2026 EDITION

AI AGENT ENGINEERING · COMPLETE GUIDE

# ハーネスエンジニアリング完全攻略

AIエージェント時代の新常識

Constrain　Inform　Verify　Correct


by バイブコーディング テイラー

@shuntailor · shuntailor.net


130+ sources · 23 chapters · Free distribution

ハーネスエンジニアリング完全攻略
AIエージェント時代の新常識

---

初版発行: 2026年3月
著者: shuntailor (テイラーのアジト)

Threads: @shuntailor
ブログ: shuntailor.net

---



本文の情報は2026年3月時点のものです。
AI分野は変化が速いため、最新情報は各公式ドキュメントを参照してください。

この本に引用された全ソースは付録Cに整理されています。

FROM THE AUTHOR

# 著者の言葉

この本を書き始めたのは、自分自身が「バイブコーダー」として壁にぶつかったからだ。

Lovableで作ったアプリが数千行を超えた瞬間、エージェントの挙動が不安定になり始めた。同じプロンプトを入れても毎回違う結果が出た。ある日は動いていたコードが翌日には壊れた。

何が悪いのかわからなかった。

そのとき出会ったのがMitchell HashimotoのブログとOpenAIのレポートだった。「問題はモデルではなく、環境（ハーネス）だ。」その一言が全てを変えた。

ハーネスエンジニアリングを学んで、AGENTS.mdを書き始めて、Hooksを設定して、Evalループを組んだ。エージェントの挙動が劇的に安定した。

この本はその学びを体系化したものだ。130以上の公式ソースを読み込み、実際のコードベースで試した内容だけを書いた。

バイブコーディングで最初のアプリを作った喜びを覚えているあなたへ。その先に進む地図が、この本だ。

**バイブコーディング テイラー**

@shuntailor

TABLE OF CONTENTS

# 目次

# 00

# プロローグ: なぜこの本なのか

> エージェントがミスをするたびに、そのミスを二度と起こさないようにする仕組みを設計する。
>
> — Mitchell Hashimoto — 'My AI Adoption Journey', 2026年2月5日

## 同じAI、違う結果

2026年2月、LangChainが一つの実験結果を公開した。

GPT-5.2-Codex。モデルは同一だ。一文字も変えていない。変わったのはモデルの外側の環境だけだった。システムプロンプト、ツール構成、エラー復旧ループ、コンテキスト管理戦略。これらをまとめて「ハーネス」と呼ぶ。

TerminalBench 2.0のスコアが52.8%から66.5%に上がった。リーダーボード順位に換算すると約30位から5位。同じモデルだ。コード1行も触っていない。ハーネスだけを変えただけだ。

**KEY INSIGHT**

同じモデルがハーネスによって33位にも5位にもなる。これがこの本全体を貫く命題だ: モデル選択よりハーネス設計が成果をより大きく左右する。

同時期にOpenAIが発表した数字も興味深い。100万行を超えるプロダクションコード。人間が直接書いたコードは0行。5ヶ月。エンジニア3-7名。彼らが5ヶ月間やったことはコーディングではなかった。ハーネスを設計し磨き上げた。マイクロVM隔離、自動テストループ、エラーフィードバック体系。モデルに「何をさせるか」ではなく「どんな環境で働かせるか」を設計したのだ。

Stripeは毎週1,000件以上のPRをエージェントが自動でマージしている。これもモデルが優れているからではない。コードレビューハーネス、テストゲート、ロールバック機構が隙なく敷かれているからだ。

## 核心テーゼ

この本は一つの文の上に建てられている。

**"モデルよりハーネスが成果を決める。"**

2025年まで業界の関心は「どのモデルがより優れているか」に集中していた。GPT-4 vs. Claude 3.5 vs. Gemini。ベンチマーク1-2%の差に一喜一憂していた。しかし2026年、ゲームのルールが変わった。

同じモデルでもハーネスが良ければ上位5位以内に入る。逆に、最高のモデルでもハーネスが貧弱なら30位

圏外に押し出される。モデルは上方平準化された。差を生むのはモデルの外側だ。

2026年2月5日、TerraformとVagrantを作ったMitchell Hashimotoがこの概念に名前を付けた。「ハーネスエンジニアリング（Harness Engineering）」。馬のエネルギーを方向に変える馬具（harness）のように、AIモデルの知能を安定した結果に変える環境を設計する技術だ。

