14

長時間動作するエージェントのための効果的なハーネス

— Effective harnesses for long-running agents —

公開日	2025-11-26
原題	Effective harnesses for long-running agents
著者	Anthropic Engineering Team
原文	https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents
翻訳	Claude（機械翻訳／Anthropic）
編集	2026-04-09

長時間動作するエージェントのための効果的なハーネス

AI エージェントの能力が上がるにつれ、開発者は数時間あるいは数日にわたる作業を伴う複雑なタスクをエージェントに任せるようになっています。しかし、複数のコンテキストウィンドウにまたがってエージェントに一貫した進捗を出させることは、まだ未解決の問題です。

長時間エージェントの核心的な課題は、離散的なセッションで作業しなければならないことと、各新セッションが以前の記憶ゼロから始まることです。シフト制で働くエンジニアのいるソフトウェアプロジェクトを想像してみてください。各新エンジニアは前のシフトで何が起きたかの記憶をまったく持たずにやってきます。コンテキストウィンドウは有限で、複雑なプロジェクトの大半は 1 つのウィンドウには収まらないため、エージェントはコーディングセッション間のギャップを埋める方法が必要です。

私たちは、多数のコンテキストウィンドウにわたって Claude Agent SDK を効果的に働かせるために、二段構えの解決策を開発しました——最初の実行で環境をセットアップする **初期化エージェント（initializer agent）** と、各セッションで漸進的な進捗を作りつつ次セッションのために明確な成果物を残す **コーディングエージェント（coding agent）** です。コード例は付属のクイックスタート にあります。

長時間エージェント問題

Claude Agent SDK は、コーディングだけでなく、モデルが文脈収集・計画・実行にツールを使う必要のあるその他のタスクにも強い、強力で汎用的なエージェントハーネスです。圧縮（compaction）のようなコンテキスト管理機能を持ち、コンテキストウィンドウを枯渇させずにエージェントがタスクを続けられます。理論的には、このセットアップならエージェントが任意の長さで有用な仕事を続けられるはずです。

しかし、圧縮だけでは十分ではありません。デフォルト設定では、Opus 4.5 のような最先端のコーディングモデルですら Claude Agent SDK をループで回し「claude.ai のクローンを作って」のような高レベルプロンプトだけを与えると、本番品質の Web アプリを作るには至りません。

Claude の失敗は 2 つのパターンで現れました。まず、エージェントは一度に多くを詰め込みすぎ、本質的にはアプリのワンショットを試みがちでした。これはしばしば、実装の途中でモデルがコンテキストを使い果たし、次セッションには機能が半実装のまま文書化もされずに残される結果になりました。すると次のエージェントは何が起きたかを推測し、基本のアプリを動かすのに時間をかけるハメになります。圧縮があっても、次のエージェントへ完璧に明確な指示が渡るとは限らないのです。

2 つ目の失敗モードはプロジェクトの後半で起きることが多かったです。いくつかの機能が既に作られた後、後のエージェントインスタンスが周りを見渡し、進捗が出ているのを見て「仕事は終わった」と宣言するのです。

これを問題として 2 つに分解できます。まず、与えられたプロンプトが要求する *すべての* 機能の基盤を敷く初期環境をセットアップし、エージェントが段階的・機能単位で作業を進められる状態を整える必要があります。次に、各エージェントがゴールに向けて漸進的に進捗を出しつつ、セッションの終わりに環境をクリーンな状態で残すようにプロンプトすべきです。ここでいう「クリーンな状態」とは、メインブランチにマージしても差し支えないようなコード——大きなバグがなく、整然として文書化されており、開発者が関係ない散らかりを先に片付けなくても新機能に取りかかれる——を指します。

社内で実験したとき、私たちはこの問題を 2 部構成の解で扱いました。

1. **初期化エージェント**: 最初のエージェントセッションは専用のプロンプトを使い、モデルに初期環境のセットアップを依頼します—— `init.sh` スクリプト、エージェントがしたことを記録する `claude-progress.txt` ファイル、追加されたファイルを示す初期 git コミット。
2. **コーディングエージェント**: それ以降の各セッションは、モデルに漸進的な進捗を作り、構造化された更新を残すよう依頼します[1]。

