10

AI エージェントのための効果的なコンテキストエンジニアリング

— Effective context engineering for AI agents —

公開日	2025-09-29
原題	Effective context engineering for AI agents
著者	Anthropic Engineering Team
原文	https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents
翻訳	Claude（機械翻訳／Anthropic）
編集	2026-04-09

AI エージェントのための効果的なコンテキストエンジニアリング

応用 AI の世界で数年にわたり注目されてきた「プロンプトエンジニアリング」に代わって、新しい用語が前景に出てきました——**コンテキストエンジニアリング** です。言語モデルで構築することは、プロンプトに最適な言葉やフレーズを見つけることから、「モデルに望ましい振る舞いをさせる可能性が最も高いコンテキスト構成はどんなものか?」というより広い問いに答えることへと移りつつあります。

コンテキスト とは、大規模言語モデル(LLM)からサンプリングするときに含められるトークン集合のことです。ここでの **エンジニアリング** 問題とは、望む結果を一貫して得るために、LLM 固有の制約に対してそれらのトークンの有用性を最適化することです。LLM を効果的に取り回すには、*コンテキストで考える* こと——つまり任意の時点で LLM が利用できる全体的な状態と、その状態がどんな振る舞いを生み出しうるかを考えること——がしばしば必要になります。

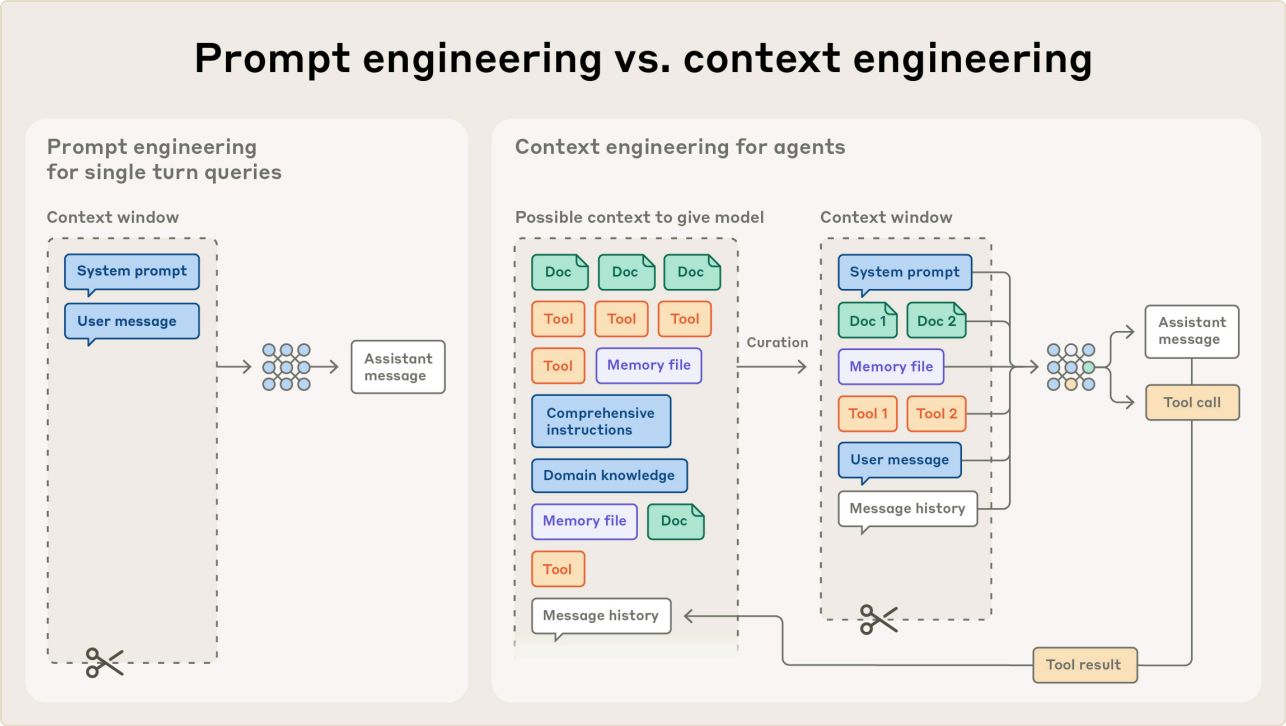
本稿では、コンテキストエンジニアリングという新しい芸術を探り、操作可能で効果的なエージェントを作るための洗練されたメンタルモデルを提示します。

