06

マルチエージェント調査システムをどう作ったか

— How we built our multi-agent research system —

公開日	2025-06-13
原題	How we built our multi-agent research system
著者	Anthropic Engineering Team
原文	https://www.anthropic.com/engineering/multi-agent-research-system
翻訳	Claude（機械翻訳／Anthropic）
編集	2026-04-09

マルチエージェント調査システムをどう作ったか

Claude には、Web、Google Workspace、各種連携を横断して検索し、複雑なタスクをこなすリサーチ（Research）機能が備わっています。

このマルチエージェントシステムをプロトタイプから本番に持っていくまでの道のりで、私たちはシステムアーキテクチャ、ツール設計、プロンプトエンジニアリングに関する重要な教訓を得ました。マルチエージェントシステムとは、複数のエージェント（ツールをループで自律的に使う LLM）が協調して動くシステムのことです。私たちのリサーチ機能は、ユーザーのクエリに基づいて調査プロセスを計画するエージェントが、ツールを使って並列エージェントを作り、情報を同時に検索するという仕組みです。複数エージェントが関与するシステムは、エージェント間の協調、評価、信頼性に新たな課題をもたらします。

本稿では、私たちにとって機能した原則を分解します。自分たちのマルチエージェントシステムを構築する際にお役に立てば幸いです。

マルチエージェントシステムの利点

調査はオープンエンドな問題が多く、必要なステップを事前に予測するのは非常に困難です。複雑なトピックの探索には固定の道筋をハードコードできません。プロセスは本質的に動的かつ経路依存だからです。人間が調査するとき、発見に応じて継続的にアプローチを更新し、調査中に出てきた手がかりを追いかけていきます。

この予測不可能性のために、AI エージェントは調査タスクにとりわけ向いています。調査には、進展に応じて方向転換したり、脇道のつながりを探したりする柔軟性が必要です。モデルは多ターンにわたって自律的に動き、途中の発見に基づいてどの方向を追うかの意思決定を行う必要があります。直線的な一発パイプラインでは、こうしたタスクは扱えません。

検索の本質は「圧縮」です。膨大なコーパスから洞察を蒸留することです。サブエージェントは、それぞれ独自のコンテキストウィンドウで並列に動作し、問いのさまざまな側面を同時に探索し、最も重要なトークンをリードエージェントに凝縮して返すことで、この圧縮を支援します。各サブエージェントは関心事の分離ももたらします——別々のツール、プロンプト、探索の軌跡——これが経路依存性を下げ、徹底した独立調査を可能にします。

知能があるしきい値を超えると、マルチエージェントシステムは性能をスケールさせる重要な方法になります。たとえば、過去 10 万年で個々の人間は賢くなってきましたが、情報化時代の人類社会は *集合知* と協調能力のおかげで *指数的に* 強力になっています。汎用的に賢いエージェントであっても、個体として動く限り限界があります。群れのエージェントはずっと多くのことを成し遂げられます。

