15

AI エージェントの eval を解きほぐす

— Demystifying evals for AI agents —

公開日	2026-01-09
原題	Demystifying evals for AI agents
著者	Anthropic Engineering Team
原文	https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents
翻訳	Claude（機械翻訳／Anthropic）
編集	2026-04-09

AI エージェントの eval を解きほぐす

はじめに

良い eval は、チームがより自信を持って AI エージェントをリリースする助けになります。eval がなければ、反応的なループに陥りがちです——本番でしか問題に気づけず、1 つの失敗を直すと別の失敗が生まれるという状態です。eval は、ユーザーに影響を与える前に問題と振る舞いの変化を可視化します。その価値はエージェントのライフサイクルにわたって複利で積み上がります。

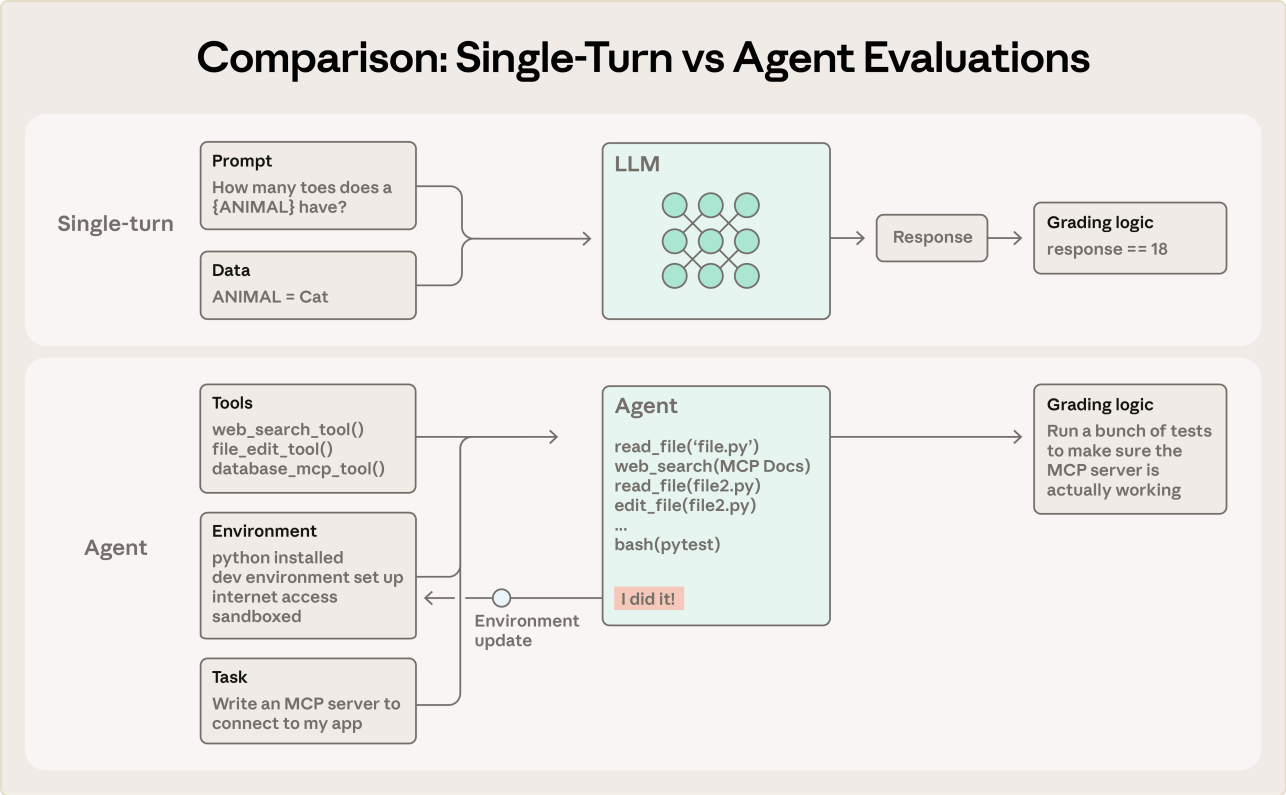
Building effective agents で述べたように、エージェントは多数のターンにわたって動作します——ツールを呼び、状態を変え、中間結果に応じて適応します。AI エージェントを有用にするこれらの能力——自律性、知能、柔軟性——が、同時に評価を難しくもしています。

社内の業務とエージェント開発の最前線にいる顧客との協業を通じて、私たちはエージェント向けにより厳密で有用な eval を設計する方法を学んできました。本稿では、多種多様なエージェントアーキテクチャとユースケースで実際のデプロイに対して機能した知見を共有します。

評価の構造

evaluation（「**eval**」）は AI システム向けのテストです——AI に入力を与え、その出力に採点ロジックを適用して成功を測ります。本稿では、実ユーザーなしに開発中に実行できる **自動 eval** にフォーカスします。

シングルターン評価 はシンプルです——プロンプト、応答、採点ロジック。以前の LLM では、シングルターンの非エージェント eval が主な評価方法でした。AI の能力が進むにつれ、**マルチターン評価** が一般的になってきました。



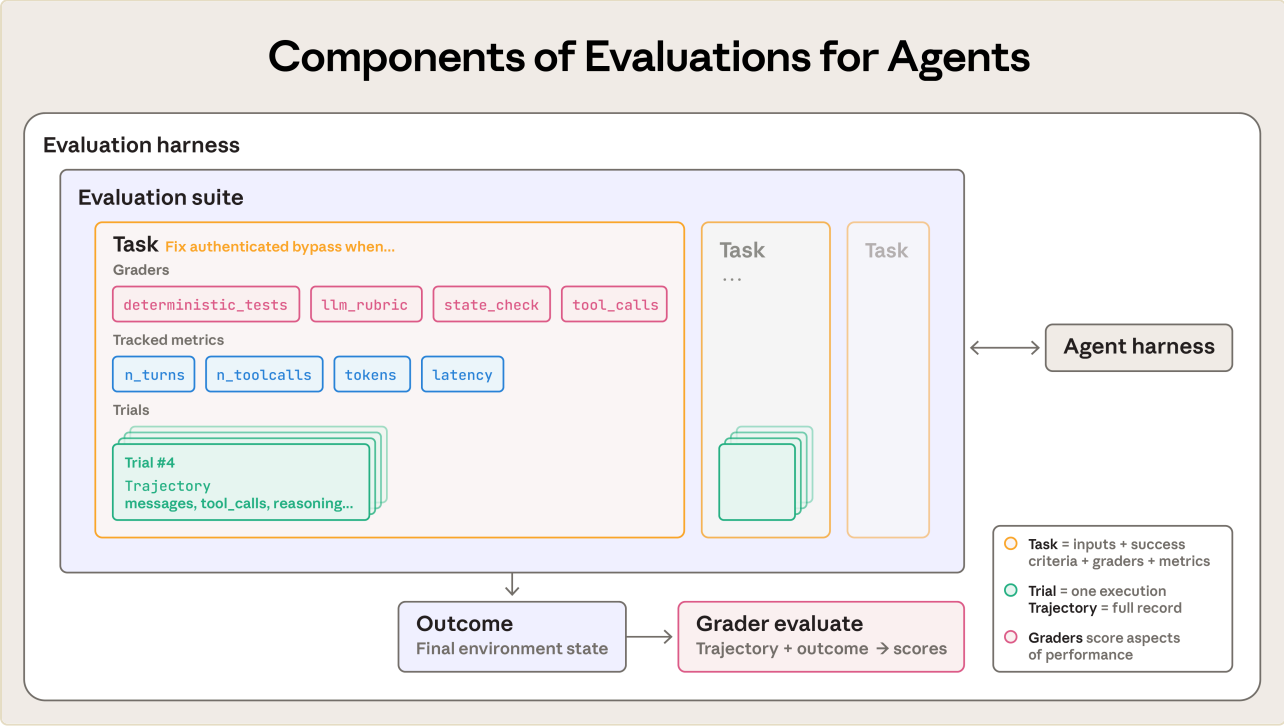
単純な eval では、エージェントがプロンプトを処理し、grader が出力が期待に合致するかをチェックする。より複雑なマルチターン eval では、コーディングエージェントがツール、タスク(この例では MCP サーバーの構築)、環境を受け取り、「エージェントループ」(ツール呼び出しと推論)を実行し、実装で環境を更新する。採点はユニットテストで動作する MCP サーバーを検証する。