コンテキストエンジニアリング vs プロンプトエンジニアリング

Anthropic では、コンテキストエンジニアリングをプロンプトエンジニアリングの自然な進化形と見なしています。プロンプトエンジニアリングは、最適な結果を得るための LLM への指示の書き方・組み立て方を指す言葉です(概観と有用な戦略はドキュメントを参照)。一方、**コンテキストエンジニアリング** は、LLM 推論中に最適なトークン(情報)集合をキュレーション・維持するための戦略の集合を指し、プロンプトの外で紛れ込むかもしれないあらゆる情報も含みます。

LLM エンジニアリングの初期には、プロンプティングが AI エンジニアリング作業の最大要素でした。日常的なチャット以外のユースケースの多くは、ワンショットの分類やテキスト生成タスク向けに最適化されたプロンプトを必要としていたからです。名前のとおり、プロンプトエンジニアリングの主眼は、特にシステムプロンプトを含めた効果的なプロンプトの書き方にありました。しかし、複数ターンの推論とより長い時間軸で動作する、より有能なエージェントを設計する段階に移ると、コンテキスト状態全体(システム指示、ツール、MCP、外部データ、メッセージ履歴など)を管理する戦略が必要になります。

ループで動くエージェントは、次の推論ターンに関連する *かもしれない* データを次々と生成していき、この情報は循環的に洗練する必要があります。コンテキストエンジニアリングとは、絶えず進化する可能な情報の宇宙から、限られたコンテキストウィンドウに入れるものをキュレーションする芸術と科学です。



プロンプトを書くという離散的な作業と対照的に、コンテキストエンジニアリングは反復的で、モデルに何を渡すかを決めるたびにキュレーション段階が走る。

有能なエージェントを作るうえでコンテキストエンジニアリングが重要な理由

LLM はますます大量のデータを扱える速度と能力を持つ一方で、人間と同じく、ある時点でフォーカスを失ったり混乱したりすることが観察されています。needle-in-a-haystack 型のベンチマーク研究は、context rot（コンテキストロット）という概念を明らかにしました——コンテキストウィンドウのトークン数が増えるにつれ、モデルがその中から情報を正確に想起する能力は低下するのです。

劣化の度合いはモデルによって異なりますが、この性質はすべてのモデルに共通して現れます。したがってコンテキストは、限界収益が逓減する有限リソースとして扱う必要があります。人間が 限られたワーキングメモリ容量 を持つのと同じく、LLM は大量のコンテキストを解釈する際に引き出す「注意予算」を持っています。新しいトークンを加えるたびにこの予算が消費され、LLM に利用可能なトークンの注意深いキュレーションの必要性が増します。

この注意の希少性は、LLM のアーキテクチャ的制約に由来します。LLM は transformer アーキテクチャ に基づいており、すべてのトークンがコンテキスト全体で他のすべてのトークンに注意を向けられるようになっています。これは n 個のトークンに対して n^2 のペア関係を生みます。

コンテキスト長が増えると、これらのペア関係を捉えるモデルの能力は薄く伸ばされ、コンテキストサイズと注意のフォーカスの間に自然な緊張が生じます。さらに、モデルは通常長い系列より短い系列の方がずっと多い訓練データ分布から注意パターンを獲得します。つまりモデルは、コンテキスト全域にわたる依存関係についての経験が少なく、専門化されたパラメータも少ないのです。