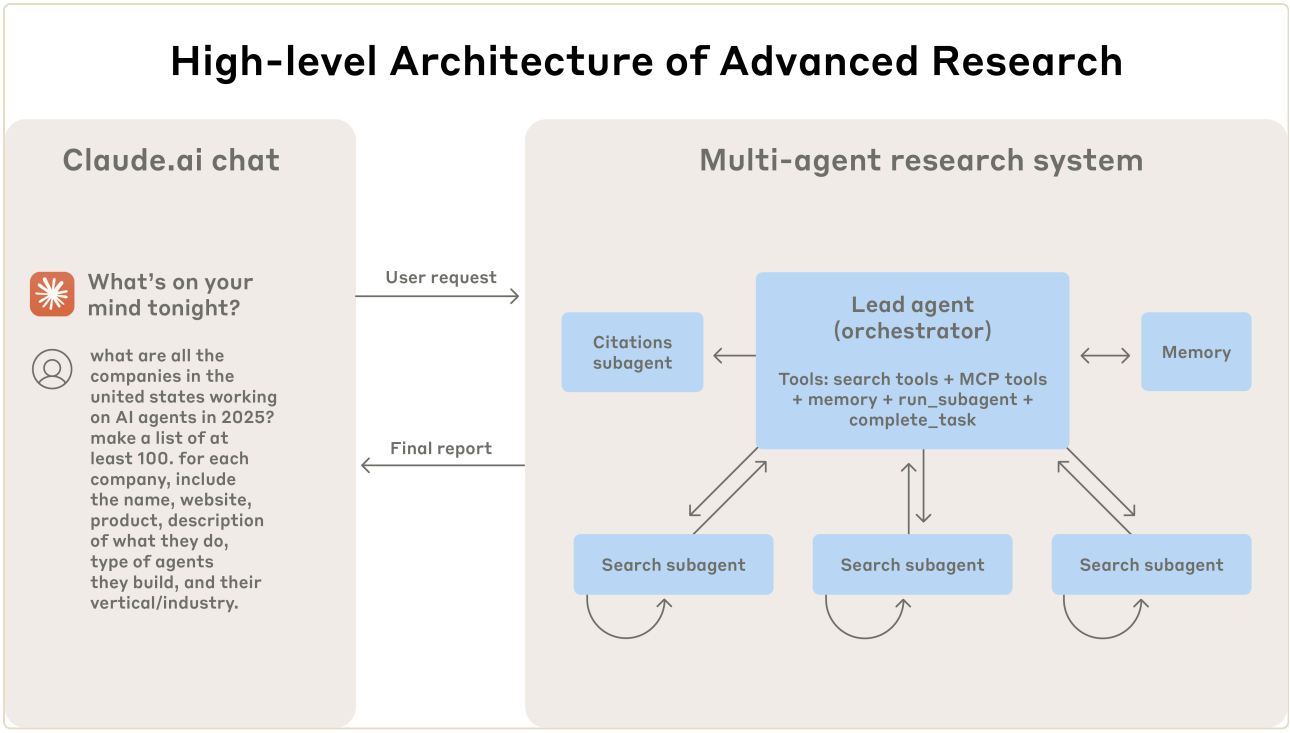
私たちの内部評価では、マルチエージェント調査システムは特に、複数の独立した方向を同時に追う「広く探索する（breadth-first）」クエリで優れた性能を発揮します。Claude Opus 4 をリード、Claude Sonnet 4 をサブエージェントとするマルチエージェントシステムは、内部調査 eval で単体の Claude Opus 4 を 90.2% 上回りました。たとえば「情報技術系 S&P 500 の全企業の取締役を特定せよ」と尋ねると、マルチエージェントシステムはこれをサブエージェント向けのタスクに分解して正答に辿り着いた一方、単体エージェントシステムは遅い逐次検索で答えに到達できませんでした。

マルチエージェントシステムがうまく働くのは、主に「問題を解くのに十分なトークンを使う」助けになるからです。BrowseComp 評価（ブラウジングエージェントが見つけにくい情報を探し当てる能力を測る）のパフォーマンス分散の 95% を、3 つの因子が説明しました。トークン使用量だけで 80% を説明し、残りの 2 因子はツール呼び出し回数とモデル選択でした。この発見は、並列推論の容量を足すために別々のコンテキストウィンドウを持つエージェント群に作業を分散する、という私たちのアーキテクチャを裏付けています。最新の Claude モデルは、トークン使用量に対する大きな効率倍率として働きます——Claude Sonnet 4 へのアップグレードは、Claude Sonnet 3.7 のトークン予算を 2 倍にするより性能向上が大きいのです。マルチエージェントアーキテクチャは、単体エージェントの限界を超えるタスクでトークン使用量を効果的にスケールさせます。

デメリットもあります。実際にはこれらのアーキテクチャはトークンを猛烈に消費します。私たちのデータでは、エージェントは普通のチャットの 4 倍ほどのトークンを使い、マルチエージェントシステムは 15 倍ほど使います。経済的に成立するには、性能向上のコストを正当化できるほど価値の高いタスクが必要です。さらに、すべてのエージェントが同じコンテキストを共有する必要があるドメインや、エージェント間に多くの依存があるドメインは、現状ではマルチエージェントシステムに向きません。たとえば、ほとんどのコーディングタスクは調査ほど真に並列化可能ではなく、LLM エージェントはリアルタイムで他エージェントに協調・委譲するのはまだ不得手です。マルチエージェントシステムが光るのは、重い並列化を伴い、単一コンテキストウィンドウに収まらない情報を扱い、多数の複雑なツールとやりとりする価値の高いタスクです。

