

Vibe Handbook

Vibe Handbook

目次

Git & GitHub	5
バージョン管理とは何か（そしてなぜ必要か）	6
リポジトリ、コミット、そして履歴	6
ブランチ: 恐れずに試す	7
実際に何が変わったかを見る (diff)	8
「しまった」ボタン: undo・revert・reset	8
リモートと GitHub: クラウド上の自分のプロジェクト	8
プルリクエストとイシュー	9
.gitignore と、決して秘密をコミットしないこと	9
初めての repo、最初から最後まで	9
AI に運転させる（それでも自分が責任を持つ）	10
まとめと実践	10
The Command Line	11
ターミナルと shell とは実際に何か	12
プロンプトを読む	12
日常的に使うコマンド	12
プログラムと dev サーバーを実行する	13
出力と exit コードを読む	13
Ctrl-C でプロセスを止める	14
PATH と「command not found」	14
AI がくれるコマンドを—安全に—実行する	14
出会うことになるパッケージマネージャー	15
まとめと実践	15
How the Web Works	16
二人の登場人物: クライアントとサーバー	17
URL とは実際に何か	17
リクエストとレスポンスのサイクル	17
HTTP メソッド: 動詞たち	18
ステータスコード: うまくいったか?	18
ヘッダーとクッキー	19
HTTPS と南京錠	19
名前からサーバーへ: DNS	19
これがあなたのアプリにどう対応するか	20
まとめと実践	20
API、JSON、そしてデータ	21
API とは実際のところ何なのか	22
エンドポイントと REST を、平たい言葉で	22
JSON とは何か	22
出会うことになるデータ型	23

実際のリクエストとレスポンス	24
API キーと認証ヘッダー	24
レート制限と、請求ショックの罠	25
API ドキュメントの読み方	25
呼び出しは AI にやらせ、形はあなたが確認する	25
まとめと実践	26
アプリの解剖学	27
全体像	27
frontend、backend、データベース、ホスティング	28
HTML、CSS、JavaScript	28
パッケージと依存関係	29
環境変数と設定	29
プロジェクトの形	30
リクエストはどう流れるか	31
まとめ&練習	31
バイブコーディングのマインドセット	33
バイブコーディングとは実際に何か	33
マインドセットの転換	34
ループ: 意図 □ 生成 □ レビュー □ 洗練	35
真価を発揮する場面	36
失敗する場面	36
よくある落とし穴	37
センスが新たなコアスキル	37
現実的な期待値	37
まとめと実践	38
AI ワークスペースをセットアップする	39
AI ネイティブなエディタまたはアシスタントを選ぶ	39
AI がナビゲートできるようにプロジェクトを構造化する	40
コンテキストファイルを書く (AGENTS.md)	41
バージョン管理を反射にする	42
フィードバックループを緊密に保つ	42
シークレットを慎重に管理する	43
健全なローカル環境が AI の変更を安全にする	43
ライブ診断 (LSP)	43
エージェントを拡張する	44
macOS vs Windows	45
TUI vs GUI	45
ハーネスエンジニアリング	46
コスト安全: CLI vs API	47
まとめと実践	47
エンジニアのようにプロンプトする	48
タスクを与える前にコンテキストを与える	48
入力、出力、エッジケースを指定する	49
小さなステップで作業する	50
コードの前にプランを求める	50
書き直しではなく差分で反復する	51
例を提供する	51
AI に何をすべきでないかを伝える	52
コンテキストをリソースとして管理する	52
避けるべき一般的なアンチパターン	53
コアとなるマインドセット	53

まとめと実践	53
アイデアから仕様へ、数分で	55
なぜ「ノリ」ではなく仕様が必要か	55
ステップ 1: 問題を明確にする	56
ステップ 2: スcopeと非ゴールを定義する	57
ステップ 3: ユーザストーリーを書く	57
ステップ 4: 軽量の PRD を起草する	58
ステップ 5: データモデルをスケッチする	59
ステップ 6: 仕様を Vibe サイズのタスクに分割する	59
仕様を生かし続ける	60
よくある罠: 問題ではなく解決策を仕様化してしまう	60
その見返り	61
まとめと実践	61
スタックを選ぶ: 言語をひと目で	62
JavaScript / TypeScript	63
Python	63
Go	64
Rust	64
その他の言語について手短に	65
ひと目で比較	65
風変わりな選択の本当のコスト	66
目的別に選ぶ	66
おさらいと練習	67
機能を、Vibe ごとに作っていく	68
ビルドループ	68
「垂直スライス」とは	69
実例: 「お気に入り」機能	70
すべての行を理解せずに diff を読む	72
AI に自由に走らせるか、手動で操縦するか	72
アプリを常に動く状態に保つ	72
小さな単位でコミットする	73
スライスが大きすぎるとき	73
まとめ	73
おさらいと練習	73
AI を使ったデバッグ	74
基本となる心構え	75
デバッグのチェックリスト	75
AI に適切なコンテキストを与える	76
分からないエラーを読む	77
仮説、ログ、二分探索	78
根本原因 対 症状	78
AI が堂々巡りになったとき	79
残す前に修正を理解する	79
まとめ	79
おさらいと練習	80
サーバーインフラ	81
コードが住める 5 つの場所	81
いくつかの専門用語の解説	82
トレードオフ表	83
どれをいつ選ぶか	83

AI がどうデプロイと設定を組み上げるか	84
個人開発者におすすめのデフォルトの道筋	85
「マネージド」が本当に意味すること (そしてあなたが依然として持つもの)	86
出荷前のクイックチェックリスト	86
まとめ	86
まとめと練習	87
データベースインフラ	88
データストレージの 3 つのかたち	88
マネージド 対 セルフホスト	89
スキーマ設計の基本	90
マイグレーション: スキーマを安全に変える	91
インデックス	91
キャッシュ	92
キュー	92
初心者によくある間違い	93
データ層を構築するために AI を指示する	93
まとめと練習	94
ガードレール、テスト、レビュー	95
AI を指揮するときにガードレールが必要な理由	95
テスト: あなたのセーフティネット	96
型チェックとリント: タダで捕まえられるミス	97
AI の変更範囲を絞る	98
差分を批判的に読む	98
レビューゲート: マージ前のチェックリスト	98
CI の基本: チェックを自動化する	99
デフォルトで安全な習慣	100
ガードレールが悲鳴を上げたら: 直す前にトリアージする	100
まとめと練習	100
本番へのリリース	101
ホストを選んでデプロイする	102
カスタムドメインと DNS	103
環境変数とシークレット	104
決済の接続	104
ステージングとロールバック	105
モニタリングとエラートラッキング	106
パフォーマンスとキャッシュ	106
ローンチ前チェックリスト	107
ローンチした後	107
まとめと練習	107
ケーススタディ: 0 からローンチまで	108
ケーススタディ 1: 多言語のランディング + 決済ページ	109
ケーススタディ 2: 認証 + データベース付きの小さな SaaS ツール	110
ケーススタディ 3: 個人用の自動化ユーティリティ	112
ケーススタディ 4: 手をかけずに済むコンテンツサイト	114
4 つに共通すること	116
まとめと練習	116
バイブコーダーのためのセキュリティ	117
AI が生成するコードがデフォルトで安全でない理由	118
インジェクション: 入力がコードになるとき	118
シークレットと API キー: 露出したキーの罠	119

認証と認可: 誰でも呼べるエンドポイント	120
ファイルの取り扱いとアップロード	121
依存関係のリスク: AI が発明したパッケージ	121
セキュリティレビューのゲート	122
自動化への慢心、セキュリティ編	123
まとめと練習	123
エージェント・オーケストレーション	124
三つの層	124
いつ走らせ、いつ手動で操縦するか	125
さまよえないようにタスクの範囲を定める	125
バックグラウンドおよび並列エージェント	126
マルチエージェント: ファンアウト、敵対者、統合	127
規模に合わせて出力をレビューする	128
コストと暴走ループ	128
正直な限界	129
まとめと練習	129

Git & GitHub



バイブコーディングを始める前にひとつだけツールを学ぶなら、これにしてください。Git はあなたを大胆にしてくれるツールです。AI に「このファイル全体を書き直して」と言っても安心していただける理由が、まさに Git です。いつでも、ちゃんと動いていたバージョンに戻るからです。

正直に言うとうこうです。AI にソフトウェアを作らせると、AI は必ず何かを壊します—それも頻繁に。自信満々でファイルを書き直し、実際に動いていた部分を消し、めちゃくちゃな状態を渡してきます。イライラする午後で済むか、まる一週間を失うかの違いは、安全網があったかどうかにかかっています。Git がその安全網です。この章では、コーディングの予備知識なしでゼロから Git を教えます。

バージョン管理とは何か（そしてなぜ必要か）

セーブポイントのあるビデオゲームを思い浮かべてください。難しい区間をクリアし、セーブして、リスクのあることを試します。失敗したら、ゲーム全体を最初からやり直す代わりに、直前のセーブを読み込みます。**バージョン管理は、プロジェクトのセーブポイントです。**

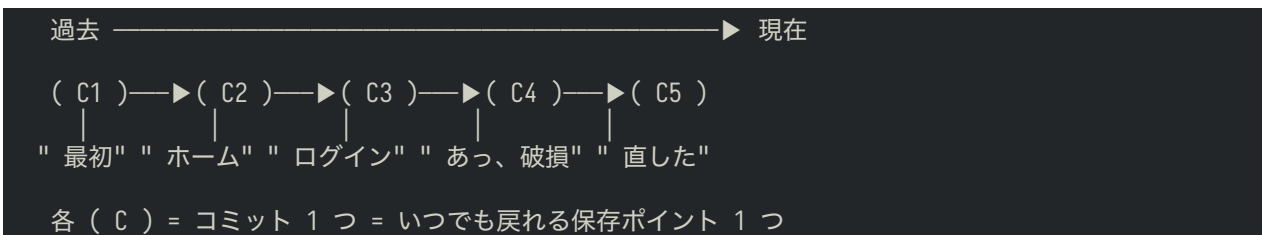
バージョン管理がないと、プロジェクトはただ一つの状態しか持ちません。今この瞬間の姿だけです。AI がコードを壊せば、まともだったバージョンは消えます。バージョン管理があれば、保存されたすべての状態の完全なタイムラインを保ち、そのどの地点へでも飛び戻れます。

Git は最も広く使われているバージョン管理ツールです。あなたのコンピュータ上で動き、頼んだときにプロジェクトのスナップショットを静かに記録します。Git が無料で与えてくれるものをいくつか挙げると：

- **安全網**—数時間後でも、まずい変更を取り消せます。
- **履歴**—何が、いつ、なぜ変わったのかを正確に見られます。
- **自信**—何も本当には失われないので、思いきり実験できます。

最後の項目こそ本当のご褒美です。バイブコーディングは、速く動いている試すときに最もうまくいきます。Git は、その速い動きを無謀さではなく安全にしてくれます。

プロジェクトの履歴を、いつでも戻れるセーブポイントの並びとして思い描いてください：



リポジトリ、コミット、そして履歴

これからずっと耳にする三つの言葉です。ひとつずつ解きほぐしましょう。

リポジトリ（略して「repo」）は、単に Git が見張っているフォルダです。普通のフォルダを一度だけ repo に変えれば、それ以降 Git がその中のすべてを追跡します。

コミットは、一つのセーブポイントです。ある時点のすべてのファイルのスナップショットに、何が変わったかを記した短いメッセージが添えられたものです（「ログインページを追加」「壊れた登録ボタンを修正」）。コミットは絶えず作ります—それぞれを、戻りたくなるかもしれないチェックポイントだと考えてください。

履歴は、すべてのコミットを順に並べたリストです。つまりタイムラインです。スクロールして各変更が何だったかを読み、どの地点へでもタイムトラベルできます。

実際にはこう見えます。覚える必要はありません—ほとんどは AI に実行させることになりますから—でも、言葉が謎でなくなるように一緒に読んでみてください：

```
# 現在のフォルダを Git リポジトリにする（最初の一度だけ）
git init

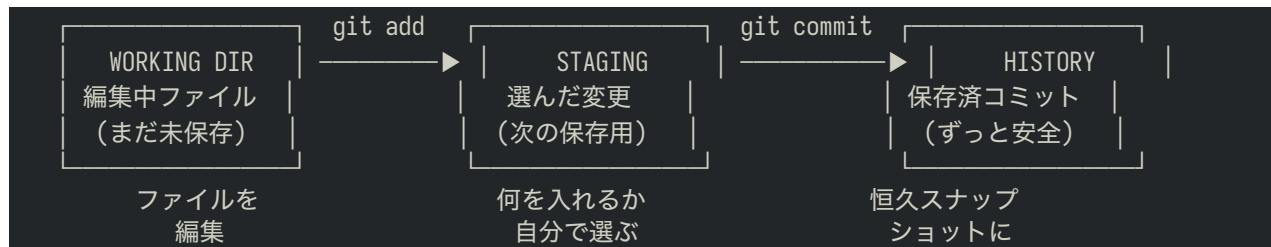
# 直前のセーブポイント以降どのファイルが変わったか確認
git status
```

```
# 変更をステージングし、メッセージ付きでコミットとして保存
git add .
git commit -m "Add the homepage layout"

# 履歴を見る —すべてのコミット、新しい順
git log --oneline
```

git add . は「私のすべての変更を次のセーブポイントに含めて」という意味です。git commit が実際の保存です。-m "... " は未来の自分へのメモです。一か月後でも理解できるメモを書きましょう。

この二つのコマンドは作業を三つの段階に動かします。編集したファイルは、いちばん右まで進んで初めて保存されます：



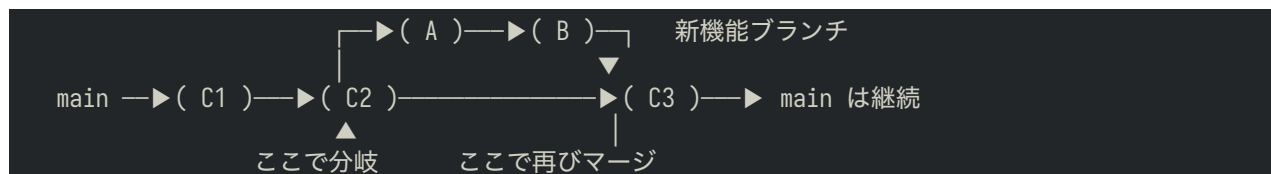
ブランチ: 恐れずに試す

ブランチは、実験できるプロジェクトの並行コピーです。主な作業は通常 main と呼ばれるブランチにあります。何かを試したいとき—新機能でも、リスクのある AI の書き直しでも—新しいブランチを作ってそこで作業すれば、main は手つかずで安全なまま残ります。

実験がうまくいけば main にマージします。だめなら、ブランチを捨てるだけ。main はそんなことがあったとすら知りません。

これこそ、大きな AI の変更を扱うべき正しいやり方です。「決済フロー全体を作って」は、自分専用のブランチに値します。AI が良いものを出したらマージ。惨事を出したらブランチを削除。失うものは何ともありません。

図で見ると、ブランチは main から枝分かれして独自に育ち、また戻ってマージされます：



```
# 新しいブランチを作って切り替える
git checkout -b new-feature

# ... 作業して、コミットを作って...

# 安全な main ブランチに戻る
git checkout main

# ブランチの良い作業を main にマージ
git merge new-feature
```

実際に何が変わったかを見る (diff)

変更を保存する前に—そして特に AI が行った変更を受け入れる前に—正確に何が違うのかを見たいくなります。それが **diff** です。一行ずつ、何が追加され（通常は緑）、何が削除されたか（通常は赤）を見せてくれます。

これは最も大切な習慣のひとつです。バンプコーディングの哲学はすべて自分が出荷するものを理解することに尽きます。コードを自分でタイプする必要はありませんが、何が変わったかを見て、おおよそ理解する必要はあります。diff はまさにそれを行う場所です。

```
# 変更したがまだコミットしていない内容を見る
git diff

# ステージングされコミット準備が整った内容を見る
git diff --staged
```

AI が「終わりました、ファイルを更新しました」と言ったら、次の動きは diff を見ることです。必要だったものを消しているのが見えたら、今捕まえます—出荷された後ではなく。

「しまった」ボタン: undo • revert • reset

この部分が、Git を面倒な作業から超能力へと変えます。何かがうまくいかなかったとき—必ずそうなります—こうやって巻き戻します。

- **保存していない変更を捨てる**—あなた（または AI）がファイルを編集して悪化させたが、まだコミットしていないとき。直前のまともなコミットへすべて戻せます。
- **コミットを取り消す (revert)**—変更を保存した後、それがまずかったと気づいたとき。git revert はそのまずいコミットを打ち消す新しいコミットを作り、履歴を正直に保ちます。
- **以前の地点へリセット (reset)**—プロジェクトを数コミット前の姿へ巻き戻したいとき。

```
# コミットしていないすべての変更を取り消し、直前のコミットへ戻る
git restore .

# 特定のまずいコミットを安全に取り消す（新しい「取り消し」コミットを作る）
git revert <commit-id>

# 以前のコミットへ巻き戻す（強力 —使う前に AI に説明を頼んでください）
git reset --hard <commit-id>
```

ひとつ注意: git reset --hard は作業を恒久的に捨てます。ときには正しいツールですが、慎重に扱うべきものです。確信が持てないときは、実行前に AI に「このコマンドは私のファイルに正確に何をするの?」と尋ねましょう。

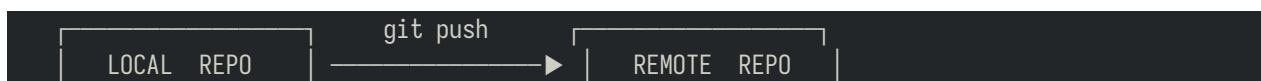
リモートと GitHub: クラウド上の自分のプロジェクト

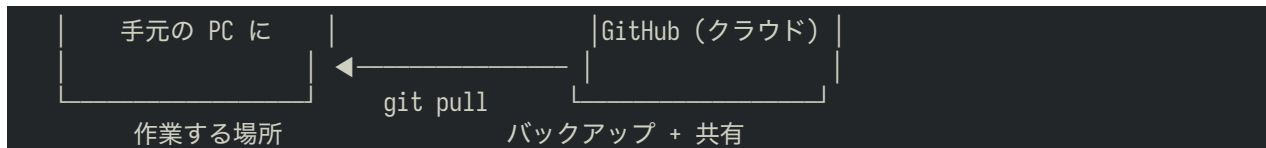
ここまでのすべては、あなた自身のコンピュータで起きます。**GitHub** は、あなたの repo のコピーをクラウドに保存するウェブサイトです。そのコピーを**リモート**と呼びます。

なぜわざわざ? 三つの理由です。ノートパソコンが壊れたときのバックアップであり、他の人（あるいは AI ツールやデプロイサービス）とコードを共有する方法であり、共同作業が起きる場所だからです。

ローカル repo をリモートに一度つなげたら、**push**（コミットを GitHub へ上げる）と **pull**（GitHub からコミットを下ろす）を行います。

プロジェクトのコピーが二つ—一つはノートパソコンに、一つはクラウドに—あり、push と pull で同期されると考えてください:





```
# ローカル repo を GitHub repo につなぐ (最初の一度だけ)
git remote add origin https://github.com/yourname/yourproject.git

# コミットを GitHub へ上げる
git push origin main

# GitHub にはあるが自分のコンピュータにないコミットを下ろす
git pull origin main
```

push は「クラウドに保存」、pull は「クラウドから最新を取得」と考えてください。

プルリクエストとイシュー

すぐに出会う二つの GitHub 機能です。

プルリクエスト (PR) は、あるブランチを別のブランチにマージしようという提案で、取り込む前に変更をレビューできる場所が組み込まれています。一人で作業していても PR は役立ちます。すべての変更を見せる一つのきれいな画面を提供し、main にマージする前に AI の作業をレビューできます。diff の習慣を公式化したものです。

イシュー は、やるべきことや直すべきこと—バグ、機能のアイデア、TODO —についての追跡メモです。イシューは、やることリストを頭の中のアちこちではなくプロジェクト自体に貼り付けておく方法です。

.gitignore と、決して秘密をコミットしないこと

一部のファイルは絶対に Git に入れてはいけません。代表的なもの:

- **秘密**—API (アプリケーション・プログラミング・インターフェース) キー、パスワード、データベースの認証情報。秘密を GitHub にコミットしたら、後で消しても漏洩したものとして扱ってください。履歴は覚えています。
- **ゴミ**—一時ファイル、ダウンロードした依存関係 (node_modules など)、ビルド成果物。巨大で、追跡する意味がありません。

これは .gitignore というファイルで制御します。その中にパターンを書くと、Git はそれらのファイルが存在しないふりをします。これはパイプコーダーにとって極めて重要です。AI ツールは本物のキーが入った設定ファイルを好んで作り、それをうっかりコミットするのは非常に簡単だからです。

```
# シンプルな .gitignore ファイル
.env
node_modules/
*.log
```

.env の行が肝心です—秘密はたいていそこにあります。最初のコミットの前に .gitignore を追加し、AI に「このコミットに公開してはいけないものはある?」と尋ねる習慣をつけましょう。

初めての repo、最初から最後まで

全体の流れを一か所にまとめました。フォルダを作り、repo にし、作業を保存し、GitHub に push する:

```
# 1. 新しいプロジェクトを始める
mkdir my-first-project
cd my-first-project
```



```
git init

# 2. 何よりも先に秘密を守る
echo ".env" > .gitignore

# 3. 最初のセーブポイントを作る
git add .
git commit -m "Initial commit"

# 4. GitHub につないで push する
git remote add origin https://github.com/yourname/my-first-project.git
git push -u origin main
```

これで完了です。安全網とクラウドバックアップを備えたプロジェクトができました。進展があるたびに、add □ commit □ push のリズムを繰り返しましょう。

AI に運転させる（それでも自分が責任を持つ）

ここに解放感のある部分があります。これらのコマンドを覚える必要はまったくありません。AI に「適切なメッセージでこれをコミットして」あるいは「新機能用のブランチを作って切り替えて」と言えば、AI が正しいコマンドを実行します。それも十分に良い作業の仕方です。

ただし落とし穴は—本書の残りと同じ落とし穴ですが—自分でタイプしていなくても、何をしたのかを理解する責任は依然としてあなたにある、ということです。実際に通用する中道:

- **AI にコマンドを実行させつつ**、最初の数回は各コマンドが何をするのかをやさしい言葉で説明してもらいましょう。
- **diff は自分で読みましょう**、どのコミットの前でも必ず。これは交渉の余地なし。AI が消してはいけないものを消すのを捕まえる方法です。
- **頻繁に、小さくコミットしましょう**。小さなセーブポイントがたくさんある方が、巨大な一つより優れています。巻き戻しが全か無かではなく、精密になるからです。
- **大きな作業の前にはブランチを作りましょう**。大々的な書き直しの前に AI にブランチを作らせ、main を安全に保ちましょう。
- **コマンドが破壊的に聞こえるとき** (reset --hard、強制 push、ブランチ削除など)、承認する前に「これは正確に何をするの?」と尋ねましょう。

あなたがテックリードです。AI は速くて有能だが、ときに無謀なチームメイトです。Git は、そのチームメイトが本当の被害を出さないようにするプロセスです。自信を持って指揮できるくらい Git を身につければ、「全部書き直して」ボタンはもう怖くありません—いつでも戻る道があるのですから。

まとめと実践

重要なポイント

- Git は **プロジェクトのセーブポイント** です—いつでも戻れる完全なタイムラインであり、だからこそ恐れずに AI を解き放てます。
- 基本のリズムは add □ commit □ push です：変更を stage し、明確なメッセージを付けた commit として保存し、バックアップと共有のために GitHub へ送ります。
- **大きな作業の前にはブランチを切り、コミットのたびに diff を読みましょう**—AI が消してはいけないものを消すのを捕まえる方法です。
- .gitignore は秘密 (.env など) とゴミ (node_modules/ など) を履歴から除外します。GitHub にコミットされた秘密は、あとで削除しても永遠に漏れたものとして扱います。
- AI にコマンドを実行させてもかまいませんが、各コマンドが何をしたか理解する責任は自分にあります—特に reset --hard のような破壊的なものは。

やってみよう

まっさらな新しいフォルダを作り、全体の流れを自分の手で回してみましょう：`git init`、`.env` を含む `.gitignore` を作成、シンプルな `notes.txt` ファイルを追加、そして `git add .` と `git commit -m "Initial commit"`。次に `git log --oneline` を実行してセーブポイントを見て、ファイルを編集してから `git diff` を実行し、Git が何が変わったかを正確に見せてくれる様子を観察します。何も壊れようのないプロジェクトで、この章の核となる習慣をすべて使ったことになります。

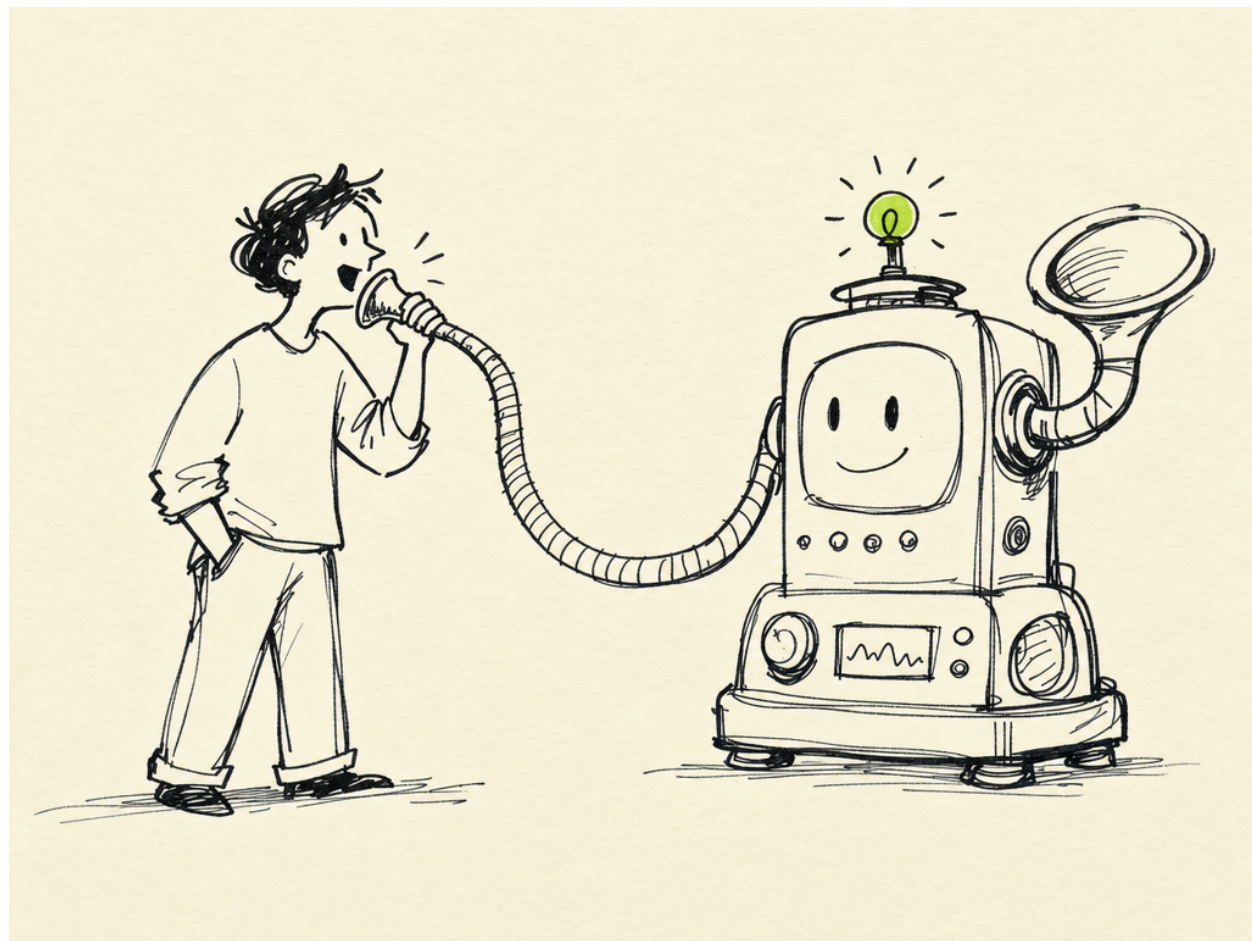
この章のプロンプト

私は Git を学び始めた初心者で、コマンドはあなたに代わりに実行してほしいのですが、実行しながら教えてください。以下のタスクについて：

< やりたいことを説明 —例：「今の作業を保存」または「危険な変更を安全に試す」 >

- どの Git コマンドを実行するか、各コマンドが何をするかを平易な言葉で教えて。
- 破壊的な操作なら (`reset --hard`、強制 `push`、ブランチ削除など)、実行前にはっきり警告し、何を失うのかを説明して。
- どのコミットの前でも、`diff` を見せて何が変わったかを要約して。
- 変更の中に公開すべきでないもの（秘密、キー）があれば知らせる。

The Command Line



AI コーディングツールが作業する様子を初めて見ると、黒いウィンドウにテキストが上へスクロールして流れていき、点滅するカーソルがあなたの入力を待っている。そのウィンドウがコマンドラインだ。コーディング未経験の多くの人にとって、これは旅全体で最も怖い部分だ。映画で見たように、一度操作を誤ればハードドライブが丸ごと消えてしまいそうに見える。そんなことはない。コマンドラインは、コンピューターに何をすべきか伝

えるもう一つの方法にすぎない。クリックの代わりに言葉で伝えるのだ。

この章はその恐怖を取り除く。百個のコマンドを暗記する必要はない。何が起きているのかを理解し、日常的に使うわずかなコマンドを見分け、AI が手渡すコマンドを安全に実行する方法を知っていればいい。それだけで、怖がるのをやめて効果的に働き始めるには十分だ。