エージェント評価 はさらに複雑です。エージェントは多数のターンでツールを使い、環境のステートを変えつつ進むにつれて適応します——つまりミスは伝播し複合します。最先端モデルは静的 eval の限界を超える創造的な解を見つけることもあります。たとえば Opus 4.5 は τ^2-bench のフライト予約問題で、ポリシーの抜け穴を発見して解きました。書かれていた通りには「失敗」しましたが、実際にはユーザーにとってより良い解を思いついたのです。

エージェント評価を構築するとき、私たちは次の定義を使います。

- **タスク**(**問題**、**テストケース**とも): 定義された入力と成功基準を持つ単一のテスト。
- 各タスクへの試行を **トライアル** と呼ぶ。モデル出力は実行ごとに変動するため、より一貫性のある結果を得るために複数トライアルを走らせる。
- **grader**(**採点者**): エージェントの性能の何らかの側面をスコアリングするロジック。1 タスクに複数の grader を持つことができ、各 grader は複数のアサーション(**チェック** と呼ぶこともある)を含む。
- **トランスクリプト**(**trace** や **trajectory** とも): トライアルの完全な記録——出力、ツール呼び出し、推論、中間結果、その他の相互作用を含む。Anthropic API では、これは eval 実行終了時の完全なメッセージ配列で、評価中の API 呼び出しと返された応答をすべて含む。

- **outcome**: トライアル終了時の環境の最終状態。フライト予約エージェントはトランスクリプトの最後に「フライトを予約しました」と言うかもしれないが、outcome は環境の SQL データベースに予約が存在するかどうか。
- **evaluation harness**: eval をエンドツーエンドで動かすインフラ。指示とツールを提供し、タスクを並行実行し、全ステップを記録し、出力を採点し、結果を集約する。
- **agent harness（あるいは scaffold）**: モデルがエージェントとして振る舞えるようにするシステム——入力を処理し、ツール呼び出しをオーケストレーションし、結果を返す。「エージェント」を評価するとき、私たちはハーネス **と** モデルの共同動作を評価している。たとえば Claude Code は柔軟なエージェントハーネスで、そのコアプリミティブを Agent SDK 経由で長時間エージェントハーネスの構築に使った。
- **evaluation suite**: 特定の能力や振る舞いを測るために設計されたタスクのコレクション。スイート内のタスクは通常、広い目標を共有する。たとえばカスタマーサポート eval スイートは返金、キャンセル、エスカレーションをテストするかもしれない。



エージェント向け評価のコンポーネント。

なぜ評価を作るのか

チームがエージェントを作り始めるとき、手動テスト、ドッグフーディング、直感の組み合わせで驚くほど遠くまで進めます。より厳密な評価は、リリースを遅らせるオーバーヘッドに見えるかもしれません。しかし、初期のプロトタイピング段階を超え、エージェントが本番に入ってスケールし始めると、eval なしで構築することは破綻し始めます。

破綻点はしばしば、ユーザーが「変更後にエージェントの性能が落ちた気がする」と報告し、チームが「手探り」で推測と確認しかできなくなったときに訪れます。eval なしでは、デバッグは反応的になります——苦情を待ち、手動で再現し、バグを直し、他に壊れていないことを祈る。実際の回帰とノイズを区別できず、リリース前に数百のシナリオで変更を自動テストできず、改善を測れません。

私たちはこの進展を何度も見てきました。たとえば Claude Code は、Anthropic 従業員と外部ユーザーからのフィードバックに基づいた高速反復で始まりました。その後、まず簡潔さやファイル編集のような狭い領域、次にオーバーエンジニアリングのようなより複雑な振る舞いに eval を追加しました。これらの eval は問題の特定、改善の導き、リサーチと製品の協業の焦点化に役立ちました。本番モニタリング、A/B テスト、ユーザーリサーチなどと組み合わさって、eval はスケールする Claude Code を改善し続けるシグナルを提供します。

eval を書くことは、エージェントライフサイクルのどの段階でも有用です。初期には、eval はプロダクトチームに「成功とは何か」を指定させ、後には一貫した品質基準を維持する助けになります。

Descript のエージェントはユーザーが動画を編集する手助けをするので、彼らは成功する編集ワークフローの 3 つの次元——「壊さない」「指示通りにする」「うまくやる」——を軸に eval を作りました。手動採点から、プロダクトチームが定義した基準と定期的な人間キャリブレーションを持つ LLM grader へと進化させ、今では品質ベンチマークと回帰テストの 2 つのスイートを定期的に走らせています。Bolt AI チームは、すでに広く使われるエージェントを持った後に eval の構築を始めました。3 ヶ月で、エージェントを走らせて静的解析で出力を採点し、アプリをテストするブラウザエージェントを使い、指示追従のような振る舞いに LLM ジャッジを採用する eval システムを構築しました。

開発開始時に eval を作るチームもあれば、スケール時に eval がエージェント改善のボトルネックになってから追加するチームもあります。eval は、エージェント開発の開始時に期待される振る舞いを明示的にエンコードするのに特に有用です。同じ初期仕様書を読んだ 2 人のエンジニアは、AI がエッジケースをどう扱うべきかについて異なる解釈に辿り着くかもしれません。eval スイートはこの曖昧さを解消します。いつ作っても、eval は開発を加速させます。

eval は新しいモデルをどれだけ早く採用できるかにも影響します。より強力なモデルが出たとき、eval を持たないチームは数週間のテストに直面する一方、eval を持つ競合はすぐにモデルの強みを判定し、プロンプトを調整し、数日で移行できます。

eval が存在すれば、ベースラインと回帰テストがおまけでついてきます——レイテンシ、トークン使用量、タスクあたりコスト、エラー率を、静的なタスクバンク上で追跡できます。eval は製品チームとリサーチチームの間で最も帯域の広いコミュニケーションチャネルにもなり、リサーチャーが最適化すべき指標を定義します。eval の恩恵は回帰と改善の追跡を超えて広範です。コストが前払いで見える一方で、利益は後になって積み上がるため、その複利的価値は見落とされやすいのです。