この名前が生まれて2週間でOpenAIが公式に採用した。Anthropic、LangChain、Martin Fowler、Philipp Schmidが相次いで関連記事を発表した。2026年3月現在、ハーネスエンジニアリングはAI開発で最もホットなキーワードだ。

---

### この本があなたに与えるもの

この本はハーネスエンジニアリングを最初から最後まで扱う。130以上の公式ソースを分析した。Hashimoto原文、OpenAI公式発表、Anthropic技術ブログ、LangChain実験レポート、Martin Fowlerの分析、arXiv論文まで。散らばった知識を一つの体系にまとめたのがこの本だ。

- 23チャプター＋3つの付録、約255ページ
- 130以上の公式ソースに基づく — ブログの感想文ではなく一次ソース分析
- 4大プラットフォーム（Claude Code、Codex、Cursor、Devin）ハーネス完全比較
- コピーしてすぐ使えるAGENTS.md、Evalテンプレート付き
- 1章から21章まで一つのプロジェクト（TaskFlow）が進化するランニングケース

読み終えるとこうなる。どのAIエージェントを使っても、「モデルを変えようか」の代わりに「環境をどう設計するか」をまず問うようになる。そしてその問いに対する答えを自ら設計できるようになる。



## この本の構成

6つのパートに分かれている。「なぜ」から始まり「何を」「どのように」「比較」「運用」「実習」へと続く。各パートは独立して読むこともできるが、順番に読めば一つの物語になる。

| パート | テーマ | 核心的な問い |
|---|---|---|
| パート1 | なぜ今なのか | バイブコーディングの限界は何で、なぜハーネスが必要なのか？ |
| パート2 | 4つの柱 | Constrain・Inform・Verify・Correctとは何か？ |
| パート3 | 実践ハーネス設計 | AGENTS.md、ツール、コンテキストをどう設計するか？ |
| パート4 | プラットフォーム別比較 | Claude Code、Codex、Cursor、Devinの違いは？ |
| パート5 | 運用と最適化 | Eval、マルチエージェント、セキュリティ、コストはどう管理するか？ |

| | | |
|---|---|---|
| パート6 | 実践プロジェクト | 実際に自分だけのハーネスをどう構築するか？ |

## 読み方

3つのルートがある。自分に合ったものを選べばいい。

**順番通りに読む**
ハーネスエンジニアリングに初めて触れる人。パート1から着実に。理論→実践→深化の自然な流れに沿って進む。

**必要なパートだけ**
すでにエージェントを運用中の人。パート4（プラットフォーム比較）やパート5（Eval、セキュリティ）から直接必要な部分を選んで読む。

**実習から始める**
手を動かして学ぶタイプ。パート6の実践プロジェクト（19-21章）をまず追体験してから、気になる理論は該当チャプターに戻って補う。

**リファレンスとして活用**
すでに概念を知っている人。付録のソースリストとテンプレートをリファレンス文書として使う。必要な時に取り出して参照する。

## ランニングケース: TaskFlow

この本には一つのプロジェクトが最初から最後まで一緒に進む。「TaskFlow」というタスク管理Webアプリだ。

1章でバイブコーディングによりMVPをサクッと作る。動作はするがテストもなく、セキュリティも気にしていない状態だ。チャプターが進むにつれてハーネスを一つずつ追加していく。4章で制約条件を、5章でコンテキストを、6章で検証体系を、7章でエラー復旧を付ける。21章頃には、TaskFlowはプロダクションに出しても大丈夫なプロジェクトになっている。

**RUNNING CASE**
TaskFlowのコードは各チャプター末の「TaskFlow進捗状況」セクションで確認できる。バイブコーディングMVPから出発して完全なハーネスフレームワークへ進化する過程を21チャプターにわたって直接追跡する。

---

著者shuntailorはAIエージェントとハーネス設計を研究するエンジニアでありコンテンツクリエイターだ。Threads(@shuntailor)でAI開発トレンドを共有しており、この本は130以上の公式ソースを分析・整理した成果物だ。

さあ、始めよう。なぜバイブコーディングだけでは不十分なのか、1章で語る。

PART 1

# なぜ今なのか

バイブコーディングの限界とハーネスエンジニアリングの登場

# 01

# バイブコーディングだけでは不十分な理由

## バイブコーディングが開いた世界

金曜日の夜、TaskFlowというタスク管理アプリを作ることにした。Claudeに「タスクリストを管理できるWebアプリを作って」と入力した。3分後、画面にきれいなTo-Doアプリが動いていた。ドラッグ＆ドロップもでき、ダークモードも適用されていた。コードを1行も書いていない。