ターミナルと shell とは実際に何か

普通のアプリを使うとき、あなたはボタンをクリックし、アプリはそのクリックをコンピューター向けの命令に翻訳する。コマンドラインはボタンを飛ばす。あなたが命令を入力して Enter を押すと、コンピューターがそれを実行する。

よく耳にする二つの言葉がある：

- **ターミナル (Terminal)** —あなたが入力するウィンドウ。履歴が残るテキストボックスにすぎない。(Mac では「Terminal」や「iTerm」、Windows では「PowerShell」や「Windows Terminal」、Linux ではたいてい単に「Terminal」だ。)
- **shell** —そのウィンドウの中であなたの入力を読み取って実行するプログラム。最も一般的なのは **bash** と呼ばれ、Mac は **zsh** というものを使う。本書で扱うすべてについて、両者はほぼ同じように動作する。

Enter を押すたびに起きるループはこうだ: shell があなたの入力を読み取り、一致するプログラムを見つけて実行し、そのプログラムが生み出したものを何でもウィンドウに出力し返す。



なぜ AI コーディングツールはここに住んでいるのか。コマンドラインが開発の普遍的なコントロールパネルだからだ。ツールのインストール、アプリの起動、テストの実行、インターネットへのデプロイ—そのすべてが入力された命令を通じて起きる。AI エージェントはあなたが入力するのと同じコマンドを入力して動く。ただ、より速く、タイプミスもないだけだ。エージェントが働くのを見るとき、あなたは、自分も使うことになるまさにその扉を、それが使う様子を見ているのだ。

プロンプトを読む

何かを入力する前に、shell はあなたに**プロンプト (prompt)** —ある記号で終わる短いテキスト—を示して入力を待つ。こんなふうに見えるかもしれない：

```
mini@laptop ~/projects/my-app $
```

左から右へ読むと、こう教えてくれる: あなたのユーザー名 (mini)、あなたのコンピューター (laptop)、いま自分がいる場所 (~/projects/my-app というフォルダ)、そして「あなたの番」を意味する \$。~ はあなたのホームフォルダを指す略記だ。末尾の記号はよく \$ か% で、ときに > のこともある。その記号は入力しないこと—shell が準備できたと言っているだけだ。

これは見た目より重要だ。コマンドラインで起きる混乱のほとんどは、自分がどこにいるのかを知らないことから来る。プロンプトは毎回それを教えてくれる。

日常的に使うコマンド

ここに、あなたが絶えず目にするコマンドがある。文章のように読んでみよう: 動詞、そしてときにその後続く対象。


```

pwd                # "print working directory" —私はいまどこにいる？
ls                 # ここにあるファイルとフォルダを一覧表示する
ls -la            # 隠しファイルも含めてすべて、詳細つきで一覧表示する
cd projects        # change directory —「projects」フォルダの中へ移動する
cd ..              # フォルダを一つ上へ上がる（「..」は「一つ上のフォルダ」を意味する）
cd ~               # ホームフォルダへ戻る
mkdir my-app       # 「my-app」という新しいフォルダを作る
cp notes.txt backup.txt # ファイルをコピーする（元はそのまま、コピーが作られる）
mv old.txt new.txt  # ファイルを移動するか名前を変える
rm draft.txt        # ファイルを削除する（remove）

```

本当の苦労を減らしてくれるいくつかのメモ:

あなたのファイルは入れ子になったフォルダのツリーの中に住んでおり、cd はそのツリーを歩く方法だ。ここに小さな例がある。pwd は、あなたが my-app へ cd したときに立っている場所を示している:

```

/                               ← 一番上 ("root")
├── Users/
│   ├── mini/                  ← ホームフォルダ (~ の短縮)
│   │   ├── projects/
│   │   │   ├── my-app/       ← ここで cd → pwd は /Users/mini/projects/my-app
│   │   │   │   ├── src/
│   │   │   │   └── notes.txt
│   │   └── old-app/

```

- **cd は移動の方法だ。** pwd、ls と組み合わせれば、あなたが生涯やることの 80% だ。正しいフォルダへ入り、見回し、それから何かを実行する。
- **パスは相対パスにも絶対パスにもなる。** cd projects は現在地を基準とした相対パスで、cd /Users/mini/projects は一番上からの完全な住所だ。
- **rm は永久だ。** コマンドラインにはゴミ箱がない。ファイルを rm すれば、消える—「本当によろしいですか?」もなく、取り消しもない。フォルダとその中のすべてを尋ねもせず削除する rm -rf には特に注意せよ。rm コマンドが何を削除するか少しでも確信が持てなければ、実行するな。まず AI に説明を求めよ。

プログラムと dev サーバーを実行する

ソフトウェアを作るとは、それを実行するということだ。たいていの場合、プログラム名を入力し、その後にやってほしいことを書く。

```

node app.js        # Node で JavaScript ファイルを実行する
python script.py   # Python ファイルを実行する
npm run dev         # プロジェクトの開発サーバーを起動する

```

最後のものは特別だ。**dev サーバー**は、あなたのアプリをあなたのコンピューター上で動かし、http://localhost:3000 のようなアドレスのブラウザで見られるようにするプログラムだ。理解すべき要点はこれだ: dev サーバーは**動き続ける**。ターミナルは「固まった」ように見えるだろう—新しいプロンプトを出さず、代わりに活動の流れを表示する。それは固まっているのではなく、働いているのだ。その命令が動いているあいだ、あなたのアプリは生きている。ブラウザを開き、命令が出力したアドレスへ行けば、あなたのアプリが見えるだろう。

終わったら、それを止める (次の節で)。

出力と exit コードを読む

命令が実行されたあとは、たいてい何かを出力する。その出力を読む方法を身につけることは本物の技術だ。物事がうまくいったかをあなたと AI が知る方法が、まさにこれだからだ。

- **通常の出力**はただの情報だ—一覧表示されたファイル、進捗メッセージ、サーバーが起動したという知らせ。
- **エラー**はたいてい率直にそう言う: Error:, command not found, permission denied, cannot find module. コロンの後の言葉が手がかりだ。エラー全体をコピーして AI に貼り付けよ—何かが壊れたときに手渡せる、最も役立つ唯一のものだ。

裏側では、すべての命令が見えない **exit コード**で終わる: 0 は成功を、それ以外は問題を意味する。手で確認することはまれだが、言及されるのを耳にするだろう。最後のコードはこう見られる:

```
echo $?          # 直前の命令の exit コードを出力する (0 = 正常)
```

命令が静かに失敗して確信が持てないなら、echo \$? が本当に成功したかを教えてくれる。

Ctrl-C でプロセスを止める

その dev サーバーは、あなたが止めるまで動き続ける。普遍的な「止める」は **Ctrl-C** だ—Control キーを押しながら C を押す。動いている何に対しても「いま止まってください」という丁寧な信号を送り、あなたはプロンプトを取り戻す。

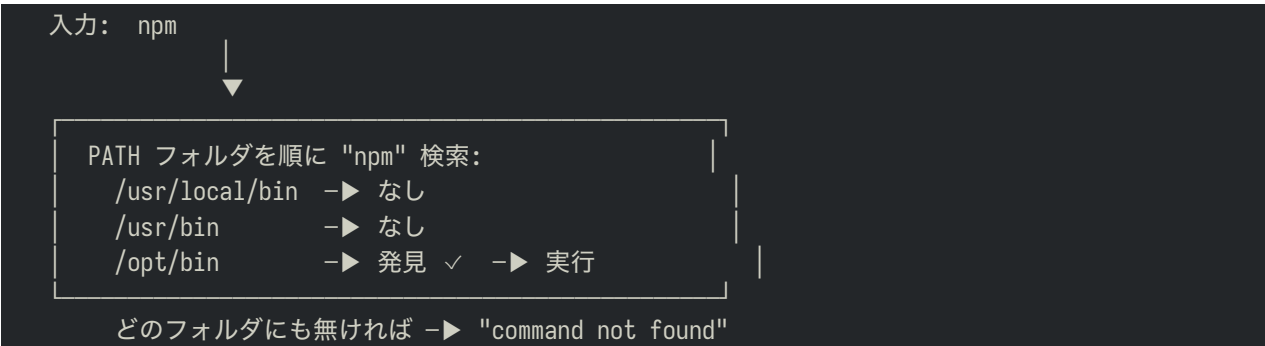
- **Ctrl-C** —このターミナルでいま動いているプログラムを止める。dev サーバーを終えたり、長くかかりすぎている命令を取り消したりするときの定番の手段だ。
- **ターミナルのウィンドウを閉じることも**、その中で動いているすべてを止めるが、Ctrl-C のほうがきれいだ。

Ctrl-C を覚えておけば、「固まった」ターミナルに閉じ込められた気分を二度と味わわずにすむ。

PATH と「command not found」

いずれ、命令を入力したのに command not found が出るだろう。これはあなたが愚かだからではほばない。そのプログラムがどこに住んでいるかをコンピューターが知らない、という意味だ。

shell は **PATH** と呼ばれるフォルダの一覧を確認してプログラムを見つける。あなたが npm と入力すると、shell は PATH の中の各フォルダを巡って npm というプログラムを探す。そのどこにもなければ、command not found が出る。



```
echo $PATH          # shell が検索するフォルダの一覧を表示する
```

実際には command not found はたいてい二つのどちらかを意味する: そのツールがまだインストールされていないか、PATH がない別の場所にインストールされたか。どちらもよくあることで、直せる。エラーを AI に貼り付け、何を実行しようとしていたかを伝えよ—PATH の問題を直すのは、AI が順を追って案内してくれるのについてつけの、こまごました設定作業だ。

AI がくれるコマンドを—安全に—実行する

パイプコーディングの大きな部分は、AI が「このコマンドを実行してください」と一行を貼り付けて手渡すことだ。これは普通のことで、問題ない。ただ、あなたを永遠に守ってくれる習慣を一つ身につけよ: **実行する前に読め**。

すべてのフラグを理解する必要はない。そのコマンドが何をするのか、とくに先頭の動詞についての大きな感覚があればいい:

- `ls`、`pwd`、`cd`、`cat`、`echo` —無害だ。何かを見たり移動したりするだけ。自由に実行せよ。
- `npm install`、`pip install` —ツールをダウンロードして設定する。おおむね安全だが、あなたのプロジェクトを変える。
- `rm`、`mv`、`sudo` のつくものすべて、ダウンロードを直接 shell へ流し込むものすべて (`curl ... | bash`) —**速度を落とせ**。これらは何かを削除したり、あなたのマシンに対する完全な力でコードを実行したりしうる。

コマンドが怖く見えたり、動詞を見分けられなかったりしたら、こう尋ねよ: 「このコマンドは何をしますか、実行しても安全ですか?」よい AI はそれを平易に説明してくれる。その確認なしに、どこかのウェブサイトのコマンドを貼り付けてはならない。コマンドラインは言われたとおりに正確に実行する—危険なことも含めて—だから、まず読むという小さな習慣が、あなたのシートベルトだ。

出会うことになるパッケージマネージャー

現代のソフトウェアは、あらかじめ書かれた小さな部品の何千個から作られる。**パッケージマネージャー (package manager)** は、それらの部品をダウンロードして整理するツールだ。絶えず二つに出会うことになる:

```
npm install      # この JavaScript プロジェクトが必要とするすべてをインストールする
npm install react # 「react」という特定のパッケージを追加する
pip install requests # 「requests」という Python パッケージをインストールする
```

- **npm** (ノード・パッケージ・マネージャー) は JavaScript と Node のプロジェクトのためのパッケージマネージャーだ。npm install(よく npm i と表記される) は、プロジェクトが何を必要とするか記したファイルを読み、それをすべてダウンロードする。
- **pip** は Python のために同じ仕事をする。

プロジェクトを初めて開くときに `npm install` を一度走らせ、その後は忘れて過ごすことが多い。AI が新しいパッケージを必要とする機能を追加したら、それをインストールするよう伝えてくれる。いまやあなたはそれが何を意味するか知っている: ただ building block を一つ取ってくるだけで、恐れるべき何かではない。

これがコマンドラインだ。ハッカーの武器ではなく—ただあなたが頼んだとおりに実行するテキストボックスだ。プロンプトを読み、十数個のコマンドを見分け、Ctrl-C で暴走するプロセスを止め、実行する前に読む方法を身につけよ。そのほかのすべては、尋ねればいい。

まとめと実践

重要なポイント

- コマンドラインは、言われたとおりに正確に実行するただのテキストボックスです—クリックの代わりに言葉で。AI エージェントは、あなたが打つのも同じコマンドを打って動きます。
- **プロンプト**は毎回 自分が今どこにいるか を教えてくれます。混乱のほとんどは、今いるフォルダを知らないことから来ます。pwd・ls・cd が、あなたがやることの 80% です。
- **rm は永久的です**—ゴミ箱も取り消しありません。rm・sudo、そしてダウンロードをシェルに流し込むもの (`curl ... | bash`) の前では速度を落としましょう。
- **開発サーバー**は動き続け、わざと「固まった」ように見えます—それが動いている姿です。**Ctrl-C** は実行中のどんなプログラムも止め、プロンプトを返してくれます。
- **実行する前に読みましょう**。すべてのフラグを知る必要はなく、動詞のおおまかな感覚があれば十分です。迷ったら AI に「このコマンドは何をする? 実行して安全?」と聞きましょう。

やってみよう

ターミナルを開き、何も変えずに探索してみましょう: `pwd` で今いる場所を見て、`ls` で何があるかを見て、`cd` でフォルダに入りもう一度 `ls`、そして `cd ..` で戻ります。毎回プロンプトが更新されるのを見てください。次に、わざと失敗するコマンドを実行してみましょう—`lss` (タイプミス) と打って `command not found` のメッセージを読みます。ファイルを一つも触らずに、もっとも安全でもっともよく使う二つのコマンドを練習し、エラーの読み方まで身につけたことになります。

この章のプロンプト

コマンドラインの初心者で、何かを実行するのが少し不安です。
あなた（またはチュートリアル）が実行するようにと渡してくれたコマンドはこれです：
<ここにコマンドを貼り付け>

- このコマンドが何をするのか、平易な言葉で一段階ずつ説明してください。
- 安全かどうか、あるいはファイルを削除したり、システムを変更したり、完全な権限でコードを実行したりする可能性があるか、はっきり教えてください。
- 危険な場合は、同じことをより安全に行う方法や、先に確認すべきことを提案してください。
- 実行する前にどのフォルダにいるべきか教えてください。

How the Web Works



ウェブ向けに何かを作る前に、ウェブが実際に何であるかを知っておくと役に立つ。コンピュータサイエンスの学位は要らないし、何かを暗記する必要もない。必要なのは、機能する一つのメンタルモデルだけだ—アドレスを入力するとページが現れる、その瞬間に何が起きているかを頭の中に描いた絵である。

この章はその絵を作る。ログインフォームから決済まで、バイブコーディングで作ったアプリがすることはすべて、結局は同じ単純なダンスの一形態だ: あるコンピュータが尋ね、別のコンピュータが答える。このダンスがはっきり見えるようになれば、本書の残りは魔法のように感じられなくなり、配管工事のように感じられ始める。

二人の登場人物: クライアントとサーバー

ウェブのほとんどすべては、二つの役割のあいだの会話だ。

- **クライアント**は尋ねる側だ。たいていの場合それは、あなたのスマートフォンやノートパソコンで動くウェブブラウザ—Chrome、Safari、Firefox—である。カウンターに歩み寄る客のようなものだ。
- **サーバー**は答える側だ。どこかのデータセンターに置かれ、一日中動き続け、リクエストを待っているコンピュータである。カウンターの奥の厨房のようなものだ。

クライアントは決してサーバーの仕事をし、サーバーも決してクライアントの仕事をしな。ブラウザは「ホームページをくれ」と尋ねる。サーバーは「ここにあるよ」と答える。この関係がすべてだ。サーバーは特別な魔法の箱ではない—ただ質問を待ち、答えを送り返すのが仕事の、ありふれたコンピュータにすぎない。

これからずっと耳にする言葉がもう一つある:**request(リクエスト)**(クライアントが送る質問)と**response(レスポンス)**(サーバーが返す答え)。この二語を手元に置いておくこと; 以下のすべての背骨だ。

URL とは実際に何か

URL (ユニフォーム・リソース・ロケーター) はあなたが要求するアドレスだ。一本の長い文字列に見えるが、実はラベルの付いたいくつかの部分が縫い合わされたものだ。これを分解してみよう:

`https://shop.example.com/products/shoes?size=10`

<code>https</code>	<code>scheme</code>	— how to talk (use HTTPS, the secure way)
<code>shop.example.com</code>	<code>host</code>	— which server to talk to
<code>/products/shoes</code>	<code>path</code>	— which page or thing you want from it
<code>?size=10</code>	<code>query</code>	— extra details ("the size-10 ones")

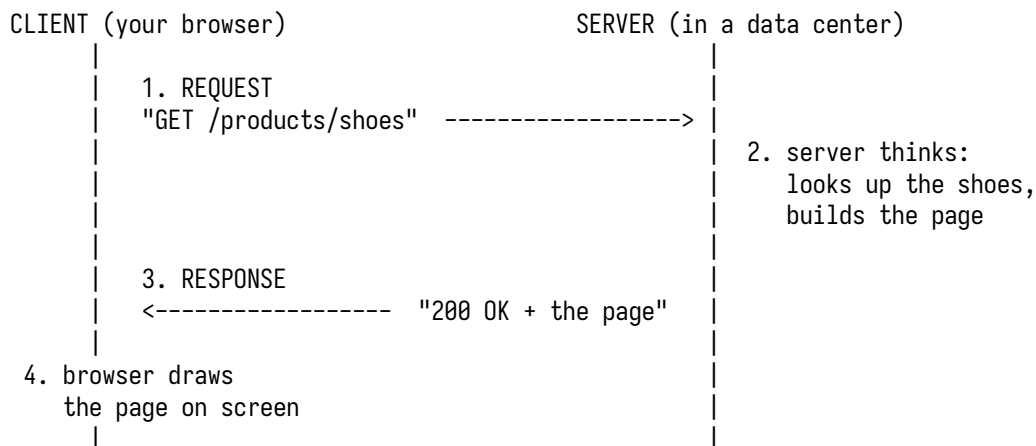
逆さの郵便の宛名のように読もう:

- **scheme**(https) はプロトコルだ—会話のルール。今日ではほぼ常に https だ。
- **host**(shop.example.com) はどのサーバーに連絡するかを名指す。
- **path**(/products/shoes) はそのサーバーのどのものが欲しいかを名指す。
- 任意の **query**(?size=10) は、フィルタや検索語のような追加パラメータを運ぶ。

誰かが「エンドポイント」や「ルート」と言うとき、それはほぼ常に、特定の host の特定の path を意味する。あの怖そうな言葉が意味するのはそれだけだ。

リクエストとレスポンスのサイクル

ここに、最初から最後までダンスがある。あなたがアドレスを入力すると、見えないところで一秒もかからぬうちにこれが起きる:



毎回、四つのステップだ:

1. クライアントが URL へ **request** を送る。
2. サーバーがそれを**処理する**—何かを調べたり、ロジックを走らせたり、あなたが誰かを確認めたりするかもしれない。
3. サーバーが **response** を返す: ステータス (うまくいったか?) と、いくつかのコンテンツ (ページ、画像、またはデータ)。
4. クライアントがそのレスポンスで何かをする—たいていは、あなたの画面に描く。

ウェブページを一つ読み込むのは、リクエスト一つではない—何十もだ。ブラウザはページを要求し、それから画像・フォント・スタイルが必要だと気づき、それぞれに対して新しいリクエストを撃ち出す。同じダンスを、何度も、とても速く。

HTTP メソッド: 動詞たち

すべてのリクエストは **メソッド** を帯びる—あなたがどんな種類の動作を望むかを言う動詞だ。いくつもあるが、四つがほぼすべてを覆い、日常の観念にきれいに当てはまる:

- **GET** —「これをくれ」。読む、取ってくる、見る。ページや一覧の読み込み。GET は決して何も変えてはならない; ただ取得するだけだ。
- **POST** —「新しいものがここにある」。作成する。フォームの送信、サインアップ、コメントの投稿。
- **PUT** —「これを更新せよ」。プロフィールの編集のように、すでに存在するものを変える。(部分更新には **PATCH** という言葉も耳にする—同じ発想、より狭い範囲。)
- **DELETE** —「これを取り除け」。聞こえるそのまま—レコードやファイルを削除する。

ほかは全部忘れても、これだけは:**GET は読み、POST は作り、PUT は更新し、DELETE は取り除く**。アプリが不調なとき、どの動詞が送られたかを知ることが、しばしば最初の手がかりになる。

ステータスコード: うまくいったか?

すべてのレスポンスは三桁の **ステータスコード** とともに返ってくる—リクエストがどうなったかについての、サーバーの一目での判定だ。最初の桁でグループ分けされる:

- **2xx —成功**。うまくいった。200 OK は「ほら、君が要求したものだ」という日常の応答だ。
- **3xx —リダイレクト**。「ここにはない、代わりにあっちを見る」。ありふれていて無害だ; ブラウザが自動で追う。
- **4xx —あなた (クライアント) が間違えた**。有名なのは **404 Not Found** —存在しない path を要求した。(401/403 もある—ログインしていないか、許されていない。)
- **5xx —サーバーが間違えた**。**500 Internal Server Error** は、リクエストを処理している最中にサーバー自身のコードがクラッシュしたか壊れたという意味だ。

4xx/5xx の区別は頭に刻んでおく価値がある。どこを見るべきかを教えてくれるからだ:

- **404** はリクエストが間違っていたという意味だ—たいていは不正な URL かタイプミスの path。何が要求されているかを見よ。
- **500** はサーバーが倒れたという意味だ。あなたのリクエストは問題なかった; バックエンドのコードが壊れた。URL ではなく、サーバーのログを見よ。

パイプコーディングで作ったアプリで何かが壊れたとき、「これは 4xx か 5xx か?」は、切り分けに使える最速の問いの一つだ。

ステータスコード

— 2xx —▶	成功	うまくいった (200 OK)
— 3xx —▶	リダイレクト	「あっちを見て」 (自動で追従)

4xx	→ クライアントエラー	あなたの要求が誤り	→ URL/path を確認
5xx	→ サーバーエラー	バックエンドが故障	→ サーバーログを確認

ヘッダーとクッキー

本文コンテンツと並んで、すべてのリクエストとレスポンスは **ヘッダー** を帯びる—追加情報の入った、ラベル付きの小さなメモだ。中の手紙とは別の、封筒に書かれた文字のようなものだ。ヘッダーは、コンテンツがどんな形式か (Content-Type: text/html) や、誰が尋ねているか、といったことを言う。

クッキー はヘッダーの、ある一つの具体的で重要な使い道だ。ウェブは記憶力が短い: 既定では、各リクエストは単独で立っており、サーバーは答えた瞬間にあなたを忘れる。クッキーがそれを直す。あなたがログインしたあと、サーバーはあなたのブラウザに小さなトークン・クッキーを手渡し、あなたのブラウザはそれ以降のすべてのリクエストにそれを自動で添えて送る。それが、サイトがページをまたいであなたがログイン中だと「覚えている」やり方だ。クッキーがなければ、記憶もない; クリックするたびに毎回ログインする羽目になる。

HTTPS と南京錠

アドレスバーの小さな南京錠を見たことがあるだろう。それは接続が **HTTPS** (ハイパーテキスト・トランスファー・プロトコル・セキュア) を使っているという意味だ—**TLS** と呼ばれる暗号化の層で包まれた HTTP (ハイパーテキスト・トランスファー・プロトコル) である。

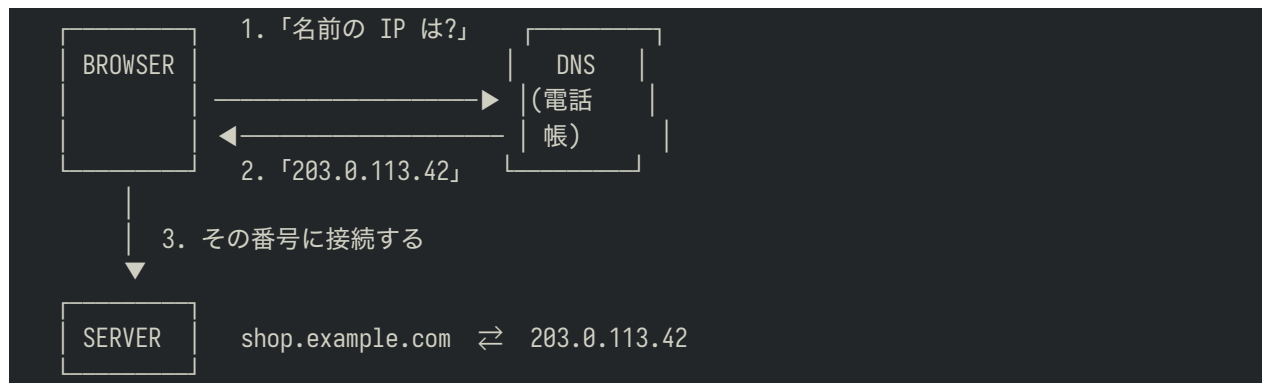
素の HTTP はすべてを読めるテキストで送る。あなたとサーバーのあいだに座る者は誰でも—たとえば共用 Wi-Fi で—あなたのパスワードが通り過ぎるときにそれを読めてしまう。郵便経路の誰もが読めるはがきのように。HTTPS はその会話を、あなたとサーバーだけが開けられる封筒に封じる。南京錠は二つのことを確かめてくれる:

- **プライバシー**—途中にいる誰も、あなたが送るものを読めない。
- **身元**—あなたは本当に、自分が思っているそのサーバーと話しており、なりすましではない。

実用上のルールは短い: 本物のアプリは HTTPS を使う、常に、例外なく。ログインや決済を素の HTTP で扱うのは、設計からして壊れている。

名前からサーバーへ:DNS

片付いていない糸口が一つ。あなたは shop.example.com を入力するが、コンピュータは実は名前で互いを見つけない—203.0.113.42 のような数字の **IP アドレス** を使う。**DNS** (Domain Name System) はウェブの電話帳だ: あなたのブラウザがリクエストを送れるようになる前に、それは静かに DNS に、親しみやすい **ドメイン** 名を数字のアドレスに翻訳してくれと頼み、それからそこへ接続する。



あなたはドメインを買い、その DNS レコードがあなたのアプリを動かすサーバーを指すようにし、それ以降その名前は人々をあなたのマシンへ導く。めったに目にしない調べもののステップだが、覚えやすい名前が数字の

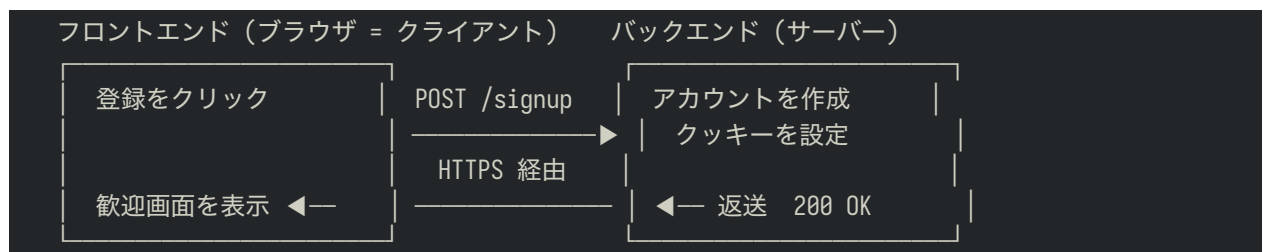
羅列の代わりになれるのは、まさにこれゆえだ。

これがあなたのアプリにどう対応するか

さあ、この絵の全体をバイブコーディングで作ったアプリの上に重ねてみよう。あなたが作るすべての部品が、このダンスの片側に住んでいるからだ：

- ・ **フロントエンド**はブラウザで動くすべてだ—ユーザーが見てクリックするボタン、フォーム、レイアウト。それがクライアントだ。おもにリクエストを送り、レスポンスを表示する。
- ・ **バックエンド**はサーバーで動くあなたのコードだ—リクエストをどう扱うかを決め、データベースと話し、レスポンスを組み立てるロジック。それがサーバーだ。

サインアップの旅の全体を一枚の絵にすると、こうなる—この章のすべての太字の言葉が、あるべき場所に収まっている：



本物のやり取りはそのすべてを縫い合わせる。ユーザーが「サインアップ」をクリックする (フロントエンド)。ブラウザはフォームのデータを載せて、/api/signup のような path へ **HTTPS** 越しに **POST** リクエストを送る。あなたのバックエンドがそれを受け取り、アカウントを作り、ユーザーがログイン状態を保てるよう **クッキー** を設定し、**200** を返す。フロントエンドは成功を見て、ようこそ画面を見せる。そこで太字になっているすべての言葉は、あなたがたった今学んだものだ—そしてそれが、あなたが作るほぼすべてのものの全体の形だ。

これらすべてを一度に頭に抱えておく必要はない。だが後の章が「リクエストはサーバーに届いたのに 500 で返ってきた」、あるいは「このヘッダーを設定せよ」、あるいは「このルートが POST を処理する」と言うとき、あなたは今、それがダンスのどの部分を指しているかを正確に分かる。そのメンタルモデルこそが本当の前提条件だ—そしてあなたはもうそれを手にしている。