リサーチのアーキテクチャ概要

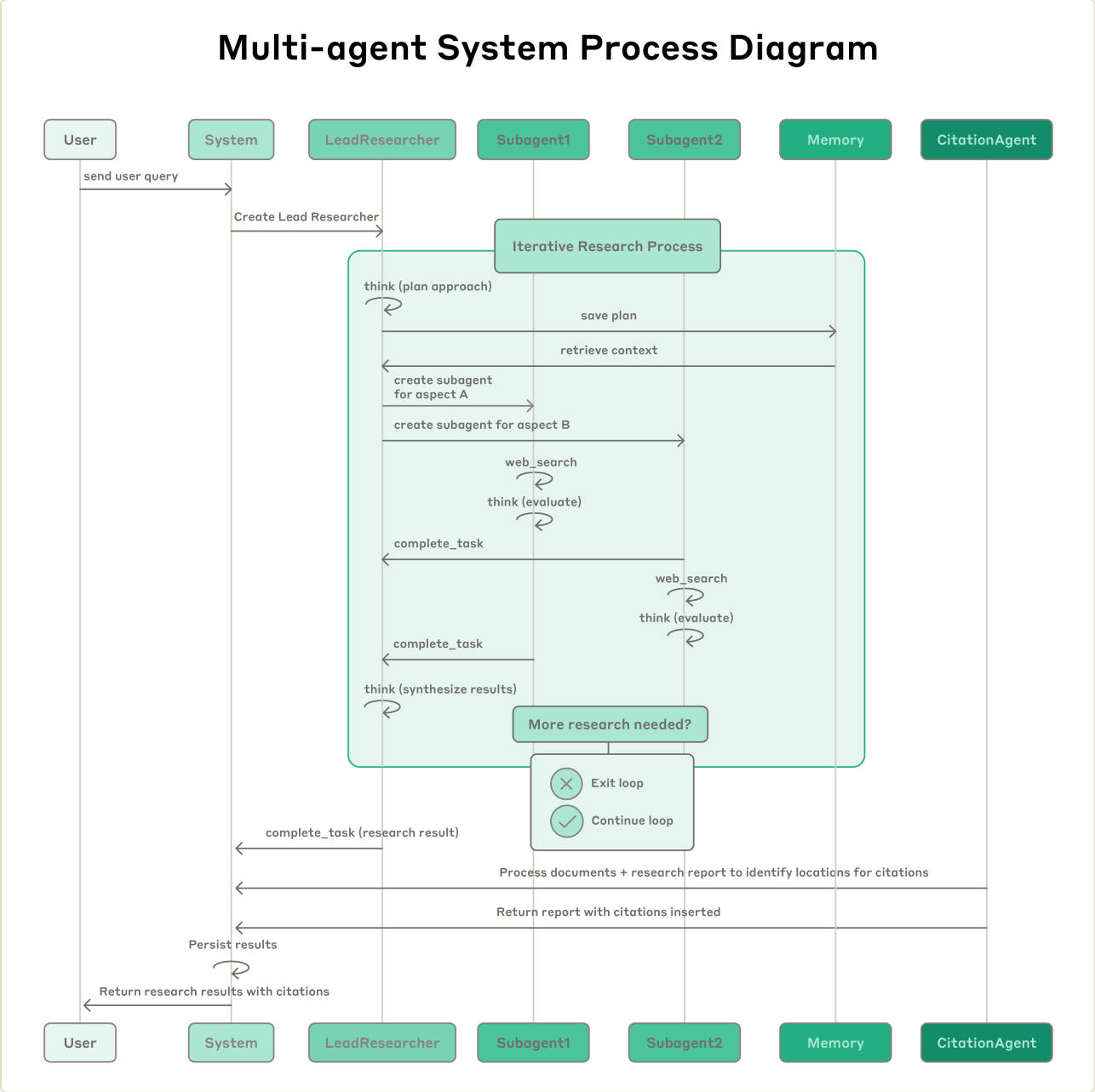
私たちのリサーチシステムは、オーケストレーター／ワーカーパターンのマルチエージェントアーキテクチャを採用しています。リードエージェントがプロセスを調整し、専門化されたサブエージェント群に並列で委譲します。