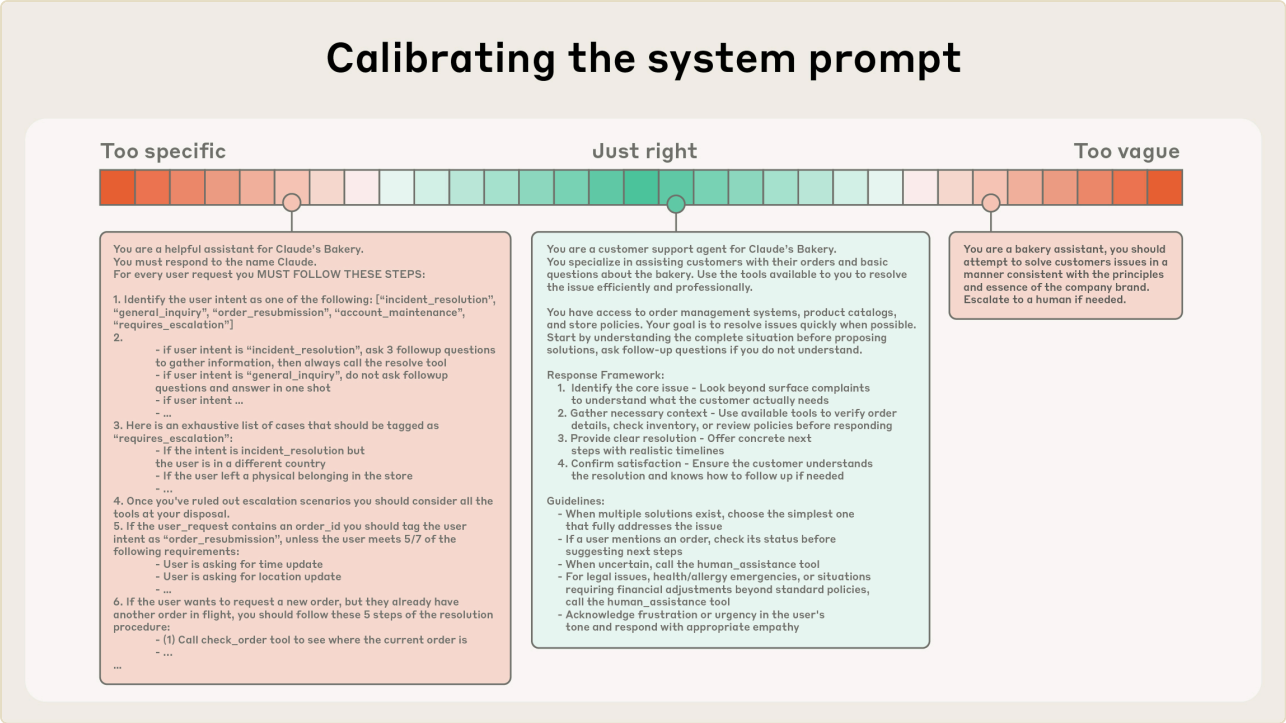
位置エンコーディングの補間のような手法は、元々訓練された短いコンテキストに適合させることで長い系列を扱えるようにしますが、トークン位置の理解にはある程度の劣化が伴います。これらの要因は、性能の「崖」ではなく「勾配」を作り出します——モデルは長いコンテキストでも高い能力を保ちますが、情報検索や長距離推論に関しては短いコンテキストの性能より精度が落ちることがあります。

これらの現実を踏まえると、有能なエージェントを作るには思慮深いコンテキストエンジニアリングが不可欠です。

効果的なコンテキストの構成

LLM が有限の注意予算で制約されている以上、*良い* コンテキストエンジニアリングとは、望ましい結果が得られる確率を最大化する、*最小の* 高信号トークン集合を見つけることを意味します。これを実践するのは言うほど簡単ではありませんが、次節ではこの指針がコンテキストの各要素ごとに実際に何を意味するかを示します。