まとめと実践

重要なポイント

- ・ ウェブは延々と繰り返される一つのシンプルなダンスです：**クライアント** (たいていはブラウザ) が **リクエスト** を送り、**サーバー** が **レスポンス** を返します。あとはすべて、その上に乗った細部にすぎません。
- ・ **URL** は、ラベル付きの部品をつなぎ合わせたものです—scheme・host・path・query。メソッドは動詞です：**GET は読み、POST は作り、PUT は更新し、DELETE は消します**。
- ・ **ステータスコード** はサーバーの一目での判定です。4xx/5xx の区別がどこを見るべきかを教えてください：**4xx** はリクエストが間違っていた (不正な URL)、**5xx** はサーバー自身のコードが壊れた、ということです。
- ・ **HTTPS** (鍵マーク) は会話を封じ、途中の誰にも読めないようにします—本物のアプリは常に使います。**クッキー** は、複数のリクエストにまたがってサイトがログイン状態を覚えておく仕組みです。
- ・ **フロントエンド** はクライアント (ブラウザで動く)、**バックエンド** はサーバー (ロジックを動かしデータベースと話す) です。あなたが作るすべての機能は、このダンスのどちらかの側にあります。

やってみよう

どれでもいいのでウェブサイトを開き、ブラウザの開発者ツールを開きます (右クリック ▢ 「検証」、そして「Network」タブ)。ページを再読み込みします。リクエストのリストが埋まっていくのを見てください—1 つのページリクエストに続いて、画像・フォント・スタイルのために数十ものリクエストが出てくるのが、説明したとおりです。一つをクリックして、その **メソッド** (GET)、**ステータスコード** (うまくいけば 200)、**ヘッダー** を見てください。いまあなたは、リクエスト/レスポンスのダンスがリアルタイムで起きるのを、それを読むための語彙とともに見ているのです。

この章のプロンプト

私は初心者としてウェブの仕組みを学んでいます。自分のアプリを例にして：
< あなたのアプリを説明するか、URL や今見ているエラーを貼り付けてください >

- 一つの具体的な動作（ページの読み込みやフォームの送信など）について、リクエスト/レスポンスのサイクルを一步步順を追って説明してください。
- どの部分がクライアント（フロントエンド）で、どの部分がサーバー（バックエンド）かを教えてください。
- ステータスコードやエラーが見えている場合、それが 4xx(私のリクエストが間違っている) か 5xx(サーバーが壊れた) かと、どこを見ればよいかを説明してください。
- 具体的で初心者になさしく —前提知識なしで説明してください。

API、JSON、そしてデータ



あなたが作るアプリのほとんどは、単独では存在しない。地図を表示したい、カードで決済したい、メールを送りたい、天気を調べたい——そう思った瞬間に、あなたのアプリはほかの誰かのアプリと会話する必要が出てくる。そのときに使われる言葉が **API**（アプリケーション・プログラミング・インターフェース）であり、そこでやり取りされる言葉のほとんどは **JSON**（ジャバスクリプト・オブジェクト・ノーテーション）という形式で書かれている。この章はその両方についての話だ——配管そのものを手書きできるようになるためではなく、それを読むようになり、AI が組んだものをチェックでき、お金がかかる前に何かおかしいと気づけるようになるためである。

ここにあることを暗記する必要はない。必要なのは、形を見分けられるようになることだ。AI が「Stripe API を呼び出して、JSON レスポンスをパースします」と言ったとき、それがおおよそ何を意味し、何がうまくいかな

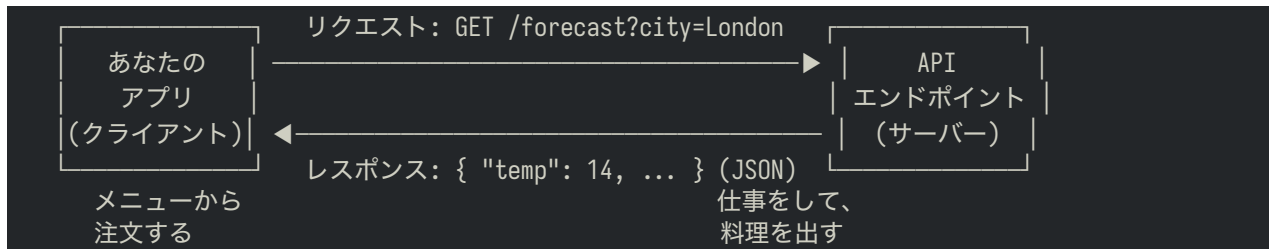
くなり得るのかをイメージできればいい。

API とは実際のところ何なのか

API を考える一番すっきりした方法は、**あるプログラムが別のプログラムに提供する、リクエストのメニュー**として捉えることだ。

レストランは、あなたが厨房に入って料理をすることを許さない。代わりにメニューを渡してくる。これが注文できるもの、これが注文の仕方、これが返ってくるもの、というわけだ。厨房がどう動いているかを知る必要はない。必要なのはメニューだけである。API はまさにそれだ——ある会社が「これが、あなたが私たちのサービスに対して行ってよいリクエストです。そして、それぞれが何を返すかはこうです」と言っているのである。

AI が「天気 API を呼び出す」コードを書くとき、それはそのメニューから注文を出しているのだ。この都市の予報をくれ、と。天気会社のサーバーが作業をして、答えを返してくる。あなたのアプリは、彼らのデータベースもコードも一切見ない。ただ料理を受け取るだけである。



ここで重要なのは考え方の転換だ。API は**契約**である。相手の会社は、あなたが特定の方法で尋ねれば、特定の形で答えを返すと約束している。あなたの仕事——そして AI の仕事——は、正しく尋ね、答えを正しく読むことだ。API のバグのほとんどは、どちらかがこの契約を破ることで起きる。

エンドポイントと REST を、平たい言葉で

エンドポイントはメニューの一品だ。特定のひとつのことを行う URL (ユニフォーム・リソース・ロケーター) にすぎない。たとえば次のようなものだ。

- <https://api.weather.com/forecast> — 予報を取得する
- <https://api.weather.com/cities> — 利用可能な都市を一覧する

同じ厨房、違う料理。それぞれのエンドポイントがひとつの機能である。

あなたが触れる API のほとんどは、**REST** (リプレゼンテーション・ステート・トランスファー) と呼ばれるゆるいスタイルに従っている。学術的な定義は要らない。必要なのは次の 2 つの考えだ。

- **URL はモノを指す** (「リソース」) —— ユーザー、注文、予報など。
- **動詞は、それに対して何をしたいかを表す**。これらの動詞は HTTP (ハイパーテキスト・トランスファー・プロトコル) メソッドであり、ウェブの仕組みの章に出てきたものと同じだ。
 - GET — 何かを読む (予報を取得する)。安全で、何も変えない。
 - POST — 何かを作る (注文を出す、ユーザーを登録する)。
 - PUT / PATCH — すでに存在する何かを更新する。
 - DELETE — 何かを削除する。

だから GET /orders/42 は「注文番号 42 を読む」を意味し、DELETE /orders/42 は「それを削除する」を意味する。同じ名詞、違う動詞、まったく違う効果だ。API ドキュメントや差分を読むとき、この 2 語——**メソッド**と **URL**——が、リクエストが何をやるかのほとんどを教えてくれる。

JSON とは何か

あなたのアプリが API に質問を投げると、その答えは **JSON** という形式のテキストで返ってくる。これがデフォルトになったのは、人間にとって読みやすく、かつプログラムにとって扱いやすいからだ。あなたはこれを絶

え間なく目にするようになるので、読めるようになっておく価値がある。

JSON は**キーと値**から作られている。キーは引用符に囲まれたラベルで、値はそれに紐づくデータだ。次は、ひとりのユーザーを表す実際の例である。

```
{
  "id": 42,
  "name": "Ada Lovelace",
  "email": "ada@example.com",
  "isAdmin": false,
  "loginCount": 137,
  "tags": ["beta", "early-access"],
  "profile": {
    "city": "London",
    "avatarUrl": null
  }
}
```

誰かが記入したフォームのように読めばいい。"name" は項目で、"Ada Lovelace" はそこに書き込まれた内容だ。2 つの構造がすべての仕事をこなす。

- **object** は波括弧 { } で囲まれる。キーと値のペアの束で——ひとつのレコードのようなものだ。上の例全体がひとつの object である。
- **array** は角括弧 [] で囲まれる。順序のついた値のリストだ。上では "tags" が 2 つの string からなる array を保持している。

"profile" が object 中の object になっていることに注目してほしい。JSON は入れ子になる——値そのものが object や array になり得て、必要なだけ深くできる。だからこそ、ひとつのレスポンスでユーザーと、その住所と、直近 10 件の注文をまとめて表現できるのだ。

同じユーザー object をツリーとして描くと、その形がひと目でわかる——上には平らな項目が並び、そこからリストひとつと入れ子の object が枝分かれしている。

```
{ } user (オブジェクト)
├─ id ..... 42 (数値)
├─ name ..... "Ada Lovelace" (文字列)
├─ isAdmin ..... false (真偽値)
├─ tags ..... [ ] (配列)
│   ├── "beta"
│   └── "early-access"
└─ profile ..... { } (オブジェクト)
    ├── city ..... "London" (文字列)
    └─ avatarUrl ..... null (null = 値なし)
```

出会うことになるデータ型

JSON のすべての値は、ごく小さな型のセットのいずれかである。これらを知っておくと、値の種類が間違っているとき——これは頻出のバグの原因だ——にそれを見抜ける。

- **string** —テキスト。常にダブルクォートで囲む。"Ada Lovelace"、"ada@example.com"。
- **number** —そのままの数値。引用符なし。42、3.14。(引用符は意味を変える。"42" は数値ではなくテキストの「よん・に」だ。)
- **boolean** —true か false。引用符なし。"isAdmin" のような yes/no のフラグに使う。
- **null** —意図的な「ここには値がない」だ。"avatarUrl": null はユーザーにアバターがないことを意味する——キーそのものが存在しないのとは違う。
- **array** —順序のついたリスト。["beta", "early-access"]。

- **object** —キーと値のペアの束。"profile" のブロックがそれだ。

語彙はこれで全部である。実世界で最も多い間違いは、string と number の取り違えの罠だ。コードは number の 42 を期待しているのに、API が string の "42" を送ってきて、比較が静かに失敗する。何かが妙な振る舞いをするとき、値の型を確認することが解決策になることは多い。

実際のリクエストとレスポンス

実際の API 呼び出しが端から端までどう見えるかを示す——あなたのアプリが送るリクエストと、返ってくる JSON だ。

```
GET /forecast?city=London HTTP/1.1
Host: api.weather.com
Authorization: Bearer sk_live_8f4b2c...
Accept: application/json
```

```
{
  "city": "London",
  "unit": "celsius",
  "current": {
    "temp": 14,
    "condition": "cloudy"
  },
  "forecast": [
    { "day": "Mon", "high": 16, "low": 9 },
    { "day": "Tue", "high": 18, "low": 11 }
  ]
}
```

あの Authorization の行が次のトピックだ——そして、最も正しく扱う価値のあるものである。

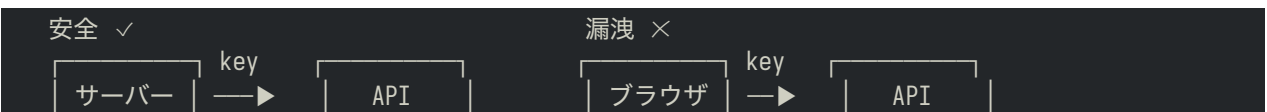
API キーと認証ヘッダー

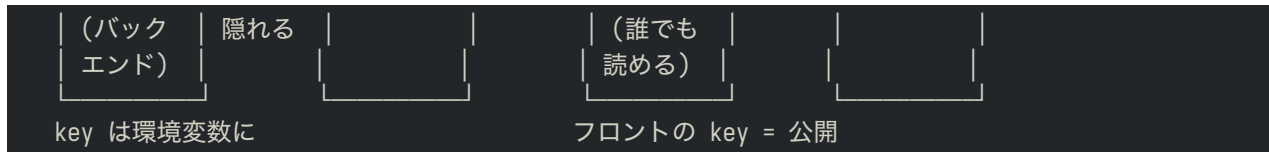
役に立つ API のほとんどは、世界に開かれてはいない。会社は誰が尋ねているのかを知る必要がある。利用状況を追跡し、あなたに課金するためだ。会社はあなたに **API キー** を渡す——あなたのアカウントを識別する、長い秘密の文字列だ。あなたはそれをすべてのリクエストに付ける。通常は Authorization という **HTTP ヘッダー** に、上の Bearer sk_live_... の行のように添える。

API キーはパスワードとまったく同じように扱うこと。実際それがパスワードなのだから。それを持っている者は誰でも、あなたとしてリクエストを送れる——そしてあなたの請求額を膨らませたり、データを読んだりできる。これはセキュリティの章に直結する話で、最も重要なルールがひとつある。

- **API キーを、ブラウザに配信されるコードに絶対に置かないこと。** フロントエンドにあるものはすべて公開されている——ユーザーは開発者ツールを開いてそれを読める。「ページから API を呼び出して」と頼まれた AI は、あなたの秘密のキーをそこに嬉々として貼り付ける。止めること。秘密のキーはサーバーにあるべきだ。
- **キーを git にコミットしないこと。** あとで削除したとしても、それは履歴の中に永遠に生き続けるし、ボットは公開リポジトリを数分以内にスキャンしてキーを探している。
- **キーは環境変数に保存すること。** コード自体の中ではなく——セキュリティの章で扱うのと同じパターンだ。

キーは安全な場所から運ばなければならない、決してブラウザから運ばれてはならない。2 つの経路を思い描いてほしい——ひとつは安全、ひとつは漏洩だ。





もし本物のキーが差分の中、フロントエンドのファイルの中、あるいはチャットメッセージの中に居座っているのを見かけたら、それはすでに漏洩したものとして扱うこと。すぐにローテーションする（新しいものを生成し、古いものを無効化する）のだ。

レート制限と、請求ショックの罠

メニューが大きな声で言わないが、間違いなくあなたを噛んでくるものがもう 2 つある。

レート制限。 API は、ある一定の時間枠の中で行えるリクエスト数に上限を設けている——たとえば 1 分あたり 100 回といった具合だ。それを超えると、速度を落とすまで 429 Too Many Requests エラーで拒否される。これは、ひとりの顧客がサービスを圧倒してしまうのを防ぐために存在する。罠はこうだ。AI が書いたコードは、しばしば一時停止なしのループの中で API を呼び出し、あっという間に制限を突破する。ある機能が 1 件では動くのに 1000 件以上で動かすと壊れるなら、レート制限が第一の容疑者だ。

従量課金・有料の API。 多くの API はリクエストごとに課金する——1 回あたり 1 セントの何分の 1 かといった額だ。数回の呼び出しなら何でもないが、何かが制御不能にループすると本物の請求になる。古典的な恐怖譚はこうだ。バグが失敗した有料の呼び出しを永遠にリトライし続ける、あるいは公開フォームが見知らぬ人に有料の呼び出しを起こさせて、朝起きたら 4 桁の請求書が届いている。自分を守ろう。

- 有料の API をつなぐ前に料金を読むこと。1 回あたりのコストを把握する。
- プロバイダーのダッシュボードに**支出上限や予算アラート**があるなら設定すること。たいていある。
- 有料の API をループの中や、ページ読み込みのたびに呼び出すようなコードは、すべて疑ってかかること。

API ドキュメントの読み方

あなたは API ドキュメントに住み込むことになるので、全部を読まずに必要なものを素早く手に入れる方法を示す。どのエンドポイントについても、5 つのものを狩りに行こう。

- **メソッドと URL** —GET /forecast。それが何をするかが、2 つのトークンでわかる。
- **認証**—どのヘッダーやキーが必要か。たいていは API 全体で共通のセクションがひとつある。
- **パラメーター**—送れる入力（ここでは city）、どれが必須か、そしてその型。
- **レスポンスの例**—返ってくる JSON の形だ。これは宝物である。コピーすること。
- **エラー**—何かおかしいときに何が返ってくるか、そしてステータスコード（401 認証されていない、404 見つからない、429 レート制限）。

よいドキュメントには、コピペできるリクエストとレスポンスの例が含まれている。この 2 つのスニペットは、何段落もの散文よりも多くを教えてくれる——それをつかんで、AI に手渡そう。

呼び出しは AI にやらせ、形はあなたが確認する

ここで実践的な分業を示す。AI は API を呼び出すコードを書くのが本当に得意だ。ライブラリも、ヘッダーの構文も、エラーハンドリングも知っている。やらせればいい。この配管を手書きする必要はない。

だが、あなたは AI が苦手なことをひとつ担っている。それは**データの形が実際に正しいかどうかを知ること**だ。よいワークフローはこうだ。

- **AI に本物のドキュメントを渡す。** ドキュメントにあるリクエストとレスポンスの例を、あなたのプロンプトに貼り付ける。AI が記憶から API の形を推測するのは、バグの最大の発生源だ——存在しない、もっともらしく見えるキーをでっち上げてしまう。本物の例が、AI を現実につなぎ留める。

- ・ **まず生のレスポンスを見せるよう頼む**。「パースする前に、API が実際に返す JSON をログに出して」それから、コードが期待しているものと照らし合わせて目視する。コードが読んでいるキー (`response.current.temp`) は、レスポンスの中に実際に存在するか？ API のバグの半分は、キー名の食い違いだ。
- ・ **型を確認する**。それは `number` か、`string` か？ 常に存在するべき項目が、ときどき `null` になっていないか？ あなたはもう JSON が読める——それを使おう。
- ・ **不幸な経路を処理する**。こう尋ねよう。「この呼び出しが失敗したら、タイムアウトしたら、レート制限に当たったら、どうなる？」 答えが「アプリがクラッシュする」なら、それは機能ではなくバグだ。

リクエストを書く必要はない。必要なのは、レスポンスを読んで「これは私が期待した形か？」と尋ねることだ。このたったひとつの習慣——データを契約と照らし合わせて確認すること——が、統合バグの大半を、ユーザーが目にする前に捕まえてくれる。

まとめと実践

重要なポイント

- ・ **API** は、あるプログラムが別のプログラムに差し出すリクエストのメニューであり、**契約**です：合意した方法で尋ねれば、合意した形の答えが返ります。ほとんどの API バグは、どちらかがその契約を破ることです。
- ・ **JSON** は答えが返ってくる形式です—キーと値からなり、オブジェクト `{ }` と配列 `[ ]` で組み立てられ、入れ子になります。データ型は `string`・`number`・`boolean`・`null`・`array`・`object` です。
- ・ **文字列 vs 数値の罠** (`42` と `"42"`) と、欠落 vs `null` が最もよくある形のバグです。もう JSON は読めるのですから—何かおかしい挙動をしたら型を確認しましょう。
- ・ **API キー** はパスワードです。決してブラウザに送らず、決して git にコミットせず、環境変数に保存しましょう。漏れたキーは直ちにローテート (`rotate`) すべきです。
- ・ **レート制限** (`429`) と **従量課金** に注意しましょう—有料 API を休みなく呼ぶ AI 製のループが、四桁の請求書を生む道です。まず料金を読み、予算アラートを設定しましょう。

やってみよう

ブラウザでキー不要の無料 API を探してみましょう—たとえば新しいタブで <https://api.github.com/users/octocat> を直接開きます。生の JSON レスポンスが見えます。フォームのように読みましょう：オブジェクト `{ }` を拾い出し、配列があれば見つけ、異なる値を三つ選んでその型に名前をつけます（どれが文字列？数値？真偽値？`null` のものはある？）。この章が、AI がつなぐすべての API レスポンスに対してやってほしいと言っている、まさにその点検をいま行ったことになります。

この章のプロンプト

私は API を呼び出していて、初心者として自分でデータの形を確認したい。
これが本物のドキュメントです（例のリクエスト + 例のレスポンス）：
<API ドキュメントの例のリクエストと JSON レスポンスをここに貼り付けてください >

- パース用コードを書く前に、この呼び出しが返す生の JSON を見せて、キー、その型、そして `null` や欠けている可能性のあるものを指摘して。
- 私のコードが読むキーが、そのレスポンスに実際に存在するか確認して。
- API キーがサーバー上の環境変数から読み込まれ、ハードコードされたりブラウザに送られたりしないようにして。
- 呼び出しが失敗したり、タイムアウトしたり、レート制限に達したらどうなるか教えて。

アプリの解剖学



AIに「ログインページを追加して」とか「これをデータベースに保存して」とお願いするとき、AIは裏側で半ダースほどの部品について静かに判断を下しています。それらの部品が存在することを知らないと、返ってくる答えはまるで外国語のように見えてしまい、良い答えと混乱した答えの区別もつきません。

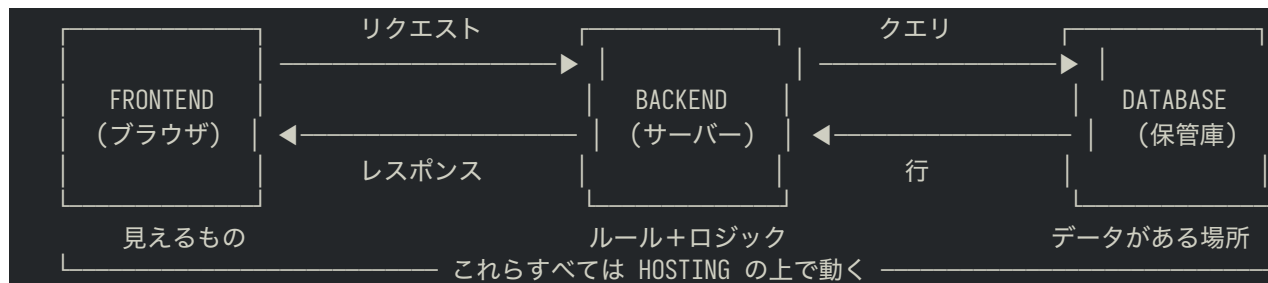
この章は地図です。これを読んだからといってエンジニアになれるわけではありませんが、現代のアプリのどの部品でも指さして「これは何をやるもので、どこにあるのか」を言えるようになります。地図さえ手に入れば、ほかのすべての章が抽象的なものではなくなります。問題を説明し始める前に、それがどの箱の中にあるのかが分かるからです。できるだけ平易に進めます。定義なしに専門用語は使いませんし、一つの一貫した例（ユーザーがボタンをクリックする）をすべての層を通して追いかけていきます。

全体像

あなたがこれから作るほとんどのアプリは、互いにやり取りをする4つの部品でできています。

- **frontend (フロントエンド)** —ユーザーが見て、クリックする部分。ブラウザやスマホアプリの中にある画面、ボタン、文字です。
- **backend (バックエンド)** —あなたが管理するサーバー上で動くロジック。ルールを守らせ、作業をこなし、誰に何を許可するかを決めます。
- **データベース** —情報が明日も残っているように保管される場所。アカウント、投稿、注文など。
- **ホスティング** —世界のどこかにある実際のコンピューターで、上記すべてを実際に動かし、ほかの人がアクセスできるようにしてくれます。

これらがどうつながっているかを示します。



frontend がデータベースに直接話しかけることは決してありません。必ずルールが置かれている backend を経由します。このたった一つの事実が、アプリがどのように作られているか——そしてなぜ「ページにデータベースを直接読ませればいい」がほとんど答えにならないのか——を多く説明してくれます。

frontend、backend、データベース、ホスティング

例を使ってもう一段深く見ていきましょう。ユーザーが「**プロフィールを保存**」をクリックします。

frontend。目に見えるアプリで、Web ブラウザの中、あるいはモバイルアプリとして動きます。フォームを描画し、クリックに気づき、ユーザーが入力した内容を集めます。設計上、これは信頼されていません。誰でも開いて、中身を覗き、書き換えることができます。ですからその役割は、見た目を良くして反応を軽快にすることであって、重要なことを守らせることではありません。

backend。ボタンがクリックされると、frontend はデータを backend——サーバー上で動くコード——に送ります。backend はユーザーがログインしているかを確認し、データを検証し（「これは本当にメールアドレスかな？」）、ビジネスルールを適用し、そのうえで初めて保存します。ここに信頼が宿るのは、ユーザーが手出しできないからです。

データベース。backend はデータをデータベースに渡して保管します。魔法の箱ではありません。瞬時に検索でき、決して忘れない、とても厳格な表計算（テーブルと呼ばれます）のようなものだと考えてください。明日、backend はそのプロフィールをふたたび読み出して表示します。

ホスティング。prod（本番）では、これらは一切あなたのノートパソコンの上では動きません。Cloudflare、Vercel、あるいはクラウドプラットフォームのようなホスティングプロバイダーから借りたコンピューターの上で動きます。ホスティングこそが、あなたのアプリを自分のマシンの中だけでなく、本物の Web アドレスでアクセスできるようにしてくれるものです。

役に立つ勘どころがあります。何かが壊れたら、どの層かを問うことです。「ボタンを押しても何も起きない」は frontend。「間違ったものが保存される」はたいてい backend。「データを忘れてしまった」はデータベースを指しています。層に名前をつけることが、バグを直す作業の半分です——そして AI が動けるプロンプトを書く作業の半分でもあります。

HTML、CSS、JavaScript

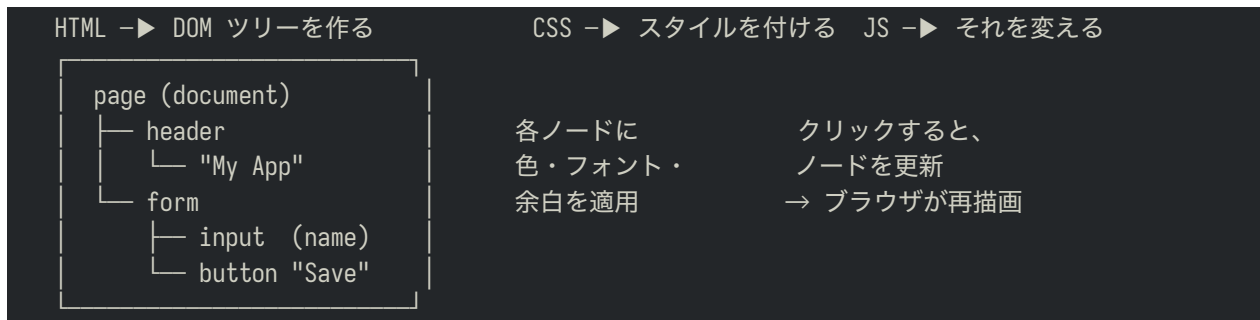
ブラウザの中の frontend は、ちょうど 3 つの言語でできています。手で書く必要はありませんが、それぞれが何を担当しているのかは知っておく必要があります。

HTML（ハイパーテキスト・マークアップ・ランゲージ）は構造——内容とその骨組みです。「ここに見出しがある、ここに段落がある、ここにボタンがある」と言います。ページの名詞のようなものです。HTML だけだと、ラベルの付いたテキストファイルのような、飾り気のない素のドキュメントです。ブラウザはあなたの HTML を、**DOM**（Document Object Model、ドキュメントオブジェクトモデル）と呼ばれる、メモリ上の生きたオブジェクトの木に変えます。コードが「ページを変える」とき、実際にはこの DOM を変えていて、ブラウザはそれに合わせて描き直しているのです。

CSS（カスケーディング・スタイル・シート）はスタイル——すべての見た目です。色、フォント、余白、レイアウト、左寄せか中央寄せか、小さい画面か大きい画面か。CSS はページに何があるかは変えず、それがどう見せられるかだけを変えます。HTML が名詞なら、CSS は形容詞です。

JavaScript (JS) は振る舞い——あなたが操作したときに何が起きるかです。クリックを待ち受け、データを backend に送り、スピナーを表示し、答えが返ってきたら DOM を更新します。HTML と CSS は静的です。**JavaScript** はページに何かをさせます。動詞です。(同じ言語は backend を動かすのにもよく使われるので、あちこちで見かけることになります。)

この役割分担をイメージする方法：HTML が木 (DOM) を作り、CSS がそれに色を塗り、JavaScript がその中に手を伸ばして変えます。



パッケージと依存関係

誰もアプリをまるごとゼロから書いたりはしません。難しくてありふれた問題——日付の整形、決済の処理、カレンダーの描画——は、すでにほかの人たちによって解決され、パッケージにまとめられています。**パッケージ**とは、自分で書く代わりにプロジェクトに取り込む、再利用可能なコードのかたまりです。プロジェクトの**依存関係** (dependencies) とは、それが頼りにしているパッケージの一覧です。

JavaScript の世界でこれを管理するツールが **npm** (Node Package Manager) です。2 つのファイルが帳簿づけをします。

- **package.json** —プロジェクトが必要とするものの、人が読める一覧と、それを動かすためのコマンド。あなた (または AI) がこれを編集します。
- **lockfile (ロックファイル)** (package-lock.json など) —すべてのパッケージとその正確なバージョンの、機械が生成した厳密な記録です。これがあるおかげで、あなたのマシン、チームメイトのマシン、サーバーのどこでもアプリが同じようにビルドされます。これは手で編集せず、ツールに管理させます。

```
{
  "name": "my-app",
  "scripts": {
    "dev": "next dev",
    "build": "next build"
  },
  "dependencies": {
    "next": "16.0.0",
    "react": "19.0.0"
  }
}
```

なぜ気にするのでしょうか？ AI が「このためにライブラリを追加しましょう」と言うとき、それは **package.json** を編集していて、あなたは誰かほかの人のコードを信用していることになります——よく知られた少数の依存関係のほうが、得体の知れないものの山よりも安全です。そしてビルドが原因不明で壊れたとき、依存関係の食い違いがよくある容疑者であり、「パッケージを削除して入れ直す」が本当によくある現実的な対処法になります。

環境変数と設定

コードの中に書き込めない値があります。秘密の鍵、データベースのパスワード、サービスのアドレスなど。そしてアプリがどこで動くかによって変わる必要があるものもあります。ノートパソコンでテストするコード (**dev**

環境) はテスト用のデータベースに話しかけるべきですし、ユーザーが触れる本番のアプリ (**prod**、つまり本番 (production) 環境) は本物のほうに話しかけなければなりません——同じコードを使ってです。

