19

長時間アプリケーション開発のためのハーネス設計

— Harness design for long-running application development —

公開日	2026-03-24
原題	Harness design for long-running application development
著者	Anthropic Engineering Team
原文	https://www.anthropic.com/engineering/harness-design-long-running-apps
翻訳	Claude（機械翻訳／Anthropic）
編集	2026-04-09

長時間アプリケーション開発のためのハーネス設計

執筆: Prithvi Rajasekaran、私たちの Labs チームのメンバー。

過去数ヶ月、私は 2 つの相互に関連する問題に取り組んできました——Claude に高品質のフロントエンドデザインを作らせること、そして人間の介入なしに完全なアプリケーションを作らせることです。この作業は、以前のフロントエンドデザイン skill と長時間コーディングエージェントハーネス の取り組みに端を発しています。同僚と私はプロンプトエンジニアリングとハーネス設計で Claude の性能をベースラインより大幅に改善できましたが、どちらも天井に達しました。

突破するために、私は主観的な趣味で定義される領域と、検証可能な正確さと使いやすさで定義される領域の 2 つの異なるドメインに共通する新奇な AI エンジニアリングアプローチを探しました。敵対的生成ネットワーク（GAN）からインスピレーションを得て、私は **生成**（**generator**）エージェントと **評価**（**evaluator**）エージェントからなるマルチエージェント構造を設計しました。出力を確実に——そして趣味を持って——採点する評価者を作るには、まず「このデザインは良いか？」のような主観的判断を具体的で採点可能な言葉に変える基準一式を開発する必要がありました。

その後、これらの手法を長時間自律コーディングに適用し、以前のハーネス作業から 2 つの教訓を引き継ぎました——ビルドを扱いやすいチャンクに分解すること、そしてセッション間の文脈を受け渡す構造化アーティファクトを使うこと。最終的な結果は、長時間自律コーディングセッションでリッチなフルスタックアプリケーションを生み出す、**プランナー・ジェネレーター・エバリュエーター** の 3 エージェントアーキテクチャになりました。

素朴な実装がうまくいかない理由

ハーネス設計が長時間エージェント型コーディングの効果に大きく影響することは以前示しました。以前の実験 では、初期化エージェントが製品仕様をタスクリストに分解し、コーディングエージェントがセッションをまたいで文脈を運ぶアーティファクトを引き渡す前に機能を 1 つずつ実装するという方式を使いました。広い開発者コミュニティも同様の洞察に収束しており、エージェントを連続反復サイクルに保つためにフックやスクリプトを使う「Ralph Wiggum」のような手法があります。

しかしいくつかの問題が残っていました。より複雑なタスクでは、エージェントはそれでも時間と共に脱線しがちです。この問題を分解すると、この種のタスクを実行するエージェントで 2 つの一般的な失敗モードを観察しました。

まず、コンテキストウィンドウが埋まるにつれて、モデルは長いタスクで一貫性を失う傾向があります（コンテキストエンジニアリング の記事参照）。一部のモデルは「コンテキスト不安（context anxiety）」を示します——自分の考える文脈上限に近づくにつれ、作業を早めに切り上げ始めるのです。コンテキストリセット——コンテキストウィンドウを完全にクリアし新しいエージェントを開始し、前のエージェントのステートと次のステップを運ぶ構造化ハンドオフと組み合わせる——は両方の問題に対処します。

これは、同じエージェントが短縮された履歴で続けられるよう、会話の初期部分を要約する **圧縮**（**compaction**）とは異なります。圧縮は連続性を保ちますが、エージェントにクリーンスレートを与えないため、コンテキスト不安が依然として続く可能性があります。リセットはクリーンスレートを提供する代わりに、次のエージェントが作業をクリーンに引き継ぐのに十分なステートをハンドオフアーティファクトに持たせるコストがかかります。以前のテストでは、Claude Sonnet 4.5 がコンテキスト不安を強く示し、圧縮だけでは強い長時間タスク性能に不十分だったため、コンテキストリセットがハーネス設計に不可欠となりました。これは中核的な問題を解きますが、各ハーネス実行にオーケストレーションの複雑さ、トークンのオーバーヘッド、レイテンシを追加します。

以前に対処していない 2 つ目の問題は **自己評価** です。自分が生み出した作業を評価するよう求められると、エージェントは自信を持って作業を褒める傾向があります——人間の目には品質が明らかに凡庸なときでも。この問題は、検証可能なソフトウェアテストのようなバイナリチェックがないデザインのような主観的タスクで特に顕著です。レイアウトが洗練されているか平凡かは判断の問題で、エージェントは自分の仕事を採点するとき確実にポジティブに偏ります。

しかし、検証可能な outcome を持つタスクでも、エージェントは時に判断が悪く、タスク完了中の性能を妨げます。仕事をするエージェントと、それを判断するエージェントを分離することは、この問題に対処する強いレバーであることが証明されました。この分離はそれ自体で寛容さを即座に排除するわけではありません——エバリュエーターも LLM 生成出力に寛大になりがちな LLM です。しかし、スタンドアロンのエバリュエーターを懐疑的にチューニングする方が、生成者に自分の作業を批判的にさせるよりはるかに扱いやすく、その外部フィードバックが存在すれば、生成者は具体的に反復できる対象を得ます。

フロントエンドデザイン: 主観的な品質を採点可能にする

自己評価問題が最も見えやすいフロントエンドデザインから実験を始めました。介入がなければ、Claude は通常、技術的には機能するが視覚的に印象的でない、安全で予測可能なレイアウトに引き寄せられます。

フロントエンドデザイン向けに作ったハーネスは、2 つの洞察で形作られました。まず、美学は完全にスコアに還元できず——個人の趣味は常に異なります——それでもデザイン原則や好みをエンコードした採点基準で改善できます。「このデザインは美しいか？」は一貫して答えにくいですが、「これは私たちの良いデザインの原

則に従っているか?」は具体的に Claude に採点対象を与えます。次に、フロントエンドの生成と採点を分離することで、生成者をより強い出力へと駆動するフィードバックループを作れます。