実際に動くマルチエージェントアーキテクチャ: ユーザーのクエリはリードエージェントに流れ、リードエージェントが異なる側面を並列検索する専門サブエージェントを作る。

ユーザーがクエリを送ると、リードエージェントがそれを分析し戦略を立て、異なる側面を同時に探索するサブエージェントを生成します。上図のように、サブエージェントは検索ツールを反復的に使って情報を集める知的フィルタとして振る舞い（この例では 2025 年の AI エージェント企業を調べる）、企業のリストをリードエージェントに返します。リードエージェントはそれを基に最終回答を組み立てます。

従来的な Retrieval Augmented Generation（RAG）は静的検索を用います。入力クエリに最も似ているチャンク群を取得し、それらで応答を生成します。対照的に、私たちのアーキテクチャは多段階の検索を使い、関連情報を動的に見つけ、新しい発見に適応し、結果を分析して高品質の回答を形成します。



マルチエージェントリサーチシステムの全体プロセス図。ユーザーのクエリ送信後、システムは LeadResearcher エージェントを作り、反復的な調査プロセスに入る。LeadResearcher はまずアプローチを考え、計画をメモリに保存して文脈を永続化する（20 万トークン超でコンテキストが切り詰められたときに計画を失わないため）。次に具体的な調査タスクを持つ専門サブエージェントを作成する（図は 2 つだが、任意の数を作れる）。各サブエージェントは独立して Web 検索を行い、インターリーブ思考（interleaved thinking）を使ってツール結果を評価し、発見を LeadResearcher に返す。LeadResearcher は結果を統合し、さらに調査が必要かを判断する——必要なら追加のサブエージェントを作ったり戦略を磨いたりする。十分な情報が集まると、システムは調査ループを抜け、すべての発見を CitationAgent に渡す。CitationAgent はドキュメントと調査レポートを処理して引用の具体的場所を特定する。これにより、すべての主張が出典に適切に帰属される。最後に、引用付きの調査結果がユーザーに返される。

リサーチエージェントのプロンプトエンジニアリングと評価

マルチエージェントシステムは単体エージェントシステムと大きく異なる点があり、特に協調の複雑さが急速に増大します。初期のエージェントは、単純なクエリに 50 個のサブエージェントを生成したり、存在しない情報を求めて Web を永遠にさまよったり、更新情報の過多で互いを混乱させたりしていました。各エージェントはプロンプトで操縦されるため、プロンプトエンジニアリングがこうした振る舞いを改善する主要なレバーでした。以下は、エージェントをプロンプトする上で学んだ原則の一部です。