システムプロンプト は極めて明確であるべきで、シンプルで直接的な言葉で、エージェントに対して *正しい「高度（altitude）」* で発想を提示すべきです。正しい高度とは、2 つのよくある失敗モードの間にあるゴルディロックス帯です。一方の極端では、エンジニアが厳密なエージェント挙動を引き出すために、複雑で脆いロジックをプロンプトにハードコードします。このアプローチは脆弱さを生み、時間とともに保守複雑性を増します。もう一方の極端では、エンジニアが曖昧で高レベルなガイダンスを与えてしまい、望ましい出力への具体的なシグナルを LLM に与えられなかったり、共有文脈を誤って前提にしたりします。最適な高度はバランスを取ります——効果的に振る舞いを導けるほど具体的でありつつ、モデルに行動を導く強いヒューリスティックを与えられるほど柔軟であること。



スペクトルの一端には脆いハードコードされた if-else プロンプトがあり、もう一端には過度に汎用的か共有文脈を誤って前提にしたプロンプトがある。

プロンプトは、`<background_information>`、`<instructions>`、`## Tool guidance`、`## Output description` のような明確な区画に分け、XML タグや Markdown 見出しでこれらを区切ることを勧めます。ただしモデルが有能になるにつれ、プロンプトの正確な書式の重要度は下がっている可能性があります。

システムプロンプトの構造をどうするにせよ、期待する振る舞いを完全に記述する最小限の情報集合を目指すべきです（最小限とは必ずしも短いことではありません——望ましい振る舞いを守ってもらうには、事前に十分な情報をエージェントに与える必要があります）。まず入手可能な最良のモデルで最小プロンプトを試し、自分のタスクでどう動くかを観察し、初期テストで見つかった失敗モードに基づいて明確な指示と例を追加していくのが良い出発点です。

ツール は、エージェントが環境と相互作用し、作業を進めながら追加の文脈を取り込むためのものです。ツールはエージェントと情報／アクション空間の契約を定義するので、ツールが効率を促進することが極めて重要です——トークン効率の良い情報を返すこと、そして効率的なエージェント行動を促すこと、の両面で。

Writing effective tools for agents — with agents では、LLM によく理解され、機能の重複が最小限のツールの作り方について議論しました。設計の良いコードベースの関数と同様に、ツールは自己完結し、エラーに強く、意図する使い方に関して極めて明確であるべきです。入力パラメータも同様に説明的・明確で、モデルの持ち味に寄り添ったものであるべきです。

よく見かける失敗モードの 1 つは、機能を詰め込みすぎたり、どのツールを使うべきかが曖昧になる意思決定点を生むような、肥大化したツールセットです。人間のエンジニアがある状況でどのツールを使うべきか断言できないなら、AI エージェントがそれ以上にうまくやれると期待するのは無理です。後述するように、エージェントのための「最小有効ツールセット」をキュレーションすることは、長い対話にわたるコンテキストの信頼できる保守と剪定にもつながります。

例示（few-shot プロンプティング）は、私たちも依然として強く勧めるベストプラクティスです。しかしチームはしばしば、特定のタスクで LLM が従うべきあらゆるルールを明示化しようとして、プロンプトにエッジケースのリストを詰め込んでしまいます。これはお勧めしません。代わりに、エージェントの期待される振る舞いを効果的に描写する、多様で正典的な例のセットをキュレーションすべきです。LLM にとって、例は千の言葉に値する「絵」です。

コンテキストの各要素（システムプロンプト、ツール、例、メッセージ履歴など）を横断する全体方針は、思慮深く、情報的でありながら引き締まった状態を保つことです。次は、実行時に動的にコンテキストを取得する話に移ります。

コンテキスト取得とエージェント型検索

Building effective AI agents では、LLM ベースのワークフローとエージェントの違いを強調しました。あの投稿以来、私たちはエージェントに関するシンプルな定義に近づいてきています——**ループの中で自律的にツールを使う LLM**、です。