これを念頭に置き、私は生成エージェントと評価エージェントの両方にプロンプトで与える 4 つの採点基準を書きました。

- **デザイン品質**: デザインは部品の集まりではなく、一貫した全体に感じられるか? ここでの強い仕事は、色、タイポグラフィ、レイアウト、画像、その他の詳細が組み合わさって特有のムードとアイデンティティを生むことを意味する。
- **独創性**: カスタムな決定の証拠があるか、それともテンプレートレイアウト、ライブラリデフォルト、AI 生成パターンか? 人間のデザイナーは意図的な創造的選択を認識すべきだ。変更されていないストックコンポーネント——あるいは白いカードの上の紫のグラデーションのような AI 生成のサイン——はここで失敗する。
- **クラフト**: 技術的実行——タイポグラフィの階層、間隔の一貫性、色の調和、コントラスト比。これは創造性チェックというより能力チェック。ほとんどの妥当な実装はデフォルトでここをうまくこなす——失敗は基礎が壊れていることを意味する。
- **機能性**: 美学とは独立した使いやすさ。ユーザーはインターフェースが何をするかを理解し、主要アクションを見つけ、推測なしにタスクを完了できるか?

デザイン品質と独創性を、クラフトと機能性より強調しました。Claude はデフォルトでクラフトと機能性で良いスコアを出していました——必要な技術能力はモデルに自然に備わっていたからです。しかしデザインと独創性については、Claude はしばしば良く言って平凡な出力を生み出していました。基準は非常に汎用的な「AI slop」パターンを明示的に罰し、デザインと独創性をより重く重み付けることで、モデルをより美学的な冒険へと押し出しました。

私は詳細なスコア内訳付きの few-shot 例でエバリュエーターをキャリブレーションしました。これによりエバリュエーターの判断は私の好みに合致し、反復間のスコアのドリフトを減らしました。

このループは Claude Agent SDK の上に構築し、オーケストレーションをシンプルに保ちました。生成エージェントがユーザープロンプトに基づいて HTML/CSS/JS のフロントエンドをまず作ります。エバリュエーターには Playwright MCP を与え、各基準を採点し詳細な批評を書く前に、ライブページと直接対話できるようにしました。実際には、エバリュエーターは自分でページをナビゲートし、スクリーンショットを取り、実装を注意深く研究してから評価を出しました。そのフィードバックは次の反復の入力として生成者に流れます。1 世代あたり 5〜15 回の反復を実行し、各反復で生成者はエバリュエーターの批評に応答してより特徴的な方向に押されました。エバリュエーターは静的スクリーンショットを採点するのではなくページを能動的にナビゲート

していたので、各サイクルには実時間がかかりました。完全な実行は最大 4 時間に及びました。また、生成者には各評価の後に戦略的決定を行うよう指示しました——スコアが順調に上がっているなら現在の方向を洗練し、アプローチが機能していないなら全く異なる美学にピボットする。

実行を通じて、エバリュエーターの評価は反復を経て改善しプラトーに達しましたが、余地は残っていました。一部の世代は漸進的に洗練されました。他は反復間で鋭い美学的転換を遂げました。

基準の言い回しは、私が完全には予想しなかった方法で生成者を操縦しました。「最良のデザインは美術館品質」のようなフレーズを含めると、デザインを特定の視覚的収束に押しやり、基準に関連するプロンプティング自体が出力の特性を直接形作ることを示唆しました。

スコアは一般に反復を経て改善しましたが、パターンは常にきれいに線形ではありませんでした。後の実装は全体として良い傾向がありましたが、私は定期的に最後のものより中間の反復を好むケースを見ました。実装の複雑性もラウンドを経て増す傾向があり、生成者はエバリュエーターのフィードバックに応じてより野心的な解に手を伸ばしました。最初の反復でも、出力はプロンプティング無しのベースラインより顕著に良く、基準とそれに関連する言語自体がエバリュエーターのフィードバックによるさらなる洗練の前に、モデルを汎用デフォルトから引き離していたことを示唆しました。

ある顕著な例では、モデルにオランダの美術館のウェブサイトを作らせるようプロンプトしました。9 回目の反復までに、架空の美術館のクリーンでダークテーマのランディングページを生み出しました。ページは視覚的に洗練されていましたが、概ね私の期待通りでした。それから 10 回目のサイクルで、アプローチを完全に廃棄し、サイトを空間体験として再構想しました——CSS パースペクティブでレンダリングされたチェッカーフロアの 3D 部屋、壁に自由形式で掛けられたアートワーク、スクロールやクリックではなくドアベースのギャラリー間ナビゲーション。これは、単一パスの生成ではこれまで見なかった種類の創造的飛躍でした。

フルスタックコーディングにスケールする

これらの発見を手にして、この GAN にインスパイアされたパターンをフルスタック開発に適用しました。ジェネレーター・エバリュエーターループは、コードレビューと QA がデザインエバリュエーターと同じ構造的役割を果たすソフトウェア開発ライフサイクルに自然にマッピングされます。

アーキテクチャ

以前の長時間ハーネス では、初期化エージェント、機能を 1 つずつ扱うコーディングエージェント、セッション間のコンテキストリセットで一貫したマルチセッションコーディングを解きました。コンテキストリセットは鍵となる解き口でした——ハーネスは前述の「コンテキスト不安」傾向を示す Sonnet 4.5 を使っていたからです。コンテキストリセットにまたがってうまく動くハーネスを作ることが、モデルをタスクに留めておく鍵でした。

Opus 4.5 はその振る舞いを概ね自分で解消したので、このハーネスからコンテキストリセットを完全に落とせました。エージェントは全ビルドを通じて 1 つの連続セッションとして動き、Claude Agent SDK の自動圧縮が途中でコンテキストの成長を扱いました。

この作業のために、元のハーネスの基盤の上に 3 エージェントシステムを作り、各エージェントが以前の実行で観察した特定のギャップに対処しました。システムには以下のエージェントペルソナが含まれます。