1. **エージェントになったつもりで考える。**プロンプトを反復するには、その効果を理解する必要があります。そのために私たちは、実際のプロンプトとツールをそのまま使ったシミュレーションをコンソールで作り、エージェントの動きをステップバイステップで観察しました。これで失敗モードが即座に可視化されました——十分な結果があるのに続けてしまう、過度に冗長な検索クエリを使う、間違ったツールを選ぶ、など。効果的なプロンプトは、エージェントの正確なメンタルモデルに依存しており、それが最もインパクトのある変更を自明にします。
2. **オーケストレーターに委譲の仕方を教える。**私たちのシステムでは、リードエージェントがクエリをサブタスクに分解してサブエージェントに伝えます。各サブエージェントには目的、出力フォーマット、使うべきツールとソースのガイダンス、明確なタスク境界が必要です。詳細なタスク記述がなければ、エージェントは仕事を重複させたり、抜けを作ったり、必要な情報を見つけ損ねたりします。最初は「半導体不足を調べて」のような短い指示をリードに許していましたが、指示が曖昧すぎてサブエージェントがタスクを誤解したり、他のエージェントと全く同じ検索を行ったりしました。例として、あるサブエージェントが 2021 年の自動車チップ危機を調べる一方、他の 2 つが 2025 年の現在の供給網を重複して調査し、労働分担が効いていませんでした。
3. **クエリの複雑さに応じて労力をスケールする。**エージェントはタスクごとの適切な労力を判断するのが苦手なので、スケーリングルールをプロンプトに埋め込みました。単純な事実調査は 3〜10 回のツール呼び出しを 1 エージェントで、直接比較は 10〜15 回のツール呼び出しを 2〜4 サブエージェントで、複雑な調査は 10 以上のサブエージェントで責任を明確に分担して、といった具合です。これらの明示的なガイドラインは、リードエージェントがリソースを効率的に配分する助けとなり、単純クエリへの過剰投資（初期バージョンでよくあった失敗モード）を防ぎます。
4. **ツール設計と選択が決定的に重要。**エージェントとツールのインターフェースは、人間とコンピュータのインターフェースと同じくらい重要です。正しいツールを使うと効率的です——しばしば、それが厳密に必要です。たとえば Slack にしかない文脈を Web で検索するエージェントは最初から失敗しています。MCP サーバーで外部ツールへアクセスできるようになると、この問題は悪化します。エージェントが初見のツールに遭遇し、その説明文の質はピンキリだからです。私たちはエージェントに明示的なヒューリスティックを与えました——たとえば、まず利用可能なツールをすべて確認する、ツール使用をユーザー意図に合わせる、広く外部を探索するなら Web を検索する、汎用より特化ツールを優先する、など。悪いツール説明はエージェントを完全に誤った経路に送り出すので、各ツールには明確な目的と明確な説明が必要です。

5. **エージェントに自己改善させる。**Claude 4 モデルは優れたプロンプトエンジニアになり得ることが分かりました。プロンプトと失敗モードを与えると、なぜエージェントが失敗しているかを診断し改善を提案できます。私たちはツールテストエージェントすら作りました——不完全な MCP ツールを与えると、そのツールを使ってみた上で失敗を避けるようにツール説明を書き直します。このエージェントはツールを何十回もテストして、主要なニュアンスとバグを発見しました。このプロセスでツールの使い勝手を改善した結果、新しい説明を使う将来のエージェントはほとんどのミスを避けられるようになり、タスク完了時間が 40% 短縮しました。
6. **広く始めて、徐々に絞る。**検索戦略は熟練した人間の調査を模すべきです——詳細に入る前に、まず全景を把握する。エージェントはしばしば過度に長く具体的なクエリをデフォルトにして、結果をほとんど返さないことになります。私たちはこの傾向を打ち消すため、短く広いクエリから始め、利用可能な情報を評価し、その後フォーカスを徐々に絞るようエージェントに促しました。
7. **思考プロセスを導く。**拡張思考（extended thinking）モードは、Claude が可視の思考プロセスとして追加トークンを出力するもので、制御可能なスクラッチパッドとして機能します。リードエージェントは思考を使ってアプローチを計画し、タスクに合うツールを判断し、クエリの複雑さとサブエージェント数を決め、各サブエージェントの役割を定義します。私たちのテストでは、拡張思考は指示追従、推論、効率を改善しました。サブエージェントも計画を立て、ツール結果の後にインターリーブ思考を使って品質を評価し、抜けを特定し、次のクエリを磨きます。これによりサブエージェントは任意のタスクに適応する能力が高まります。
8. **並列ツール呼び出しが速度と性能を変える。**複雑な調査タスクは自然と多くの情報源を扱います。初期のエージェントは検索を逐次実行していて、これは辛いほど遅かったのです。速度のために私たちは 2 種類の並列化を導入しました。(1) リードエージェントが 3〜5 個のサブエージェントを逐次ではなく並列に立ち上げる、(2) サブエージェントが 3 つ以上のツールを並列で使う。これらの変更は、複雑クエリの調査時間を最大 90% 削減し、他のシステムより多くの情報をカバーしつつ、時間単位ではなく分単位でより多くの作業を可能にしました。