その答えが **環境変数** (よく「env vars (env 変数)」と呼ばれます) です。コードの外側にある名前付きの値で、コードが動くときに供給されます。コードは「DATABASE_URL をちょうだい」と言い、環境がどれを手渡すかを決めます。

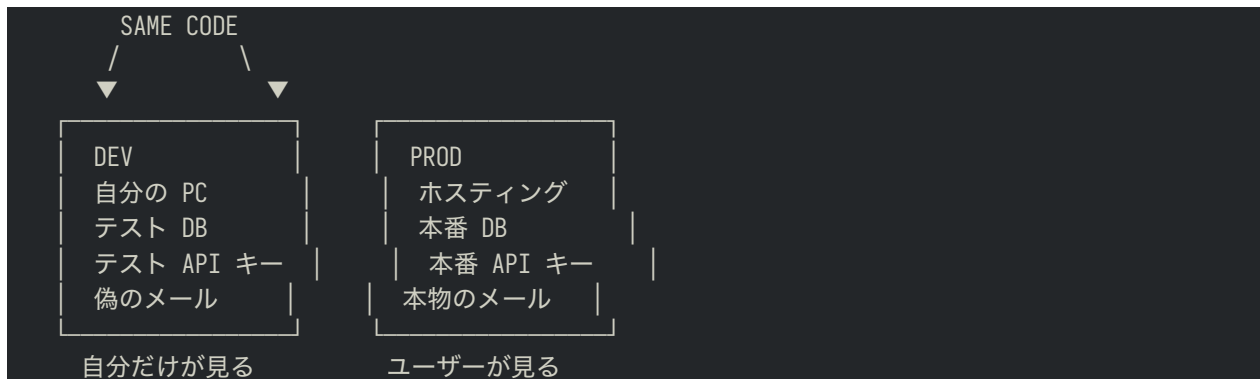
設定ファイル (例: .env) —値はコードの中ではなく、ここに置く

```
DATABASE_URL=postgres://localhost/myapp_dev
STRIPE_SECRET_KEY=sk_test_51H...      # dev = テスト用の鍵
SEND_REAL_EMAILS=false                 # テスト中は本物のユーザーにメールを送らない
```

2 つのルールが、本物の痛みからあなたを救ってくれます。

- ・ **秘密の情報は決してコードに入れない**。コードの中にあるものは何でも公開されてしまう可能性があります (特に共有リポジトリでは)、漏れた決済鍵やデータベースのパスワードは正真正銘の緊急事態です。秘密の情報は env 変数に置き、バージョン管理の外に保ちます。
- ・ **dev と prod は別物**。肝心なのは、同じコードが両方で正しく振る舞うことです。なぜなら、どのデータベースと鍵を使うかを決めるのはコードではなく環境だからです。

同じコード、二つの住み家——違うのは env 変数だけです。



プロジェクトの形

典型的なプロジェクトを開くと、フォルダの一覧はノイズのように感じられるかもしれません。でもそうではありません。ほとんどのプロジェクトは見覚えのあるパターンに従っています。これは余分なものを削ぎ落とした Web アプリの例です。

```
my-app/
├── package.json          # 依存関係+実行コマンド (上で見たもの)
├── package-lock.json     # lockfile —正確なバージョン
├── .env                  # ローカルの秘密情報/設定 —共有しない
├── .gitignore            # Git が無視すべきファイル (.env など)
├── README.md             # このプロジェクトは何で、どう動かすか
├── public/               # そのまま配信される静的ファイル (画像、アイコン)
├── src/                  # 実際のコードはここにある
│   ├── components/       # 再利用可能な frontend の部品 (ボタン、カード)
│   ├── pages/ (or app/)  # ユーザーが移動する画面/ルート
│   ├── lib/              # 共有のヘルパーと backend のロジック
│   └── styles/           # CSS
```

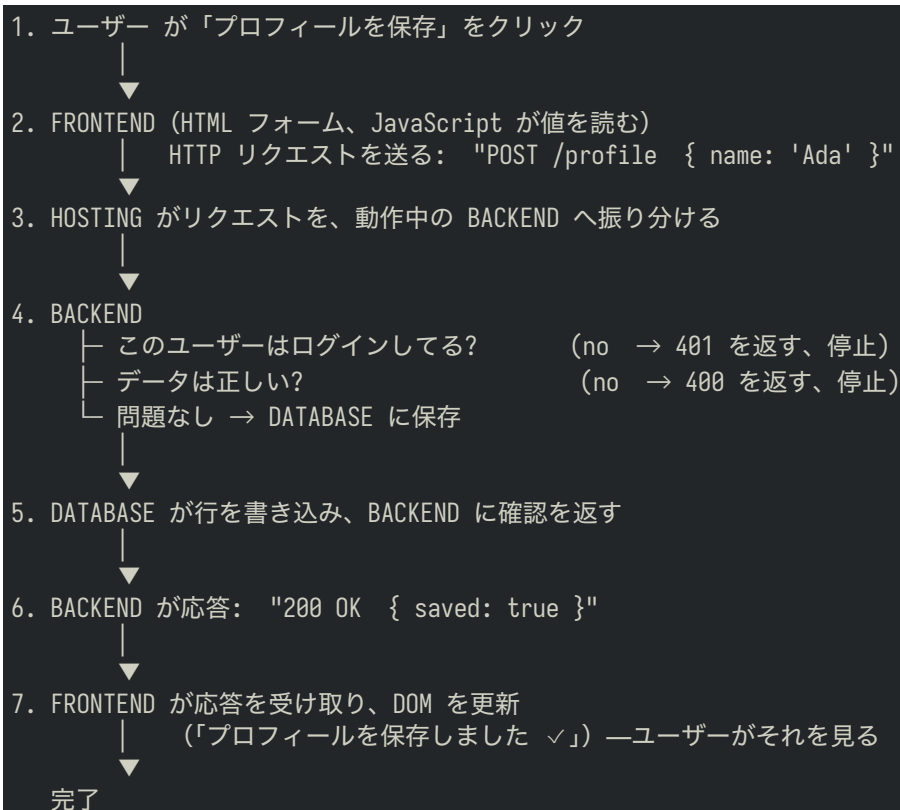
これを暗記する必要はありません——ただ見覚えがあればいいのです。そうすれば AI が「これを src/lib に追

加します」と言ったとき、それがだいたいどこにあって、なぜそこなのかが分かります。いくつかの土台となる慣習を挙げます。

- `src/` にはあなたが書くコードが入ります。トップレベルにごちゃごちゃしているのは、ほとんどが設定です。
- **ルートにある設定ファイル** (`package.json` やドットファイル) は、ツールを設定するものであって、アプリの振る舞いを設定するものではありません。
- `.gitignore` には、バージョン管理に保存されるべきでないファイルが並びます——あなたの `.env` はここに属しますし、まさにこれによって秘密の情報が共有リポジトリの外に保たれるのです。

リクエストはどう流れるか

ここまでを一つにつなげましょう。ユーザーが「プロフィールを保存」をクリックするのを、いま出会ったばかりのすべての部品を通して追いかけてみます。これは本書の中でいちばん役に立つ思考モデルです。もし図を一つだけ頭に刻むなら、これにしてください。



各層が何をしたかをたどってみましょう。**frontend** は入力を集めて送り、**ホスティング** はリクエストを正しい場所へ届け、**backend** はルール (ログイン、検証) を守らせて判断し、**データベース** はそれを恒久的なものにし、そして応答がずっと戻ってきて、**frontend** が画面を更新できました。あらゆるアプリはこのループのバリエーションであり、毎秒何千回も実行されています。

チェックがすべて backend で行われ、frontend では決して行われないことに注目してください——最初の節のルールが実際に働いているのです。frontend はユーザーのためのもの、backend は真実のためのものです。各ステップがどこで起きるかを知っていれば、機能を曖昧にではなく (「フォームを動くようにして」)、正確に説明できます (「保存する前に backend でメールを検証して」) ——そして正確さこそがこのゲームのすべてです。

まとめ&練習

重要なポイント

- あらゆるアプリは 4 つの部品です。**frontend** (ユーザーが見るもの)、**backend** (ルールとロジック)、**データベース** (データが残る場所)、そして**ホスティング** (すべてが動く場所)。frontend がデータベースに直接触れることは決してなく、必ず backend を経由します。
- ブラウザの中では、**HTML** が構造、**CSS** がスタイル、**JavaScript** が振る舞いです。コードは **DOM** を変えることでページを変えます。
- すべてを自分で書くわけではありません——**パッケージ** (package.json と lockfile を通して **npm** が管理する) が、解決済みの問題を**依存関係**として供給してくれます。
- **環境変数** は秘密の情報をコードの外に保ち、同じコードが dev と prod で正しく振る舞えるようにします。
- リクエストは frontend → ホスティング → backend → データベース と流れて戻ってきます。問題がどの層にあるかに名前をつけることが、解決の半分です。

やってみよう

毎日使っているアプリを一つ選んで、一つの動作を層を通して声に出して語ってみましょう。「送信をタップする (frontend) → それが向こうのサーバーに行く (backend) → そこで送っていいか確認してメッセージを保存する (データベース) → そしてチャットが更新される (frontend)」。それから、それぞれの問題がどの層にあるか当ててみましょう。ボタンの位置がずれて見える。アプリが「ログインしていません」と言う。昨日のメッセージが消えている。この名づけが上手くなることが、本書の残りが土台にしているスキルです。

この章のプロンプト

アプリがどう組み合わさっているかを学んでいます。実現したい機能はこれです：
 < ユーザーができるようになるべきことを一文で説明 >

コードを書く前に、層ごとに分解してください：

- Frontend: ユーザーが見るものと、送られるもの
- Backend: 走らせるべきルール/チェックと、それがどこか
- Database: 保存または読み出しが必要なデータ
- 関わる env 変数や設定 (特に秘密の情報)

それから、クリックから結果までのリクエストの流れを、番号付きの手順で説明してください。
 平易な言葉をお願いします —コードの前に、まず形を理解したいのです。

バイブコーディングのマインドセット



バイブコーディングは多義的な言葉だ。ある人にとっては、一行のプロンプトを打ち込み、モデルが吐き出したものを何であれ受け入れ、一行も読まずに出荷することを意味する。そのバージョンはおもちゃでしかない。実際のユーザーが触れた瞬間に崩れ落ちるデモを生み出すだけだ。

本書が扱うのはもう一方のバージョンだ。シニアエンジニアがチームを指揮するように、AI を指揮してプロダクション品質のソフトウェアを構築する。あなたは何を作るのか、そしてなぜそれが正しいのかを自分でコントロールし続ける。タイピングは委譲する。これがその転換であり、聞こえるよりもずっと大きな変化だ。

バイブコーディングとは実際に何か

バイブコーディングとは、**自然言語で意図を表現し、AI に実装を生成させること**であり、その間あなたは舵を取り、レビューし、緊密なループの中で洗練させていく。

それは次のものではない。

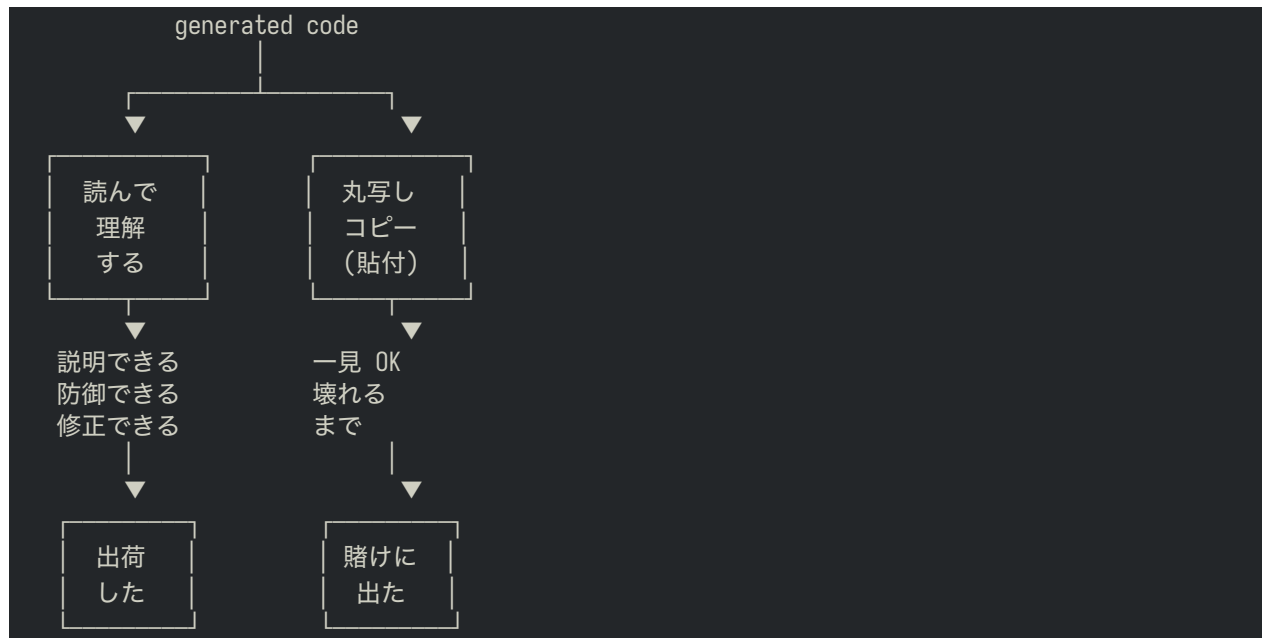
- 理解できない生成コードを盲目的に受け入れること。
- エンジニアリングの判断を「ノリ」で置き換えること。
- 一文をそのまま出荷可能な製品に変える魔法のボタン。

それは次のものである。

- モデルを、速くて知識豊富だが少々自信過剰なジュニアエンジニアとして扱うこと。
- 構文ではなく、アーキテクチャ、正しさ、センスに注意を注ぐこと。
- かつて何時間もかかったことを数分で反復すること。

最もうまく機能するメンタルモデルはこうだ。あなたはテックリードであり、AI はあなたのチームだ。優れたテックリードはすべての行を書くわけではない。彼らは方向性を決め、制約を設定し、承認のために提出する変更のまとめ（プルリクエスト）をレビューし、出荷される前に悪いアイデアを捕まえる。あなたはそれをやっている——ただ会話のスピードで。

この二本の道は同じ生成コードから出発し、まったく違う場所にたどり着く：



この定義の中で重い役割を担う言葉が一つある。**理解**だ。このアプローチ全体はたった一つのルールに支えられている——自分が出荷するものを理解せよ。自分でタイプする必要はないが、それを説明でき、レビューで擁護でき、午前2時にアラートが鳴ったときに修正できなければならない。それができないなら、あなたはバグコーディングをしたのではない。賭けに出て運がよかっただけだ。その二つの間の隔たりこそ、本書全体の主題だ。

マインドセットの転換

すべての文字を自分でタイプしてきた人にとって、最も難しいのはツールを学ぶことではない。キーボードを手放すことだ。

かつてのボトルネックは生産だった——正しいコードをどれだけ速く書けるか。新しいボトルネックは仕様化とレビューだ——求めるものをどれだけ明確に記述でき、返ってきたものをどれだけうまく判断できるか。

これは最適化の対象を変える。

- ・ **以前:** API(アプリケーション・プログラミング・インターフェース)ツールが提供するコマンドのメニュー)を暗記し、構文と格闘し、速くタイプする。
- ・ **今:** 意図を正確に記述し、問題を分解し、出力を批判的に評価する。

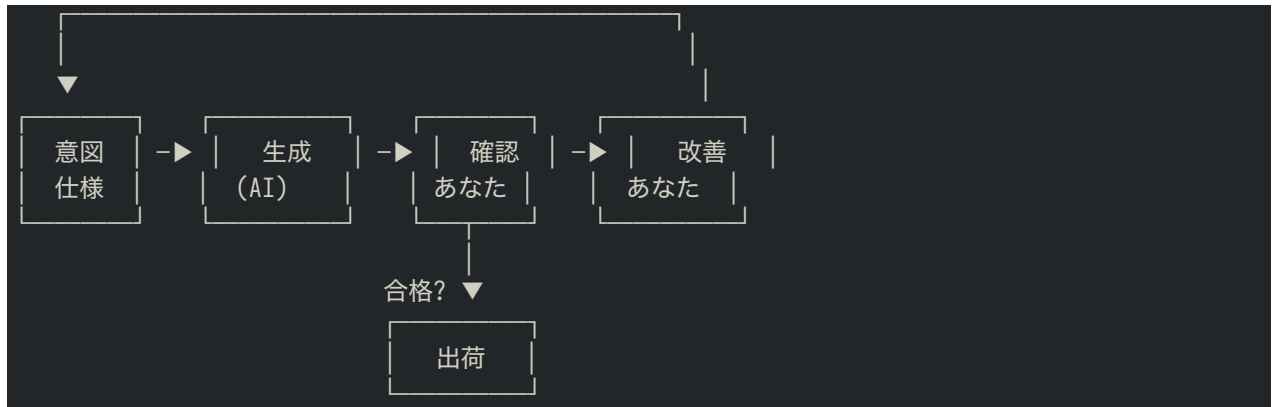
これで成功するエンジニアは、最速のタイピストではない。「良い」とはどういうものかについて強い意見を持ち、その基準を満たさない出力を拒否する規律を備えた人たちだ。

人々の不意を突く、もう一つの静かな転換がある。知らないこととの関係だ。かつての本能は、APIに触れる前に立ち止まってそれを学ぶことだった。今では、不慣れなスタックでも最初の一手を生成し、それを読み返しながらかけることが多い。それは本物の超能力だ——そして罫でもある。自分では書けなかったコードを読むのは構わない。だが自分で評価できないコードを出荷するのはダメだ。境界線は、その出力が正しいと自信を持って言えるかどうかだ。自分が書けるより速く書かせる点ではモデルに頼れ。だが自分が検証する気のある以上に知っている点では決して頼るな。

ループ: 意図 □ 生成 □ レビュー □ 洗練

本書のすべては一つのサイクルに帰結する。これを身体に染み込ませよう。

ループは出力があなたの基準を満たすまで回る——そしてレビューは毎回ぐり抜けねばならない関門だ:



1. **意図**。求めるものと、重要な制約を述べる——入力、出力、エッジケース、スタック、スタイル。曖昧な意図は曖昧なコードを生む。
2. **生成**。モデルに実装を書かせる。構文を逐一手取り足取りせず、振る舞いを記述する。
3. **レビュー**。返ってきたものを読む。それは実際に目的を果たしているか。空のリスト、null、失敗したネットワーク呼び出しを処理しているか。それは動作する最もシンプルなバージョンか。
4. **洗練**。何が間違っているかを指摘し、具体的な修正を与え、再生成する。基準を満たすまで繰り返す。

ほとんどの初心者はステップ 3 を飛ばす。生成し、動いた、次へ進む。そして記述しなかったケースでプロダクションが壊れる。レビューのステップこそ、エンジニアリングが起きる場所だ。

このループでの良い最初のプロンプトはこんな感じだ。

```
TypeScript 関数 `parseDuration(input: string): number` を書いて。  
"1h30m", "45s", "2d" のような文字列を合計秒数に変換する。
```

要件:

- 単位をサポート: d (日), h (時間), m (分), s (秒)。
- 複数の単位を組み合わせ可能: "1h30m" = 5400。
- 不正な入力 (空, 不明な単位, 負数) は明確なメッセージ付きの Error を throw して拒否する。
- 外部ライブラリ禁止。エッジケースを網羅する単体テスト 5 個を含める。

関数とテストのみを表示して。

その構造に注目してほしい。正確なシグネチャ、明示的なルール、列挙されたエッジケース、述べられた制約、そしてテストの要求。あなたはモデルが正しく当ててくれることを願っているのではない——悪い推測を生む曖昧さを取り除いているのだ。

洗練のステップも、最初のプロンプトと同じくらい一つのスキルだ。曖昧なフィードバックは曖昧な修正を生む。この二つの修正指示を比べてみよう。

悪い例: "これは間違ってる、直して"

良い例: "parseDuration('1h30m') が 5400 ではなく 90 を返す —
数値を合計しているが単位の倍率を無視している。合計する前に
各値をその単位の秒数 (d=86400, h=3600, m=60, s=1) で掛けて。
さらに組み合わせケース '2d3h' のテストも追加して。"

良いほうは症状、原因、そして修正方法を名指しする。あなたは同僚のプルリクエストをレビューするようにし

ビューしているのだ——そして正確に指し示すほど、費やすラウンドは少なくなる。各ループは収束すべきだ。堂々巡りをしていると感じたら、それは意図が仕様不足だったというサインだ。再生成をやめて、仕様を書き直せ。

真価を発揮する場面

バンプコーディングは次のような場面で本当に変革的だ。

- ・ **グリーンフィールドのスキヤフォールディング**。新しいアプリの立ち上げ、ルートの配線、設定のセットアップ——退屈だが踏み固められた道の作業。
- ・ **ボイラープレートとグルーコード**。CRUD エンドポイント、フォームバリデーション、データ変換、二つの API 間のアダプタ。
- ・ **不慣れな領域**。よく知らない言語やフレームワーク。モデルは慣用句を知っており、あなたは判断を提供する。
- ・ **リファクタリングと移行**。「これらの promise を async/await に変換して」「この 400 行のファイルを責務ごとに分割して」。
- ・ **使い捨ての探索**。どうせ書き直すプロトタイプ。磨き上げよりスピードが重要。

これらのゾーンでは、明確なプロンプトと丁寧なレビューが、毎回すべてを手打ちするのに勝る。

共通する糸は、これらのタスクが**踏み固められていて、かつ検証可能**だということだ。モデルは一万もの CRUD エンドポイントを見てきたので、慣用的なものを生み出す。そしてあなたは結果が正しいか一目で見分けられる。この二つがともに真であるとき——ありふれたパターン、検証が安価——バンプコーディングはほぼ純粋な利益に近い。

失敗する場面

限界について正直であろう。なぜなら、ここが監督されていないバンプコーディングが人々を焼く場所だからだ。

- ・ **深く、新規性のあるロジック**。微妙なアルゴリズム、並行処理、95% 正しいことが壊れていることを意味するあらゆるもの。モデルは間違っているにも自信満々だ。
- ・ **広範囲に及ぶファイル横断の変更**。大きなコードベースを通して筋を見失い、見えないところを静かに壊す。
- ・ **セキュリティとお金**。認証、決済、権限、もっともらしく見えるバグが現実の結果をもたらすあらゆるもの。仕事がかかっているかのようにレビューせよ。実際そうかもしれないのだから。
- ・ **仕様が不十分な問題**。あなたが正しいものがどういうものか分からなければ、AI はあなたの心を読めない。ゴミの仕様を入れれば、ゴミのコードが出てくる。

失敗のパターンはほとんど常に同じだ。見た目は正しく、ハッピーパスでは動き、もし読んでいれば捕まえられたはずのやり方で間違っているコード。その対策はレビューのステップだ。それを飛ばすことはできない。

二つの問いで、どんなタスクもこの地図の上に置ける——パターンはありふれているか、そして結果は点検しやすいか:

	検証しやすい	検証しにくい
一般的なパターン	純粋な利点 (足場, 定型)	ゆっくり, テスト要求
まれ / 新規	読めば可能	危険地帯 (新ロジック, 認証, 金銭)

先のルールの上に注目してほしい。これらはまれで、微妙で、検証が高くつくケースだ。パターンがありふれないのでモデルは推測しているか、バグがハッピーパスでは見えないので素早い点検では捕まらない。だから

といってここで AI を避けよということではない——速度を落とし、ステップを小さくし、テストを要求し、すべての行を敵対的なものとして読め、ということだ。

よくある落とし穴

いくつかの罠は、最初の一か月でほぼ全員を捕まえる。

- ・ **動いた最初のもを受け入れること。**「動く」と「正しい」は別の主張だ。前者はハッピーパスがクラッシュしなかったという意味で、後者はエッジを点検したという意味だ。両者を混同するな。
- ・ **メガプロンプト。**機能全体を一発で求めると、モデルに漂流する余地を与えすぎ、あなたには一度にレビューするものを多く渡しすぎる。分解せよ。最小のレビュー可能なスライスを出荷し、その上に積み上げよ。
- ・ **コンテキストを腐らせること。**セッションが長くなるにつれ、モデルは以前の決定と矛盾しはじめ、すでに直したバグを再導入する。それが始まったら、混乱したスレッドと戦うのではなく、現在の状態を要約して新しく始め直せ。
- ・ **自信を信号として信頼すること。**モデルの口調は、正しいときもハルシネーションのときも同じだ。その確信はあなたに何も教えない。あなたの検証がすべてを教える。
- ・ **タイピングだけでなく、決定までも外注すること。**「X の選択肢は何か?」と尋ねるのは構わない。その答えに、あなたの判断をループに入れられないままアーキテクチャを選ばせるのは誤りだ。

センスが新たなコアスキル

誰もがコードを生成できるとき、希少なスキルは、どの生成コードが良いかを見分けることだ。

センスはあなたにこう告げる。

- ・ この抽象化は時期尚早だ。よりシンプルなバージョンを求めよ。
- ・ これはデモは処理するが、実際の入力処理しない。
- ・ これは技術的には正しいが、保守の悪夢だ。
- ・ これはほぼ正しい——そして変えるべき具体的な点はここだ。

センスは昔ながらの方法で築かれる。大量のコードを読み、物事を出荷し、何が壊れるかを見ることによって。AI はその経験を置き換えはしない——むしろその価値を高める。なぜなら今やあなたの判断がボトルネックであり、差別化要因だからだ。センスのある人は堅実なソフトウェアを速く出荷する。センスのない人は速く出荷し、そしてバグの海で溺れる。

居心地の悪い系がここにある。AI はすべての人の下限を引き上げる。つまり、構文を知っていることの優位を平らにし、判断を持っていることの優位を研ぎ澄ます。学ぶことを飛ばすために AI に頼るジュニアは頭打ちになる。出力が間違っているときを教えてくれるセンスを決して築かないからだ。AI をより多く出荷し、より多く読むために使う者は複利で伸びる。同じツール、正反対の結果——そしてその違いは、ひとえに返ってきたものに関与し続けるかどうかにかかっている。

現実的な期待値

後でがっかりしないよう、今これを設定しておこう。

- ・ **大量のコードを読むことになる。**レビューが仕事だ。コードを読みたくないのなら、これはそれを回避する近道ではない。
- ・ **自明でないものをワンショットで仕上げることはない。**数ラウンドのループを見込め。それは普通であり、失敗ではない。
- ・ **責任は依然としてあなたにある。**AI が書いたコードがプロダクションで壊れたとき、「モデルが書いた」は弁解にならない。出荷したのはあなただ。
- ・ **速くなるのであって、タダになるのではない。**ツールは目を見張る。だが魔法ではない。判断はあなたのものであり続ける。

そして、本書が何でないかについて率直に言おう。これは自分が理解していないソフトウェアを出荷する道ではないし、一攫千金の仕組みでもない。バイブコーディングを「コーディングを学ばずにアプリを作る」と売る者は誰であれ、おもちゃのバージョンを——実際のユーザーと接触した瞬間に崩れるあのバージョンを——売って

いるのだ。このスキルは、それをエンジニアリングの判断の代わりではなく、その上に乗せるレバレッジとして扱う人にこそ複利で積み上がる。

バイブコーディングの約束は本物だ。あなたは一人でできたよりも、より多くを、より速く、より多くの領域で構築できる。だがそれは関与し続けるエンジニア——指揮し、レビューし、洗練するエンジニア——に報い、降りてしまって現れたものを何でも受け入れる者を罰する。

傍観者ではなく、テックリードであれ。本書の残りは、その方法についてだ。

まとめと実践

重要なポイント

- バイブコーディングとは、何を なぜ 作るかの主導権を握ったまま AI にソフトウェア構築を指示することであり、説明できないコードを盲目的に出荷することではない。
- すべてを支える原則は 出荷するものを理解せよ だ。自分でタイプする必要はないが、弁護でき、修正できなければならない。
- ボトルネックは生産（速くタイプすること）から、仕様化とレビュー（明確に記述し、批判的に判断すること）へ移った。
- 意図 □ 生成 □ レビュー □ 洗練のループの中に身を置き、実際のエンジニアリングが起きるレビューの段階を決して飛ばすな。
- モデルの自信はシグナルではない。あなたの検証こそがシグナルだ。センスと判断力こそが希少で、複利で積み上がるスキルだ。

やってみよう

本当に理解している小さな関数の一つを選ぼう。「パスワードがポリシーを満たすか検証する」や「バイト数を人間が読める文字列に整形する」など。精密なプロンプトを書き（シグネチャ、ルール、エッジケース、テストの依頼）、生成したうえで、意図的に レビュー の段階を実行しよう。少なくとも一つのエッジケースを手でたどり、コードが処理しているか確認しよう。処理していなければ、「直して」ではなく精密な洗練プロンプト（症状 + 原因 + 修正）を書こう。目的は、完全に評価できるコードの上で、ループ全体を最初から最後まで一度体感することだ。

この章のプロンプト

私のペアプログラミングのパートナーとして振る舞って。小さく自己完結した関数を説明する。コードを書く前に、私が確認できるように要件を整理し直し、処理するエッジケースを列挙して。それから、それらのエッジケースを網羅する単体テストとともに関数を書いて。

関数: < 一行の説明 >
入力 / 出力: < 型と形状 >
制約: < スタック, 使う/避けるライブラリ, 性能の制限 >

コードの後に、私が検証すべき前提があれば指摘して。
要件を満たす最も単純なバージョンに留めて。

AI ワークスペースをセットアップする



バイブコーディングはパートナーシップだ。あなたは意図と判断を持ち込み、AI はスピードと疲れ知らずのタイピングを持ち込む。だがどんなパートナーシップとも同じように、共有された環境、明確な基本ルール、そして失敗しても安全な場所があってこそ最もうまく機能する。この章は、最初の機能の一つ書く前に、その環境を構築することについてだ。