こうした体験をした人が2025年の1年間だけで数十万人生まれた。2025年2月、Andrej Karpathyがツイートを一つ投稿した。

> There's a new kind of coding I call 'vibe coding', where you fully give in to the vibes, embrace exponentials, and forget that the code even exists.
>
> — Andrej Karpathy, 2025年2月

コードを読みもしない。コードが存在するという事実自体を忘れてもいい。AIにフィーリングだけ伝えればいい。この宣言は開発文化の盤面を変えた。

バイブコーディングがもたらした変化は単なる生産性向上ではなかった。コーディング経験のないデザイナー、マーケター、企画者も自分のアイデアを直接プロトタイプとして実装できるようになった。Lovable、Bolt、Replit Agentのようなツールが登場し「コーディングの民主化」が現実になった。TaskFlowのようなアプリを作るのに必要な時間が数週間から数時間に短縮された。

### プロトタイプとプロダクションの間の谷

問題はその次だった。TaskFlowにユーザー認証を付けようとした。メールログイン、パスワードリセット、セッション管理。AIが作ってくれたコードは動作したが、パスワードを平文で保存していた。セッショントークンに有効期限がなかった。

「動くこと」と「信頼できること」の間に深い谷があった。

バイブコーディングは0から1に行くには圧倒的だ。個人プロジェクト、ハッカソンプロトタイプ、週末サイドプロジェクトではこれ以上効率的な方法を見つけるのは難しい。しかしその1を10に、100にスケールさせようとした瞬間、壁が現れる。

## 壁にぶつかる瞬間

## 「目視確認」の限界

バイブコーディングの検証方式は「目視確認（human eyeballing）」だ。AIがコードを生成すると、人間が目で結果を確認し「大丈夫そうなら」先に進む。10行の関数なら問題ない。しかしTaskFlowが500行、1,000行、5,000行に膨らんだら？

エージェントが一度に200行のコードを出力するとき、その中に潜むSQLインジェクション脆弱性やレースコンディションを目で見つけられるだろうか。2025-2026年のセキュリティ分析結果、AI生成コードの約45%でセキュリティ脆弱性が発見された。動作するかではなく安全かどうかが問題だった。

> **WARNING**
> AIが書いたコードが「動く」ということは「安全だ」という意味ではない。バイブコーディングで最も危険な錯覚は「動いてるから大丈夫だろう」だ。

## 再現不可能な結果

TaskFlowにタグ機能を追加してほしいとリクエストした。月曜日にはタグを配列で実装し、水曜日に同じリクエストをすると別のTagモデルを作って多対多リレーションで実装した。金曜日にはまた別のアプローチが出てきた。

同じプロンプトを3回入力すれば3つの異なるコードが出てくる。LLMは確率的に次のトークンを生成するため、同一の入力に対して同一の出力を保証しない。アーキテクチャの決定がセッションごとに変わり、命名規則がバラバラになり、エラーハンドリング戦略がファイルごとに異なる。

プロトタイプではこれは問題にならない。コードが10,000行を超えた瞬間、一貫性のないコードベースはメンテナンス不可能な状態になる。バイブコーディングには「テスト」がないことが、ここで致命的に作用する。

## コンテキスト喪失と技術的負債

TaskFlowの開発を3週目に続けていた。ある日エージェントが既存のAPI構造を完全に無視したコードを生成した。初週に設計したRESTfulエンドポイント規則を3週目のエージェントはまったく記憶していなかった。

コンテキストウィンドウには限界がある。セッションが長くなったり新しいセッションを始めたりすると、エージェントは以前の設計決定を忘れる。ドキュメント化されていない決定が積み重なり、誰も「なぜこう実装したのか」わからないコードが増えていく。これがバイブコーディングが生む技術的負債（technical debt）だ。