私たちのプロンプト戦略は、厳格なルールよりも良いヒューリスティックを植え付けることにフォーカスしています。熟練した人間が調査タスクにどう向かうかを研究し、その戦略をプロンプトにエンコードしました——難問を小タスクに分解する、情報源の品質を注意深く評価する、新情報に応じて検索アプローチを調整する、深さ（1 トピックを詳しく）と広さ（複数トピックを並列）をいつ使い分けるかを見極める、といった戦略です。また、エージェントが制御不能に陥らないよう明示的なガードレールを設定して副作用を先回りで緩和しました。最後に、観測可能性とテストケースによる高速な反復ループにこだわりました。

エージェントの効果的な評価

良い評価は信頼できる AI アプリケーションを作るために不可欠で、エージェントも例外ではありません。ただし、マルチエージェントシステムの評価には独自の難しさがあります。従来の評価は、AI が毎回同じステップを踏むと仮定しがちです——入力 X ならシステムは Y の経路を辿って出力 Z を出す、と。しかしマルチエー

ジェントシステムはそう動きません。同じ出発点でも、エージェントは完全に異なる正当な経路でゴールに辿り着きます。あるエージェントは 3 つのソースを検索し、別のエージェントは 10 を検索するかもしれません。同じ答えを違うツールで見つけるかもしれません。何が「正しいステップ」かを常に知っているわけではないので、事前に規定したステップを踏んだかを単に確認することはできません。代わりに必要なのは、エージェントが適切なプロセスを辿りつつ正しい結果を達成したかを判断する柔軟な評価方法です。

小サンプルですぐに評価を始める。エージェント開発の初期では、変更が劇的なインパクトを持つことが多く、熟した低位の果実が豊富にあります。プロンプトのちょっとした調整が成功率を 30% から 80% に押し上げることもあります。効果がこのくらい大きいなら、わずかなテストケースで変化が見て取れます。私たちは、実使用パターンを代表する約 20 件のクエリセットから始めました。これらをテストするだけで変更の影響がよく見えました。「eval は数百のテストケースで作らないと役に立たない」と信じて eval の作成を遅らせるチームをよく見かけますが、より徹底した eval を作れるようになるまで待つより、少数例ですぐに小規模テストを始めた方が良いのです。

うまく使えば LLM-as-judge 評価はスケールする。調査の出力は自由形式のテキストで唯一の正解があまりないため、プログラム的に評価しづらいものです。LLM は出力の採点に自然に馴染みます。私たちは、各出力をルーブリックの基準に照らして評価する LLM ジャッジを使いました。事実の正確さ（主張は情報源と一致しているか）、引用の正確さ（引用された情報源は主張と一致しているか）、網羅性（要求された全側面をカバーしているか）、情報源の質（低品質な二次情報よりも一次情報を使っているか）、ツール効率（正しいツールを妥当な回数使ったか）。各要素を評価する複数のジャッジを試しましたが、0.0〜1.0 のスコアと合否判定を出力する単一 LLM 呼び出し+単一プロンプトが最も一貫性があり、人間の判断とも最も整合しました。この方法は、eval テストケースに明確な答えがあり、LLM ジャッジにその正答性をチェックさせるだけで済む場合（例: R&D 予算トップ 3 の製薬会社を正しく列挙したか）に特に有効でした。LLM ジャッジを使うことで数百の出力をスケーラブルに評価できるようになりました。

人間の評価は自動化が見逃すものを捕まえる。人間がエージェントをテストすると、eval が見逃すエッジケースが見つかります。異常なクエリでの幻覚、システム障害、あるいは微妙な情報源選択のバイアスなどです。私たちの場合、人間テスターは、初期のエージェントが学術 PDF や個人ブログのような権威あるが低ランクのソースよりも SEO 最適化されたコンテンツファームを一貫して選んでいることに気づきました。プロンプトに情報源品質のヒューリスティックを追加することで解決しました。自動評価の世界でも、手動テストは必須のままです。

マルチエージェントシステムは、特定の実装なしに生まれる創発的な振る舞いを持ちます。たとえば、リードエージェントへの小さな変更が、サブエージェントの振る舞いを予測不能に変えることがあります。成功には、個々のエージェントの振る舞いだけでなく相互作用のパターンを理解することが必要です。したがって、これらのエージェントにとっての最良のプロンプトは、厳格な指示ではなく、労働分担、問題解決のアプローチ、労力