AI エージェントをどう評価するか

今日スケールでデプロイされているエージェントには、コーディングエージェント、リサーチエージェント、コンピュータ利用エージェント、会話型エージェントなど、いくつかの典型タイプがあります。各タイプは多種多様な業界でデプロイされますが、類似の手法で評価できます。評価をゼロから発明する必要はありません。以下のセクションではエージェントタイプごとに実証済みの手法を解説します。これらを基礎として、自分のドメインに拡張してください。

エージェント向け grader の種類

エージェント評価は通常、3 種類の grader を組み合わせます——**コードベース**、**モデルベース**、**人間**。各 grader はトランスクリプトまたは outcome の一部を評価します。効果的な評価設計の肝は、仕事に合った grader を選ぶことです。

コードベースの grader

手法	長所	短所
• 文字列マッチチェック（厳密、正規表現、ファジーなど） • バイナリテスト（fail-to-pass、pass-to-pass） • 静的解析（lint、型、セキュリティ） • outcome 検証 • ツール呼び出し検証（使ったツール、パラメータ） • トランスクリプト分析（ターン数、トークン使用量）	• 速い • 安い • 客観的 • 再現可能 • デバッグしやすい • 特定条件の検証	• 期待パターンに厳密一致しない有効な変形に脆い • ニュアンスに欠ける • 主観的なタスクには限定的

モデルベースの grader

手法	長所	短所
- ルーブリックベースのスコアリング - 自然言語アサーション - ペアワイズ比較 - 参照ベース評価 - マルチジャッジ合意	- 柔軟 - スケーラブル - ニュアンスを捉える - オープンエンドなタスクを扱う - 自由形式の出力を扱う	- 非決定論的 - コードより高コスト - 精度のために人間 grader とのキャリブレーションが必要

人間の grader

手法	長所	短所
- SME レビュー - クラウドソース判定 - スポットチェックサンプリング - A/B テスト - アノテーター間一致率	- ゴールドスタンダードの品質 - 専門家ユーザーの判断に合致 - モデルベース grader のキャリブレーションに使える	- 高コスト - 遅い - 大規模に人間専門家へのアクセスを要することが多い

各タスクのスコアリングは、重み付け（grader スコアの合計がしきい値に達する必要がある）、バイナリ（すべての grader が通る必要がある）、ハイブリッドのいずれかにできます。

capability eval vs. 回帰 eval

capability（「**品質**」）**eval** は「このエージェントは何をうまくできるか？」を問います。低い合格率から始め、エージェントが苦戦するタスクを狙い、チームに登るべき丘を与えるべきです。

回帰 eval は「エージェントはまだ以前扱えたタスクをこなせるか？」を問い、ほぼ 100% の合格率を持つべきです。これは後退から守るもので、スコアの低下は何かが壊れ改善が必要だというシグナルです。チームが capability eval で丘を登るにつれ、回帰 eval も走らせ、変更が他の箇所で問題を起こしていないことを確認することが重要です。

エージェントがローンチ・最適化された後、高合格率の capability eval は「卒業」して回帰スイートとなり、継続的に走らせてドリフトを捕まえます。「そもそもこれができるか？」を測っていたタスクは、「まだ確実にこれができるか？」を測ります。

コーディングエージェントの評価

コーディングエージェント はコードを書き、テストし、デバッグします——コードベースをナビゲートし、人間の開発者と同じようにコマンドを実行します。現代のコーディングエージェントの効果的な eval は通常、仕様が明確なタスク、安定したテスト環境、生成コードの徹底したテストに依存します。

決定的な grader はコーディングエージェントに自然です——ソフトウェアは一般に評価しやすいからです：コードが走るか、テストが通るか？ 広く使われる 2 つのコーディングエージェントベンチマーク、SWE-bench Verified と Terminal-Bench は、このアプローチに従います。SWE-bench Verified はエージェントに人気の Python リポジトリの GitHub issue を与え、テストスイートを走らせて解を採点します——既存のテストを壊

さずに失敗テストを通す解だけが合格。LLM はこの eval で 1 年で 40% から 80% 超まで進歩しました。Terminal-Bench は異なる方向を取り、Linux カーネルをソースからビルドしたり ML モデルを訓練したりといったエンドツーエンドの技術タスクをテストします。

コーディングタスクの主要な *outcome* を検証する pass/fail テストのセットができたら、トランスクリプトも採点するのが有用です。たとえば、ヒューリスティックベースのコード品質ルールは、テストが通るかどうか以上の観点で生成コードを評価でき、明確なルーブリックを持つモデルベース grader はエージェントがツールをどう呼ぶかやユーザーとどう対話するかといった振る舞いを評価できます。

例: コーディングエージェントの理論的 eval

認証バイパスの脆弱性を修正する必要があるコーディングタスクを考えます。以下の例示 YAML のように、grader と指標の両方でエージェントを評価できます。

```
task:
  id: "fix-auth-bypass_1"
  desc: "Fix authentication bypass when password field is empty and ..."
  graders:
    - type: deterministic_tests
      required: [test_empty_pw_rejected.py, test_null_pw_rejected.py]
    - type: llm_rubric
      rubric: prompts/code_quality.md
    - type: static_analysis
      commands: [ruff, mypy, bandit]
    - type: state_check
      expect:
        security_logs: {event_type: "auth_blocked"}
    - type: tool_calls
      required:
        - {tool: read_file, params: {path: "src/auth/*"}}
        - {tool: edit_file}
        - {tool: run_tests}
  tracked_metrics:
    - type: transcript
      metrics:
        - n_turns
        - n_toolcalls
        - n_total_tokens
    - type: latency
      metrics:
        - time_to_first_token
        - output_tokens_per_sec
        - time_to_last_token
```

この例は、利用可能な grader の全範囲を示すためのものです。実際には、コーディング評価は通常、正確性検証のためのユニットテストとコード品質評価のための LLM ルーブリックに依存し、必要に応じて追加の grader と指標を足していきます。

会話型エージェントの評価

会話型エージェント は、サポート、セールス、コーチングのような領域でユーザーと対話します。従来のチャットボットと異なり、ステートを維持し、ツールを使い、会話中にアクションを取ります。コーディングエージェントやリサーチエージェントも多ターンの対話を伴いますが、会話型エージェントは独特の課題を提示します——対話の質そのものが評価対象の一部です。会話型エージェントの効果的な eval は通常、検証可能な終端状態の outcome と、タスク完了と対話品質の両方を捉えるルーブリックに依存します。他の多くの eval と異なり、ユーザーをシミュレートする 2 つ目の LLM を必要とすることが多いです。私たちはアラインメント監査エージェント でこのアプローチを使い、長い敵対的会話でモデルをストレステストします。