1つのファイルを直すと別のファイルが壊れる。それを直すとまた別の箇所が壊れる。この終わりなき修正ループに一度でもハマったことがある人なら、どこかで根本的に間違っていることに気づいているはずだ。

| 次元 | バイブコーディング | ハーネスエンジニアリング |
|---|---|---|
| アプローチ | 非公式的、対話型プロンプティング | 体系的インフラ設計 |
| 検証方式 | 人間が目で確認 | 自動テスト + Evalパイプライン |
| 適用範囲 | プロトタイプ、個人プロジェクト | プロダクション、チーム開発、高リスクシステム |
| スケール | 単独開発者、短期セッション | チーム、数ヶ月のコードベース |
| 品質保証 | 「動けばOK」 | 自動化されたエージェント相互レビュー |
| コンテキスト管理 | セッションに依存 | AGENTS.md、メモリアーキテクチャ |

## 直感からシステムへ

ここまで読むと「バイブコーディングは役に立たない」という結論に向かうようだが、そうではない。

バイブコーディングは入門であり、目的地ではない。

バイブコーディングを通じて私たちはAIエージェントがどう動作するかについての直感を養った。どのプロンプトが良い結果を生み、どの状況でAIが混乱に陥るかを経験で知った。この直感がなければハーネスを設計することもできない。

Martin Fowlerのブログに寄稿したBirgitta Bockeler（Thoughtworks）がこの視点を明快にまとめた。

> バイブコーディングは「コーディングの民主化」だった。誰でも作れるようになった。しかし「ちゃんと動くものを作ること」には依然としてエンジニアリングが必要だ。そのエンジニアリングの対象が「コード」から「環境（ハーネス）」に変わったのだ。
>
> — Birgitta Bockeler, Martin Fowler's blog, 2026年2月

対象が変わっただけで、エンジニアリングの必要性は消えていない。コードを直接書く代わりに、AIがコードをうまく書ける環境を設計する。それがハーネスエンジニアリングであり、この本で扱おうとする核心テーマだ。

### バイブコーダーからハーネスエンジニアへ

転換の核心は視点の変化だ。「AIに何をさせるか」から「AIがうまく動作するにはどんな環境が必要か」へ問いが変わる。

| 以前の問い（バイブコーディング） | 以後の問い（ハーネスエンジニアリング） |
|---|---|
| どのモデルが一番いいか？ | どんな環境を設計すべきか？ |
| プロンプトをどう書くか？ | コンテキストをどう管理するか？ |
| AIがうまく作ってくれたらいいな... | AIがミスしてもシステムがキャッチする |
| 成果物 = モデルの能力 | 成果物 = ハーネスの品質 |

TaskFlowの話に戻ろう。バイブコーディングで作ったTaskFlow MVPはデモ用には十分だった。しかし実際のユーザーにデプロイするには、認証セキュリティ、データ整合性、エラー復旧、テストカバレッジ——これらすべてが必要だった。これらを手作業で解決するのではなく、エージェントが自ら守るようにするシステムを作ること。それがこれからこの本で学ぶ内容だ。

**POINT**
あなたがバイブコーダーなら、まずやるべきことはバイブコーディングを捨てることではなく、その上に検証レイヤーを積み重ねることだ。AGENTS.md 1枚を書くことから始められる。

バイブコーディングの限界を感じたなら、解決策を定義した人の話を聞いてみよう。Mitchell Hashimotoが「harness engineering」という名前を付けるまでの旅路だ。

# 02

# Agent = Model + Harness

## 命名者 Mitchell Hashimoto

Terraform、Vagrant、Consul。インフラ自動化分野で働く人ならこれらの名前を知らないはずがない。すべてMitchell Hashimotoが作った。HashiCorpを創業し数十億ドル企業に育てた後に退社し、現在はGhosttyというターミナルエミュレーターを作っている。

2026年2月5日、彼がブログに「My AI Adoption Journey」という記事を投稿した。Ghostty開発にAIエージェントを導入する中で経験した試行錯誤を記録した記事だった。この投稿から一つの用語が誕生した。

> I don't know if there is a broad industry-accepted term for this yet, but I've grown to calling this 'harness engineering.' It is the idea that anytime you find an agent makes a mistake, you take the time to engineer a solution such that the agent never makes that mistake again.
>
> — Mitchell Hashimoto, 'My AI Adoption Journey', 2026年2月5日