プランナー: 以前の長時間ハーネスは、ユーザーが詳細な仕様を前もって提供することを要求しました。このステップを自動化したかったので、シンプルな 1〜4 文のプロンプトを取り、完全な製品仕様に拡張するプランナーエージェントを作りました。スコープについて野心的であり、詳細な技術実装ではなく製品文脈と高レベルの技術設計にフォーカスするようプロンプトしました。この強調は、プランナーが前もって細かい技術詳細を指定し何かを間違えた場合、仕様のエラーが下流の実装にカスケードするという懸念から来ています。生成する成果物でエージェントを制約し、彼らが働きながら道を見つけ出させる方が賢明に思えました。また、プランナーには AI 機能を製品仕様に織り込む機会を見つけるよう依頼しました。(付録の例を参照。)

ジェネレーター: 以前のハーネスの「一度に 1 機能」アプローチはスコープ管理にうまく機能しました。ここでも同様のモデルを適用し、ジェネレーターに仕様から 1 機能ずつ取り上げてスプリントで働くよう指示しました。各スプリントは React、Vite、FastAPI、SQLite(後に PostgreSQL)スタックでアプリを実装し、ジェネレーターは QA に引き渡す前に各スプリントの終わりに自己評価するよう指示されました。バージョン管理用の git も持っていました。

エバリュエーター: 以前のハーネスからのアプリケーションはしばしば印象的に見えましたが、実際に使おうとすると実際のバグがありました。これらを捕まえるため、エバリュエーターは Playwright MCP を使ってユーザーのように走行中のアプリケーションをクリックして回り、UI 機能、API エンドポイント、データベースの状態をテストしました。そして各スプリントを、見つけたバグとフロントエンド実験をモデルにした——製品の深さ、機能性、視覚デザイン、コード品質をカバーするよう適応した——基準一式に対して採点しました。各基準には厳しいしきい値があり、1 つでもその下回れば、スプリントは失敗し、ジェネレーターは何が悪かったかの詳細なフィードバックを得ました。

各スプリントの前に、ジェネレーターとエバリュエーターはスプリント契約を交渉しました——コードが書かれる前に、そのチャンクの作業に対して「完了」がどう見えるかに合意するのです。これは製品仕様が意図的に高レベルだったためで、ユーザーストーリーとテスト可能な実装の間のギャップを橋渡しするステップが必要だったからです。ジェネレーターは何を作るか、どう成功を検証するかを提案し、エバリュエーターはジェネレーターが正しいものを作っているか確認するためにその提案をレビューしました。2 人は合意するまで反復しました。

通信はファイル経由で扱われました——あるエージェントがファイルを書き、別のエージェントがそれを読み、そのファイル内で応答するか、前のエージェントが次に読む新しいファイルで応答します。その後ジェネレーターは QA に仕事を引き渡す前に、合意された契約に対して構築しました。これは仕様への忠実性を保ちつ

つ、実装を早すぎる段階で過剰指定しないものでした。

ハーネスを走らせる

このハーネスの最初のバージョンには Claude Opus 4.5 を使い、ユーザープロンプトを完全なハーネスと単一エージェントシステムの両方で実行して比較しました。この実験を始めたときに私たちの最良のコーディングモデルだったので Opus 4.5 を使いました。