顧客との協働の中で、業界がこのシンプルなパラダイムに収束していくのを目にしてきました。基盤モデルが有能になるにつれて、エージェントの自律レベルもスケールできます——賢いモデルがあれば、エージェントは微妙な問題空間を独立に進み、エラーから回復できます。

エンジニアがエージェント向けのコンテキストをどう設計するかについても、シフトが起きています。今日、多くの AI ネイティブアプリケーションは、推論前の何らかの埋め込みベースの検索を使って、エージェントが推論する重要な文脈を事前に浮かび上がらせています。業界がよりエージェント型のアプローチに移るにつれ、こうした検索システムに「just in time」コンテキスト戦略を組み合わせるチームが増えています。

関連データを前もって一括前処理するのではなく、「just in time」アプローチで作られたエージェントは軽量な識別子（ファイルパス、保存済みクエリ、Web リンクなど）を保持し、これらの参照をツール経由で実行時に動的にコンテキストに読み込みます。Anthropic のエージェント型コーディングソリューション Claude Code は、大きなデータベースに対する複雑なデータ分析をこのアプローチで行っています。モデルは的を絞ったクエリを書き、結果を保存し、`head` や `tail` のような Bash コマンドを活かして、全データオブジェクトをコン

テキストに読み込むことなく大量データを分析できます。このアプローチは人間の認知を映しています——私たちは通常、コーパス全体を暗記するのではなく、ファイルシステム、受信箱、ブックマークといった外部の整理・索引システムを導入し、関連情報を必要に応じて取り出します。

ストレージ効率を超えて、これらの参照のメタデータは、明示的であれ直感的であれ、振る舞いを効率的に洗練するメカニズムを提供します。ファイルシステム上で動作するエージェントにとって、`tests` フォルダ内の `test_utils.py` と、`src/core_logic/` にある同じ名前のファイルは異なる目的を示唆します。フォルダ階層、命名規則、タイムスタンプは、情報をどう・いつ使うかを人間にもエージェントにも理解させる重要なシグナルを提供します。

エージェントに自律的にデータをナビゲート・取得させることは、漸進的な開示（progressive disclosure）も可能にします——つまり、探索を通じて関連文脈を段階的に発見できるのです。各相互作用が次の決定を知らせる文脈を生みます。ファイルサイズは複雑さを示唆し、命名規則は目的をほのめかし、タイムスタンプは関連度の代理になります。エージェントは理解を層ごとに組み上げ、ワーキングメモリに必要なものだけを保持し、ノート取り戦略で追加の永続性を活かせます。こうした自己管理型コンテキストウィンドウは、網羅的だが関係ないかもしれない情報に溺れるのではなく、関連部分集合にフォーカスし続けさせます。

もちろんトレードオフがあります。実行時の探索は事前計算されたデータの取得より遅いです。それだけでなく、LLM が情報ランドスケープを効果的に navigating するための適切なツールとヒューリスティックを持たせるには、意見を持って思慮深いエンジニアリングが必要です。適切なガイダンスがないと、エージェントはツールを誤用し、袋小路を追い、重要な情報を特定し損ねて文脈を無駄にしかねません。

特定の設定では、最も効果的なエージェントはハイブリッド戦略を採るかもしれません——速度のために一部のデータは前もって取得し、必要に応じて自律探索を追加する、というものです。「適切な」自律レベルの境界はタスクに依存します。Claude Code はこのハイブリッドモデルを採用するエージェントです——CLAUDE.md ファイルは素朴に前もってコンテキストに落とし込まれる一方、`glob` や `grep` のようなプリミティブで環境をナビゲートし、ファイルをジャストインタイムに取得できるので、古い索引と複雑な構文ツリーの問題を事実上回避しています。

