利用（computer use）」のリファレンス実装。Claude がコンピュータを操作してタスクを達成します。



コーディングエージェントの概略フロー

これらのパターンを組み合わせてカスタマイズする

これらのビルディングブロックは規範ではありません。開発者が用途に合わせて形を変え、組み合わせることができる、ごく一般的なパターンにすぎません。成功のカギは、LLM 機能全般と同じで、性能を測定し実装を反復することにあります。繰り返し強調します。複雑さは、結果が明確に改善する *場合にのみ* 追加すべきです。

まとめ

LLM 領域での成功は、もっとも洗練されたシステムを作ることではなく、自分のニーズに *合った* システムを作ることにかかっています。シンプルなプロンプトから始め、包括的な evaluation で磨き上げ、シンプルな解が不十分になったときだけ多段階のエージェント型システムを追加すること。

エージェントを実装するとき、私たちは次の 3 つの基本原則に従うよう努めています。

1. エージェントの設計において **シンプルさ** を保つこと。
2. エージェントの計画ステップを明示的に見せることで **透明性** を優先すること。

3. 徹底したツールの **ドキュメント化とテスト** を通じて、エージェント／コンピュータ間のインターフェース（ACI）を丁寧に作り込むこと。

フレームワークは立ち上げを加速してくれますが、本番に進むにあたっては抽象のレイヤーを減らし、基本コンポーネントで組み上げることを躊躇しないでください。これらの原則に従えば、強力なだけでなく、信頼できてメンテナンス可能で、ユーザーから信用されるエージェントを作ることができます。

謝辞

本稿は Erik Schluntz と Barry Zhang によって書かれました。本稿は、Anthropic におけるエージェント構築の経験と、お客様から共有いただいた貴重な知見に基づいています。皆様に深く感謝いたします。

付録 1: 実践におけるエージェント

お客様との取り組みから、AI エージェントのとりわけ有望な応用領域が 2 つ浮かび上がってきました。いずれも、これまで議論してきたパターンの実用的価値を示しています。2 つの応用に共通するのは、会話とアクションの両方を必要とし、成功基準が明確で、フィードバックループが回せ、意味のある人間の監督を組み込めるタスクにおいて、エージェントが最大の価値を加えるという点です。

A. カスタマーサポート

カスタマーサポートは、見慣れたチャットボットのインターフェースと、ツール連携による拡張された能力を組み合わせたものです。これはよりオープンエンドなエージェントにとって自然な適合です。理由は以下のとおりです。

- サポートのやりとりは自然に会話の流れに沿う一方で、外部情報やアクションへのアクセスを必要とします。
- 顧客データ、注文履歴、ナレッジベース記事を引き出すためのツールが統合できます。
- 返金処理やチケット更新といったアクションをプログラム的に扱えます。
- 成功をユーザー定義の「解決（resolution）」として明確に測定できます。

複数の企業が、解決したケースに対してのみ課金する利用量ベースの価格モデルでこのアプローチの有効性を示しています。それは、自社エージェントの有効性に対する自信の表れでもあります。

B. コーディングエージェント

ソフトウェア開発領域は、コード補完から自律的な問題解決へと能力が進化し、LLM 機能にとって驚くべき可能性を見せてきました。エージェントがとりわけ効果的である理由は次のとおりです。

- コードの解決策は自動テストによって検証可能です。
- エージェントはテスト結果をフィードバックとして、解決策を反復できます。
- 問題空間は十分に定義・構造化されています。
- 出力品質を客観的に測定できます。

私たち自身の実装では、エージェントは今や SWE-bench Verified ベンチマークの実際の GitHub issue を、プルリクエストの説明文のみに基づいて解くことができます。ただし、自動テストは機能を検証するには役立つものの、システム全体の要件に解決策が整合しているかを確認するには、依然として人間のレビューが欠かせません。

付録 2: ツールをプロンプトエンジニアリングする

どのようなエージェント型システムを作るにせよ、ツールはエージェントの重要な構成要素になるはずです。ツール（Tools）は、API 上で正確な構造と定義を指定することで Claude が外部サービスや API とやり取りできるようにする仕組みです。Claude が応答する際、ツールを呼び出そうと計画していれば、API レスポンスに tool use ブロック が含まれます。ツールの定義と仕様には、全体のプロンプトと同じくらいのプロンプトエンジニアリング上の注意を払うべきです。この短い付録では、ツールをどうプロンプトエンジニアリングするかを説明します。

同じアクションを表す方法は、しばしば複数あります。たとえばファイル編集は、diff を書くことで指定することもできれば、ファイル全体を書き直すことで指定することもできます。構造化出力についても、markdown の中にコードを入れて返すことも、JSON の中に入れて返すこともできます。ソフトウェアエンジニアリングの観点では、こうした違いは表層的で、相互に無損失で変換できます。しかし、ある形式は LLM にとって他の形式よりもはるかに書きにくいことがあります。diff を書くには、新しいコードを書き始める前にチャンクヘッダー中の変更行数を知っておかなければなりません。markdown ではなく JSON の中にコードを書くには、改行と引用符を追加でエスケープする必要があります。

ツールのフォーマットを決めるにあたっての私たちの提案は次のとおりです。

- モデルが行き詰まる前に「考える」ための十分なトークンを与えること。
- モデルがインターネット上の文章で自然に見てきたものに近いフォーマットを保つこと。
- 数千行のコードの正確な行数管理や、自身が書くコードの文字列エスケープのような、フォーマット上の「オーバーヘッド」を持ち込まないこと。

1 つの目安として、ヒューマン・コンピュータ・インターフェース（HCI）にどれだけの労力が割かれているかを考え、同じくらいの労力を *エージェント・コンピュータ・インターフェース*（ACI）の設計にも投じる、と考えるとよいでしょう。その進め方についてのいくつかの考えを紹介します。

- モデルの立場で考えてみること。説明とパラメータを見れば、このツールの使い方は明らかですか、それとも注意深く考える必要がありますか？ 後者ならば、モデルにとっても同じでしょう。良いツール定義はしばしば、使用例、エッジケース、入力フォーマットの要件、他のツールとの明確な境界を含みます。
- パラメータ名や説明を変えることで、どうすればより明快になるか。チームの新人開発者のために素晴らしい docstring を書くつもりで、と考えてください。似たようなツールを多数使うときほど重要になります。
- モデルがあなたのツールをどう使うかテストすること。私たちの workbench で多くの例を流し、モデルがどんな間違いをするかを観察して反復すること。
- ツールを ポカヨケ すること。引数を変更して、間違えにくくすること。

SWE-bench 用のエージェントを作っていたとき、私たちは実際、全体プロンプトよりもツールの最適化に多くの時間を使いました。たとえば、エージェントがルートディレクトリから移動した後に相対パスを使うツールで間違いを犯すことがわかったため、ツールを常に絶対パスを要求するように変更しました。すると、モデルはこの方式を一度も間違えずに使うようになりました。