私はこんなプロンプトを書いてレトロビデオゲームメーカーを生成しました。

> *「レベルエディタ、スプライトエディタ、エンティティ動作、プレイ可能なテストモードを含む 2D レトロゲームメーカーを作って」*

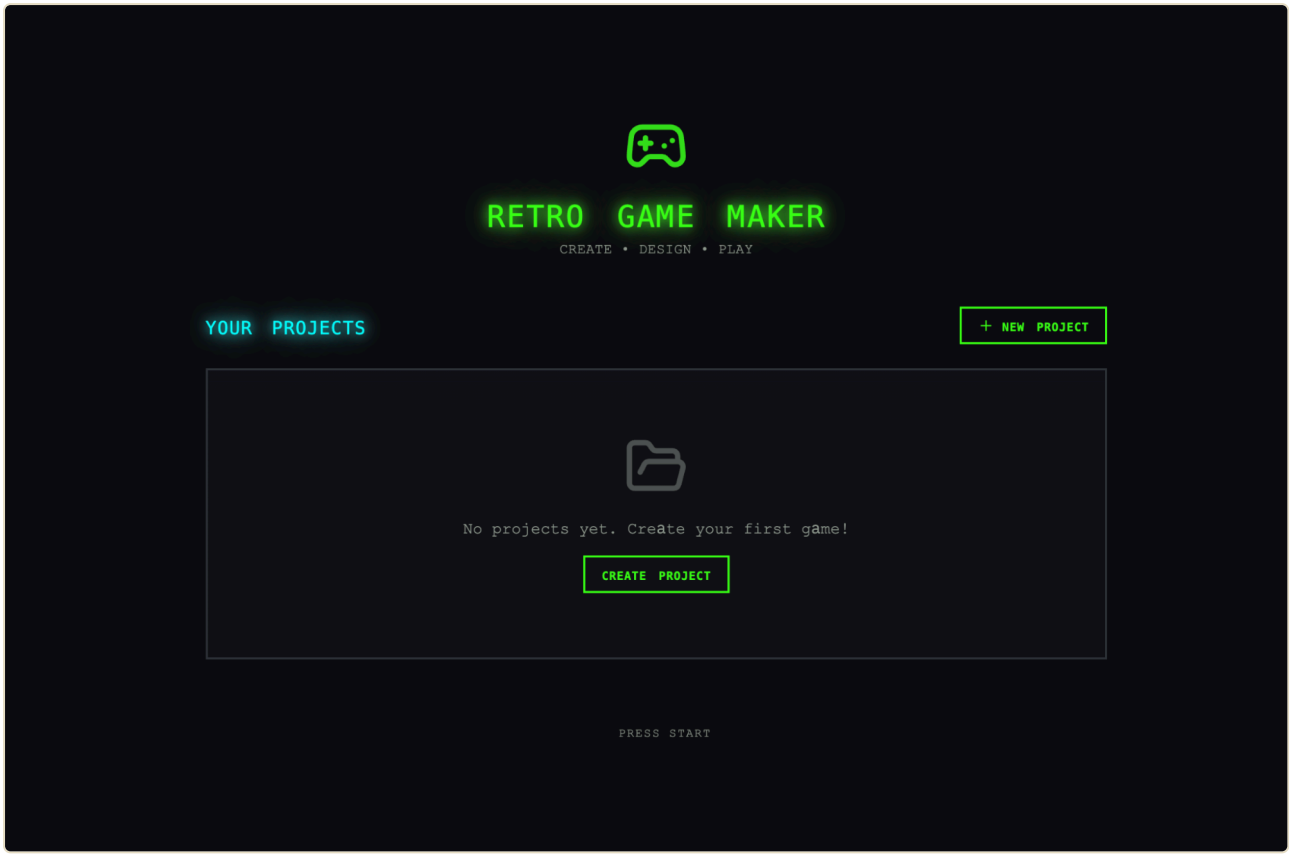
以下の表はハーネスタイプ、実行時間、総コストを示します。

ハーネス	所要時間	コスト
単独	20 分	$9
完全ハーネス	6 時間	$200

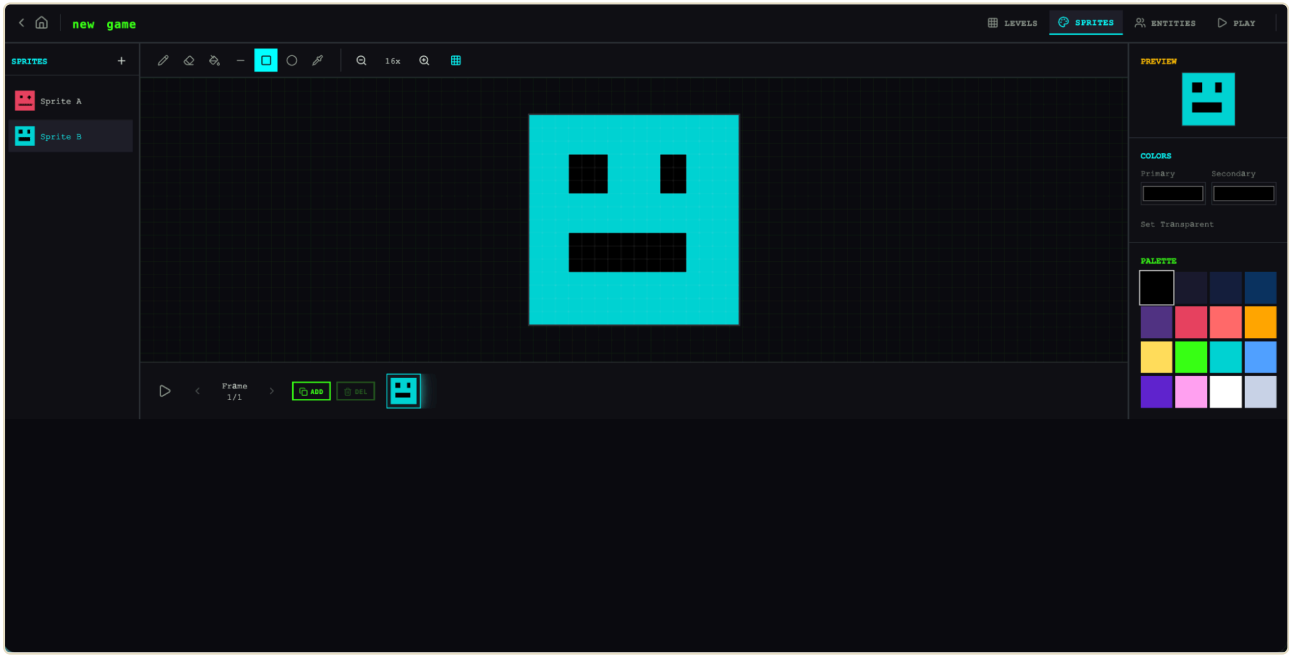
ハーネスは 20 倍超高価でしたが、出力品質の違いは即座に明らかでした。

レベルとその構成部品（スプライト、エンティティ、タイルレイアウト）を構築し、プレイボタンを押して実際にレベルをプレイできるインターフェースを期待していました。まず単独実行の出力を開きましたが、初期アプリケーションはそうした期待に沿っていました。

しかしクリックして回るうちに問題が見え始めました。レイアウトはスペースを無駄にし、固定高さのパネルがビューポートの大半を空けていました。ワークフローは硬直していました。レベルを populating するには先にスプライトとエンティティを作るよう促されましたが、UI には何もその順序を導くものがありませんでした。さらに重要なのは、実際のゲームが壊れていたことです。エンティティは画面上に現れましたが、入力に何も反応しませんでした。コードを掘り下げると、エンティティ定義とゲームランタイムの配線が壊れていることが分かりましたが、どこが壊れているかの表面的な指示は何もありませんでした。