ハイブリッド戦略は、法律や金融業務のような変動の少ない文脈により向いているかもしれません。モデルの能力が向上するにつれ、エージェント設計は賢いモデルに賢く行動させる方向に進み、人間によるキュレーションは徐々に減っていくでしょう。この分野の急速な進歩ペースを考えると、「動く最もシンプルなことをする」は、Claude の上でエージェントを作るチームへの最良のアドバイスであり続けるでしょう。

長時間タスクのためのコンテキストエンジニアリング

長時間タスクでは、アクションの連なりの中でエージェントが一貫性・文脈・目標志向の振る舞いを維持することが必要で、そのトークン数は LLM のコンテキストウィンドウを超えます。大規模コードベースの移行や包括的な調査プロジェクトのように、数十分から数時間の連続作業にわたるタスクでは、エージェントはコンテキス

トウィンドウのサイズ制約を回避する特別な手法が必要です。

より大きなコンテキストウィンドウを待つのは一見明白な戦術に思えるかもしれません。しかし当面の間、あらゆるサイズのコンテキストウィンドウは、コンテキスト汚染と情報の関連性の問題に晒されるでしょう——少なくとも最強のエージェント性能が望まれる状況では。拡張された時間軸でエージェントが効果的に働けるように、私たちはこうしたコンテキスト汚染の制約に直接対処するいくつかの手法を開発してきました——**圧縮**（**compaction**）、**構造化ノート取り**（**structured note-taking**）、**マルチエージェントアーキテクチャ** です。

圧縮（Compaction）

圧縮とは、コンテキストウィンドウの上限に近づいた会話を要約し、その要約で新しいコンテキストウィンドウを再初期化する実践です。圧縮は通常、長期的な一貫性を高めるためのコンテキストエンジニアリング上の最初のレバーとして働きます。本質的には、コンテキストウィンドウの内容を高忠実度で蒸留し、最小限の性能劣化でエージェントが継続できるようにするものです。

たとえば Claude Code では、メッセージ履歴をモデルに渡して、最も重要な詳細を要約・圧縮することで実装しています。モデルはアーキテクチャ上の決定、未解決のバグ、実装の詳細を保ちながら、冗長なツール出力やメッセージを捨てます。エージェントはその圧縮されたコンテキストに、直近アクセスした 5 つのファイルを加えて継続できます。ユーザーはコンテキストウィンドウの制約を気にせず継続性を得られます。

圧縮の芸術は、何を残し何を捨てるかの選別にあります。積極的すぎる圧縮は、後になってから重要性が判明するような微妙だが決定的な文脈を失わせかねません。圧縮システムを実装するエンジニアには、複雑なエージェントのトレース上で圧縮プロンプトを注意深くチューニングすることを勧めます。まず再現率（recall）を最大化して、圧縮プロンプトがトレースから関連情報をすべて捕まえることを保証し、その後精度を上げるために余計な内容を除去していきます。

低い枝に成る余計な内容の例は、ツール呼び出しと結果のクリアです——ツールがメッセージ履歴の深いところで既に呼ばれているなら、エージェントが生の結果をもう一度見る必要があるでしょうか？ 最も安全で軽い圧縮の形の 1 つがツール結果のクリアで、Claude Developer Platform の機能として最近リリースされました。

構造化ノート取り（Structured note-taking）

構造化ノート取り、別名「エージェント型メモリ」は、エージェントがコンテキストウィンドウの外のメモリに永続化されるノートを定期的に書く手法です。これらのノートは後でコンテキストウィンドウに引き戻されます。

この戦略は最小限のオーバーヘッドで永続的メモリを提供します。Claude Code が to-do リストを作るように、あるいは独自エージェントが `NOTES.md` を維持するように、このシンプルなパターンは複雑なタスクにわたる進捗の追跡を可能にし、さもなくば数十のツール呼び出しにわたって失われていたであろう重要な文脈と依存関係を保ちます。