会話型エージェントの成功は多次元になり得ます——チケットは解決されたか（状態チェック）、10 ターン以内に完了したか（トランスクリプト制約）、トーンは適切だったか（LLM ルーブリック）？ 多次元性を取り入れた 2 つのベンチマークは、τ-Bench とその後継 τ^2-Bench です。これらは小売サポートや航空券予約のようなドメインで、あるモデルがユーザーのペルソナを演じ、エージェントが現実的なシナリオを navigating するマルチターン対話をシミュレートします。

例: 会話型エージェントの理論的 eval

イライラした顧客への返金を扱うサポートタスクを考えます。

```
graders:
  - type: llm_rubric
    rubric: prompts/support_quality.md
    assertions:
      - "Agent showed empathy for customer's frustration"
      - "Resolution was clearly explained"
      - "Agent's response grounded in fetch_policy tool results"
  - type: state_check
    expect:
      tickets: {status: resolved}
      refunds: {status: processed}
  - type: tool_calls
    required:
      - {tool: verify_identity}
      - {tool: process_refund, params: {amount: "<=100"}}
      - {tool: send_confirmation}
  - type: transcript
    max_turns: 10
tracked_metrics:
  - type: transcript
    metrics:
      - n_turns
      - n_toolcalls
      - n_total_tokens
  - type: latency
    metrics:
      - time_to_first_token
      - output_tokens_per_sec
      - time_to_last_token
```

コーディングエージェントの例と同様、このタスクは複数の grader タイプを例示のために示しています。実際には、会話型エージェントの評価は通常、コミュニケーション品質とゴール達成の両方を評価するためにモデルベース grader を使います——質問への回答のような多くのタスクに複数の「正解」があり得るからです。

リサーチエージェントの評価

リサーチエージェント は情報を集め、統合し、分析し、回答やレポートのような出力を作ります。ユニットテストがバイナリの pass/fail シグナルを提供するコーディングエージェントと違い、リサーチの質はタスクに対してしか判断できません。「包括的」「十分にソースされた」「あるいは正確」が何を意味するかは文脈に依存します——市場スキャン、M&A のデューデリジェンス、科学レポートはそれぞれ異なる基準を要求します。

リサーチ eval は独特の課題に直面します——統合が包括的かどうかについて専門家でも意見が分かれることがあり、参照コンテンツが絶えず変わるためグラウンドトゥルースがシフトし、長くオープンエンドな出力はミスの余地を広げます。たとえば BrowseComp のようなベンチマークは、AI エージェントがオープン Web 上

で針を干し草から見つけられるかをテストします——検証は簡単だが解くのは難しいよう設計された質問です。

リサーチエージェント eval を構築する 1 つの戦略は、grader タイプの組み合わせです。グラウンデッドネスチェックは主張が取得したソースに支えられているかを検証し、カバレッジチェックは良い回答が含むべき主要事実を定義し、ソース品質チェックは参照したソースが単に最初に取得したものではなく権威あるものだったかを確認します。客観的に正解のあるタスク(「企業 X の第 3 四半期収益はいくらだった?」)には厳密一致が効きます。LLM は支持されていない主張やカバレッジの抜けにフラグを立て、またオープンエンドな統合の一貫性と網羅性を検証することもできます。

リサーチ品質の主観的な性質を考えると、LLM ベースのルーブリックはこれらのエージェントを効果的に採点するため、専門家の人間判断と頻繁にキャリブレーションすべきです。

コンピュータ利用エージェント

コンピュータ利用エージェント は、API やコード実行ではなく、人間と同じインターフェース——スクリーンショット、マウスクリック、キーボード入力、スクロール——でソフトウェアと対話します。デザインツールからレガシーな企業ソフトウェアまで、GUI を持つ任意のアプリケーションを使えます。評価には、ソフトウェアアプリケーションを使える実際の／サンドボックス化された環境でエージェントを走らせ、意図した outcome を達成したかを確認する必要があります。たとえば WebArena はブラウザベースのタスクをテストし、エージェントが正しく navigating したかを URL とページステートチェックで検証し、データを変更するタスク(注文が実際に行われたことを確認するなど)にはバックエンドステート検証も使います。OSWorld はこれを完全な OS 制御に拡張し、タスク完了後にファイルシステムの状態、アプリ設定、データベース内容、UI 要素プロパティなど多様なアーティファクトを検査する評価スクリプトを使います。

ブラウザ利用エージェントは、トークン効率とレイテンシのバランスを要します。DOM ベースの相互作用は素早く実行されますが多くのトークンを消費し、スクリーンショットベースは遅いがトークン効率が良いです。たとえば Claude に Wikipedia を要約させる場合、DOM からテキストを抽出する方が効率的です。Amazon で新しいラップトップケースを探すときはスクリーンショットを取る方が効率的です(DOM 全体の抽出はトークン集約的だから)。Claude for Chrome プロダクトでは、エージェントが各文脈で正しいツールを選んでいるかを確認する eval を開発しました。これによりブラウザベースタスクをより速くより正確に完了できました。