単独ハーネスが作ったアプリを開いたときの初期画面。



単独ハーネスが作ったスプライトエディタ。

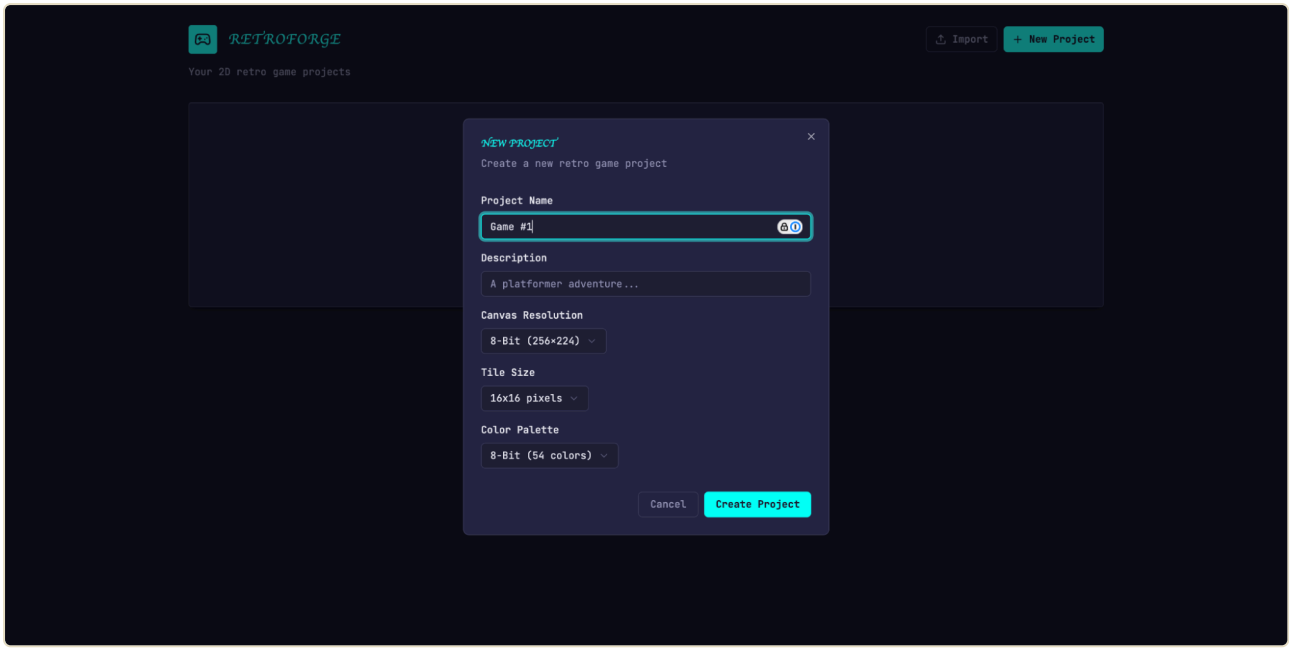
作ったレベルをプレイしようとしたが失敗。

単独実行を評価した後、ハーネス実行に注意を向けました。この実行は同じ 1 文プロンプトから始めましたが、プランナーステップがそのプロンプトを 10 スプリントにわたる 16 機能の仕様に拡張しました。単独実行が試みたものをはるかに超えていました。コアエディタとプレイモードに加えて、仕様はスプライトアニメーションシステム、動作テンプレート、効果音と音楽、AI 支援のスプライトジェネレータとレベルデザイナ、共有可能リンク付きのゲームエクスポートを要求しました。プランナーには フロントエンドデザイン skill へのアクセスを与え、仕様の一部としてアプリの視覚デザイン言語を作るのに読み使いました。各スプリントで、ジェネレーターとエバリュエーターはスプリントの具体的な実装詳細と完了検証に使うテスト可能な振る舞いを定義する契約を交渉しました。

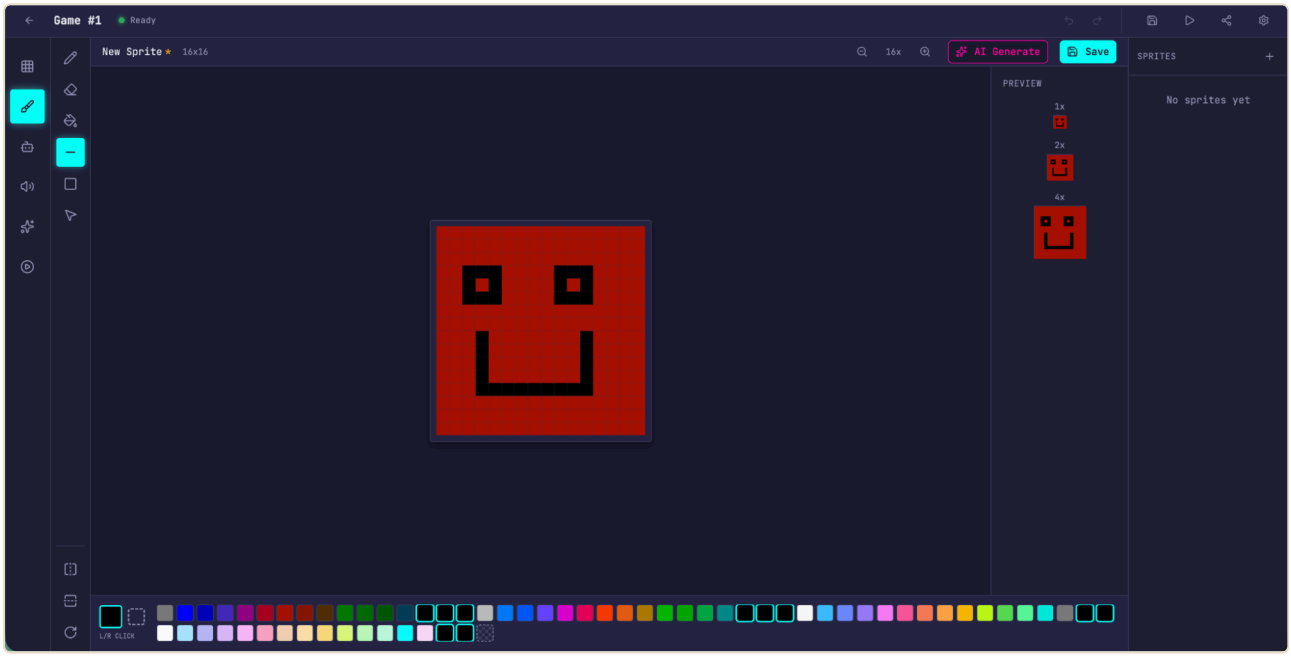
アプリは単独実行よりも即座に洗練さとスムーズさを示しました。キャンバスは完全なビューポートを使い、パネルは妥当にサイズ設定され、インターフェースは仕様からのデザイン方向を追う一貫した視覚的アイデンティティを持ちました。単独実行で見たクランキーさの一部は残っていました——ワークフローはそれでもレベルを populating しようとする前にスプライトとエンティティを構築すべきだと明確にしておらず、私が周りを探ってそれを理解しなければなりませんでした。これは、ハーネスが対処するように設計されたものではなく、ベースモデルの製品直感のギャップとして読めましたが、ハーネス内の的を絞った反復が出力品質をさらに改善するのを助ける場所を示唆しました。