予算を定義した「協調のためのフレームワーク」です。これを正しくやるには、注意深いプロンプトとツール設計、堅牢なヒューリスティック、観測可能性、タイトなフィードバックループが必要です。私たちのシステムのサンプルプロンプトは、クックブックのオープンソースプロンプトを参照してください。

本番運用の信頼性とエンジニアリング課題

従来のソフトウェアでは、バグは機能を壊すか性能を劣化させるか障害を起こす程度です。エージェントシステムでは、わずかな変更が振る舞いの大きな変化にカスケードし、長時間実行でステートを維持する複雑なエージェントのコードを書くことを著しく難しくします。

エージェントはステートフルでエラーは累積する。エージェントは長時間走り、多数のツール呼び出しにまたがってステートを維持します。つまりコードを持続的に実行し、その途中で起きるエラーを扱う必要があります。効果的な緩和策がないと、軽微なシステム障害がエージェントにとって壊滅的になります。エラー発生時、最初からやり直すわけにはいきません——リスタートはコストが高く、ユーザーにとってフラストレーションの原因です。代わりに、エラーが起きた時点から再開できるシステムを作りました。また、モデルの賢さを使って問題に優雅に対処します——ツールが失敗していることをエージェントに知らせて適応させるのは、驚くほどうまく動きます。Claude の上に作られた AI エージェントの適応性と、リトライロジックや定期チェックポイントのような決定的な安全策を組み合わせています。

デバッグには新しいアプローチが必要。エージェントは動的に意思決定し、同じプロンプトでも実行間で非決定論的です。これがデバッグを難しくします。たとえばユーザーがエージェントが「明らかな情報を見つけられない」と報告しても、なぜかが見えないことがあります。悪い検索クエリを使っているのか？ 情報源が悪いのか？ ツール障害か？ 本番の完全なトレーシングを追加することで、エージェントがなぜ失敗したかを診断し、問題を体系的に修正できるようになりました。標準的な観測可能性に加えて、エージェントの意思決定パターンと相互作用構造を監視します——ユーザープライバシーのため、個別会話の中身は監視しません。このハイレベルな観測可能性により、根本原因の診断、予期せぬ振る舞いの発見、よくある失敗の修正が可能になりました。

デプロイには注意深い協調が必要。エージェントシステムは、プロンプト、ツール、実行ロジックが絡み合い、ほぼ継続的に動くステートフルなウェブです。つまり更新をデプロイするとき、エージェントはプロセスのどこにでもいる可能性があります。そのため、善意のコード変更が既存エージェントを壊さないようにする必要があります。すべてのエージェントを同時に新バージョンに更新することはできません。代わりに rainbow デプロイを使い、古いバージョンと新しいバージョンを同時に動かしながらトラフィックを徐々に移行します。

同期実行はボトルネックを生む。現状、私たちのリードエージェントはサブエージェントを同期的に実行し、サブエージェント群の完了を待ってから次へ進みます。これで協調は単純化できますが、エージェント間の情報フローにボトルネックを生みます。リードはサブエージェントを操縦できず、サブエージェント同士は協調できず、1 つのサブエージェントの検索待ちでシステム全体がブロックされかねません。非同期実行ならさらに並列化

できます——エージェントが同時に働き、必要に応じて新しいサブエージェントを作る。しかし、この非同期性は結果の協調、ステートの一貫性、サブエージェント間のエラー伝播という新たな課題を持ち込みます。モデルがより長くより複雑な調査タスクをこなせるようになれば、性能向上が複雑さを正当化すると期待しています。

結論

AI エージェントを作るとき、ラストマイルがしばしば旅全体になります。開発者のマシンで動くコードベースが信頼できる本番システムになるには、大きなエンジニアリングが必要です。エージェントシステムにおけるエラーの複合性は、従来ソフトウェアなら小さな問題でもエージェントを完全に脱線させ得るということを意味します。1 ステップの失敗が、エージェントを全く違う軌跡に送り込み、予測不能な結果を生みかねません。本稿で述べた理由すべてから、プロトタイプと本番の隔たりは予想よりしばしば広いのです。