「業界で通用する用語があるかどうかはわからないが、私はこれを'ハーネスエンジニアリング'と呼ぶことにした。」彼の表現は謙虚だったが、この言葉は爆発的に広まった。

### Ghostty AGENTS.mdの実態

HashimotoはGhosttyプロジェクトにAGENTS.mdというファイルを用意した。このファイルの構造がユニークだ。過去にエージェントが犯したミス1件につき、そのミスを防止するルール1行が追加される。

markdown

```
# AGENTS.md — Ghostty Project
## Rules
- Never modify files in src/core/ without explicit permission
- Always run `zig build test` before committing
- Use snake_case for all function names

## Past Failures → Prevention
- Do not use `std.mem.alloc` directly → use arena allocator
- Config parsing: always validate UTF-8 before processing
- Terminal escape sequences: test with all 6 major terminals
```

エージェントが直接メモリ割り当てを使用してメモリリークを引き起こしたことがある。→ arena allocatorを使えというルールが追加された。UTF-8検証なしに設定ファイルをパースしてクラッシュが発生した。→ 必ずUTF-8を検証せよというルールが追加された。失敗から学んだ防止策が1行ずつ積み重なり、エージェ

ントの作業環境そのものが進化する。

Hashimotoはこうまとめた。

> エージェントがミスをするたびに、そのミスを二度と起こさないようにする仕組みを設計する。それがハーネスエンジニアリングだ。
>
> — Mitchell Hashimoto

同月、OpenAIが「Harness Engineering: Leveraging Codex in an Agent-First World」という記事を公式に発表し、この用語を採用した。OpenAIの発表は単なる用語の使用ではなかった。「100万行以上のプロダクションコードを、人間が1行も書かずに生成した。核心はモデルではなくハーネスだった。」ハーネスエンジニアリングはあっという間に業界の共通言語になった。

---

## Agent = Model + Harness の公式

**Agent = Model + Harness**

生のLLMはエージェントではない。GPT-5であろうとClaude Opusであろうと、モデルそのものは「知能」しか持っていない。状態を記憶できず、ツールを使えず、ミスを検知できず、自分自身を制御できない。

ハーネスがこれらすべてを付与する。状態管理、ツール実行、フィードバックループ、制約条件。ハーネスがモデルを包んで初めてエージェントになる。



### 各要素がない場合どうなるか

この公式の各要素が欠けたときに何が起きるかを見れば、なぜハーネスが必須なのかが明確になる。

| 欠けた要素 | 現象 | 結果 |
|---|---|---|
| Modelのみでハーネスなし | APIを直接呼び出し、チャットのみ可能 | バイブコーディングレベルで停滞 |
| ハーネスのみでModelなし | ツール・ルールはあるが推論能力なし | ルールベース自動化（従来のスクリプト） |
| Model + 部分的ハーネス | ツールはあるが検証・修正なし | 動作するが信頼不可 |
| Model + 完全なハーネス | 推論 + ツール + 検証 + 修正 | プロダクションレベルのエージェント |

### コンピュータアーキテクチャの比喩

Hugging FaceのPhilipp Schmidがこの関係をコンピュータに例えて説明した。業界で広く引用されている比喩だ。

| コンピュータ | AIエージェント | 役割 |
|---|---|---|
| CPU | 基盤モデル（LLM） | 推論能力、思考力 |
| RAM | コンテキストウィンドウ | 揮発性ワーキングメモリ |
| OS | エージェントハーネス | コンテキスト管理、ブートシーケンス、ライフサイクル管理 |
| Application | エージェント | ユーザー固有のロジックとワークフロー |

この比喩で核心的な洞察が一つある。同じCPUを搭載しても、OSが違えばパフォーマンスが変わる。同一のハードウェアでLinuxとWindowsの性能が異なるように、同一のLLMでもハーネスが異なればアウトプットの品質がまったく変わる。TerminalBench 2.0で同じOpus 4.6モデルがハーネスによって33位にも5位にもなったのがまさにこの原理だ。

TaskFlowに戻って考えてみよう。1章でバイブコーディングで作ったTaskFlow MVPがある。Modelはすでにある（Claude）。しかしハーネスがない。テスト自動化も、コードスタイルルールも、エラー復旧ロジックも、コンテキスト管理体系もない。今のTaskFlowは「Modelのみでハーネスなし」の状態だ。バイブコーディングレベルにとどまっている理由がここにある。