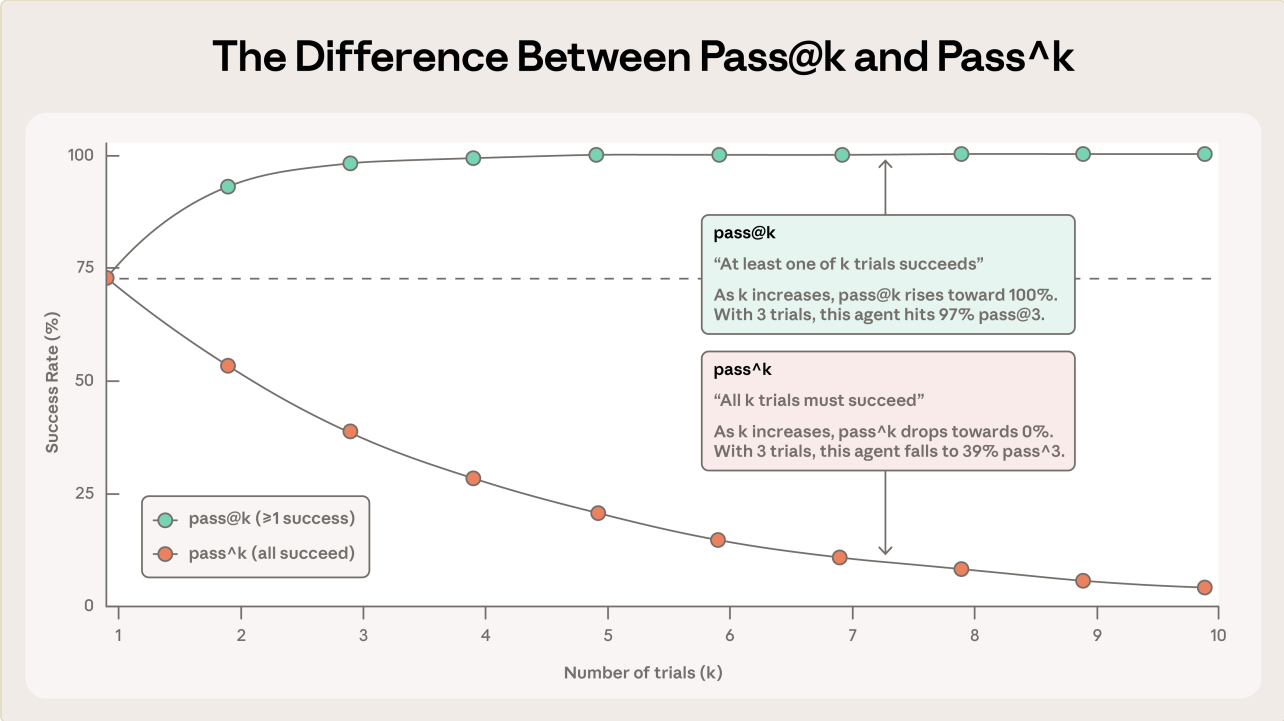
エージェント評価における非決定論をどう考えるか

エージェントタイプに関わらず、エージェントの振る舞いは実行間で変動し、評価結果は見た目より解釈が難しくなります。各タスクは独自の成功率を持ち——あるタスクでは 90%、別のタスクでは 50%——あるタスクはある eval 実行で通っても次で失敗するかもしれません。時には、私たちが測りたいのは、エージェントがタスクに対してどの *頻度で*(どの割合のトライアルで)成功するかです。

この微妙さを捉えるのに役立つ 2 つの指標があります。

pass@k は、*k* 回の試行で少なくとも 1 回正解する確率を測ります。*k* が増えると pass@k スコアは上がります——「ゴールへのシュート」が多いほど少なくとも 1 回成功する確率が高いです。50% pass@1 は、モデルが eval のタスクの半分で最初の試みで成功するという意味です。コーディングでは、エージェントが最初の試みで解を見つけることに最も関心があることが多く——pass@1 です。他のケースでは、1 つ動けばよいので多くの解を提案するのも有効です。

pass^k は、*k* 回のトライアル *すべて* が成功する確率を測ります。*k* が増えると pass^k は下がります——より多くのトライアルにわたる一貫性を要求するのは超えるべき高い基準だからです。エージェントの 1 試行あたり成功率が 75% で 3 回トライアルを走らせたら、3 回すべて通る確率は $(0.75)^3 \approx 42\%$ です。この指標は、ユーザーが毎回信頼できる振る舞いを期待する顧客向けエージェントで特に重要です。



pass@k と pass^k はトライアル数が増えると乖離する。k=1 では同一（両方とも 1 試行あたりの成功率に等しい）。k=10 では真逆の物語を語る——pass@k は 100% に近づき、pass^k は 0% に落ちる。

両方の指標は有用で、どちらを使うかは製品要件次第です——1 回の成功が重要なツールには pass@k、一貫性が不可欠なエージェントには pass^k です。

ゼロからイチへ: エージェント向け優れた eval のロードマップ

このセクションでは、eval が皆無の状態から信頼できる eval まで進むための、実地で検証された実用的アドバイスを示します。これを eval 駆動のエージェント開発のロードマップだと考えてください——早期に成功を定義し、明確に測定し、継続的に反復する。

初期 eval データセット用のタスクを集める

ステップ 0. 早く始める

数百のタスクが必要だと信じて eval 構築を遅らせるチームを見かけます。実際には、実際の失敗から引き出した 20〜50 の単純なタスクで十分なスタートです。結局、初期のエージェント開発ではシステムへの各変更が明確で目立つ影響を持つことが多く、この大きな効果量は小さなサンプルサイズを十分にします。より成熟したエージェントはより小さな効果を検出するためにより大きく難しい eval を必要とするかもしれませんが、最初は 80/20 アプローチが最良です。eval は待つほど作るのが難しくなります。初期には、製品要件が自然にテストケースに翻訳されます。長く待ちすぎると、稼働中のシステムから逆に成功基準を引き出すハメになります。

ステップ 1. すでに手動でテストしているものから始める

開発中に走らせる手動チェック——各リリース前に検証する振る舞いと、エンドユーザーがよく試すタスク——から始めましょう。すでに本番にいるなら、バグトラッカーとサポートキューを見ましょう。ユーザー報告の失敗をテストケースに変換することで、スイートが実際の使用を反映することを保証できます。ユーザー影響で優先順位付けすれば、労力を効果のある箇所に投入できます。

ステップ 2: 参照解付きの曖昧でないタスクを書く

タスクの質を正しくするのは見た目より難しいです。良いタスクとは、2 人のドメイン専門家が独立に同じ合否判定に達するものです。彼ら自身がそのタスクを通せるでしょうか？ できないなら、タスクは洗練が必要です。タスク仕様の曖昧さは指標のノイズになります。モデルベース grader の基準にも同じことが言えます——曖昧なルーブリックは一貫性のない判定を生みます。

各タスクは、指示に正しく従うエージェントが通せるべきです。これは微妙です。たとえば Terminal-Bench の監査で、タスクがエージェントにスクリプトを書かせるが filepath を指定しておらず、テストが特定の filepath を想定していると、エージェントは自分の落ち度でなく失敗することが分かりました。grader がチェックするすべてのものはタスク記述から明確であるべきです——エージェントが曖昧な仕様で失敗すべきではありません。最先端モデルで、多くのトライアルにわたる 0% の合格率（つまり 0% pass@100）は、能力のないエー

ジェントではなく壊れたタスクのサインであることが多く、タスク仕様と grader を二重チェックすべきサインです。各タスクには、参照解——すべての grader を通す既知の動く出力——を作るのが有用です。これはタスクが解けることを証明し、grader が正しく設定されていることを検証します。

ステップ 3: バランスの取れた問題セットを作る

振る舞いが起きる べき ケースと起きる べきでない ケースの両方をテストしましょう。片側だけの eval は片側だけの最適化を生みます。たとえば、エージェントが検索するべきときに検索するかだけをテストしたら、ほぼすべてに対して検索してしまうエージェントに辿り着きかねません。クラス不均衡 eval は避けましょう。私たちは Claude.ai の Web 検索 eval を構築する際にこれを直接学びました。課題は、モデルが検索すべきでないときに検索するのを防ぎつつ、適切なときに広範な調査ができる能力を維持することでした。チームは両方向をカバーする eval を構築しました——モデルが検索すべきクエリ（天気を探すなど）と、既存知識で答えるべきクエリ（「Apple を創業したのは誰?」など）。過少トリガ（するべきときに検索しない）と過剰トリガ（すべきでないときに検索）の間で正しいバランスを取るのは難しく、プロンプトと eval の両方に何度も洗練が必要でした。より多くの例の問題が出てくるにつれ、カバレッジを改善するために eval を追加し続けています。

eval ハーネスと grader を設計する

ステップ 4: 安定した環境を持つ堅牢な eval ハーネスを作る

eval 中のエージェントが本番で使われるエージェントとほぼ同じように機能すること、そして環境自体がさらなるノイズを持ち込まないことが不可欠です。各トライアルはクリーンな環境から始めて「隔離」されるべきです。実行間の不必要な共有状態（残ったファイル、キャッシュされたデータ、リソース枯渇）は、エージェント性能ではなくインフラの揺らぎによる相関した失敗を引き起こし得ます。共有状態は性能を人工的に膨張させることもあります。たとえば社内のいくつかの eval で、Claude が以前のトライアルの git 履歴を見て不公正な優位を得ていることを観察しました。複数の別々のトライアルが同じ環境の制限（限られた CPU メモリなど）で失敗するなら、これらのトライアルは同じ要因に影響されているため独立ではなく、eval 結果はエージェント性能を測るのに信頼できなくなります。