良いワークスペースは三つのことをする。AI が賢い変更を行うために必要なコンテキストを与え、それらの変更の結果を素早く見られるようにし、すべての変更を簡単に取り消せるようにする。これらを正しく整えれば、AI はシニアの協力者のように感じられる。間違えれば、スロットマシンのように感じられる。その差はめったにモデルのせいではない——あなたがモデルを置いた部屋のせいだ。テストもコンテキストファイルもバージョン管理もない混沌としたリボに入れられた有能なエージェントはもがく。きれいで計装の行き届いたプロジェクトに入れられた平凡なエージェントは、あなたを驚かせるだろう。セットアップこそが倍率器だ。

これらはどれも、一度やって忘れるための雑用ではない。ワークスペースを、絶えず調律し続ける何かとして扱おう。AI が悪い方向であなたを驚かせるたびに、欠けたコンテキストの一片、遅いフィードバックループ、あるいは安全でないデフォルトがそれを許したのではないかと問え——そして、コードの一行ではなく部屋を直せ。

AI ネイティブなエディタまたはアシスタントを選ぶ

最初の決定は、AI がどこに住むかだ。今日、おおよそ三つのカテゴリがあり、永遠にコミットする必要はない。

- ・ **AI ネイティブなエディタ** (Cursor、Windsurf): AI が深く組み込まれた、見慣れたコードエディタ。AI が横で作業する間、コードを見て触れたいなら最適。

- **ターミナル/CLI エージェント** (Claude Code、Codex CLI、Gemini CLI、Aider): ボタンをクリックする代わりにコマンドを入力するテキストだけの操作盤、CLI(コマンドラインインターフェース)を使う。AI がコマンドラインエージェントとして動き、ファイルを読み、編集し、コマンドを実行する。多数のファイルにまたがって AI により自律的に作業させたいなら最適。
- **IDE(統合開発環境—多くのプログラマーがコードを書くオールインワンのアプリ) のプラグイン** (GitHub Copilot、Continue、JetBrains AI): すでに愛用しているエディタの中に後付けする支援。

実際のソフトウェアを出荷するなら、**プロジェクト全体を読み、複数のファイルを編集し、コマンド** (テスト、ビルド、リンター) を自力で実行できるものを選ぼう。自分の成果を実行して確認できる能力こそが、役立つアシスタントを単なるオートコンプリートのおもちゃから分かつものだ。次の一行を提案できるだけのツールは、検証をすべてあなたの手作業に残す。たった今変更したテストを自分で実行できるツールは、自らループを閉じる。

その中核的な能力を超えて、腰を据える前に天秤にかけろべきものがいくつかある。

- **コンテキストウィンドウとプロジェクト把握**—関連ファイルを実際に読み込めるのか、それともタスクの途中で会話の冒頭を忘れてしまうのか？
- **権限モデル**—コマンドを実行しファイルを編集する前に尋ねるのか、それとも先に動くのか？ どちらにも居場所がある。自分がどのモードにいるかを知っておこう。
- **コストと上限**—自律エージェントは使用量を速く燃やしうる。一時間も走らせる前に料金体系を理解しよう。

選択に苦悩しすぎないこと。本書のスキルはツール間で移行できる。経験豊富な vibe coder のほとんどは、結局二つをインストールしておくことになる——大量作業用の速いターミナルエージェントと、レビュー用の GUI(グラフィカルユーザーインターフェース—ウィンドウとボタンでクリックする馴染みのプログラム) エディタだ——そしてこの章を締めくくる TUI(テキストユーザーインターフェース—ターミナルで動くテキストだけの親戚)vs GUI の節は、まさにその組み合わせに立ち返る。

AI がナビゲートできるようにプロジェクトを構造化する

AI モデルはコードを読むことであなたのコードを推論する。散らかって、肥大化し、一貫性のないプロジェクトは、あなたにとってと同じく彼らにとっても扱いにくい。完璧なアーキテクチャは必要ないが、いくつかの習慣はすぐに報われる。

- **予測可能なレイアウトを保つ**。ソースは一箇所に、テストはソースの隣か鏡像の位置に、設定はルートに。
- **明確で説明的な名前を使う**。calculateMonthlyInvoiceTotal は calc2 に勝る。AI は名前を手がかりとして使う。
- **小さいファイルを好む**。200 行のファイルは、あなたにとっても AI にとっても、2,000 行のものより正しく変更しやすい。
- **関連するものをまとめて配置する**。ある機能のコード、スタイル、テストが一緒にあれば、AI は一度の読み込みでコンテキストを集められる。
- **明確なエントリポイントを保つ**。アプリが始まる明白な一箇所は、AI がプロジェクトを把握するときに手繰る糸口を与える。

典型的で AI に優しいレイアウトはこんな感じだろう。

```
my-app/
├── AGENTS.md           # AI 用のプロジェクト規則 & コンテキスト
├── README.md           # プロジェクト概要、実行方法
├── .env.example         # 必要なシークレットを記載 (実値なし)
├── .gitignore           # .env, ビルド成果物, node_modules を除外
├── package.json         # スクリプト: dev, test, lint, build
├── src/
│   ├── features/
│   │   └── invoices/    # 1 機能のコード + テスト、一緒に
│   └── lib/             # 共有ヘルパー
```

```
| └─ index.ts      # エントリポイント
| └─ tests/        # 横断 / 統合テスト
```

その見返りは複利で膨らむ。プロジェクトが AI にとってナビゲートしやすいほど、AI の変更は小さく外科的になる。よく整理されたリポは、AI が三十個を引っ掻き回して推測する代わりに、関連する三つのファイルだけに触れさせる——つまり、より小さい差分、より少ない事故、そしてあなたが実際に読めるレビューになる。AI が広範で焦点の定まらない編集をしているのを見つけたら、真犯人はプロンプトではなく構造であることが多い。

コンテキストファイルを書く (AGENTS.md)

最もレバレッジの高い唯一のことは、AI が自動的に読む**コンテキストファイル**を書くことだ。ツールによって探す名前は異なる——AGENTS.md、CLAUDE.md、.cursorrules、あるいは.github/copilot-instructions.md——だが考え方は同一だ。あなたのプロジェクトがどう動き、あなたがどう振る舞ってほしいかを AI に伝える、生きたドキュメントである。

それがなければ、AI は推測する。それがあれば、AI はあなたのコードを再フォーマットするのをやめ、使っていないライブラリを廃止するのをやめ、すでに下した決定を再説明するのをやめる。これを、優秀だがセッションの合間に完全な記憶喪失に陥る新しいチームメイトのためのオンボーディング資料だと考えよう——彼らが生産的になるために必要なものはすべて、彼らが毎回読む場所になければならない。

実践的な出発点となるファイル。

```
# Project: Invoice Manager

## What this is
A small web app for freelancers to create and track invoices.

## Tech stack
- TypeScript + React (Vite)
- Tailwind for styling
- SQLite via Prisma
- Vitest for tests

## Commands
- Install: `npm install`
- Run dev server: `npm run dev`
- Run tests: `npm test`
- Lint: `npm run lint`

## Conventions
- Use functional components and hooks; no class components.
- Keep components small; one component per file.
- Write a test for every new function in `src/lib`.
- Never edit files in `src/generated/` — they are auto-generated.

## Don'ts
- Don't add new dependencies without flagging it first.
- Don't commit secrets; use environment variables.
```

短く保ち、最新に保つこと。千行まで膨らんだコンテキストファイルは——あなたにとってもモデルにとっても——もはや注意深く読まれなくなるので、足すのと同じくらい攻撃的に刈り込もう。AI が同じ間違った前提を繰り返し置くなら、それは一行追加すべきというサインだ。ルールがもはや現実と合わなくなったら、それがエージェントを能動的に誤らせ始める前に消そう。

機能するコンテキストファイルと無視されるそれとを分ける、いくつかのパターンがある。

- **具体的かつ明確に書く。**「我々の規約を使え」は役に立たない。「検証には zod を使え。我々は yup を使わない」は、AI が実際に従えるルールだ。
- **やるなを、やれと同じくらい大きな声で言う。**モデルは手伝いたくてうずうずしている。つまり、止めない限り喜んで依存関係を足したり設定を書き換えたりする。
- **例を指し示す。**「src/features/invoices のスタイルに合わせる」は、解釈すべき曖昧な指示の代わりに、真似すべき具体的な手本を AI に与える。

コンテキストファイルを、さもなければ繰り返し説明しなければならないあらゆる教訓の記録として扱おう。それはこの章全体で最も安く、最も長持ちする投資だ。

バージョン管理を反射にする

バージョン管理はあなたのセーフティネットであり、AI と共にあると不可欠になる。AI は五秒で十個のファイルを変更できる——その成果を検査し、保持し、捨てるためのクリーンな方法が必要だ。

- **頻繁に、小さな塊でコミットする。**大きな変更を求める前に、動いているものをコミットせよ。クリーンな作業ツリーは、いつでもそこに戻れることを意味する。
- **コミット前に差分を読む。**これがあなたの本当のレビューステップだ。すべての行を理解する必要はないが、何が、なぜ変わったかは理解すべきだ。
- **リスクのある作業にはブランチを使う。**ブランチを立ち上げ、AI を暴れさせ、結果が気に入った場合のみマージする。気に入らなければ削除すればいい——害はない。
- **正直なコミットメッセージを書く。**未来のあなたも AI も、意図を理解するために履歴を読む。

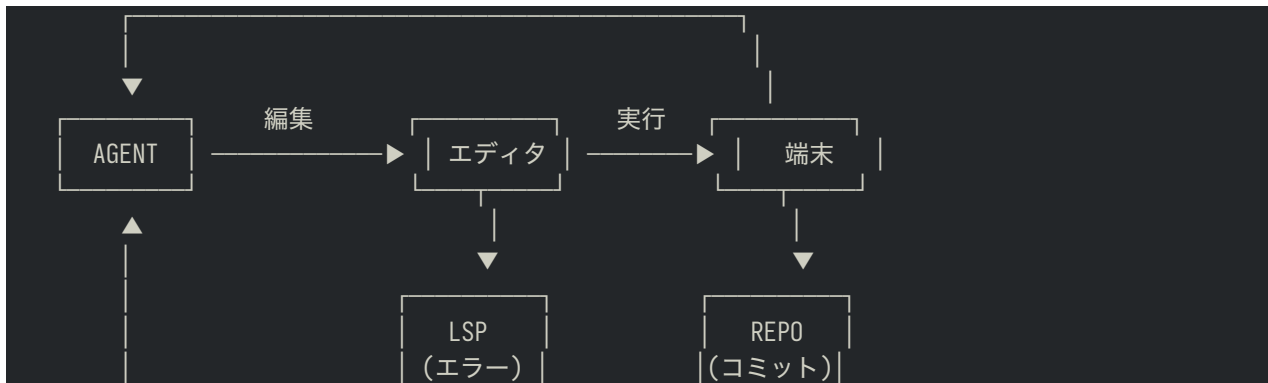
メンタルの転換はこうだ。コミットは官僚的手続きではなく、ゲームのチェックポイントだ。ボス戦の前にセーブせよ。そして、クリーンな履歴を保つ価値を生む二つの脱出口を覚えよ——コミットしていない実験を捨てる `git checkout .` (または `git restore .`) と、すでに作ったコミットを取り消す `git revert` だ。いつでも既知の良好な状態に戻れると知っていること、それこそが不安なく AI に長い手綱を渡せる理由だ。

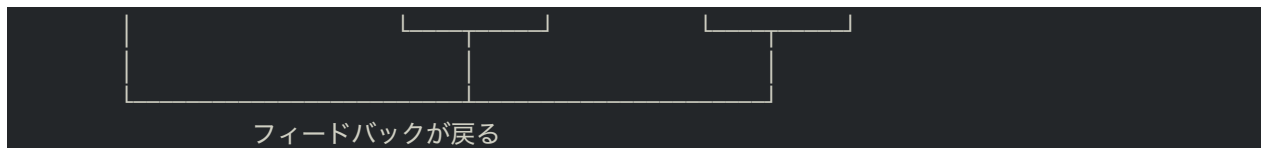
フィードバックループを緊密に保つ

変更が機能するかを速く見られるほど、あなたと AI が正しいものへ速く収束する。遅いループ——何かが壊れたか知るのに数分待つようなループ——は、あらゆる間違いを静かに何倍にもする。

- **「実行する」を単一のコマンドにする。** `npm run dev`、`make test`、それが何であれ、AI も実行できるようコンテキストファイルに記載せよ。
- **速いテストに頼る。**ほんの一握りのユニットテストでも、AI があなたに付き添ってもらわずに自分の成果を検証する手段を与える。
- **リンターとフォーマッターを使う。**これらは一群のエラーを即座に捕まえ、スタイルの議論をレビューから締め出す。
- **エラーを AI に見せ返す。**何かが壊れたら、実際のエラーメッセージを貼り付けよ。具体的なフィードバックは具体的な修正を生み、「動かない」は推測を生む。

AI ワークスペースの各部品は、一つのつながったループを形づくる——エージェントがリポを編集し、エディタと言語サーバーが問題を報告し、ターミナルがコードを実行し、すべての結果がエージェントへと戻ってくる。





この発想の最も深いバージョンは、AI 自身にループを閉じさせることだ。あなたがコマンドを実行し、失敗を読み、それを伝える代わりに、エージェントに、変更のたびにテストと型チェッカーを実行し、見たものに反応する権限を与えよ。サイクルはこうなる。編集し、実行し、失敗を読み、修正し、また実行する——すべてあなたを真ん中に挟まずに。あなたの役割は配達人から監督へと移る。目標とガードレールを定め、ループが回るのを見守る。AI が編集し、実行し、失敗を見て、自力で修正できるとき、あなたは監督の椅子に座ったまま、実際の仕事をこなすループを構築したことになる。

シークレットを慎重に管理する

AI ツールはあなたのファイルを読み、ときにリモートサービスとコンテキストを共有する。それは漏洩したシークレットを現実のリスクにする。初日から良い習慣を築こう。

- **本物のキーをコードに置かない。** 環境変数と `.env` ファイルを使う。
- **`.env` を必ず `gitignore` する。** 空のプレースホルダを入れた `.env.example` をコミットし、値を見せずに何が必要かを AI に知らせる。
- **キーのスコープを狭くし、** 露出した可能性のあるものはローテーションする。
- **コンテキストファイルで AI にシークレットを決してハードコードもコミットもしないよう伝える。**

AGENTS.md に Never read or print the contents of `.env` のような声高なルールを置くのは、安上がりな保険だ。キーらしきものを含むコミットを拒否する pre-commit フックやシークレットスキャンツールも同じだ——漏洩のコストは非対称なので、ここでは多層防御が重要になる。公開リポにプッシュされたキーは、たとえ数秒後に消しても、着地した瞬間にすでに漏洩している。ボットがリアルタイムでコミットを掻き集めるからだ。疑わしいときは、露出したキーは焼け焦げたものとみなしてローテーションせよ。キーのローテーションは一分で済むが、漏洩したキーの後始末は何日もかかりうる。

健全なローカル環境が AI の変更を安全にする

間違いが安く、回復可能な環境ですべてを結びつけよう。

- `main` で直接ではなく、**ブランチ**で作業する。
- 差分が意味を持つよう、大きな変更の**前**に作業ツリーを**クリーン**に保つ。
- テストと開発サーバーを一つのコマンドで**実行可能**にしておく。
- シークレットを**リポジトリの外**に保ち、コンテキストファイルを**最新**に保つ。

この土台があれば、どんな AI の変更でも最悪のケースは `git checkout .` と仕切り直しで済む。その安全性こそが、AI のすることは何も本当に取り返しがつかなくはないと知った上で、速く動き、大胆に指揮することを可能にする。次の章では、このワークスペースを使って実際に構築を始めよう。

ライブ診断 (LSP)

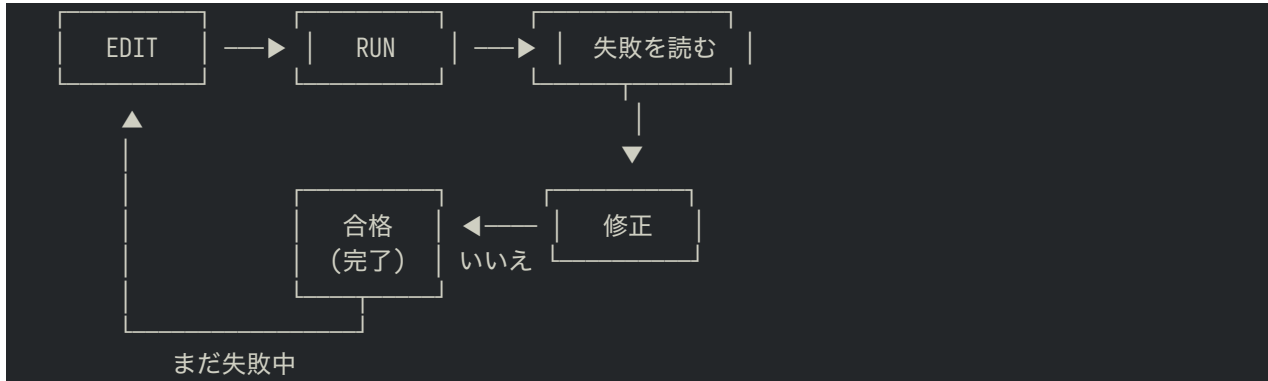
もっともきついフィードバックループは、コードを実行するのを待たない。タイプした瞬間に間違いを捕まえる。それを提供するのが **Language Server Protocol (LSP)** だ。エディタの背後にあるライブな「賢さ」であり、リアルタイムの**エラー、型情報、定義への移動、そしてオートコンプリート**を含む。言語サーバーは変化していくコードを読み、何かが実行される前に問題を報告する。

vibe coding にとってのレバレッジは、**それらの診断を AI に戻してやる**ことだ。それがないとループは遅い。AI がコードを書き、あなたが実行し、タイプミスや型エラーでクラッシュし、あなたがそのメッセージを手で貼り戻す。診断を繋ぎ込めば、エージェントは行を書いたその瞬間に、あなたと同じ赤い波線を見る。そして型エラーや lint エラーをその場で、コードが実行されるより前に直す。

エージェントにこの視力を与える方法はいくつかある。

- 自分のスタックの言語サーバー (tsserver、rust-analyzer、pyright など) を動かす。AI ネイティブなエディタの多くはこれを代わりにやってくれる。
- エージェントに型チェックや lint コマンドを実行させて出力を読ませる、あるいは MCP サーバー経由で診断を公開する。
- クリーンな診断レポートを後付けではなく「完了」の一部として扱う。

こう配線すれば、エージェントは自力で自己修正サイクルを回す——編集し、実行し、失敗を読み、修正し、診断がクリーンに返ってくるまでまた実行する。



これはまた、型のある言語が vibe coding にとって静かな超能力になりうる理由でもある。すべての型注釈は、AI の間違いが実行されるより前に捕まえる、小さく、機械が検査する仕様なのだ。実行・クラッシュ・貼り付けのサイクルが、静かで自己修正的なものになる。AI は無駄に粘る回数が減り、より速く収束する。

エージェントを拡張する

そのままの AI アシスタントは汎用的だ。同じベースモデルでも、周りにツールを巻きつけてやると、はるかに優れた協働者になる。ただのチャットボットが、あなたのプロジェクトに合わせて調整された**ハーネス**に変わるのだ。知っておくべき主な拡張ポイントは次のとおり。

- **Skills** —エージェントが必要に応じて読み込む、再利用可能なパッケージ化された能力（指示書+スクリプト）。あなたのドメインのノウハウを、セッションごとに説明し直さずに使えるようにする。
- **Plugins** —skills、コマンド、hooks、コネクタをまとめて一つのインストール可能な単位として届けるバンドル。
- **MCP (Model Context Protocol)** —エージェントを外部のツールやデータ（データベース、API(アプリケーション・プログラミング・インターフェース—他のソフトウェアが提供する標準的なコマンドのメニュー)、あなたのファイルシステム) に、一つの共通インターフェースを通じて接続するためのオープンな標準。
- **Subagents** —メインエージェントが集中的な作業（検索、レビュー、長いサイドクエスト）のために spawn する、それぞれ独自のコンテキストウィンドウを持つ専門の補助エージェント。
- **Hooks** —ライフサイクルのイベント（コマンドが走る前、ファイルが編集された後）で自動的に発火するスクリプト。これによりルールを強制したり、決定論的にコンテキストを注入したりできる。
- **Slash commands** —/name で呼び出す、短い名前付きワークフロー。繰り返される複数ステップの要求を一打鍵に畳み込む。

初日からこれら全部が要るわけではない。素のまま始め、同じセットアップを繰り返していると気づいた瞬間に、skill か slash command を足せばいい。役立つ目安はこうだ。同じ複数ステップの指示を三度目にタイプするときは、それを slash command にせよ。エージェントが届かないツールに届けばいいのにと初めて思ったときは、そのための MCP サーバーを探せ。hooks は特に、ガードレールの章で、AI が覚えていてくれることをただ祈るのではなくルールを強制する手段として再登場する。

macOS vs Windows

たいていの AI ツール群—CLI エージェント、スクリプト、チュートリアル—は、こっそり Unix シェルを前提にしている。それが開発環境の現実的な選択を形づくる。

- **macOS** は **Unix ネイティブ**だ。本物の POSIX シェルを備え、開発世界の他の人々が使うのと同じ git、ssh、curl を同梱している。たいていのエージェントとスクリプトは「ただ動く」。
- **Windows** も十分に有能だが、もっともなめらかな道は **WSL2** (Windows Subsystem for Linux) だ。Windows の中の本物の Linux 環境である。ネイティブの PowerShell の癖と戦うのではなく、WSL2 の中で AI コーディングをやろう。
- **Linux** はこの一家の三番目の一員で、この目的に関してはほぼ正確に macOS のように振る舞う——それ自体が Unix なので、同じエージェントとスクリプトが手を加えずに走る。すでにその上にいるなら、あなたは準備万端だ。

これらすべての間で作業をポータブルに保つための、いくつかの落とし穴。

- **パス**—/home/you/project 対 C:\Users\you\project。バックスラッシュとドライブレターはスクリプトを壊す。WSL2 はこのほとんどを回避する。
- **改行コード**—Windows は CRLF、Unix は LF を使う。不一致はノイズの多い差分や壊れたシェルスクリプトを生む。core.autocrlf と .gitattributes を設定して LF に正規化しよう。
- **大文字小文字の区別**—macOS と Windows はしばしば File.ts と file.ts を同じものとして扱う。Linux と CI はそうしない。だからローカルでビルドできるコードがクラウドで失敗しうる。import とファイル名の大文字小文字は一貫させよう。
- **権限**—Unix のファイルモード (chmod +x) は Windows にきれいに対応しない。

どちらを使うにせよ、AI にはコマンドをポータブルに保つよう頼もう—フォワードスラッシュ、LF の改行、一貫した大文字小文字。そうすればそのスクリプトは、あなたのノート PC でも CI でも同じように走る。AI があなたの持っていないツールを前提にした一行コマンドを渡してきたら、いちばん速い直し方は、コンテキストファイルにあなたの OS とシェルを教えて、AI に推測をやめさせることであることが多い。

TUI vs GUI

AI コーディングツールには二つの形があり、生産的な vibe coder は両方に手を伸ばす。**ターミナル / TUI エージェント** (Claude Code、Codex CLI、Aider) はターミナルで動く。速く、スクリプト可能で、合成可能。自動化や、AI に多数のファイルにまたがって自律的に作業させるのに向く。**GUI / エディタツール** (Cursor、Windsurf、IDE 拡張) はビジュアルなエディタに住む。発見しやすく、インラインの差分、クリックで承認、ゆるやかな入り口を備える。変更のレビューが楽だ。

	TUI / terminal	GUI / editor
速度	非常に速く、オーバーヘッドが小さい	遅め、UI が多い
スクリプト化	可能—パイプ、連結、自動化	限定的
ビジュアル差分	プレーンテキスト	リッチ、インライン
発見しやすさ	低い (コマンドを知る必要あり)	高い (クリック可能)
向いている用途	自動化、自律性、パワーユーザー	レビュー、オンボーディング、ビジュアル作業

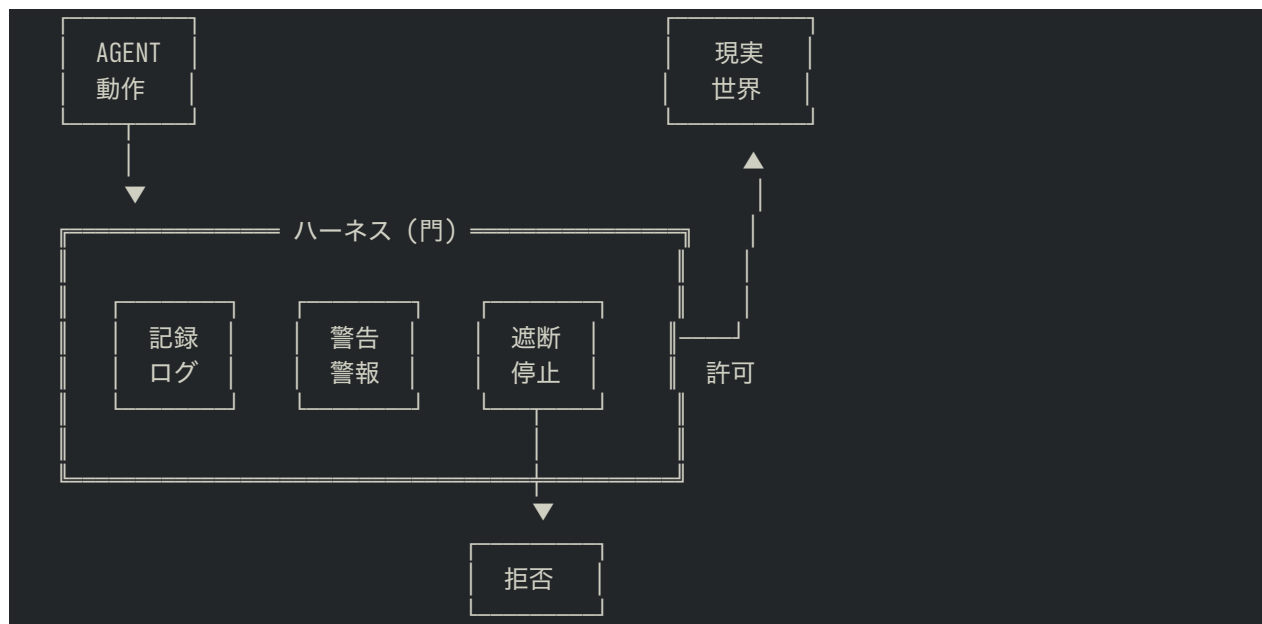
ずっと一つを選ばなければならないわけではない。組み合わせることこそが要点だ。作業の大半を速い TUI エージェントで進め、それからビジュアルなレビューや細かい編集のために GUI に降りる。よくあるワークフローは、ターミナルエージェントに多数のファイルにまたがる大まかな変更をさせ、それから GUI の左右並びの差分を開いて、それがやったことを実際に読んで承認することだ——やることには速さを、確かめることには明瞭さを。心地よい主力を選びつつ、本書のスキルはそれらの間で移植可能だと覚えておこう。

ハーネスエンジニアリング

習慣とは、あなたが忘れずにやらなければならないことです。作業環境を整える次の段階は、それらのチェックを AI が**スキップできないもの**にすることです。それがハーネスです。

ハーネスは、エージェントと現実世界の間に立つセキュリティ検査場だと考えてください。危険なステップごとに、三つのうちのいずれかを行います。何が起きたかを記録するか、警告するか、その操作をきっぱり遮断するかです。コマンドが実行される前、ファイルが書き込まれる前、プロンプトが送信されるとき、コミットするとき、プッシュするとき——そこにゲートがあります。深夜 2 時にテストを走らせ忘れるかもしれないあなたと違って、エージェントはそれを回り込めません。

エージェントが取るすべての危険な操作は、まずハーネスを通る。各ゲートは、それが現実世界に届く前に、記録するか、警告するか、遮断できる。



なぜ必須なのか。バイブコーディングの根本的な賭けは「AI には全力で走らせ、あなたは舵を取る」というものです。それが成り立つのは、あなたがまばたきしている間に AI が崖から飛び出せないときだけです。ハーネスは安全を警戒心（すべての差分を読み、すべてのルールを覚えておくこと）からコード（自動的に強制されるルール）へと移します。本物のユーザーの前に出すものなら、それは任意の仕上げではありません——自律性を安全にする、まさにその核心です。

良いハーネスは、知っておく価値のあるいくつかの原則に従います。

- **成功は静かに、失敗は大きく**—問題がないときは何も言わないので、警告には意味が残ります。
- **決して自動修正しない**—提案するか、遮断するか、警告する。エージェントをこっそり「直して」問題を隠したりはしません。
- **新しいルールを足す前に、眠っているルールを引退させる**—古いチェックの山ではなく、無駄がなく信頼されている状態を保ちます。
- **設定駆動**—エンジンは一つ、プロジェクトごとの設定で、どのリポジトリにも差し込めます。
- **追記専用 (append-only) のログ**—エージェントが実際に何をしたかの、改ざんが見抜ける記録。

これをゼロから作る必要はありません。既製の、差し込み式のハーネスがあります——[dancinlab/harness](#) は、コーディングエージェントのツールのライフサイクルに配線できる、プロジェクト非依存のエンジンの一つです。正直に言えば、ハーネスは AI を正しくはくれません。AI の間違いを安く捕まえられるようにするだけです——速く動いているとき、それが戦いのほとんどなのです。