エディタを通じて作業すると、新しい実行の単独に対する優位はより明らかになりました。スプライトエディタはより豊かで、よりフル機能で、よりクリーンなツールパレット、より良いカラーピッカー、より使えるズームコントロールを持っていました。

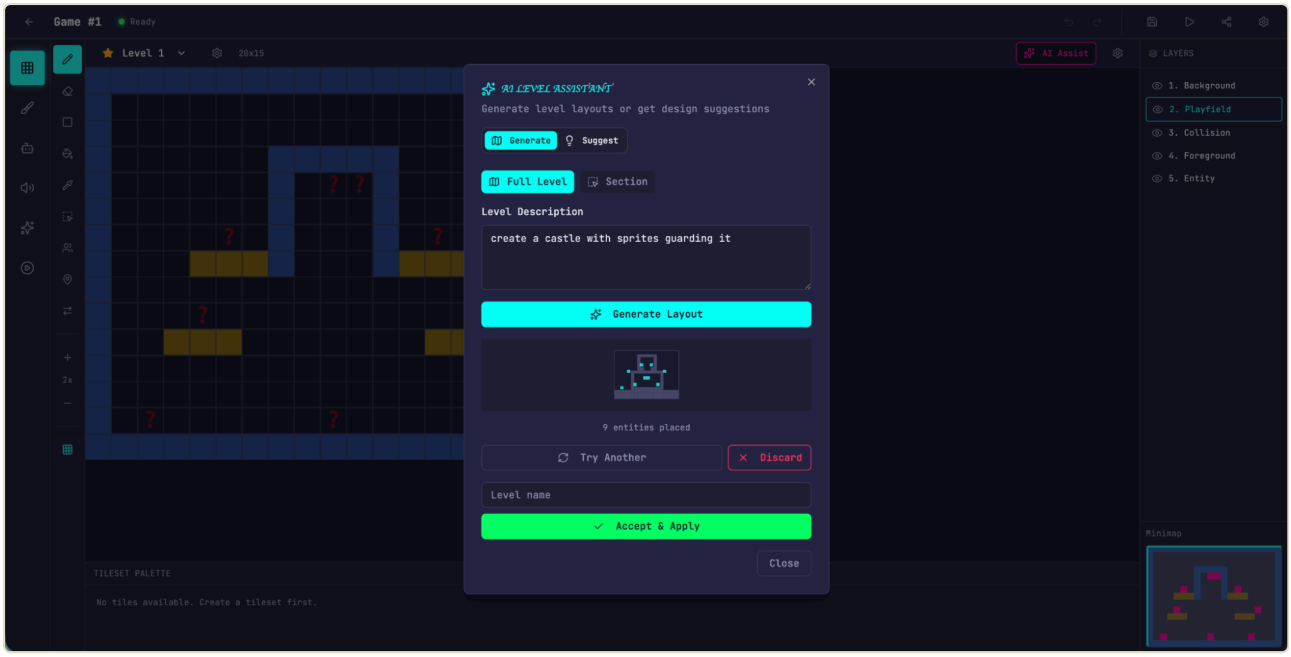
プランナーに仕様に AI 機能を織り込むよう依頼したので、アプリにはプロンプティングを通じてゲームの異なる部分を生成できる組み込みの Claude 連携も付いていました。これはワークフローを大幅に高速化しました。



完全ハーネスで作られたアプリで新しいゲームを作成する初期画面。



スプライトエディタはよりクリーンで使いやすく感じた。



組み込みの AI 機能を使ってレベルを生成。

生成したゲームをプレイ。

最大の違いはプレイモードにありました。実際にエンティティを動かしてゲームをプレイできたのです。物理には粗いエッジがありました——キャラクターがプラットフォームに跳び乗ったものの、重なって終わり、直感的に間違っていると感じました——しかしコアの機能は動きました。単独実行はそれを達成できませんでした。少し動き回った後、AI のゲームレベル構築にいくつかの制限にぶつかりました。跳び越えられない大きな壁があり、行き詰まりました。これは、ハーネスがアプリをさらに洗練するために扱える常識的改善とエッジケースがいくつかあることを示唆しました。

ログを読むと、エバリュエーターが実装を仕様に沿って保っていることが明らかでした。スプリントごとに、スプリント契約のテスト基準を走査し、Playwright を通じて動作中のアプリケーションを行使し、期待された振る舞いから逸脱したものすべてに対してバグを記録しました。契約は粒度が細かく——スプリント 3 だけでレベルエディタをカバーする 27 の基準がありました——エバリュエーターの発見は追加調査なしに行動できるほど具体的でした。以下の表はエバリュエーターが特定した問題の例を示します。

契約基準	エバリュエーターの発見
Rectangle fill ツールがクリック・ドラッグで選択タイルを矩形領域に埋められる	**FAIL** — ツールはドラッグ開始／終了点にのみタイルを置き、領域を埋めない。`fillRectangle` 関数は存在するが mouseUp で適切にトリガされていない。
ユーザーが配置したエンティティスポーンポイントを選択・削除できる	**FAIL** — `LevelEditor.tsx:892` の delete キーハンドラは `selection` と `selectedEntityId` の両方が設定されていることを要求するが、エンティティをクリックしても `selectedEntityId` のみが設定される。条件は `selection || (selectedEntityId && activeLayer === 'entity')` であるべき。
ユーザーが API 経由でアニメーションフレームを並べ替えできる	**FAIL** — `PUT /frames/reorder` ルートが `/{frame_id}` ルートの後に定義されている。FastAPI は `reorder` を frame_id 整数としてマッチしようとし、`422: unable to parse string as an integer` を返す。