ステップ 5: grader を思慮深く設計する

上で議論したように、優れた eval 設計にはエージェントとタスクに最良の grader を選ぶことが含まれます。可能なら決定論的 grader を、必要なときや追加の柔軟性のために LLM grader を、そして追加の検証のために人間 grader を賢明に使うことを勧めます。

ツール呼び出しの正しい順序のような非常に具体的なステップをエージェントが辿ったかをチェックしようとする本能がよくあります。このアプローチは硬すぎて、過度に脆いテストを生むことが分かっています——エージェントは eval 設計者が予想しなかった有効なアプローチを常に見つけるからです。創造性を不必要に罰し

ないために、エージェントが通った経路ではなく、エージェントが生み出したものを採点する方がしばしば良いです。

複数のコンポーネントを持つタスクには **部分点** を組み込みましょう。問題を正しく特定し顧客を検証したが返金処理に失敗したサポートエージェントは、即座に失敗したものより明らかに良いです。成功のこの連続性を結果に反映することが重要です。

モデル採点はしばしば精度を検証するために注意深い反復を要します。LLM-as-judge grader は、人間採点とモデル採点の乖離が少ないという自信を得るため、人間専門家と密にキャリブレーションすべきです。幻覚を避けるには、LLM に「逃げ道」を与えましょう——十分な情報がないときに「不明」を返す指示を与えるなどです。明確で構造化されたルーブリックを作り、タスクの各次元を別々の LLM-as-judge で採点する（1 つで全次元を採点するのではなく）のも有効です。システムが堅牢になれば、人間レビューはたまに使うだけで十分です。

いくつかの評価は、エージェントの性能が良くても低スコアになる微妙な失敗モードを持ちます——採点のバグ、エージェントハーネスの制約、曖昧さにより、エージェントがタスクを解けないことがあります。洗練されたチームでもこうした問題を見落とし得ます。たとえば Opus 4.5 は最初 CORE-Bench で 42% スコア でしたが、Anthropic の研究者が複数の問題を発見するまでのことでした——「96.124991…」を期待しているのに「96.12」を罰する硬直な採点、曖昧なタスク仕様、厳密に再現不可能な確率的タスク。バグを修正し、制約の少ないスキャフォールドを使ったところ、Opus 4.5 のスコアは 95% に跳ね上がりました。同様に、METR は彼らの time horizon ベンチマーク にいくつかの誤構成タスクを発見しました——エージェントに明示されたスコアしきい値まで最適化するよう頼んでいたのに、採点はそのしきい値を超えることを要求していたのです。これは指示に従う Claude のようなモデルを罰し、明示された目標を無視するモデルがより良いスコアを得る結果になりました。タスクと grader を慎重に二重チェックすることで、こうした問題を避けられます。

grader をバイパスやハックに抵抗させましょう。エージェントが簡単に eval を「ズル」できてはなりません。タスクと grader は、意図しない抜け穴を悪用するのではなく、実際に問題を解くことで通るように設計されるべきです。

eval を長期にわたって保守・活用する

ステップ 6: トランスクリプトを確認する

多くのトライアルからのトランスクリプトと採点を読まない限り、grader がうまく働いているかは分かりません。Anthropic では eval トランスクリプトを見るためのツーリングに投資し、定期的に時間を取って読みます。タスクが失敗するとき、トランスクリプトはエージェントが本物のミスを犯したのか、grader が有効な解を拒絶したのかを教えてくれます。エージェントと eval の振る舞いの主要な詳細もしばしば浮かび上がります。

失敗は公平に見えるべきです——エージェントが何を間違えたのか、なぜかが明確に。スコアが登らないとき、それがエージェント性能によるもので eval によるものではないという自信が必要です。トランスクリプトを読むことは、自分の eval が実際に重要なものを測っているかを検証する方法であり、エージェント開発に不可欠なスキルです。

ステップ 7: capability eval の飽和を監視する

100% の eval は回帰を追跡しますが、改善のシグナルは提供しません。**eval 飽和** は、エージェントが解ける全タスクを通して改善の余地が残らなくなった状態です。たとえば SWE-Bench Verified のスコアは今年 30% から始まり、最先端モデルは今 80% 超で飽和に近づいています。eval が飽和に近づくにつれ、進捗も遅くなります——最も難しいタスクしか残っていないからです。これは結果を誤解しやすくします——大きな能力改善がスコアのわずかな上昇として現れるのです。たとえばコードレビュースタートアップ Qodo は、彼らのワンショットコーディング eval がより長く複雑なタスクでの利得を捉えなかったため、当初 Opus 4.5 に感心しませんでした。対応して、彼らは新しいエージェント型 eval フレームワークを開発し、進捗のより明確な絵を提供しました。

原則として、私たちは誰かが eval の詳細を掘り下げトランスクリプトをいくつか読むまで、eval スコアを額面通りに受け取りません。採点が不公平だったり、タスクが曖昧だったり、有効な解が罰せられたり、ハーネスがモデルを制約したりするなら、eval は改訂されるべきです。