Claude がポケモンをプレイする のは、コーディング以外のドメインでメモリがエージェント能力をどう変えるかを示しています。エージェントは数千のゲームステップにわたって正確な集計を維持します——「直近 1,234 ステップ、ピカチュウを 1 番道路で訓練中。目標レベル 10 に対して 8 レベル上げた」のような目標を追跡します。メモリ構造について何も指示されずに、探索した地域のマップを作り、解錠した主要な実績を記憶し、異なる相手にどの攻撃が効くかを学ぶのに役立つ戦闘戦略のメモを維持します。

コンテキストリセット後、エージェントは自分のノートを読み、数時間にわたる訓練シーケンスやダンジョン探索を継続します。この要約ステップを挟んだ一貫性は、LLM のコンテキストウィンドウだけに情報を保持しておくなら不可能な長時間戦略を可能にします。

Sonnet 4.5 ローンチの一環で、私たちは Claude Developer Platform にファイルベースでコンテキストウィンドウの外に情報を保存・参照しやすくするメモリツールのパブリックベータをリリースしました。これによりエージェントは、時間をかけて知識ベースを構築し、セッションをまたいでプロジェクト状態を維持し、すべてをコンテキストに入れずに以前の作業を参照できます。

サブエージェントアーキテクチャ

サブエージェントアーキテクチャは、コンテキスト制約を回避するもう 1 つの方法です。1 つのエージェントがプロジェクト全体にわたるステートを維持しようとする代わりに、専門化されたサブエージェントがきれいなコンテキストウィンドウで集中したタスクを担当できます。メインのエージェントは高レベル計画で協調し、サブエージェントは深い技術作業を行ったり、ツールを使って関連情報を見つけたりします。各サブエージェントは数万トークン以上を使って広範に探索するかもしれませんが、その作業の凝縮した蒸留要約（しばしば 1,000〜2,000 トークン）だけを返します。

このアプローチは関心事の明確な分離を達成します——詳細な検索文脈はサブエージェント内に孤立したままで、リードエージェントは結果の統合と分析にフォーカスできます。How we built our multi-agent research system で議論したこのパターンは、複雑な調査タスクで単体エージェントシステムに対する大きな改善を示しました。

これらのアプローチの選択はタスクの性質に依存します。例:

- **圧縮**は、広範なやりとりを要するタスクで会話の流れを維持する。
- **ノート取り**は、マイルストーンが明確な反復的開発で光る。
- **マルチエージェントアーキテクチャ**は、並列探索が報いをもたらす複雑な調査・分析を扱う。

モデルが改善し続けても、拡張された対話にわたる一貫性の維持は、より効果的なエージェント作りの中心的な課題であり続けるでしょう。

結論

コンテキストエンジニアリングは、LLM で構築する私たちの方法の根本的なシフトを表します。モデルが有能になるにつれ、課題は完璧なプロンプトを作ることではなく、各ステップでモデルの限られた注意予算に何を入れるかを思慮深くキュレーションすることです。長時間タスク向けに圧縮を実装するにせよ、トークン効率の良いツールを設計するにせよ、エージェントが環境を just-in-time に探索できるようにするにせよ、指針は同じです——望む結果が得られる確率を最大化する、最小の高信号トークン集合を見つけることです。

本稿で示した手法はモデルが改善するにつれて進化し続けます。賢いモデルはより少ない規範的エンジニアリングで済み、エージェントがより自律的に動けるようになることは既に見え始めています。しかし、能力がスケールしても、コンテキストを貴重で有限なリソースとして扱うことは、信頼できる効果的なエージェントを作る中心であり続けるでしょう。

今すぐ Claude Developer Platform でコンテキストエンジニアリングを始めましょう。有用なヒントとベストプラクティスは、memory と context management クックブックで確認できます。

謝辞

Anthropic Applied AI チーム: Prithvi Rajasekaran、Ethan Dixon、Carly Ryan、Jeremy Hadfield が執筆。チームメンバーの Rafi Ayub、Hannah Moran、Cal Rueb、Connor Jennings が貢献。Molly Vorwerck、Stuart Ritchie、Maggie Vo にはサポートで特別に感謝します。