エバリュエーターをこのレベルで性能を出させるには作業が必要でした。デフォルトでは、Claude は下手な QA エージェントです。初期の実行では、正当な問題を特定した後、それが大した問題ではないと自分を納得させて作業をそのまま承認する様子を見ました。また、エッジケースを探るより表層的にテストする傾向があり、より微妙なバグがしばしばすり抜けました。チューニングループは、エバリュエーターのログを読み、判断が私のものと乖離した例を見つけ、それらの問題を解くよう QA のプロンプトを更新することでした。私が妥当だと感じる方法で採点するようになるまで、この開発ループには数ラウンドかかりました。それでもハーネス出力はモデルの QA 能力の限界を示しました——小さなレイアウト問題、場所によって直感的でないインタラク

ション、エバリュエーターが徹底的に行使しなかったより深くネストされた機能における未発見バグ。さらなるチューニングで捕捉する検証余地は明らかにまだありました。しかし、アプリの中心機能がまったく動かなかった単独実行と比べれば、リフトは明白でした。

ハーネスを反復する

最初のハーネス結果のセットは励みになりましたが、それは嵩張り、遅く、高価でもありました。論理的な次のステップは、性能を劣化させずにハーネスを単純化する方法を見つけることでした。これは部分的には常識で、部分的にはより一般的な原則の関数でした——ハーネスのすべてのコンポーネントは、モデルが単独でできないことについての仮定をエンコードしており、これらの仮定はストレステストする価値があります。なぜなら、それらは間違っているかもしれないからで、モデルが改善するにつれて急速に古くなり得るからです。Building Effective Agents のブログ記事は、基礎となる考えを「可能な最もシンプルな解を見つけ、必要なときだけ複雑さを増す」としてフレーミングしており、これはエージェントハーネスを維持する人にとって一貫して現れるパターンです。

最初の単純化の試みで、ハーネスを抜本的に削ぎ落としていくつかの創造的な新しいアイデアを試しましたが、元の性能を再現できませんでした。どの部分が実際にハーネス設計の荷重を支えているのか、どのように支えているのかを判断することも難しくなりました。その経験に基づいて、1 つのコンポーネントを一度に取り除き、最終結果にどんな影響があるかをレビューする、より体系的なアプローチに移りました。

これらの反復サイクルを進めていた間に、Opus 4.6 もリリースされ、ハーネスの複雑性を減らす動機がさらに与えられました。4.6 は 4.5 より少ない足場で済むと期待するに足る理由がありました。ローンチブログ から:「[Opus 4.6 は] より注意深く計画し、エージェント型タスクをより長く維持し、より大きなコードベースでより確実に動作でき、自分のミスを捕まえるより良いコードレビューとデバッグスキルを持つ」。長文脈検索も大幅に改善されていました。これらはすべて、ハーネスが補完するために構築されてきた能力でした。

スプリント構造を取り除く

私はスプリント構造を完全に取り除くことから始めました。スプリント構造は、モデルが一貫して作業できるよう作業をチャンクに分解するのに役立っていました。Opus 4.6 の改善を考えると、この種の分解なしにモデルがネイティブに仕事を扱えると信じる理由がありました。

プランナーとエバリュエーターは両方とも残しました——どちらも明白な価値を加え続けていたからです。プランナーがないと、生成者はスコープを過小にします——生の指示を与えると、先に仕事を仕様化せずに構築を始め、プランナーがしたよりも機能の少ないアプリケーションを作ってしまうのです。

スプリント構造を取り除いて、エバリュエーターを実行の最後の 1 パスに移しました。モデルがはるかに有能になったので、エバリュエーターが特定の実行にどれだけ荷重を支えているかが変わり、その有用性はタスクがモデルが単独で確実にできることに対してどこに位置するかに依存しました。4.5 では、境界が近く、私たち

のビルドはジェネレーターが単独でうまくできることの縁にあり、エバリュエーターはビルド全体で意味のある問題を捕まえていました。4.6 では、モデルの生の能力が上がり、境界が外に移動しました。以前はエバリュエーターのチェックなしには一貫性を持って実装できなかったタスクが、今ではジェネレーターが単独でうまく扱えるものになることが多く、その境界内のタスクでは、エバリュエーターは不要なオーバーヘッドになりました。しかし、ジェネレーターの能力の縁にあるビルドの部分については、エバリュエーターは依然として実際のリフトを与え続けました。

実践的な含意は、エバリュエーターは固定の yes/no の決定ではないということです。タスクが現在のモデルが単独で確実に行うことを超えるときに、そのコストを支払う価値があります。

構造的単純化と並んで、ハーネスが各アプリに AI 機能をどう構築するかを改善するためのプロンプティングも追加しました——具体的には、生成者にツールを通じてアプリ自身の機能を駆動できる適切なエージェントを構築させることです。関連知識は最近のもので、Claude の訓練データが薄くしかカバーしていないため、実際の反復が必要でした。しかし十分にチューニングすれば、ジェネレーターはエージェントを正しく構築していました。

更新されたハーネスの結果

更新されたハーネスをテストに回すため、以下のプロンプトで Digital Audio Workstation（DAW）——楽曲の作曲、録音、ミキシングのための音楽制作プログラム——を生成しました。

> *「Web Audio API を使ってブラウザで完全機能の DAW を作って」*

実行はまだ長く高価で、約 4 時間、トークンコストで $124 でした。

時間の大半はビルダーにかかり、Opus 4.5 が必要としていたスプリント分解なしに 2 時間以上一貫して動きました。

エージェント & フェーズ	所要時間	コスト
プランナー	4.7 分	$0.46
ビルド（ラウンド 1）	2 時間 7 分	$71.08
QA（ラウンド 1）	8.8 分	$3.24
ビルド（ラウンド 2）	1 時間 2 分	$36.89
QA（ラウンド 2）	6.8 分	$3.09
ビルド（ラウンド 3）	10.9 分	$5.88
QA（ラウンド 3）	9.6 分	$4.06
合計 V2 ハーネス	**3 時間 50 分**	**$124.70**

以前のハーネスと同様、プランナーは 1 行プロンプトを完全な仕様に拡張しました。ログから、生成モデルがアプリとエージェント設計の計画、エージェントの配線、QA に引き渡す前のテストをうまくこなしているのが見えました。