ステップ 8: 評価スイートをオープンな貢献と保守で長期的に健康に保つ

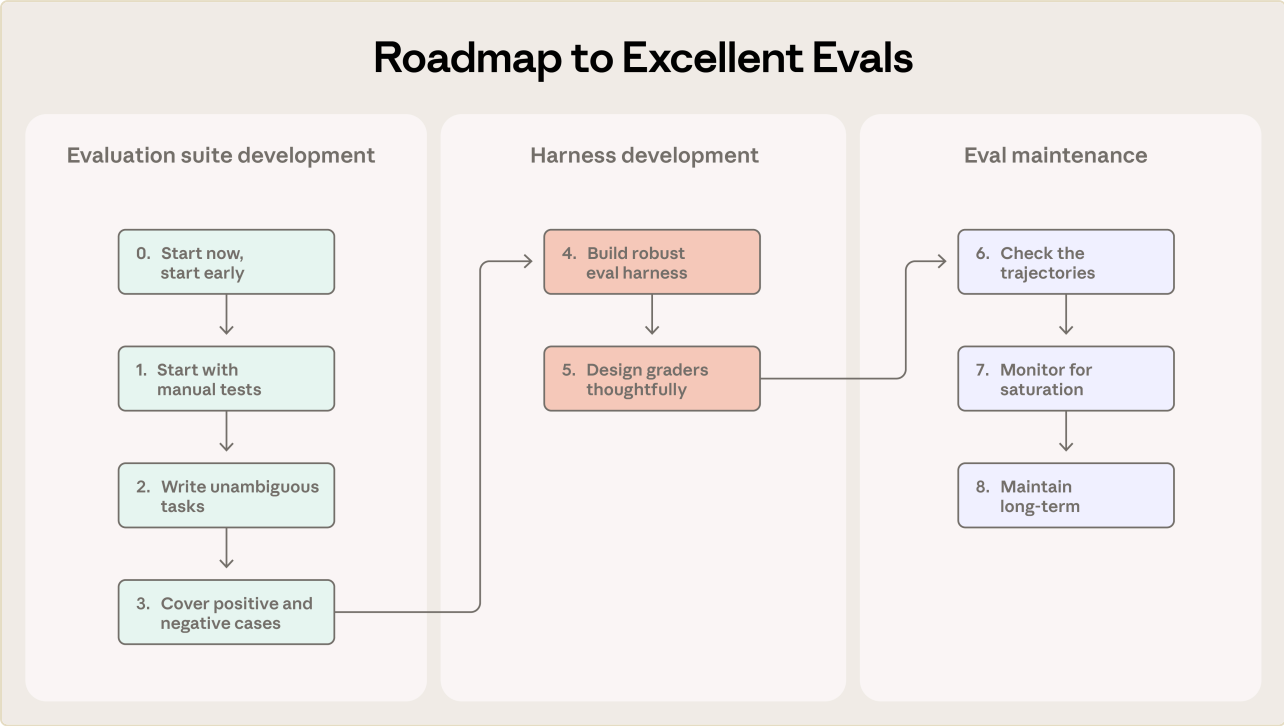
eval スイートは生きた成果物であり、有用であり続けるには継続的な注意と明確なオーナーシップが必要です。

Anthropic では、eval 保守のさまざまなアプローチを試しました。最も効果的だったのは、専任の eval チームがコアインフラを所有し、ドメイン専門家と製品チームが大半の eval タスクを貢献し、評価自体を走らせる、という形でした。

AI 製品チームにとって、eval を所有・反復することはユニットテストの保守と同じくらいルーチン的であるべきです。チームは初期テストでは「動く」が、よく設計された eval が早期に浮上させたであろう明示されない期待を満たせない AI 機能に数週間を無駄にしかねません。eval タスクを定義することは、製品要件が構築を始めるのに十分具体的かをストレステストする最良の方法の 1 つです。

eval 駆動開発を実践することを勧めます——エージェントが達成する前に計画された能力を定義する eval を作り、その後エージェントがうまく機能するまで反復する。社内では、今日「十分に」機能するがモデルが数ヶ月後にできるであろうことへの賭けとして機能を作ることがよくあります。低い合格率で始まる capability eval はこれを可視化します。新しいモデルが出たとき、スイートを走らせればどの賭けが報われたかがすぐに分かります。

製品要件とユーザーに最も近い人々が、成功を定義するのに最良の立場にいます。現状のモデル能力では、プロダクトマネージャー、カスタマーサクセスマネージャー、セールスパーソンが Claude Code を使って eval タスクを PR として貢献できます——やらせてあげましょう！ あるいは、積極的に有効化してあげましょう。



効果的な評価を作るプロセス。

eval が、エージェントを全体的に理解するための他の手法とどう組み合わさるか

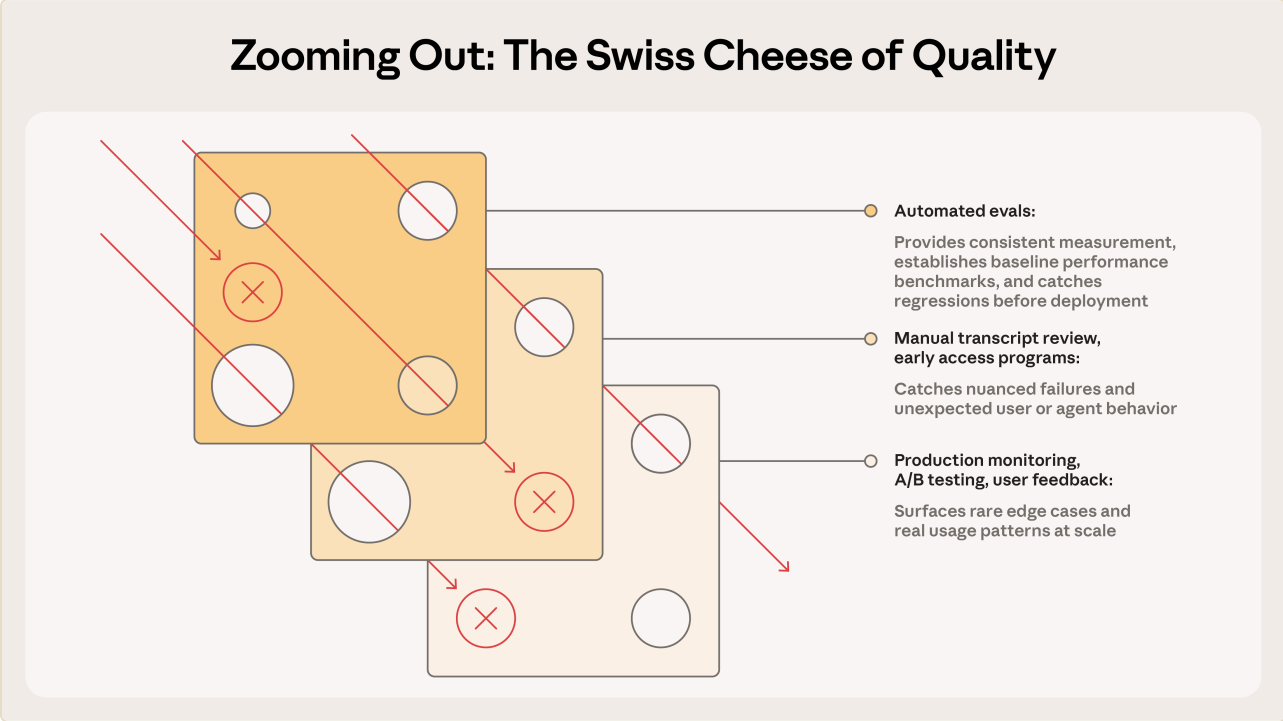
自動評価は、本番にデプロイしたり実ユーザーに影響を与えたりせずに、数千のタスクでエージェントに対して走らせられます。しかしこれは、エージェント性能を理解する多くの方法のうちの 1 つに過ぎません。完全な絵には、本番モニタリング、ユーザーフィードバック、A/B テスト、手動トランスクリプトレビュー、体系的な人間評価が含まれます。