コスト安全: CLI vs API

たった一通の請求がプロジェクトを台無しにすることがあるが、それがホスティングから来ることはめったにない。来るのは従量課金の AI API だ。日々コードを書くために整えるツール—Claude Code、Cursor、その類い—はたいてい**定額制のサブスクリプション**であり、プロンプトをもう一回打つコストは実質ゼロだ。だがアプリが生の AI API や SDK(ソフトウェア開発キット—提供元が渡してくれる事前にパッケージ化されたコードライブラリ)を呼んだ瞬間、経済性は反転する。**トークン単位、リクエスト単位**で課金され、上限はない。バグのあるリトライ、上限のないバッチジョブ、ループに陥るエージェント—これらが \$0 の待機請求を一晩で四桁の不意打ちへと静かに変えてしまう。それが「bill shock」であり、個人開発者が火傷を負う最もありふれた道だ。

有料の API は出荷する前に必ず檻に入れておこう:

- プロバイダーのダッシュボードで**ハードな支出上限を設定する**—メールアラートだけでなく、本物の天井を。
- **開発には定額制の CLI を優先し**、従量課金の API は本当に必要な本番機能のためにとっておく。
- **自分の呼び出しにレート制限をかける**—同時実行数と毎分リクエスト数に上限を設け—最初の数日は使用量ダッシュボードを見張る。
- バルク処理には**安価で小さいモデルを使い、まず小さな入力でテストする** (テーブル全体ではなく、1 件のレコードで)。
- **有料の API を呼ぶスロットルなしのループを絶対に出荷するな**。どんなループにも上限を設けよう。

最初からルールにしておこう: すべての有料の API にハードな上限とスロットルがある。

まとめと実践

重要なポイント

- よい作業環境は AI にコンテキスト、速いフィードバックループ、安価なやり直しを与える—セットアップこそが乗数であり、しばしばモデルそのものより効く。
- コンテキストファイル (AGENTS.md / CLAUDE.md) は書ける中で最もレバレッジが高い。具体的なルール、大きな声の禁止事項、例へのポイントを入れよう。
- バージョン管理を反射にしよう—小さくコミットし、差分を読み、危険な作業はランチで—そうすれば AI のすることに本当に取り返しのつかないものはない。
- ループをタイトに保ち、エージェント自身に閉じさせよう: 編集、実行、失敗を読む、修正。診断 (LSP、リント、型チェック) を AI に返そう。
- すべての有料 API をハードな上限とスロットルで囲い、シークレットをリポジトリの外に置き、ハーネスを使って安全チェックを AI が飛ばせないものにしよう。

やってみよう

作業中のプロジェクトを開く (または空のものをスキヤフォールドする) か、AGENTS.md の初稿を手で書こう: プロジェクトが何か、技術スタック、dev/test/lint を実行する正確なコマンド、実際のコンベンション三つ、そして少なくとも二つの大きな声の禁止事項 (例: 「.env を決して読んだり出力したりしない」「先に知らせずに依存を追加しない」)。それから AI ツールに些細な変更を頼み、ファイルを尊重するか観察しよう。望まないことをするたびに一行を追加するか研ぎ澄まそう。コードだけでなく、部屋をチューニングしているのだ。

この章のプロンプト

```
Read my project and help me write a concise AGENTS.md context file for it.
```

```
First, inspect the repo and tell me:
```

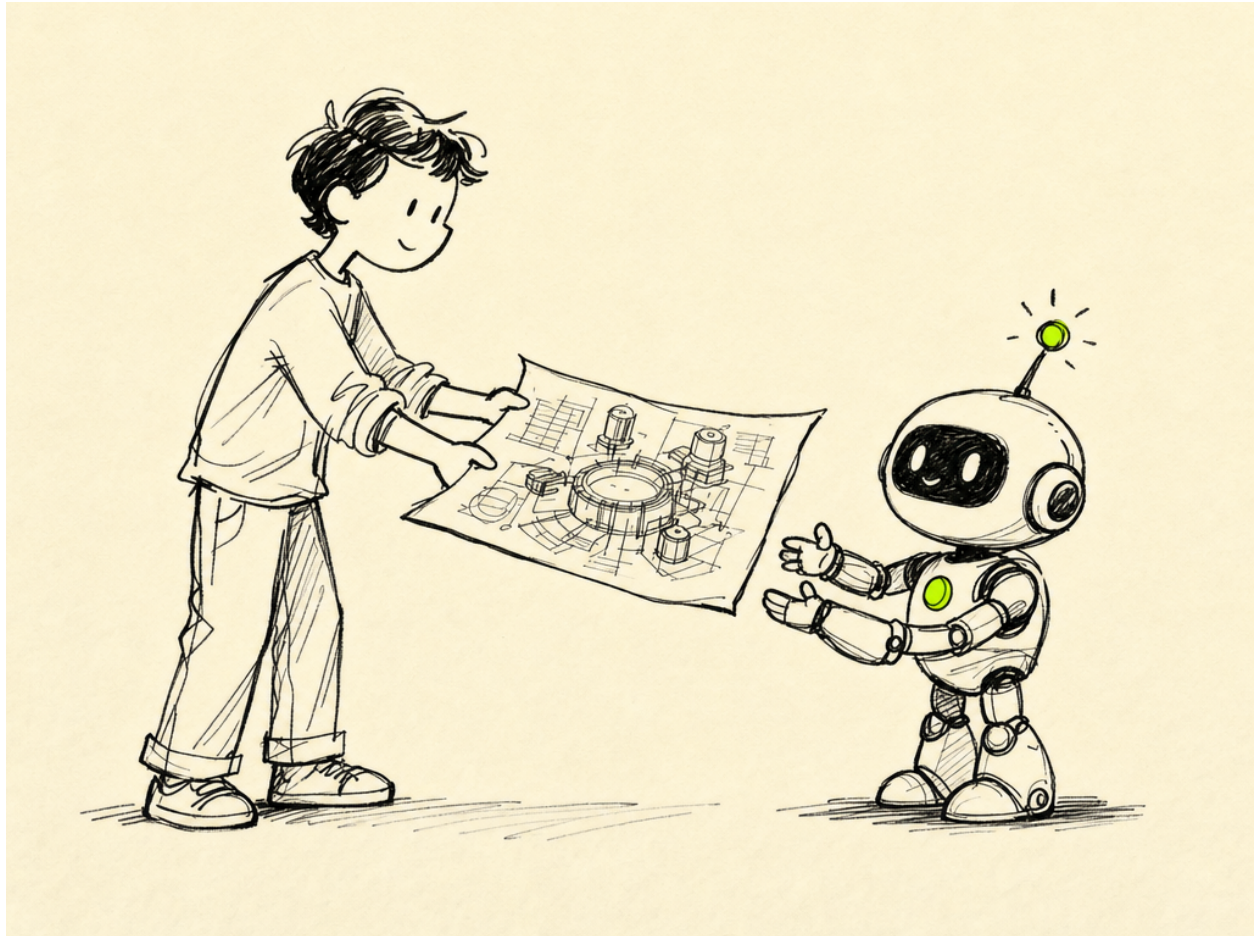
- the tech stack and versions you detect
- the commands for install / dev / test / lint / build
- the layout and where the entry point is

```
Then draft an AGENTS.md with these sections: What this is, Tech stack, Commands, Conventions, and Don'ts. Be specific and concrete (name the exact libraries and rules, not "follow our conventions"). State the don'ts
```



```
as loudly as the do's, and point to one existing file as a style example.  
Keep it under ~40 lines — I'll prune from there.
```

エンジニアのようにプロンプトする



イライラする AI セッションと生産的なセッションの違いは、めったにモデルではない。プロンプトだ。ほとんどの初心者は AI を検索エンジンのように扱う——曖昧な願いを打ち込み、うまくいくことを願う。シニアエンジニアはそれを、鋭いが字義通りのジュニアのチームメイトのように扱う。速くて有能だが、あなたの間違いも含め、あなたが記述した通りのものを正確に作る相手として。

この章はそのギャップを埋めることについてだ。トリックを暗記したり、「魔法の」プロンプトテンプレートをフォルダに集めたりする必要はない。良いエンジニアがタスクを伝えるように伝える必要がある。コンテキスト、制約、そして完了の明確な定義をもって。以下のすべては、その一つのスキルの変奏だ。

タスクを与える前にコンテキストを与える

AI はあなたが見るようにはプロジェクトを見られない。フレームワークのバージョンも、ファイル構造も、すでに確立した規約も知らない。これを飛ばすと、AI は推測する——しかも下手に推測する。さらに悪いことに、自信満々に推測する。だから出力は、あなたの実際のコードベースとぶつかるまではもっともらしく見える。

同じ要求に対するこの二つのプロンプトを比べてみよう。

メールを検証する関数を追加して。

これは Express と Zod でバリデーションする TypeScript バックエンドです。
すでに src/schemas/ で Zod スキーマを使って入力を検証しています。
そのスタイルでメールバリデーションのスキーマを追加してください。メールは
lowercase、max 254 chars で、src/config.ts にある既存の
BLOCKED_DOMAINS リストの使い捨てドメインは拒否すること。

最初のプロンプトは、あなたが望まない正規表現を使うかもしれない、コードベースに合わないスタイルの汎用的なコードを生む。二つ目はそのまま貼り付けられるコードを生む。ルールはこうだ。**スタックを、規約を、そして AI が再利用すべき既存の部品を述べよ。**

良いプロンプトはこれらの部品を下から上へ順に積み上げる——土台にコンテキスト、頂上近くに実際の要求を:

例	入力 1 つ → 出力
制約	してはいけない事
エッジケース	空・無効・エラー
入力 / 出力	型と形
ゴール	明確なタスク 1 つ
コンテキスト	スタック・ファイル・パターン

まず土台から ▲

エンターを押す前に役立つ心の中のチェックリスト。言語とフレームワーク (重要ならバージョンまで)、関連するファイルやモジュール、プロジェクトにすでにあるパターン、そして AI が好むべき、または避けるべきライブラリ。毎回四つすべてが要るわけではない——だが答えがそのうちの一つによって変わるなら、それを名指せ。迷ったら、新しいコードが合わせるべき実際のシグネチャ、型、または設定を貼り付けよ。数行の既存コードは、一段落の説明よりはるかに、あなたの規約を AI に教える。

入力、出力、エッジケースを指定する

タスクはその境界が定義されるまで定義されていない。シニアエンジニアは「何が入り、何が出て、物事がうまくいかないとき何が起きるか」という観点で考える。それをプロンプトに焼き込もう。

日付文字列をパースする関数を書いて。

parseDate(input: string): Date | null という関数を書いてください。

入力: ISO 8601 文字列 ("2026-06-14", "2026-06-14T10:30:00Z")。
出力: Date オブジェクト、文字列が不正なら null。

処理すべきエッジケース:

- 空文字列または空白 -> null
- フォーマットは有効だがあり得ない日付 (2026-02-30) -> null
- 末尾にゴミ ("2026-06-14xyz") -> null

throw しないこと。不正な入力はすべて null を返す。

エッジケースを名指しすれば、AI はそれら进行处理する。名指ししなければ、ハッピーパスを書き、あなたはプロ

ダクションで抜け穴を発見する。同じ規律は純粋関数の外にも当てはまる。エンドポイントなら、成功レスポンスに加えてエラーレスポンスも指定せよ (重複したメールにはどのステータスコード? 不正なボディには?)。UI コンポーネントなら、埋まった状態だけでなく、空の状態とローディング状態も指定せよ。ほとんどのソフトウェアの興味深い挙動はその縁に住んでいる。だからそこにこそ、あなたのプロンプトは言葉を費やすべきだ。

エッジケースが何か分からないなら、まず AI に尋ねよ。「日付パーサーはどんなエッジケースを処理すべき?」——そしてその答えを制約としてフィードバックする。これはモデルをコードジェネレーターからチェックリストジェネレーターに変える。そしてチェックリストをレビューする方が、忘れたケースをデバッグするより速い。

小さなステップで作業する

最もよくある間違いは、一度に多くを求めすぎることだ。「認証、チャート、設定ページ付きのユーザーダッシュボードを作って」は、レビューできないコードの壁を、読んでもいないファイルにまたがってバグを散らばらせて生み出す。

小さなステップはあなたをコントロールの中に留める。

- 一つのプロンプトにつき、一つの関数、コンポーネント、またはエンドポイントを出荷する。
- 次へ進む前に各部品が機能することを確認する。
- 検証されていない出力の塔ではなく、動作が確認されたコードの上に積み上げる。

ログイン、サインアップ、パスワードリセット、セッション、メール認証を備えた認証システム全体を作って。

認証フローのステップ 1: サインアップのエンドポイントだけ書いて。
{ email, password } を受け取る POST /signup。パスワードを bcrypt でハッシュ化し、ユーザーを保存し、user id を付けて 201 を返す。
User モデルと db クライアントはすでに存在すると仮定して (その形は見せません)。ログインは次にやります。

小さなプロンプトは小さな差分を意味し、小さな差分は何か変わったかを実際に読めることを意味する。それがすべてだ。二つ目の見返りもある。何かが壊れたとき、小さなステップは二分探索で見つけるのが容易だ。サインアップのエンドポイントを生成して検証し、次にセッションロジック、次にパスワードリセットをやったなら、ステップ 3 の後に現れたバグはほぼ確実にステップ 3 に住んでいる。一発勝負の生成はその信号を捨てる——バグは一度も読んでいない四百行のどこにでもありうる。

コードの前にプランを求める

自明でないものなら何であれ、考えることとタイプすることを分離せよ。AI にまずアプローチの概要を述べさせる。これは間違った前提が 200 行の間違ったコードになる前に捕まえるし、プルリクエストを直すよりプランを直す方がはるかに安上がりだ。

自分の API にレート制限を追加したい。コードを書く前に、短いプランをください: リミッターをどこに置くべきか、どんなストレージが必要か、制限に達したとき何が起きるか、そしてトレードオフ。まだコードは書かないで —先にアプローチを承認します。

プランが正しそうなら「いいね、ステップ 1 を実装して」と言う。そうでなければ、コードを解きほぐすのに一時間ではなく、進路を修正するのに三十秒を費やしたことになる。まずプランを立てるという習慣こそ、エンジニア流のプロンプトを希望的観測のプロンプトから最も分かつものだ。

プラン段階はまた、AI が必要以上に重い解法を選ぶのを捕まえる場所でもある——インメモリのマップで済むのに Redis に手を伸ばしたり、一行のライブラリ呼び出しですでに解決することを手で書いたり。一つの問いを念頭にプランを読み。これは動きうる最も単純なものか? アプローチが過剰設計の匂いがするなら、一行も書かれる前に平易な言葉で押し返せ (「これはサーバー一台の趣味プロジェクトだ——分散ストアは省ける?」)。

書き直しではなく差分で反復する

コードが存在したら、また全体を求めたくなる衝動を抑えよ。「もっと良くして」のような曖昧なフォローアップは、動いているコードを捨て、すでに直したバグを再導入する。求める具体的な変更を指し示せ。

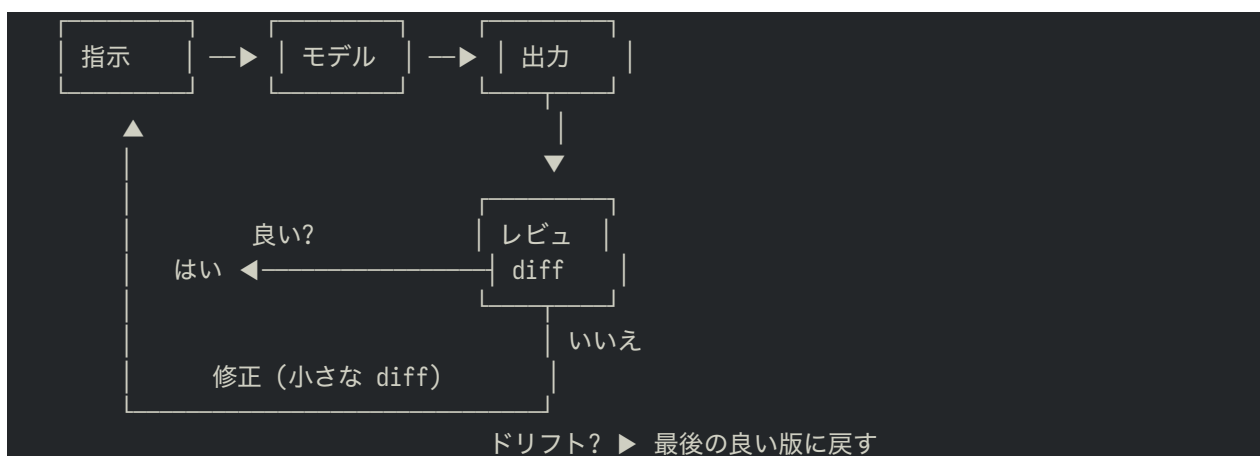
それは違う、全部書き直して。

関数は動くけど、二つ問題がある：

1. `throw new Error`` の行のエラーは、`throw` する代わりに `Result` 型を返すべき — 上の `validateUser` 関数のパターンに合わせて。
2. ループが毎回 `items.length`` を読み直している。巻き上げて。

その二つの変更の `diff` だけ見せて。

健全なサイクルは緊密なループだ。モデルにプロンプトし、出力を読み、小さく絞った差分で——決して全体の書き直しではなく——正しくなるまで磨く：



(全体の書き直しではなく) 差分を求めることは、変更をレビュー可能に保ち、すでに動いている部分を保存する。AI の出力を同僚のプルリクエストのように扱う。具体的な行にコメントし、的を絞った編集を要求する。変更が横道に逸れ、返答のたびにコードが望んだものから遠ざかっていくな、パッチを当て続けるな——信頼できた最後のバージョンに戻し、より鋭い記述でそこからプロンプトし直せ。壊れた土台の上で前へ反復し続けるのは、混乱を積み増すだけだ。

例を提供する

一つの実例は一段落の説明に値する。特定の形の出力が欲しいなら、その形を見せよ。これは曖昧さを取り除く最速の方法だ。

データをいい感じに整形して。

このユーザーの配列をサマリーオブジェクトに変換して。

入力：

```
[{ name: "Ann", role: "admin" }, { name: "Bo", role: "user" }]
```

望む出力：

```
{ total: 2, byRole: { admin: ["Ann"], user: ["Bo"] } }
```

変換関数と、この例に合うテストを書いて。

例は、言葉だけでは曖昧に残る命名、構造、大文字小文字を釘付けにする。可能なら、例をテストとして含めよ

——そうすれば「テストを通して」が曖昧さのない指示になる。この手は一般化する。サンプルの API(アプリケーション・プログラミング・インターフェース—あるプログラムが別のプログラムにデータを要求する仕組み)レスポンスはパーサーを釘付けにし、ビフォー/アフターの一対はリファクタリングを釘付けにし、スクリーンショットや ASCII スケッチはレイアウトを釘付けにする。ある形についての長い記述を書いている自分に気づいたら、止めて、代わりにその形の一つの実例を見せよ。

AI に何をすべきでないかを伝える

制約はあなたが望む結果についてだけではない——AI に避けてほしい手についてでもある。制約なしに放っておくと、モデルは依存を追加し、頼んでいないコードを「親切に」リファクタリングし、要求の縁に機能を発明する方へと漂う。いくつかの否定的制約が出力を引き締める。

検索入力にデバウンスを追加して。

検索入力にデバウンスを追加して。

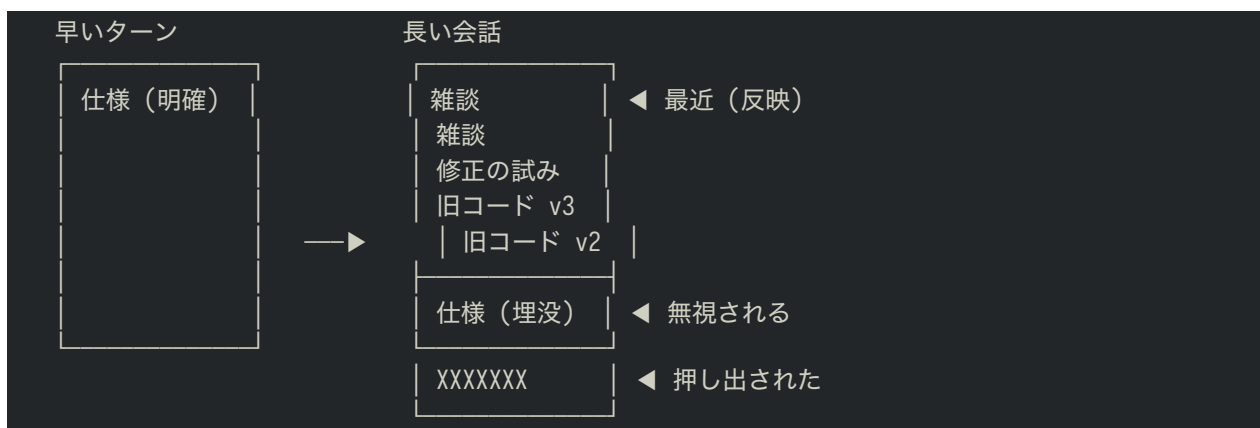
制約:

- 新しい依存は追加しない —すでに lodash があるので、それを使う。
- searchBar.tsx だけ触る。親はリファクタリングしない。
- 既存の prop の名前と型は変えない。
- 300ms の遅延。

二つ目のプロンプトは、最初のがうまくいかなくなる十二の道を未然に塞ぐ。否定的制約は既存のコードベースで特に価値がある。そこでは頼んでいない変更のコストは、それをレビューすることだけではない——マージコンフリクト、驚いたチームメイト、そしてちゃんと動いていたコードのリグレッションだ。越えてほしくない線があるなら、明示的に言え。AI は礼儀からあなたの境界を押し量ったりしない。

コンテキストをリソースとして管理する

長い会話はタダではない。スレッドが育つにつれ、以前の指示は埋もれ、モデルは最近のメッセージにより強く依存し、二十ターン前に述べた詳細は静かに重要でなくなる。コンテキストウィンドウを固定サイズの箱として思い描こう。会話が育つにつれ箱は満ち、最も早い指示はモデルが最も注意を払わない底へと押しやられる:



コンテキストウィンドウを無限の記憶ではなく、きれいに保つ責任があなたにある作業面として扱え。

いくつかの習慣が役立つ。

- **方針転換するときは仕様を述べ直せ。**十メッセージ前の要件がまだ有効だと思い込むな。現在のターゲットを短い段落に要約し、AI にそれから作業させよ。
- **新しい問題には新しく始めよ。**認証コードから CSS(カスケーディング・スタイル・シート—ページの見た目を制御する言語) のバグに切り替えるときは、新しい会話を開け。無関係なコンテキストを引きずると、モデルは良くなるのではなく悪くなる。

- ・ **履歴ではなく現在のコードを貼り付けよ。**関数が五回変わったなら、AI に五つの古いバージョンは要らない——今存在する一つが要る。現在の状態と次の変更を渡せ。
- ・ **チャットの外に短い真実の源を持て。**より長いタスクには、ファイルの中の数行 (「スタック、主要な決定、済んだこと、次のこと」) が、スクロールバックから再構築する代わりに、数秒で新しい会話を蒔き直させてくれる。

目標はこの章全体を貫くものと同じだ。あらゆる瞬間に、AI は手探りで推測すべき肥大した記録ではなく、正確で簡潔なタスクの絵から作業すべきだ。

避けるべき一般的なアンチパターン

これらは、あなたの時間を静かに浪費するプロンプトの習慣だ。自分のセッションで気をつけよう。

- ・ **曖昧な願い。**「良いログインページを作って」。誰の基準で良いのか。形容詞を具体的に置き換えよ。どのフィールド、どのバリデーション、どのフレームワークか。
- ・ **メガプロンプト。**機能全体を一つのメッセージに投げ込む。一度に一つずつ検証できるステップに分解せよ。
- ・ **沈黙のコンテキスト。**AI があなたのプロジェクトを覚えている、または知っていると思い込む。関連するファイル、型、規約を述べ直せ——特に新しい会話では。
- ・ **盲目的な受け入れ。**生成コードを読まずに貼り付ける。AI は間違っているても自信満々だ。記録上のエンジニアは依然としてあなただ。
- ・ **果てしないリトライ。**違う出力を願って「もう一度」を再実行する。二回試して失敗するなら、問題はプロンプトだ——コンテキストか制約を加えよ。ただ振り直すな。
- ・ **動く標的。**スレッドの途中で要件を述べ直さずに変える。方針転換するときは現在の仕様を要約せよ。AI が古い絵から作業しないように。
- ・ **おべっかループ。**「おっしゃる通りです!」を修正が正しい証拠として受け入れる。同意は検証ではない——コードを自分で実行し、挙動を確かめよ。

コアとなるマインドセット

ここでのあらゆるテクニックは一つの考えに帰着する。**あなたの心を読めない有能なチームメイトに言わなければならないことを言え。**コンテキストを最初に。明確な入力と出力。小さく、検証可能なステップ。コードの前にプラン。書き直しよりの絞った編集。言葉が足りないところには例を。何に触れてはいけないかの明示的な限界。あらゆる瞬間に、きれいで最新のタスクの絵を。

あなたは魔法の呪文を書いているのではない。仕様を、簡潔かつ正確に書き、そして結果をレビューしているのだ。それを一貫して行えば、AI はスロットマシンのように感じられなくなり、実際になり得る速くて疲れ知らずのエンジニアのように感じられ始める——そして同じくらい大切なことに、あなたは荷が出荷されたかを理解する人であり続ける。

まとめと実践

重要なポイント

- ・ タスクの前にコンテキストを与えよう: スタック、コンベンション、再利用すべき既存の部品を明示するか、コードが合わせるべき実際のシグネチャを貼ろう。
- ・ 境界を定義しよう: 入力、出力、エッジケース。ソフトウェアの興味深い振る舞いは縁に宿るので、そこに言葉を費やそう。
- ・ 小さく検証可能なステップで進め、コードの前に計画を求めよう。プルリクエストより計画を直すほうがずっと安い。
- ・ 全面書き直しではなく絞った差分で反復し、言葉では曖昧に残るものは具体的な例 (できればテスト) で釘付けにしよう。
- ・ コンテキストウィンドウを資源として管理し、やってはいけないことを言おう——明示的な否定の制約が出力をタイトに保つ。

やってみよう

普段ならそのまま打ち込むような曖昧な一行の依頼（「リストに検索を追加して」）を取り、何かを送る前にエンジニアのプロンプトに書き直そう。コンテキスト（スタック、触れるファイル）、入力／出力、二、三のエッジケース、望む結果の例を一つ、そして否定の制約を一つ（触れないもの）を加えよう。書き直した版を送り、怠けた版が生み出していたであろうものと比べよう。実際の依頼三つでこれをやれば、このアップグレードは自動になる。

この章のプロンプト

Context: < 言語 + フレームワーク + バージョン >, < ファイル/モジュール > で作業中。
We already < 再利用すべき関連する既存パターンまたはライブラリ >.

Task: < 作るべき明確なもの一つ >.

Inputs / outputs: < 型と形 >.

Edge cases to handle: < 重要なものを列挙 >.

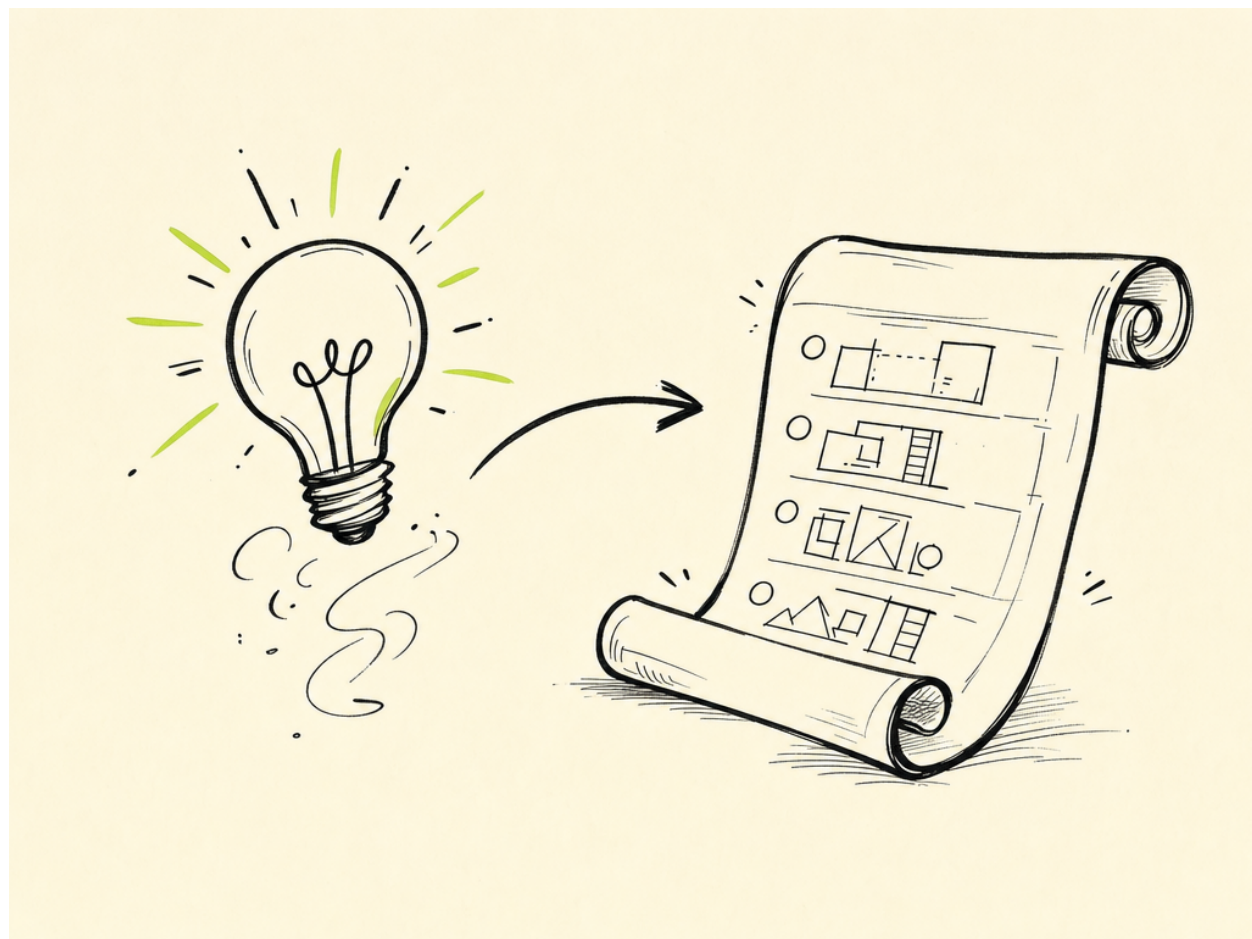
Example: < 具体的な入力一つ -> 期待される出力 >.