ここでの重要な洞察は、新しいコンテキストウィンドウで始めたエージェントが作業状態を素早く理解できる方法を見つけることであり、それは `claude-progress.txt` と git 履歴によって達成されます。これらのプラクティスは、効果的なソフトウェアエンジニアが日々何をしているかを知ることから着想を得ました。

環境管理

更新された Claude 4 プロンプティングガイド で、私たちは「最初のコンテキストウィンドウでは別のプロンプトを使う」ハーネス構造を含む、マルチコンテキストウィンドウワークフローのベストプラクティスを共有しました。この「別のプロンプト」は、将来のコーディングエージェントが効果的に働くのに必要な文脈をすべて備えた環境を、初期化エージェントにセットアップさせるものです。ここでは、そうした環境の主要コンポーネントを掘り下げます。

機能リスト

エージェントがアプリをワンショットで作ろうとしたり、プロジェクトを早期に完了と見なしたりする問題に対処するため、初期化エージェントにユーザーの初期プロンプトを拡張した包括的な機能要件ファイルを書かせるようプロンプトしました。claude.ai クローンの例では、これは 200 以上の機能を意味しました——たとえば「ユーザーが新しいチャットを開き、クエリを入力し、エンターを押し、AI の応答を受け取る」といったものです。これらの機能は最初「失敗（failing）」としてマークされ、後のコーディングエージェントが完全な機能の全容を明確に把握できるようにしました。

```
{
    "category": "functional",
    "description": "New chat button creates a fresh conversation",
    "steps": [
      "Navigate to main interface",
      "Click the 'New Chat' button",
      "Verify a new conversation is created",
      "Check that chat area shows welcome state",
      "Verify conversation appears in sidebar"
    ],
    "passes": false
  }
```

コーディングエージェントには、このファイルを `passes` フィールドのステータスを変えることでしか編集しないよう促し、「機能の欠落やバグを招きかねないのでテストの削除・編集は許容できない」といった強い表現を使っています。実験の末、Markdown ファイルより不適切に変更・上書きされにくい JSON を使うことに落ち着きました。

漸進的な進捗

この初期の環境足場があれば、次に走るコーディングエージェントには一度に 1 つの機能だけを扱うよう依頼します。この漸進的アプローチは、エージェントが一度に多くをやろうとする傾向への対処として決定的に重要であると分かりました。