こうした課題にもかかわらず、マルチエージェントシステムはオープンエンドな調査タスクで価値が証明されています。ユーザーからは、Claude のおかげで考えもしなかったビジネス機会を見つけられた、複雑な医療オプションを乗り越えられた、ややこしい技術バグを解決できた、自力では見つけられなかった調査のつながりを発見できて何日分もの作業を節約できた、といった声が届いています。マルチエージェントリサーチシステムは、注意深いエンジニアリング、包括的なテスト、詳細志向のプロンプトとツール設計、堅牢な運用プラクティス、そして現在のエージェント能力を深く理解した研究・プロダクト・エンジニアリング各チームの緊密な協業があれば、スケールする信頼できる動作が可能です。すでにこうしたシステムが、人々が複雑な問題を解く方法を変えつつあるのを私たちは目の当たりにしています。

Clio 埋め込みプロットで見るリサーチ機能の主な用途。上位カテゴリは、専門領域を横断したソフトウェアシステムの開発（10%）、プロフェッショナルおよび技術コンテンツの開発と最適化（8%）、ビジネス成長と収益創出戦略の開発（8%）、学術研究と教育資料開発の支援（7%）、人物・場所・組織に関する情報の調査と検証（5%）。

謝辞

執筆: Jeremy Hadfield、Barry Zhang、Kenneth Lien、Florian Scholz、Jeremy Fox、Daniel Ford。本作業は、リサーチ機能を実現した Anthropic の複数チームの集合的な努力を反映しています。この複雑なマルチエージェントシステムを本番に持っていった Anthropic のアプリエンジニアリングチームに特別な感謝を。素晴らしいフィードバックをくれた初期ユーザーにも感謝します。

付録

マルチエージェントシステム向けの追加のヒントをいくつか。

多ターンでステートを変化させるエージェントの終端状態評価。マルチターン会話でステートを変化させるエージェントの評価には独自の課題があります。読み取り専用の調査タスクと違い、各アクションが後続ステップの環境を変えるため、従来の評価方法が扱いづらい依存関係を生みます。私たちはターン単位の分析ではなく、終端状態評価に焦点を当てることで成功しました。特定のプロセスを辿ったかを判断する代わりに、正しい最終状態を達成したかを評価するのです。このアプローチは、エージェントが同じゴールに至る代替経路を見つけうることを認めつつ、意図した結果を届けることを保証します。複雑なワークフローでは、評価を、特定のステート変化が起きるべき離散チェックポイントに分け、すべての中間ステップを検証しようとしない方が良いです。

長期会話の管理。本番エージェントはしばしば数百ターンにまたがる会話を行い、注意深い文脈管理戦略を必要とします。会話が延びるにつれ、標準コンテキストウィンドウでは不十分になり、知的な圧縮とメモリ機構が必要になります。私たちは、エージェントが完了した作業段階を要約し、新しいタスクに進む前に不可欠情報を外部メモリに保存するパターンを実装しました。コンテキスト上限が近づくとき、エージェントは慎重なハンドオフで連続性を保ちながら、クリーンなコンテキストを持つ新しいサブエージェントを作れます。さらに、コンテキスト上限で以前の作業を失う代わりに、調査計画のような保存された文脈をメモリから取り出すことができます。この分散アプローチは、文脈オーバーフローを防ぎつつ、延長された対話の一貫性を保ちます。

「伝言ゲーム」を最小化するための、サブエージェントのファイルシステムへの出力。特定タイプの結果については、サブエージェントの出力がメインコーディネーターをバイパスでき、忠実度と性能の両方が改善します。サブエージェントがすべてをリードエージェント経由で伝える代わりに、専門エージェントが独立して永続する出力を作成できるアーティファクトシステムを実装しましょう。サブエージェントはツールを呼んで外部システムに作業を保存し、軽量な参照をコーディネーターに返します。これは多段階処理での情報損失を防ぎ、大きな出力を会話履歴経由でコピーするトークンオーバーヘッドを減らします。このパターンは、コード、レポート、データ可視化のような構造化された出力で特によく効きます——サブエージェントの専門プロンプトは、汎用のコーディネーターを介したフィルタリングより良い結果を生みます。