とはいえ、QA エージェントは依然として実際のギャップを捕まえました。最初のラウンドのフィードバックで、こう指摘しました。

> これは優れたデザイン忠実性、堅固な AI エージェント、良いバックエンドを持つ強いアプリである。主要な失敗点は機能網羅性（Feature Completeness）——アプリは印象的に見え、AI 連携はうまく動くが、いくつかのコア DAW 機能はインタラクティブな深さのない表示専用である：クリップはタイムライン上でドラッグ／移動できず、楽器 UI パネル（シンセのつまみ、ドラムパッド）がなく、ビジュアルエフェクトエディタ（EQ カーブ、コンプレッサーメーター）もない。これらはエッジケースではない——DAW を使用可能にするコアインタラクションであり、仕様は明示的にそれらを要求している。

2 回目のラウンドフィードバックでも、いくつかの機能ギャップを捕まえました。

> 残りのギャップ: - オーディオ録音はまだスタブのみ（ボタンは切り替わるがマイク録音しない） - クリップのエッジドラッグによるリサイズとクリップ分割が実装されていない - エフェクト可視化は数値スライダーで、グラフィカルではない（EQ カーブなし）

ジェネレーターは自分のデバイスに任されると詳細を見落としたり機能をスタブにしたりしがちで、QA は依然としてジェネレーターが修正する最後のマイル問題を捕まえるのに価値を加えました。

プロンプトに基づいて、メロディ、ハーモニー、ドラムパターンを作り、それらを曲にアレンジし、途中で統合エージェントから助けを得られるプログラムを期待していました。以下の動画は結果を示します。

アプリはプロフェッショナルな音楽制作プログラムにはほど遠く、エージェントの楽曲作曲スキルは明らかにさらなる作業が必要です。加えて、Claude は実際には聞こえないので、音楽的趣味に関する QA フィードバックループの効果が低くなっていました。

しかし最終アプリは、機能する音楽制作プログラムのコアピースすべてを持っていました——ブラウザで動く動作するアレンジメントビュー、ミキサー、トランスポート。それを超えて、完全にプロンプティングだけで短い曲スニペットをまとめられました——エージェントはテンポとキーを設定し、メロディを置き、ドラムトラックを構築し、ミキサーレベルを調整し、リバーブを追加しました。楽曲作曲のコアプリミティブが存在し、エージェントはそれらを自律的に駆動でき、ツールを使ってエンドツーエンドで単純な制作を作れました。完璧なピッチにはまだ至らないと言えるかもしれませんが、そこに近づいています。

今後の展望

モデルが改善し続けるにつれ、より長くより複雑なタスクで機能できると大まかに期待できます。ある場合には、モデルを取り巻くスキャフォールドが時間と共に重要でなくなり、開発者は次のモデルを待って特定の問題が自然に解決するのを見られるかもしれません。一方で、モデルが良くなるほど、ベースラインでモデルができることを超える複雑なタスクを達成できるハーネスを開発する余地が広がります。

これを念頭に置くと、この作業から持ち越す価値のあるいくつかの教訓があります。構築対象のモデルで実験し、現実的な問題でのトレースを読み、望む結果を達成するためにその性能をチューニングすることは常に良い慣行です。より複雑なタスクに取り組むとき、タスクを分解し問題の各側面に専門エージェントを適用することで余地が得られることがあります。そして新しいモデルが来たら、一般にハーネスを再検討し、もはや性能を支えていない部分を削り、以前は不可能だったかもしれないより大きな能力を達成する新しい部分を追加するのが良い慣行です。

この作業から、モデルが改善してもハーネスの興味深い組み合わせの空間は縮まない、というのが私の確信です。代わりにそれは動き、AI エンジニアにとっての興味深い仕事は、次の新奇な組み合わせを見つけ続けることです。

謝辞

本作業への貢献に対し、Mike Krieger、Michael Agaby、Justin Young、Jeremy Hadfield、David Hershey、Julius Tarng、Xiaoyi Zhang、Barry Zhang、Orowa Sidker、Michael Tingley、Ibrahim Madha、Martina Long、Canyon Robbins に特別に感謝します。

投稿の形に関する助けに Jake Eaton、Alyssa Leonard、Stef Sequeira にも感謝します。

付録

プランナーエージェントが生成した計画例。

RetroForge – 2D Retro Game Maker

Overview
RetroForge is a web-based creative studio for designing and building 2D retro-style video games. It combines the nostalgic charm of classic 8-bit and 16-bit game aesthetics with modern, intuitive editing tools—enabling anyone from hobbyist creators to indie developers to bring their game ideas to life without writing traditional code.

The platform provides four integrated creative modules: a tile-based Level Editor for designing game worlds, a pixel-art Sprite Editor for crafting visual assets, a visual Entity Behavior system for defining game logic, and an instant Playable Test Mode for real-time gameplay testing. By weaving AI assistance throughout (powered by Claude), RetroForge accelerates the creative process—helping users generate sprites, design levels, and configure behaviors through natural language interaction.

RetroForge targets creators who love retro gaming aesthetics but want modern conveniences. Whether recreating the platformers, RPGs, or action games of their childhood, or inventing entirely new experiences within retro constraints, users can prototype rapidly, iterate visually, and share their creations with others.

Features
1. Project Dashboard & Management
The Project Dashboard is the home base for all creative work in RetroForge. Users need a clear, organized way to manage their game projects—creating new ones, returning to works-in-progress, and understanding what each project contains at a glance.

User Stories: As a user, I want to:

- Create a new game project with a name and description, so that I can begin designing my game
- See all my existing projects displayed as visual cards showing the project name, last modified date, and a thumbnail preview, so that I can quickly find and continue my work
- Open any project to enter the full game editor workspace, so that I can work on my game
- Delete projects I no longer need, with a confirmation dialog to prevent accidents, so that I can keep my workspace organized
- Duplicate an existing project as a starting point for a new game, so that I can reuse my previous work

Project Data Model: Each project contains:

Project metadata (name, description, created/modified timestamps)
Canvas settings (resolution: e.g., 256x224, 320x240, or 160x144)
Tile size configuration (8x8, 16x16, or 32x32 pixels)
Color palette selection
All associated sprites, tilesets, levels, and entity definitions

...