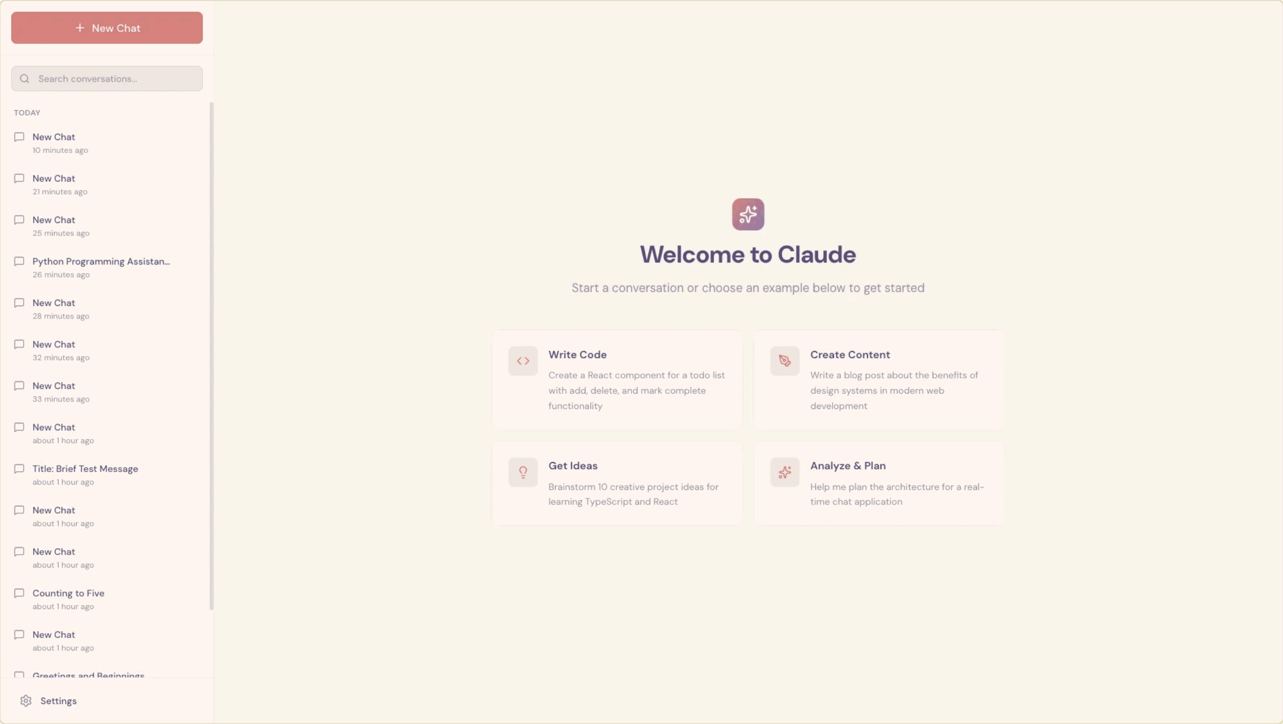
漸進的に働くようになっても、コード変更後にエージェントが環境をクリーンな状態で残すことは依然として必須です。実験では、この振る舞いを引き出す最良の方法は、記述的なコミットメッセージで進捗を git にコミットし、プログレスファイルに進捗サマリーを書くようモデルに依頼することでした。これによりモデルは git を使って悪いコード変更を revert し、動く状態のコードベースを取り戻せます。

これらのアプローチは効率も高めました――エージェントが「何が起きたか」を推測したり、基本のアプリを動かすのに時間を費やしたりする必要がなくなったからです。

テスト

もう 1 つの主要な失敗モードは、Claude が適切なテストなしに機能を完了とマークしがちなことでした。明示的なプロンプトなしでは、Claude はコード変更やユニットテスト、開発サーバーへの `curl` コマンドでのテストさえ行いますが、機能がエンドツーエンドで動いていないことに気づけないことがありました。

Web アプリ構築の場合、Claude はブラウザ自動化ツールを使って人間ユーザーと同じ方法ですべてのテストをするよう明示的に促されれば、機能をエンドツーエンドで検証するのはおおむね上手にできました。



Welcome to Claude

Start a conversation or choose an example below to get started

Create a React component for a todo list with add, delete, and mark complete functionality

Write a blog post about the benefits of design systems in modern web development

Brainstorm 10 creative project ideas for learning TypeScript and React

Help me plan the architecture for a real-time chat application

Puppeteer MCP サーバー経由で Claude が claude.ai クローンをテストしたときに取ったスクリーンショット。

Claude にこうしたテストツールを提供することで性能は劇的に改善しました——コードを見ているだけでは明らかでないバグをエージェントが特定して直せるようになったからです。

いくつかの問題は残っています——Claude のビジョンとブラウザ自動化ツールの制限により、あらゆる種類のバグを特定するのは難しいのです。たとえば Claude は Puppeteer MCP 経由でブラウザネイティブの alert モーダルを見られないため、こうしたモーダルに依存する機能はバグが出やすい傾向がありました。

状況把握

以上をすべて整えたうえで、各コーディングエージェントは以下の一連のステップで状況把握するよう促されます。一部は非常に基本的ですが、それでも有用です。

1. *`pwd` を実行して作業ディレクトリを確認する。このディレクトリ内のファイルのみ編集可能である。*
2. *git ログと progress ファイルを読んで、最近何が作業されていたかを把握する。*
3. *機能リストファイルを読み、まだ完了していない最優先の機能を選んで取り組む。*