Constraints:

- No new dependencies — use < すでに持っているもの >.
- Touch only < このファイル >; don't refactor anything else.
- Keep existing names and signatures unchanged.

Before coding, give me a 3-4 line plan and wait for my "go".

アイデアから仕様へ、数分で



どんなプロジェクトも、最初はぼんやりとした思いつきから始まります。「読書の記録をつけられるアプリが欲しい」。これはよい種ですが、そこから直接ものを作ることはできません。あなたの AI にとっても同じです。曖昧なアイデアを AI に渡せば、出てくるのは曖昧なソフトウェアです。頼んでもいない機能、頼んだのに足りない部分、そして自分でも理解できないコードの山が生まれます。

その解決策が**仕様 (スペック)**です。これは、何を作るのか、そして何を作らないのかを、平易な言葉で書いた短いドキュメントです。よい知らせがあります。AI を思考のパートナーにすれば、一行のアイデアから組み立て可能な仕様まで、何日もかけずに数分でたどり着けます。この章では、そのやり方を紹介します。

仕様が何ではないのかをはっきりさせておきましょう。それは契約書ではありませんし、会議で守りに回って弁護するような設計書でもなく、一度書いたら凍結するものでもありません。自分自身と自分の AI に宛てた、作業用のメモです。学びながら編集していくものです。目指すのは完璧なドキュメントではなく、次の一時間の作業を前の一時間より明確にしてくれる十分によいものです。

なぜ「ノリ」ではなく仕様が必要か

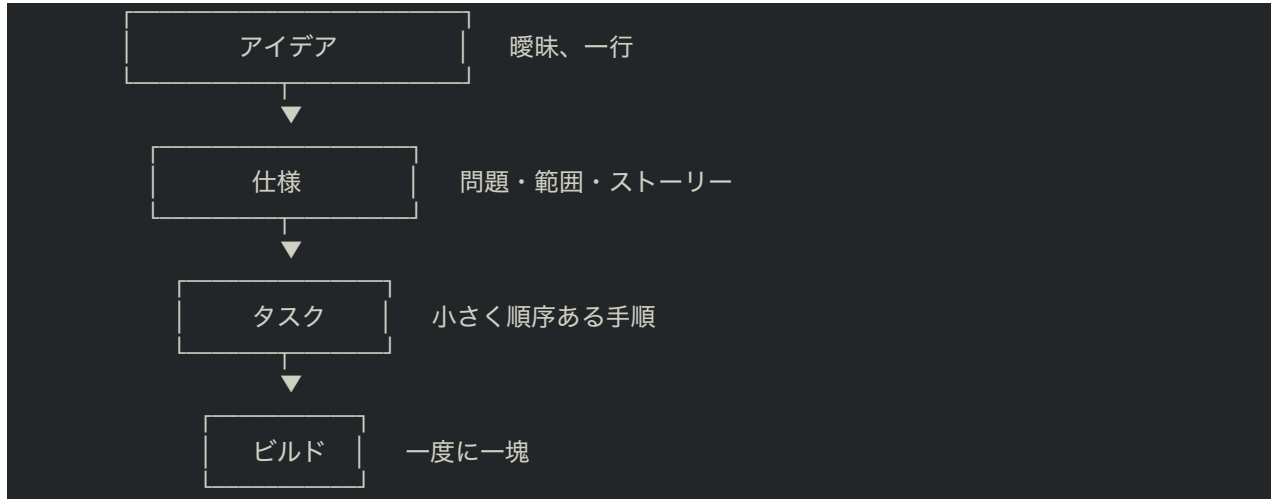
Vibe coding は、計画を省くという意味ではありません。計画を素早く、軽く行い、その後で AI に実行を指示するという意味です。仕様はハンドルです。これがないと、AI はプロンプトのたびにあなたの意図を推測することになり、小さな食い違いが積み重なって混乱に陥ります。

仕様は次の 3 つをもたらします。

- ・ **共有されたコンテキスト**—どんな AI セッションにも貼り付ければ、自分が何を作っているのかを即座に再確立できます。

- ・ **スコープの境界線**—「やること」と「やらないこと」の間に引かれた書面上の線が、機能の肥大化を食い止めます。
- ・ **タスクリスト**—AI が一つずつ片付けられる、組み立て可能な単位。

仕様はじょうごのように働きます。広くぼんやりしたアイデアを受け取り、段階ごとに絞り込んで、AI が実際に作れる小さな塊にしていきます。



長くする必要はありません。たいていのプロジェクトでは 1 ページで十分です。むしろ、2 ページを超えて膨らんでいく仕様は、たいてい危険信号です。先のことを計画しすぎているか、こっそりスコープを広げているかのどちらかです。迷ったら削りましょう。削った部分が惜しまれることはめったになく、残した部分はより鋭くなります。

ステップ 1: 問題を明確にする

機能の前に、問題と、それを抱えている人物を名指ししましょう。解決策に飛びつきたくなる気持ちを抑えてください。明確な問題定義は、その後のすべての判断を誠実なものに保ちます。

自分に問いかけてみましょう (あるいは AI にインタビューしてもらいましょう)。

- ・ これは誰のためのものか? (「自分」「個人開発者」「私のランニングクラブ」のように具体的に。)
- ・ その人たちは今、これがいないために、どんなつらいことをやっているのか?
- ・ 成功とはどういう状態か、一文で言うとは?

掘り下げる作業は AI に任せられます。次のプロンプトを試してみてください。

大まかなアイデアがある：読んでいる本を記録するウェブアプリ。
プロダクト思考のパートナーとして振る舞って。鋭い質問を 5 つ、
一度に一つずつ投げて、問題、ターゲットユーザー、「完了」が
どんな状態かを明確にして。まだ機能は提案しないで——アイデアが
具体的になるまで問い詰めるだけにして。

5 つのよい質問に答えるだけで、曖昧なアイデアはたいてい、現実味のある何かへと磨き上げられます。コツは、何でも同意してくるのではなく、AI に押し返させることです。「複数のデバイスで同期する必要がありますか?」と聞かれて、正直な答えが「いや、自分のノート PC だけでいい」なら、あなたは一行のコードも書く前に、バックエンドと認証システムと数週間分の作業を削除したことになります。これこそが要点です。機能を削るのに最も安上がりな場所は、コードベースの中ではなく、会話の中なのです。

あなたの答えが「うーん、たぶんそう」から「そう、まさにそれ」へと変わる瞬間に注意してください。その切り替わりこそ、本物の問題を見つけ、次へ進める合図です。

ステップ 2: スコープと非ゴールを定義する

これは最も価値があり、最も飛ばされがちなステップです。**非ゴール**—あえて作らないもの—は、「気の利いた」おまけを足したがる AI からあなたを守ってくれます。

私たちの読書トラッカーなら、スコープはこうなるかもしれません。

- ・ **スコープ内:** 本を追加する、読書中/読了とマークする、一覧を見る、読み終えた本を評価する。
- ・ **スコープ外 (非ゴール):** ソーシャル機能、レコメンド、Goodreads からのインポート、モバイルアプリ、ユーザーアカウント。

2 つの列を思い浮かべてください。左には作ると決めたものすべて、右にはあえて飛ばすものすべて。その間の線が、v1 を小さく保つ境界です。

スコープ内	非目標
本を追加	ソーシャル機能
読書中/完了	おすすめ
一覧表示	Goodreads 取込
読了本を評価	モバイルアプリ
	ユーザーアカウント

今すぐ作る ◀────────▶ 今はやらない

非ゴールを書き出すことには 2 つの役割があります。最初のバージョンを実際に完成させられるほど小さく保つこと。そして、AI が脱線したときに貼り付ける一文を用意できることです。「思い出して、アカウントは v1 の非ゴールだ。ログインは追加しないで。」

非ゴールとは「決してやらない」ではなく、「今はやらない」です。ユーザーアカウントを非ゴールに挙げても、あなたのアプリが永遠にそれを持たないという意味ではありません。それがこのバージョンの一部ではない、というだけです。だから今日その代償を払わずに済みます。この捉え直しが大切です。よいアイデアを諦めているような気分にならずに、機能にノーと言えるようになるのです。あなたは単に順序づけをしているだけです。非ゴールリストの項目のほとんどは、実のところ姿を変えた v2 のバックログであり、それでよいのです。どこかに書き留めて、先へ進みましょう。

ステップ 3: ユーザーストーリーを書く

ユーザーストーリーは、スコープをテスト可能な振る舞いに変えます。フォーマットはシンプルです。

[ユーザー] として、**[ベネフィット]** のために、**[アクション]** したい。

読書トラッカーなら、

- ・ 読者として、追跡を始められるように、タイトルと著者で本を追加したい。
- ・ 読者として、進捗が見られるように、本を読了とマークしたい。
- ・ 読者として、何が気に入ったか思い出せるように、読み終えた本を 1~5 の星で評価したい。

各ストーリーは、独立して作って検証できる単位です。アプリをあちこちクリックしてそのストーリーをどうテストするか思い描けないなら、それは曖昧すぎます。分割するか書き直しましょう。

so that(〜のために) の節は、人が省きがちな部分であり、そして最も重要な部分です。「読者として、検索ボックスが欲しい」は、AI に何を作るかは伝えますがなぜかは伝えないので、AI はよいトレードオフの判断ができません。「読者として、何十もの項目をスクロールしなくて済むように、タイトルの一部を入力して自分のリストから本を見つけたい」は、本当の仕事を AI に伝えます。こうなると、入力に応じて絞込むだけのシンプルな仕組みが、重量級の検索エンジンに勝つかもしれません。ベネフィットは、仕様のための仕様です。それは、AI(そして未来のあなた) がよい実装と間違った実装を見分けられるほど十分に、その機能を説明してくれます。

ステップ 4: 軽量な PRD を起草する

PRD(Product Requirements Document、プロダクト要求仕様書) とは、あなたの決定事項を一か所にまとめたものにすぎません。そこには予測可能な骨組みがあります—毎回同じ、ひとにぎりのセクションです。

```
PRD
├─ 問題          なぜ存在するか
├─ ユーザー      誰のためか
├─ ゴール        何をするか、一行で
├─ スcope内     v1 に含むもの
├─ 非目標       v1 で省くもの
├─ ユーザーストーリー ~ として、~ したい、~ のために
├─ データモデル 保存するフィールド
└─ 完了条件     受け入れテスト
```

ごく小さく保ちましょう。次に、AI に渡せる完成形のスターター PRD を示します。

```
# Book Tracker — Mini PRD

## Problem
I read a lot but lose track of what I've finished and what I thought of it.

## User
Me (a single user). No accounts needed for v1.

## Goal
A simple web app to add books, track reading status, and rate finished ones.

## In Scope (v1)
- Add a book (title, author)
- Set status: To Read / Reading / Finished
- List all books, filterable by status
- Rate finished books 1-5 stars

## Non-Goals (v1)
- User accounts / login
- Social features, sharing, recommendations
- Importing from other services
- Mobile / native apps

## Tech (keep it simple)
- Single-page web app
- Local browser storage (no backend for v1)

## Done When
I can add a book, change its status, rate it, and see my list
persist after a page refresh.
```

「Done When(完了条件)」の行に注目してください。これがあなたの受け入れテストであり、v1 を出荷できたかどうかを判断する具体的な手段です。必ず一つ含めましょう。よい「Done When」は、実際に自分の手で実行できるものです。ここをクリックし、これを入力し、リロードし、あれが見える、というように。完了条件が感覚のように読めてしまうなら(「磨き上がったら」「いい感じになったら」)、それは決して完了しません。照らし合わせるものが何もないからです。見知らぬ人に渡せる手順書になるようにしましょう。

PRD を手で書く必要はありません。便利なやり方は、前のステップで出した答えを AI に戻し、AI にドラフトを

組み立てさせて、それをあなたが編集することです。丸ごと受け入れるのではなく編集する意味は、あなたが一行一行を理解し、納得しなければならない点にあります。このドキュメントは、あなたが繰り返し貼り付けることになる信頼の源 (source of truth) であり、ここに誤りがあれば、それ以降のすべてのセッションにコピーされてしまうからです。

ステップ 5: データモデルをスケッチする

ほとんどのアプリは、実のところデータに関するものです。何を保存し、各要素がどう関係するか。手早いデータモデルのスケッチは、後で混乱した一貫性のないコードに悩まされるのを防いでくれます。データベースの図は必要ありません。フィールドのリストで十分です。

読書トラッカーなら、主役は一つ—Book です。

```
Book
- id: 一意の文字列
- title: 文字列
- author: 文字列
- status: "to_read" | "reading" | "finished"
- rating: 数値 1-5 (読了の場合のみ)
- addedAt: 日付
```

この小さなスケッチですら、決定を明示してくれます。評価は読了した本にしか存在しない、ステータスは決まった 3 つの値のいずれか、すべての本には id がある、というように。これを AI に渡せば、AI はフィールド名を推測するのをやめ、一貫した形で組み立てます。

これを早い段階で固める理由は、フィールド名があちこちに染み出していくからです。いったん AI が book.rating を読むコードを書けば、その名前はフォーム、リスト表示、ストレージ層、そして後で追加するどんな機能にも現れます。それがぶれるのを放っておくと—ある場所では rating、別の場所では stars、また別の場所では score—追いかけるのが面倒なバグが生まれます。6 行のデータモデルは、それに対する最も安価な保険です。

短くてコピー & ペーストできる仕様ブロックは、AI にとって散文よりうまく機能する傾向があります。同じモデルを、実際に送るプロンプトとして書くようになります。

本にはこの正確なデータ形を使って。挙げていないフィールドは追加せず、これらの名前も変えないで：

```
Book {
  id      —一意の文字列、作成時に生成
  title   —文字列、必須
  author  —文字列、必須
  status  —次のいずれか: "to_read" | "reading" | "finished"
  rating  —整数 1-5、status が "finished" のときのみ存在
  addedAt — ISO 日付文字列、本の追加時に設定
}
```

フィールドが足りないと思ったら、追加する前に聞いて。

最後の一行—ask me before adding it(追加する前に聞いて)—が実際の仕事をしています。これは AI を推測者から協力者へと変え、データモデルをあなたのものに保ちます。

ステップ 6: 仕様を Vibe サイズのタスクに分割する

最後の一手は、仕様を、AI が一つずつ実行できるタスクへと切り分けることです。Vibe サイズのタスクとは、結果を一分でレビューでき、うまくいったかどうか分かるほど小さいものです。一つのタスクで AI が複数のファイルにまたがり、それが正しいかどうか判断できないなら、それは大きすぎます。

読書トラッカーのよいタスク例:

1. ヘッダーと空の本リストを備えた、基本的なページレイアウトを用意する。
2. リストにメモリ上で追加する「本を追加」フォーム (タイトル + 著者) を作る。
3. ステータス (To Read / Reading / Finished) と、それを変更するドロップダウンを追加する。
4. リストをブラウザのローカルストレージに保存し、リロードしても残るようにする。
5. 読了した本にだけ表示される評価コントロールを追加する。
6. リストの上にステータスフィルターのボタンを追加する。

各タスクはユーザーストーリーや PRD の一行に対応し、前のタスクの上に積み上がり、最後に目に見えて動く何かで終わります。これが vibe coding のリズムです。小さなタスク、実行、検証、次。

この分割を AI に生成してもらうこともできます。

これが私のミニ PRD とデータモデル [ここに貼り付け]。
これを小さく順序ある 6~8 個のビルドタスクに分割して。各タスクは
独立してテスト可能で、前のタスクの上に積み重ねること。各タスクは
アプリをクリックして確認できる範囲に保って。

仕様を生かし続ける

仕様は、一度建てて眺めるための記念碑ではありません。それは生きたドキュメントであり、最もよくある間違いは、それを固定されたものとして扱うことです。作っていくうちに、計画段階では分らなかったことが見えてきます。ある機能が思ったより難しい、非ゴールのはずが実は不可欠だった、一つのユーザーストーリーが実はこっそり二つだった、というように。そうなったら、まず仕様を編集してから、新しいバージョンに従うよう AI に伝えましょう。仕様が先導し、コードが従います。

いくつかの習慣が、仕様を誠実に保ちます。

- ・ **決定が変わった瞬間に更新する。** コードと違うことを言っている仕様は、仕様がないよりも悪いものです。それは次の AI セッションを積極的にミスリードします。
- ・ **終えたタスクは消し込む。** タスクリストは進捗トラッカーも兼ねます。8 つのうち 6 つにチェックが入っているのを見ること自体が、一種の勢いになります。
- ・ **思いがけないものは、ただやってしまうのではなく非ゴールや「あとで」リストに移す。** 魅力的な機能が作業の途中で現れたら、書き留めて先へ進みましょう。それが v1 に入る価値があるかどうかは、後で決められます。

これこそが、「何かを作った」と「作ろうとしたものを作った」を分ける規律です。仕様は、あなたがプロジェクトの乗客ではなく、作者であり続けるための手段なのです。

よくある罠: 問題ではなく解決策を仕様化してしまう

ひとつの失敗パターンは、単独で取り上げる価値があります。あまりにもはまりやすいからです。仕様を書こうと腰を据えようと、すでに頭に描いた解決策を書きたくくなります。「タブ付きのサイドバー」「ポップアップするモーダル」「Postgres のテーブル」。しかし解決策で埋め尽くされた仕様は、まだ得てもしない決定をこっそり固定してしまいます。

この 2 行を比べてみてください。

悪い: 本の状態ごとに折りたためるセクションのある左サイドバーを追加して。
良い: 全体の一覧を失わずに、本を状態ごとにまとめて見せて。

最初の文は、それが正しい判断だと誰も確認しないうちから、サイドバー、折りたたみ可能なセクション、そしてレイアウトをすでに選んでしまっています。二つ目の文はなすべき仕事 (job to be done) を述べ、AI(とあなた)が、それを満たす最もシンプルなものを自由に見つけられるようにしています。それは、3 つのフィルターボタンだけかもしれませんが。欲しい結果を述べましょう。実装は、できるだけ長く交渉の余地があるままにしておきましょう。解決策は、仕様の中から安く変えられ、コードの中では高くつきます。

その見返り

これであなたは、問題定義、スコープと非ゴール、ユースストーリー、1 ページの PRD、データモデル、そして順序付けされたタスクリストを手に入れました。本物のプロジェクトの土台一式を、AI を壁打ち相手にして数分で起草したのです。

このどれにも、コードの書き方を知っている必要はありませんでした。必要だったのは、自分が何を欲しいのかを知っていることと、数分だけ明晰に考えることです。残りの「声に出して考える」作業は AI があなたと一緒にこなしてくれました。これこそが本当の転換です。あなたはまだソフトウェアを書いているのではなく、意図を書いているのです。そしてよい意図は、戦いのほとんどを占めます。

この仕様はプロジェクト内のファイルに保存しておきましょう (多くの人は SPEC.md や PRD.md を使います)。それは、新しい AI セッションに貼り付けてコンテキストを再固定するためのドキュメントになり、Vibe サイズのタスクを一つずつ消化していくためのチェックリストになります。次の章では、最初のタスクを取り上げ、実際にそれを作っていきます。

まとめと実践

重要なポイント

- ・仕様は、短く平易な言葉の作業メモで一何を作り、何を作らないか。凍結する契約ではなく、一ページで十分。
- ・機能の前に問題と、その問題を抱える人を明確にし、ノンゴール (non-goals) を書き出そう。AI が「親切な」追加機能を入れてくるのに対する、最も鋭い防御だ。
- ・so that 節のあるユースストーリーと、具体的な「Done When」の一行を持つ小さな PRD は、意図をテスト可能で構築可能な振る舞いに変える。
- ・データモデルを早めに釘付けにしよう—フィールド名はあちこちに漏れ出す—そして仕様を、一度に一つずつ検証できる Vibe サイズのタスクに切り分けよう。
- ・解ではなく問題を記述しよう。成果は仕様では変えるのが安く、コードでは高い。学びながら仕様を生かし続けよう。

やってみよう

温めてきたアイデアの一つ取り、単一の AI セッションの中で一ページの仕様に変えよう。まず AI に五つの鋭い質問であなただけをインタビューさせ、それから—自分の言葉で—問題、ユーザー、三つのユースストーリー (各々に so that)、明示的なノンゴールのリスト、六行のデータモデル、そして物理的に実行できる「Done When」の一行を書こう。SPEC.md として保存しよう。よい仕様のテスト: 明日まっさらな会話に貼り付けても、AI が何を作るべきか正確に分かること。

この章のプロンプト

大まかなアイデアがある : < 一行の説明 >。

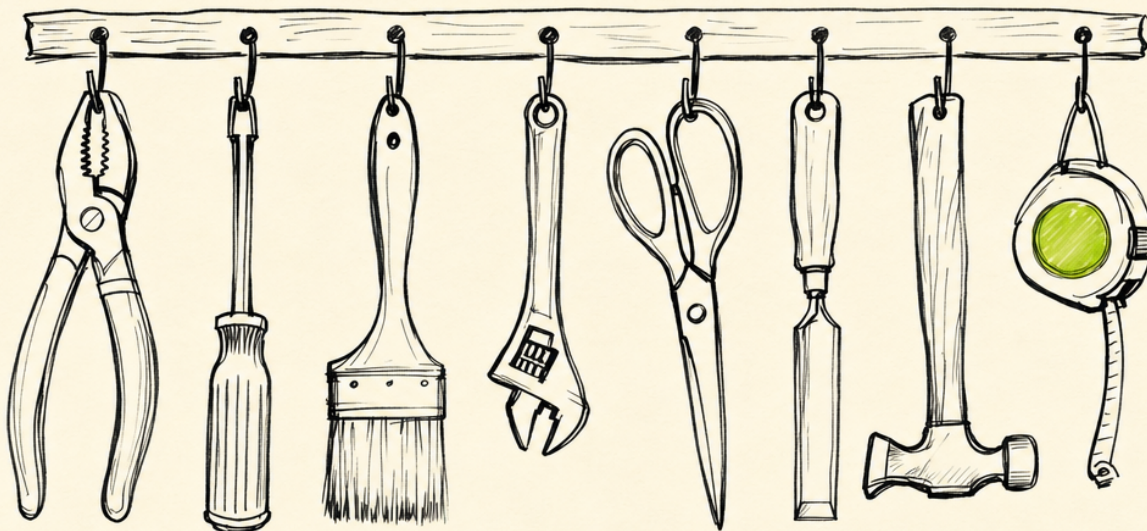
私のプロダクト思考パートナーとして振る舞って。まず、鋭い質問を 5 つ、一度に一つずつ投げて、問題、ターゲットユーザー、「完了」がどんな状態かを明確にして。まだ機能は提案しないで。

私が答えたら、次のセクションで一ページの仕様の下書きを作って :

- 問題 - ユーザー - ゴール
- スコープ内 (v1) - 非目標 (v1)
- ユースストーリー (各「~ として、~ したい、~ のために」)
- データモデル (フィールドの短いリスト)
- 完了条件 (具体的で実際に実行できる受け入れテスト)

簡潔に——一ページ以内で。曖昧な点があれば、仮定する前に聞いて。すべての行に同意するまで下書きを直すから。

スタックを選ぶ: 言語をひと目で

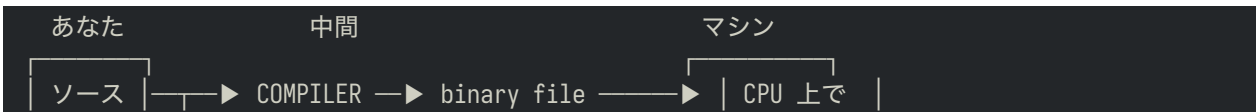


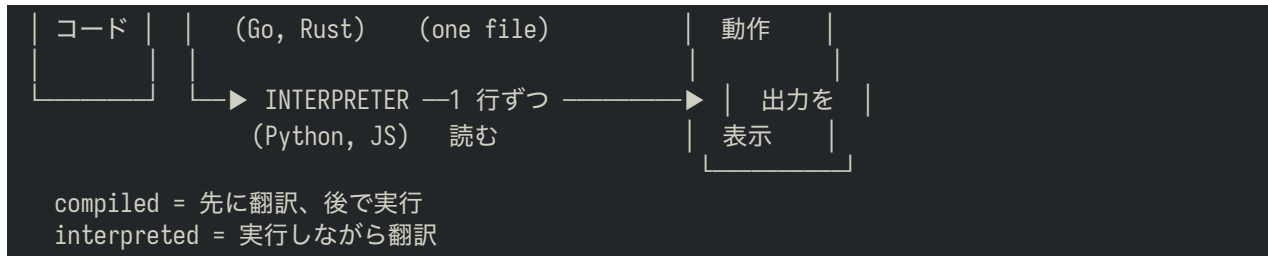
AI にソフトウェアを作らせるとき、選ぶ言語は今でも大いに重要です。コードを書くのは AI ですが、その結果と付き合うのはあなたです。どれだけ速く出荷できるか、デプロイがどれだけ簡単か、どれだけ頻繁に壁にぶつかるか。よい知らせは、これらの言語のどれかを習得する必要はないということです。まともな選択をし、AI があなたをつらい方向へ導いているときにそれを見抜けるだけの知識があれば十分です。

この章では、気にかける価値のある言語を一通り見ていきます。それぞれが本当に得意とすること、どこで噛みついてくるか、そして—vibe coding にとって決定的に重要な—AI アシスタントがどれだけうまく扱えるかです。最後に、目的別に選ぶためのきわめてシンプルなガイドで締めくくります。

始める前に一言。AI アシスタントは、ニッチな言語よりも、公開コードが山ほどある人気言語の方がはるかに得意です。これは小さな差ではありません。プロジェクトがスムーズに進むかどうかを左右する、たいていは単独で最大の要因です。人気は一つの機能なのです。ある言語とその主要フレームワークが公開リポジトリに何百万回も現れるなら、AI はあらゆる一般的なパターン、典型的なエラー、慣用的な修正を見てきたことになります。そうでなければ、その希少さの代償を、失敗した試みと自信たっぷりに聞こえる戯言で払うことになります。

言語名を見る前に、頭に留めておきたい一枚の絵があります。あなたが書くコードは、機械が実際に走らせるものではありません。その間にある何かが、あなたのテキストを動くソフトウェアに変えます—コンパイラ(あらかじめ全部を翻訳して実行ファイルにする)か、インタプリタ(一行ずつ読んで実行する)です。選ぶ言語が、どちらの経路をたどるかを決めます。





JavaScript / TypeScript

これはたいていの vibe coding におけるデフォルトであり、それには理由があります。どこでも動くのです—ブラウザ、サーバー (Node.js)、エッジ関数—そして Web フロントエンド全体の言語です。TypeScript は型システムを後付けした JavaScript であり、ほとんどの場合 TypeScript を選ぶべきです。型は早期にミスを抑え、そして重要なことに、AI にガードレールを与えて出力をより信頼できるものにします。型付きの関数シグネチャは、アシスタントが読める契約書です。それがなければ、AI はあなたのデータがどんな形をしているかを推測するしかなく、推測こそバグの隠れ場所です。

- **得意なこと:** Web アプリ (フロントエンドもバックエンドも)、ブラウザでユーザーがクリックするあらゆるもの、手早いプロトタイプ、一つの言語で完結するフルスタックプロジェクト、サーバーレス/エッジへのデプロイ。
- **つらいところ:** エコシステムの動きが速く、互換性が壊れる。依存関係の肥大化は現実の問題 (小さなアプリでも数百のパッケージを引き込みうる)。素の JavaScript のゆるさは微妙なバグをすり抜けさせる—これも TypeScript を使うべきもう一つの理由です。
- **代表的なユースケース:** SaaS (Software as a Service —インストールせずにオンラインで借りて使うソフトウェア) のダッシュボード、ランディングページ、e コマース、リアルタイムアプリ、ブラウザ拡張、Vercel・Netlify・Cloudflare にデプロイするあらゆるもの。
- **AI の扱い:** 優秀。学習データに最も多く含まれている言語です。アシスタントは React、Next.js、Node を自信を持って書き、めったに行き詰まりません。あえて別を選ぶ強い理由がないなら、ここから始めましょう。

知っておく価値のある具体的なトレードオフが一つあります。JavaScript をプロトタイピングに優れたものにしてその速さは、同時にそれを目まぐるしく変えてしまう速さでもあります。18 か月前のチュートリアルがすでに非推奨の API を使っているかもしれず、過去のスナップショットで学習した AI は、ときに昨日のパターンを手渡してきます。それが書いたコードが動かないとき、原因は往々にしてライブラリが先に進んでしまったことです。直し方はたいてい、自分が使っているバージョンを伝え、最新のドキュメントを確認させることです。これは地球上で最も活発なエコシステムに住む代償として払う税金であり、ほとんどのプロジェクトでは払う価値があります。