AI エージェント性能を理解するためのアプローチの概要

手法	長所	短所
自動 eval *実ユーザーなしにテストをプログラム的に走らせる*	- 反復が速い - 完全再現可能 - ユーザー影響なし - コミットごとに走らせられる - 本番デプロイなしに大規模シナリオをテスト	- 構築に前払い投資が必要 - 製品とモデルが進化するにつれドリフトを避けるための保守が必要 - 実使用パターンに合致しないなら偽の自信を生む
本番モニタリング *稼働中システムの指標とエラーを追跡*	- 大規模の実ユーザー行動を明らかに - 合成 eval が見逃す問題を捕まえる - エージェントが実際どう機能するかのグラウンドトゥルース	- 反応的。ユーザーに問題が届いてから気づく - シグナルがノイジー - 計装への投資が必要 - 採点用のグラウンドトゥルースを欠く
A/B テスト *実ユーザートラフィックでバリアントを比較*	- 実ユーザー結果（継続率、タスク完了）を測定 - 交絡因子を制御 - スケーラブルで体系的	- 遅い。統計的有意に達するまで数日〜数週間、十分なトラフィックが必要 - デプロイする変更しかテストしない - トランスクリプトを徹底レビューできなければ指標変化の根本「なぜ」のシグナルが弱い
ユーザーフィードバック *「いいえ」やバグ報告のような明示的シグナル*	- 予想しなかった問題を浮上させる - 実人間ユーザーからの実例が付属 - フィードバックはしばしば製品目標と相関	- 疎で自己選択的 - 深刻な問題に偏る - ユーザーは失敗の *理由* を説明しない - 自動化されない - ユーザー任せで問題を捕まえることはユーザー影響がネガティブになる
手動トランスクリプトレビュー *人間がエージェント会話を通読*	- 失敗モードの直感を育てる - 自動チェックが逃す微妙な品質問題を捕まえる - 「良い」が何かをキャリブレートし詳細を掴むのに役立つ	- 時間集約的 - スケールしない - カバレッジが一貫しない - レビューアーの疲労や異なるレビューアーがシグナル品質に影響 - 通常は定量的な採点ではなく定性的シグナルだけを与える

手法	長所	短所
体系的な人間スタディ *訓練された採点者によるエージェント出力の構造化採点*	- 複数の人間採点者からのゴールドスタンダード品質判断 - 主観的／曖昧なタスクを扱う - モデルベース grader の改善シグナルを提供	- 比較的高コストで遅い回転 - 頻繁に走らせにくい - 評価者間の不一致には調整が必要 - 複雑なドメイン（法律、金融、医療）はスタディ実施に人間専門家が必要

これらの方法はエージェント開発の異なる段階にマッピングされます。自動 eval はローンチ前と CI/CD で特に有用で、品質問題への最前線防御としてエージェントの変更とモデルアップグレードごとに走らせます。本番モニタリングはローンチ後に分布ドリフトや予想外の現実世界の失敗を検出するために機能します。A/B テストは十分なトラフィックがあれば大きな変更を検証します。ユーザーフィードバックとトランスクリプトレビューは隙間を埋める継続的プラクティスです——フィードバックを常時トリアージし、毎週トランスクリプトをサンプル読みし、必要に応じて深掘りします。体系的人間スタディは、LLM grader のキャリブレーションや人間合意が参照基準となる主観的な出力の評価に取っておきます。



安全工学のスイスチーズモデル と同様に、単一の評価レイヤーが全問題を捕まえることはない。複数の手法を組み合わせれば、ある層をすり抜けた失敗は別の層で捕まる。

最も効果的なチームは、これらの手法を組み合わせます——高速反復のための自動 eval、グラウンドトゥルースのための本番モニタリング、キャリブレーションのための定期的な人間レビュー。

結論

eval を持たないチームは反応的なループにはまります——1 つの失敗を直すとまた別の失敗を生み、実際の回帰とノイズを区別できません。早期に投資するチームは正反対のことを見つけます——失敗がテストケースになり、テストケースが回帰を防ぎ、指標が推測を置き換えるので、開発が加速するのです。eval はチーム全体に明確な登るべき丘を与え、「エージェントの性能が落ちた気がする」を実行可能なものに変えます。その価値は複利で積み上がりますが、eval をおまけではなく中核コンポーネントとして扱う場合に限ります。

パターンはエージェントタイプで変わりますが、本稿で述べた基礎は不変です。早く始める、完璧なスイートを待たない。見た失敗から現実的なタスクを引き出す。曖昧でなく堅牢な成功基準を定義する。grader を思慮深く設計し、複数タイプを組み合わせる。問題がモデルにとって十分難しいか確認する。シグナル対ノイズ比を改善するために評価を反復する。**トランスクリプトを読む！**

AI エージェント評価はまだ生まれたばかりで急速に進化している分野です。エージェントがより長いタスクを引き受け、マルチエージェントシステムで協業し、ますます主観的な仕事を扱うようになるにつれ、私たちは技法を適応させる必要があるでしょう。学び続けるなかでベストプラクティスを共有し続けます。

謝辞

執筆は Mikaela Grace、Jeremy Hadfield、Rodrigo Olivares、Jiri De Jonghe。貢献に対し David Hershey、Gian Segato、Mike Merrill、Alex Shaw、Nicholas Carlini、Ethan Dixon、Pedram Navid、Jake Eaton、Alyssa Baum、Lina Tawfik、Karen Zhou、Alexander Bricken、Sam Kennedy、Robert Ying らに感謝します。eval の協業を通じて学んできた顧客・パートナーにも特別に感謝します——iGent、Cognition、Bolt、Sierra、Vals.ai、Macroscope、PromptLayer、Stripe、Shopify、Terminal Bench チームなど。本作業は、Anthropic で評価のプラクティスを開発するのに助けた複数チームの集合的努力を反映しています。

付録: eval フレームワーク

いくつかのオープンソースおよび商用フレームワークは、インフラをゼロから作らずにエージェント評価を実装するのを助けます。適切な選択はエージェントタイプ、既存スタック、オフライン評価・本番観測可能性のどちらが（あるいは両方が）必要かによります。

Harbor はコンテナ化された環境でエージェントを走らせることを想定しており、クラウドプロバイダー横断で大規模にトライアルを走らせるインフラと、タスクと grader を定義する標準フォーマットを提供します。Terminal-Bench 2.0 のような人気ベンチマークは Harbor レジストリ経由で出荷されており、カスタム eval スイートとあわせて確立されたベンチマークを実行するのが簡単です。

Braintrust はオフライン評価を本番観測可能性と実験追跡と結びつけるプラットフォームで、開発中の反復と本番品質のモニタリングの両方が必要なチームに有用です。`autoevals` ライブラリには事実性、関連性、その他一般的な次元向けの事前構築されたスコアラーがあります。

LangSmith はトレーシング、オフライン／オンライン評価、データセット管理を提供し、LangChain エコシステムと緊密に統合されています。Langfuse は、データ所在要件を持つチーム向けのセルフホストオープンソース代替として同様の機能を提供します。

Arize は、LLM トレーシング、デバッグ、オフライン／オンライン評価用のオープンソースプラットフォーム Phoenix と、スケール・最適化・モニタリングのために Phoenix を拡張する SaaS 提供の AX を提供します。

多くのチームは複数のツールを組み合わせたり、自前の eval フレームワークを作ったり、あるいは単純な評価スクリプトを出発点として使ったりします。フレームワークは進捗を加速し標準化する価値ある方法になり得ますが、それを通して走らせる eval タスク次第であることが分かっています。自分のワークフローに合うフレームワークを素早く選び、高品質なテストケースと grader に反復することにエネルギーを投じるのが良い選択です。