## 4つの機能的な柱

ではハーネスは具体的に何をするのか？OpenAI、Martin Fowler、Anthropicの分類を総合すると、ハーネスの機能は4つの柱に集約される。

**Constrain（制約）**
エージェントが「できること」を制限する。サンドボックス、ツール許可リスト、ファイルパス制限、コスト上限。解空間を狭めるほどエージェントの精度は上がる。

**Inform（情報提供）**
エージェントに「何をすべきか」を伝える。AGENTS.md、CLAUDE.md、プロジェクト文書の整備。適時に必要な最小限の高品質コンテキストを提供する。

**Verify（検証）**
エージェントが「正しく実行したか」を確認する。Evalパイプライン、自動テスト、セルフ検証ループ。確率的出力に対する許容範囲ベースの検証。

**Correct（修正）**
エージェントが「ミスしたとき」に復旧する。エラーリカバリ、リトライロジック、Human-in-the-loop。失敗を学習に転換するフィードバックループ。

この4つの柱はパート2でそれぞれ1チャプターずつ深く扱う。今は全体構造を頭の中に捉えておけばいい。重要なのは、この4つが独立ではなく互いに結びついているということだ。Constrainが解空間を狭めればVerifyの負担が減り、Informが正確であるほどCorrectが作動する場面が少なくなる。

### 馬のハーネスからAIのハーネスへ

「ハーネス」という言葉の語源は乗馬だ。馬に付ける手綱と鞍——強力な動物のエネルギーを生産的な方向へ導く装置。馬の力そのものを変えるのではなく、その力が正しい方向に発揮されるように制御する。

AIのハーネスも同じ原理だ。モデルの推論能力そのものには触れない。モデルの「外側」を設計して、その知能が安定的で予測可能な成果物に変換されるようにする。ファインチューニングやモデル交換ではなく環境設計で成果を引き上げること。このアプローチが2026年AI開発のパラダイムになった。

---

## 用語整理: ハーネスエンジニアリングの位置づけ

### 3つの時代: プロンプト → コンテキスト → ハーネス

AI開発は3つの時代を経てきた。各時代は「人間が何を設計するか」で区分される。このフレームワークを理解すれば、今自分がどこにいてどこへ向かうべきかが見えてくる。

**第1期: プロンプトエンジニアリング**
核心的な問い: どんな言葉を使うか？
作業単位: 単一APIコール
人間の役割: プロンプト作成者
問題解決: プロンプトを書き直す

**第2期: コンテキストエンジニアリング**
核心的な問い: どんな情報が必要か？
作業単位: マルチターンセッション
人間の役割: 情報アーキテクト
問題解決: コンテキストデータを調整する

**第3期: ハーネスエンジニアリング**
核心的な問い: どんな環境が必要か？
作業単位: 完成された機能
人間の役割: 環境デザイナー
問題解決: ツール・ガードレールを追加する

決定的に、この3つの時代は置換ではなく累積だ。ハーネスエンジニアリングはプロンプトエンジニアリングとコンテキストエンジニアリングを包含する。前の時代の技術が無用になるのではなく、より大きなフレームワークの中に吸収されるのだ。

### ハーネス vs. エージェンティック vs. プロンプト

似た用語が多くて混乱するかもしれない。関係を整理しよう。

| 用語 | 範囲 | 焦点 | 比喩 |
| --- | --- | --- | --- |
| プロンプトエンジニアリング | 単一のインタラクション | 入力テキストの最適化 | 質問する技術 |
| コンテキストエンジニアリング | マルチターンセッション | 情報設計・管理 | 図書館司書 |

| ハーネスエンジニアリング | エージェントライフサイクル | インフラ設計（ツール・制約・検証・修正） | OS設計 |
| --- | --- | --- | --- |
| エージェンティックエンジニアリング | 開発プロセス全体 | エージェントオーケストレーション＋ハーネス＋デプロイ | システムインテグレーション |

エージェンティックエンジニアリングは最も広い上位概念だ。AIエージェントの計画・実装・テスト・デプロイの全プロセスを扱う。ハーネスエンジニアリングはその中で「インフラ設計」に特化した部分集合だ。LangChainのVivek Trivédyはこう区分した。「オーケストレーターはロジックと制御フローを担当し、ハーネスはツールとサイドエフェクトを担当する。両者は一緒に動作する。」

**POINT**

HashimotoのAGENTS.md作成原則: エージェントがミスを犯すたびに、防止策を1行追加する。削除しない。時間が経てばこのファイル自体がプロジェクトの「学習された知恵」になる。