このアプローチは、毎セッションでコードのテスト方法を考え直す手間を省くので、トークンを節約します。さらに、初期化エージェントに、開発サーバーを起動できる `init.sh` スクリプトを書いてもらい、新機能を実装する前に基本のエンドツーエンドテストを走らせるよう依頼するのも有用です。

claude.ai クローンの場合、これはエージェントが常にローカル開発サーバーを起動し、Puppeteer MCP で新しいチャットを開始し、メッセージを送り、応答を受け取ることを意味しました。これにより、Claude はアプリが壊れた状態で残されていないかを素早く特定し、既存バグを即座に直せます。もしエージェントが代わりに新機能の実装を始めていたら、問題を悪化させる可能性が高かったでしょう。

これらを踏まえると、典型的なセッションはこんなアシスタントメッセージで始まります。

```
[Assistant] I'll start by getting my bearings and understanding the current state of the project.
[Tool Use] <bash - pwd>
[Tool Use] <read - claude-progress.txt>
[Tool Use] <read - feature_list.json>
[Assistant] Let me check the git log to see recent work.
[Tool Use] <bash - git log --oneline -20>
[Assistant] Now let me check if there's an init.sh script to restart the servers.
<Starts the development server>
[Assistant] Excellent! Now let me navigate to the application and verify that some fundamental features are still working.
<Tests basic functionality>
[Assistant] Based on my verification testing, I can see that the fundamental functionality is working well. The core chat features, theme switching, conversation loading, and error handling are all functioning correctly. Now let me review the tests.json file more comprehensively to understand what needs to be implemented next.
<Starts work on a new feature>
```

エージェントの失敗モードと解決策

問題	**初期化エージェントの振る舞い**	**コーディングエージェントの振る舞い**
プロジェクト全体を早すぎる段階で「勝利宣言」する	入力仕様に基づき、エンドツーエンドの機能記述リストを含む構造化 JSON を書く	セッション開始時に機能リストを読み、作業する機能を 1 つ選ぶ
バグや文書化されていない進捗を残したまま環境を去る	初期 git リポジトリと progress ノートファイルを書く	progress ノートと git コミットログを読んでセッションを開始し、開発サーバーで基本テストを走らせて未文書化バグを捕まえる。セッション終了時に git コミットと progress 更新を書く
機能を早期に完了マーク	機能リストを作る	全機能を自己検証。注意深いテストの後にのみ「passing」マーク
アプリの動かし方を考えるのに時間を使う	開発サーバーを起動できる `init.sh` スクリプトを書く	セッション開始時に `init.sh` を読む

長時間 AI エージェントにおける 4 つの一般的失敗モードと解決策の要約。

今後の課題

本研究は、複数のコンテキストウィンドウにわたってモデルが漸進的な進捗を作れるようにする長時間エージェントハーネスの解決策を 1 例として示しました。しかしいくつかの未解決課題が残ります。

最も顕著なのは、単一の汎用コーディングエージェントがコンテキスト横断で最良の性能を出すのか、マルチエージェントアーキテクチャでより良い性能が得られるのかがまだ不明なことです。テストエージェント、QA エージェント、コードクリーンアップエージェントのような専門エージェントが、ソフトウェア開発ライフサイクルのサブタスクでさらに良い仕事をできるのは合理的に思えます。

加えて、このデモはフルスタック Web アプリ開発に最適化されています。将来の方向性は、これらの知見を他分野にも一般化することです。ここで得た教訓の一部または全部は、たとえば科学研究や金融モデリングに求められる種類の長時間エージェント型タスクにも応用できる可能性が高いです。

謝辞

執筆は Justin Young。David Hershey、Prithvi Rajasakeran、Jeremy Hadfield、Naia Bouscal、Michael Tingley、Jesse Mu、Jake Eaton、Marius Buleandara、Maggie Vo、Pedram Navid、Nadine Yasser、Alex Notov の貢献に特別に感謝します。

本作業は、Claude が安全に長時間軸の自律ソフトウェアエンジニアリングを行えるよう可能にした Anthropic の複数チームの集合的な努力を反映しています——特に code RL と Claude Code チーム。貢献に関心のある候補者は anthropic.com/careers から応募してください。

脚注

1. ここで別々の「エージェント」と呼んでいるのは、初期ユーザープロンプトが異なるという意味だけです。システムプロンプト、ツールセット、全体のエージェントハーネスはそれ以外同一でした。