概念は理解した。ではこれは本当に効果があるのか？「モデルよりハーネスが重要だ」という主張がスローガンではなく事実なのか、数字で検証してみよう。

# 03

# 数字が証明すること

## ベンチマークが示した逆転

2026年2月17日、LangChainがブログに投稿した記事1本が業界を揺るがした。「Improving Deep Agents with Harness Engineering.」タイトルだけ見れば平凡な技術ポストのようだった。しかし中に含まれた数字は平凡ではなかった。

### LangChain: モデル固定、ハーネスのみ変更で14%p向上

LangChainチームはdeepagents-cliというコーディングエージェントの性能を改善する実験を行った。実験条件はシンプルだった。モデルはGPT-5.2-Codexで完全に固定する。変えるのはハーネスのみだ。

具体的に何を変えたか。3つのミドルウェアを追加した。

- PreCompletionChecklistMiddleware — エージェントが答えを提出する前にチェックリストをセルフ検証させるセルフ検証ループ
- LocalContextMiddleware — 作業環境のディレクトリ構造、パッケージマネージャー、フレームワーク情報を自動収集してコンテキストに注入
- LoopDetectionMiddleware — エージェントが同じファイルを繰り返し修正する「ドゥームループ」を検知して中断

さらに「推論サンドイッチ（reasoning sandwich）」戦略を適用した。計画段階で高い推論予算を使い、実装段階では標準レベルに下げ、最終検証段階で再び上げる。すべてのトレースはLangSmithに記録してデータドリブンで反復改善した。

2行の表がすべてを物語る。

| 指標 | 変更前 | 変更後 | 変化 |
|---|---|---|---|
| TerminalBench 2.0 スコア | 52.8% | 66.5% | +13.7%p |
| リーダーボード順位 | 約30位 | Top 5 | 約25段階上昇 |
| 変更したもの | — | ハーネスのみ | モデル同一 |

モデルを1段階アップグレードしてもベンチマークスコアが2-5%p上がりにくい時代に、ハーネス変更だけで13.7%pが上がった。モデルはまったく同一だった。

### TerminalBench: 同一モデルで28段階の順位差

さらに衝撃的なデータがある。TerminalBench 2.0リーダーボードで、まったく同一のモデルであるOpus 4.6が使用するハーネスによってまるで異なる順位を記録した。

| モデル | ハーネス | 順位 |
|---|---|---|
| Claude Opus 4.6 | Claude Code 標準ハーネス | 33位 |
| Claude Opus 4.6 | 最適化された代替ハーネス | 5位 |

同じ頭脳。異なる環境。28段階の順位差。この1枚の表が「モデルよりハーネスが成果を決める」というこの本の核心テーゼを証明する。

**KEY INSIGHT**

モデルを交換して5%を改善するよりも、ハーネスを設計して15%を改善する方が現実的だ。これが2026年AI開発の新しい常識だ。

---

## プロダクションでの証拠

ベンチマークは実験室だ。「実戦でもそうなのか？」という問いは自然だ。答えは「そうだ」であり、規模はベンチマークよりはるかに大きい。

### OpenAI: 100万行、人間のコードゼロ

OpenAIは自社のCodexエージェントを開発する中で、5ヶ月間ハーネス設計に集中した。投入人員は3-7名のエンジニア。このチームが生産したコードは100万行を超える。人間が直接書いたコードは1行もない。

1日平均3.5件のPRをエンジニア1人当たり処理し、合計約1,500件のPRがマージされた。手動コーディング比で10倍の速度と推定された。プロジェクト初期のAGENTS.mdファイルさえCodexが作成した。

OpenAIが共有した5つの核心教訓を見れば、彼らが5ヶ月をどこに注いだかがわかる。

1. 環境設計 > モデル性能 — 進捗が遅いならAIの限界ではなく環境の欠陥だ
2. 「地図を与えよ」 — リポジトリを唯一の真実の源泉（Single Source of Truth）にする
3. 解空間を制約せよ — パターン、境界、構造を明示するほどエージェントの精度が上がる
4. 失敗から反復せよ — エージェントが詰まったら、不足しているツール・ガードレール・文書を見つけて補充する
5. ガベージコレクション — 定期的にエージェントを走らせてドキュメントの不整合やルール違反を検出する

5つの教訓のうち1つも「もっと良いモデルを使え」ではない。すべて環境設計に関するものだ。

### Stripe: 週間1,000件PR完全自動マージ

Stripeは毎週1,000件以上のプルリクエストを完全に無人（unattended）で自動マージしている。この規模を可能にしたのはマルチエージェントハーネスだ。