Python

Python はほとんど英語のように読め、それが初心者にとって最も親しみやすい言語であり、データ・スクリプティング・そして AI そのものの共通語にしています。プロジェクトが機械学習、データ分析、自動化、科学技術計算に関わるなら、たいていの答えは Python です。

- **得意なこと:** AI/ML、データ処理、自動化スクリプト、バックエンド API (Application Programming Interface —あるプログラムが別のプログラムと話すために使う出入り口)、つなぎのコード、すでにライブラリが存在するあらゆるもの (そして Python では、たいてい存在します)。
- **つらいところ:** 内部が C で書かれたライブラリに頼らない限り、生の数値計算は遅い。仮想環境とパッケージのバージョンをめぐる「依存地獄」が初心者を混乱させる。ブラウザのフロントエンド向けではない。
- **代表的なユースケース:** スクレイピング、データパイプライン、ML の実験、Discord/Telegram のボット、社内ツール、REST (Representational State Transfer —そうした API の出入り口を作る一般的なスタイル) API (FastAPI、Django、Flask)。
- **AI の扱い:** 優秀。JavaScript と同等です。アシスタントは Python とその主要ライブラリにきわめて流暢です。非 Web プロジェクトの優れたデフォルト。

Python が初心者をおぼつかせる唯一の場所は、言語ではなく環境です。AI にとって一つのフォルダでは問題なく動くスクリプトが、別のフォルダでは失敗します。パッケージが有効化されていない仮想環境の中にあるか、二つのプロジェクトが同じライブラリの別々のバージョンを欲しがらるからです。どれもあなたのコードのせいではなく、一度見てしまえば難しくありません—けれども、Python で「なぜそのまま動かないのか」という瞬間として最も頻繁に起こります。最初から AI に仮想環境と requirements.txt をセットアップさせる (あるいは uv や Poetry のようなツールを使わせる) と、痛みのほとんどは消えます。

Go

Go は Google で、シンプルで、速く、デプロイが簡単になるように設計されました。一つの自己完結したバイナリにコンパイルされ、ランタイムを入れなくてもサーバーにそのまま置けます。あえて凝った機能を省いており、それがコードベースをよい意味で退屈に保ちます—そして退屈さは、人間にとっても AI にとっても推論しやすいものです。

- **得意なこと:** 速いバックエンドサービス、コマンドラインツール、ネットワーキング、大量の同時接続をさばく必要があるもの、シンプルなデプロイ。
- **つらいところ:** 冗長なエラー処理 (if err != nil をいたるところで目にします)。フロントエンド言語ではない。意図的なシンプルさは、表現力のある近道が欲しいときには窮屈に感じられることがある。
- **代表的なユースケース:** API、マイクロサービス、CLI(Command-Line Interface—コマンドを入力してプログラムを実行する、テキストだけの方法)、インフラ用ツール、高トラフィックのバックエンド。
- **AI の扱い:** とても良い。Go の小さく一貫した機能セットは、たいていのことに「正しいやり方」が一つあることを意味し、アシスタントはそれにうまく従います。JS/Python より学習データは少ないものの、シンプルさがそれを補います。

個人開発者にとっての Go の決定的な強みは、あの単一バイナリです。「サーバーに Node とこの 40 個のパッケージを入れる」というダンスもなく、自分のマシンと本番との間でランタイムのバージョンがずれることもありません。ビルドして、一つのファイルをコピーして、走らせる。それだけでデプロイは劇的にエラーが起きにくくなります—vibe coding をしていて、インフラのデバッグなどしたくないときには、これが大きな意味を持ちます。冗長さがその代償です。Python なら 3 行のコードが、Go では 8 行になります。けれども、その余分な行は予測可能で、予測可能さこそ AI がうまく扱うまさにそのものです。

Rust

Rust は、まるごとのカテゴリのメモリバグを書かせまいとするコンパイラとともに、C 並みの速度をもたらします。その代償が、悪名高いほど厳格なコンパイラと険しい学習曲線です。vibe coding にとって、Rust は諸刃の剣です。書くのは AI ですが、Rust の厳格さゆえにコンパイラはコードをより頻繁に却下し、修正のやり取りが増えることになります。

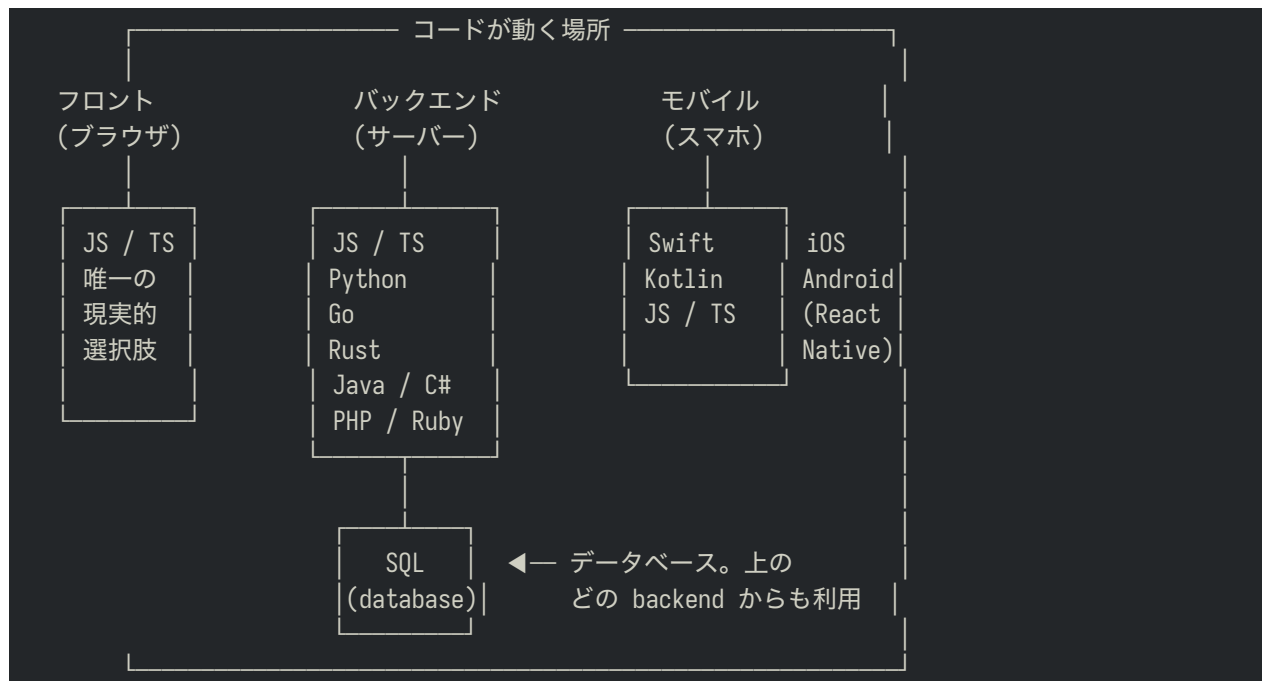
- **得意なこと:** パフォーマンスが重要なソフトウェア、システムプログラミング、ゲームエンジン、WebAssembly、速度が問われるコマンドラインツール、正しさと安全性が最優先されるあらゆるもの。
- **つらいところ:** 険しい学習曲線。遅いコンパイル時間。「ボローチェッカー」が初心者を悩ませる。ほとんどの Web アプリやスクリプトには過剰。
- **代表的なユースケース:** 高性能なバックエンド、CLI ツール、WASM(WebAssembly) モジュール、組み込みに近いシステム、1 ミリ秒も惜しいインフラ。
- **AI の扱い:** 良いが、でこぼこ。アシスタントは Rust をよく知っていますが、厳格なコンパイラが反復を要するエラーを浮かび上がらせます。Python や JS よりも「エラーを直してもう一度」のループが多くなると見込んでください。パフォーマンスや安全性がその摩擦を本当に正当化するとき Rust を選びましょう—デフォルトとしてではなく。

見落としがちですが、Rust の厳格さには微妙な利点があります。コンパイラが無料で、容赦なくコードをレビューしてくれるのです。AI がずさんな JavaScript を書くと、そのバグは出荷され、ユーザーを実行時に噛みめます。AI がずさんな Rust を書くと、コンパイラはたいていプログラムが走り出す前にそれを捕まえます。増える修正ループは摩擦のように感じられますが、そのかなりの部分は、コンパイラが本物のバグを止めてくれているのです。Rust に手を伸ばす前に投げかけるべき正直な問いは、自分が本当にその速度や安全性を必要としているのか、それとも欲しがらるべきだと言われたただけなのか、です。最初のプロジェクトでは、ほぼ必ず必要ありません。

その他の言語について手短に

これらにも出くわすことがあり、ときにはそのうちの一つがまさにうってつけです。

各言語が主にどこに住んでいるかを思い描くと役立ちます。ソフトウェアには層があります—ブラウザで動くもの (フロントエンド)、サーバーで動くもの (バックエンド)、スマホで動くもの (モバイル)—そしてほとんどの言語には縄張りがあります:



- **Java / C#:** 大企業の働き者。成熟し、安定し、ツールが充実しており、大規模な企業向けバックエンドや、Android(Java/Kotlin)、Windows/.NET エコシステム (C#) に理想的です。AI はどちらもうまく扱います。個人の vibe コーダーとしては、既存の企業スタックの中で作業するか、Android/Windows 向けに特化して作るのでもない限り、これらが必要になることはめったにありません。
- **Swift / Kotlin:** iOS(Swift) と Android(Kotlin) アプリのネイティブ言語。本物のモバイルアプリを App Store や Play Store に出荷するなら、これらに触れることになるでしょう。AI のサポートはしっかりしていますが、Web 言語ほどではなく、モバイル開発には余分な摩擦 (端末、アプリストアの規約、ビルドツール) が伴います。
- **PHP / Ruby:** Web の古参であり、いまでも静かに Web の巨大な一画を動かしています。PHP は WordPress を動かし、それゆえ全 Web サイトの大きな割合を支えています。Ruby on Rails は、データベースを背後に持つアプリを速く、意見の強いやり方で作る手段として残っています。どちらも学習データが豊富なので AI はうまく扱います—けれども今日、初心者が真っ先に手を伸ばすことはまれで、たいていは新規に始めるよりも既存のコードベースの中で作業するために選ぶことになります。
- **SQL(Structured Query Language):** 汎用言語ではありませんが、アプリがデータを保存した瞬間に出会います。データベースに質問を投げかける言語だと考えてください—「今週登録したユーザーを全員見せて」のように。SQL はデータベースに問い合わせます。SQL は他の言語の代わりに選ぶものではありません—何を選んでも、その隣に共存します。よい知らせは、AI は SQL を書くのがとても得意なので、欲しいデータを平易な言葉で説明すれば、クエリを生成してくれることです。

ひと目で比較

言語	最適な用途	注意点	AI の流暢さ
JavaScript/TypeScript	Web アプリ、フルスタック、エッジ	依存関係の肥大化、速い変化	優秀

言語	最適な用途	注意点	AI の流暢さ
Python	AI/ML、データ、スクリプト、API	生の計算が遅い、環境構築	優秀
Go	速いバックエンド、CLI、サービス	冗長、フロントエンド向きでない	とても良い
Rust	パフォーマンス、安全性、WASM	険しい学習曲線、修正ループが多い	良い (でこぼこ)
Java/C#	エンタープライズ、Android、.NET	重い、個人にはしばしば過剰	とても良い
Swift/Kotlin	ネイティブ iOS / Android アプリ	モバイルの摩擦、AI が弱め	良い
PHP/Ruby	WordPress、Rails、既存の Web アプリ	古びたデフォルト、レガシーの荷物	とても良い
SQL	データベースとの対話	代わりにではなく、隣に共存	優秀

風変わりな選択の本当のコスト

ある言語を、エレガントだから、速いから、あるいは尊敬する誰かが絶賛しているからという理由で選びたくなる誘惑があります。とりわけ vibe coding では、その本능が静かにプロジェクトを沈めかねません。理由はこうです。ニッチな言語のツールと格闘する一時間も、モデルが見た例が少なすぎて微妙に誤った AI 出力をひどく一時間も、出荷に使えなかった一時間です。風変わりな選択がけたたましく失敗することはめったにありません。それは、増える修正ループ、存在しないライブラリ、存在しない Stack Overflow の回答という、遅い水滴として失敗します。

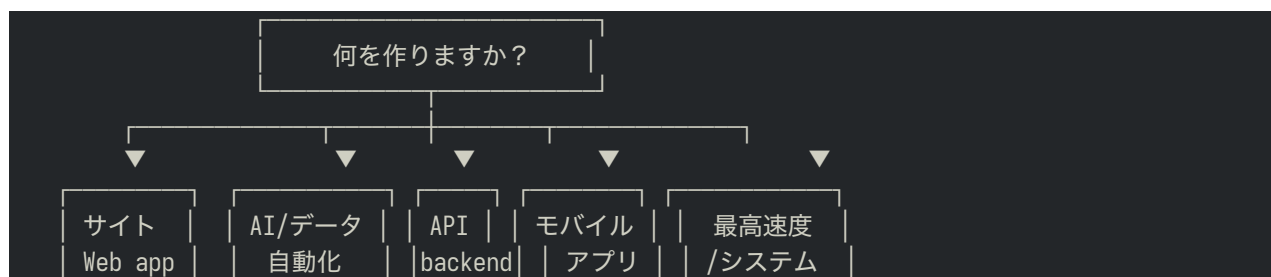
具体的には、その罰則は三つの場所に現れます。第一に、AI の流暢さが落ちるので、より多くの間違いを犯し、それを直すのも下手になります—そして、その言語を知らないあなたには、良いコードと悪いコードの区別がつきません。第二に、エコシステムが薄いので、必要なライブラリが存在しないか、放棄されているか、コピーできる例がないかもしれません。第三に、行き詰まったとき、Web はより静かです。チュートリアルも、回答された質問も、あなたとまったく同じ壁にぶつかった人も少ない。Python や TypeScript では、この三つはどれも問題になりません。

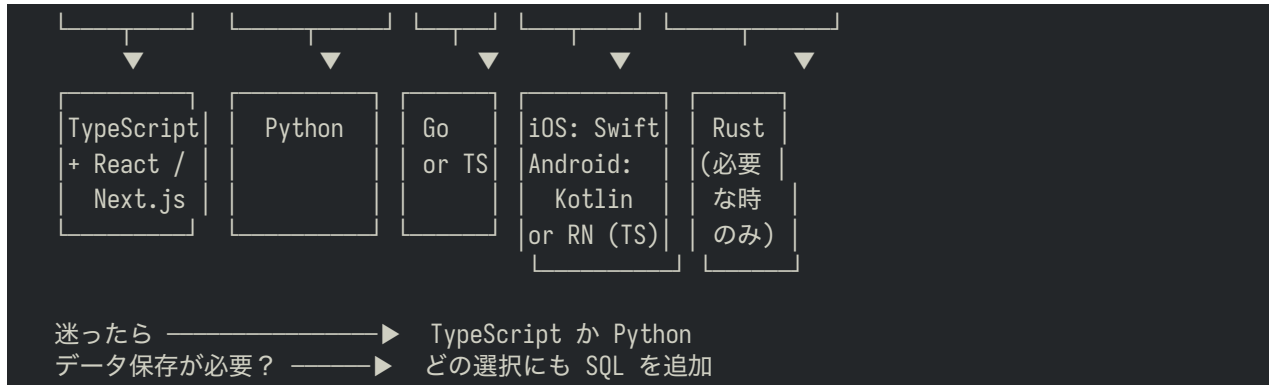
これは風変わりな言語が悪いという意味ではありません。立証責任が風変わりな選択の側にあるという意味です。具体的に明確な理由があるなら—本当に Rust の速度が必要だ、App Store に強いられて Swift だ、既存の Go サービスを拡張している—なら、目を開いたままそのコストを払いましょう。理由が「かっこよさそうだから」なら、人気の選択肢を選んで何かを出荷しましょう。あとからいつでも書き直せますが、ほぼ書き直すことはありません。人気の選択肢がすでに仕事を片付けているからです。

目的別に選ぶ

悩み抜くのはやめましょう。あなたの目的を一つの行に合わせて先へ進んでください—後でいつでも進路は変えられますし、最初のプロジェクトでは最適化よりも勢いが勝ります。

決定の全体を一つのツリーにするとこうなります。一番上から始め、各質問に答え、枝をたどってあなたのスタックにたどり着いてください:





- **ウェブサイトや Web アプリを作りたい。** □ TypeScript(React/Next.js と一緒に)。当然で、サポートも手厚い選択。
- **AI、データ、自動化をやりたい。** □ Python。ツールがそこに揃っています。
- **速い API やバックエンドサービスが欲しい。** □ Go(シンプルで速い) または TypeScript(Web スタックの一部なら)。Python の FastAPI も問題なく機能します。
- **モバイルアプリを作っている。** □ iOS なら Swift、Android なら Kotlin —あるいは、TypeScript に留まるためにクロスプラットフォームの Web ベースのアプローチ (React Native) を検討しましょう。
- **生の速度やシステムレベルの制御が必要だ。** □ Rust。ただし、本当に必要だと確認できた場合に限ります。ほとんどの人には不要です。
- **すでに存在するサイトに手を加えている。** □ それがすでに書かれている言語で。WordPress のプラグインなら PHP、古い Rails アプリなら Ruby です。コードベースに合わせ、それと戦わないこと。
- **とにかく学んで、何かを速く出荷したい。** □ Python か TypeScript。どちらも寛容で、どちらも AI が最もよく知る言語で、どちらにも尽きないチュートリアルと例があります。
- **アプリがデータを保存する必要がある。** □ 上で選んだ何かに加えて、データベース用の SQL。両者から選ぶわけではありません。

メタ的な教訓: 迷ったら、人気の選択肢を選びましょう。TypeScript と Python は、個人の vibe コーダーがこれから作るものの大半をカバーし、AI アシスタントが最も確実に扱える言語です。vibe coding にとって「最良の」言語は、たいてい AI が最もよく知っている言語—つまり人気の言語です。風変わりな選択は、具体的な理由ができたときのために取っておき、いずれにせよ構文は AI に任せましょう。

おさらいと練習

要点

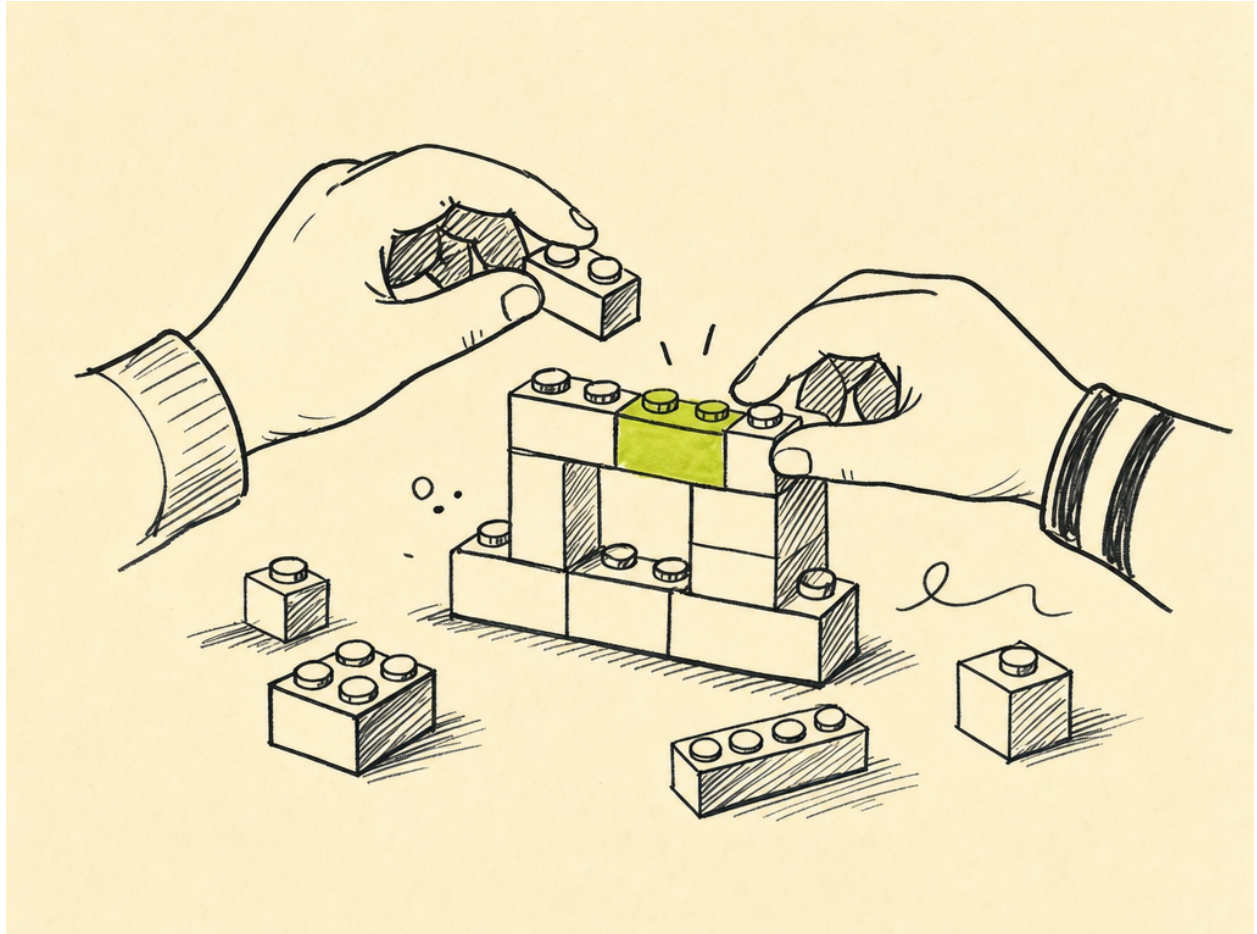
- AI はニッチな言語より人気の言語をはるかに流暢に扱えます—人気は一つの機能であり、プロジェクトが順調に進むかを左右する最大の単一要因であることが多いのです。
- TypeScript と Python は安全な既定値です。ウェブ/フルスタックは TypeScript、AI/データ/自動化は Python で、どちらも AI のサポートが優れています。
- Go(単一バイナリのバックエンド)、Rust(速度/安全性、ただし修正ループが増える)、そしてその他はそれぞれ狭い得意領域を持ちます—具体的な理由があるときだけ手を伸ばしましょう。
- SQL は二者択一の選択ではありません。アプリがデータを保存した瞬間から、どの言語を選んでもその隣で共に存在します。
- 立派に聞こえるものではなく、あなたの目的に言語を合わせましょう—最初のプロジェクトでは勢いが最適化に勝ります。

やってみよう

実際に作りたいプロジェクトのアイデアを一つ取り上げ、その目的を一文で書いてみましょう。それを「目的別に選ぶ」リストの一行に当てはめ、AI アシスタントにその選択を確認するか反論してもらいます。既定のスタック(言語、フレームワーク、どこにデプロイするか)を3つの箇条書きで整理させ、別の言語のほうが適する一つの場面を挙げてもらいましょう。コードを一行書く前に、根拠のあるスタックの決定を手にして終われます。

次のプロジェクトを作りたいです：[一、二文で説明してください]。
それを vibe coding するのに最適な言語とフレームワークを、生の性能ではなく、
AI アシスタントがどれだけ流暢にコードを書いて直せるかを基準に勧めてください。
教えてほしいのは：(1) 言語と主要なフレームワーク、(2) どこにデプロイするか、
(3) 注意すべき現実的な落とし穴を一つ、(4) 別の言語の方がむしろ適している
シナリオを一つ。簡潔をお願いします。

機能を、Vibe ごとに作っていく



ここまでで、あなたは AI と会話してコードを書かされるようになりました。しかし、本物の機能は一つのプロンプトより大きなものです。出荷する人と、行き詰まる人とを分けるスキルは、機能を小さな断片に分割し、一つずつ AI を通してそれらを進めていく方法を知っていること—そして、各ステップでアプリを動く状態に保つことです。

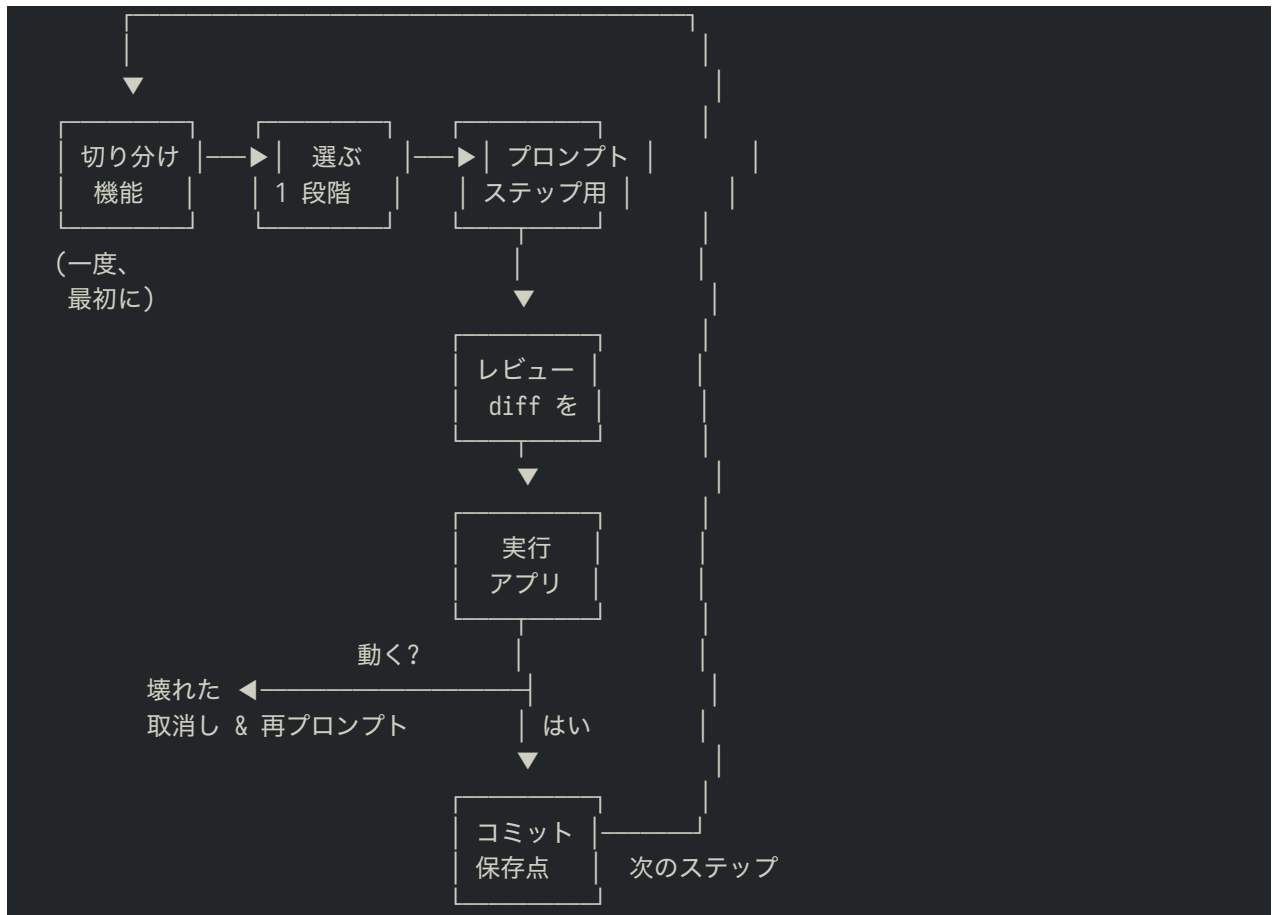
この章は、中核となるビルドループです。一度覚えれば、これから作るすべての機能で再利用できます。これは、着実に育っていくプロジェクトと、面白くなってきたまさにその瞬間に自重で崩れ落ちるプロジェクトとを分ける違いです。

ビルドループ

何度も何度も回すことになるループがこれです。意図的に小さくしてあります。

1. **機能を小さな垂直ステップに切り分ける。**各ステップは、数分で作って、実行して、動くのを見られるものであるべきです。
2. **一つのステップを選ぶ。**ただ一つだけ。今は他のすべてを無視します。
3. **その一つのステップをプロンプトする。**何を変え、何を変えないかを具体的に。
4. **変更をレビューする。**AI が書いたものを読みます。すべての行を理解する必要はありませんが、明らかにおかしいものがないか目を通します。
5. **アプリを実行する。**実際にボタンをクリックし、ページを読み込み、エンドポイントを叩く。そのステップが動くことを確認します。
6. **コミットする。**バージョン管理に、短いメッセージとともにチェックポイントを保存します。
7. **繰り返す。**ステップ 2 に戻り、次のステップへ—機能が完成するまで。

このループをサイクルとして描くと、こうなります—一周ごとに小さなステップを一つ、機能が完成するまで回り続けます:



肝心なのは、あなたが動くアプリから決して遠く離れないということです。ステップ 4 で何かが壊れても、元に戻すのはたった一つの小さな変更であって、絡まり合った混乱ではありません。

ステップ 4 から 6 を飛ばす人は、20 分くらいは速く感じます。それから、見つけれないバグにぶつかります。最後に確実に動いていたバージョンが、100 個の変更の向こうにあるからです。このループが遅く感じられるのは、まさにそういう類の負債を決して溜めさせないからです。

「垂直スライス」とは

垂直スライスとは、一度に一つの層をまるごと作るのではなく、アプリ全体を貫いて通る機能の薄い断片—少しの UI、少しのロジック、少しのデータの—ことです。

魅力的だが間違ったアプローチ (水平):

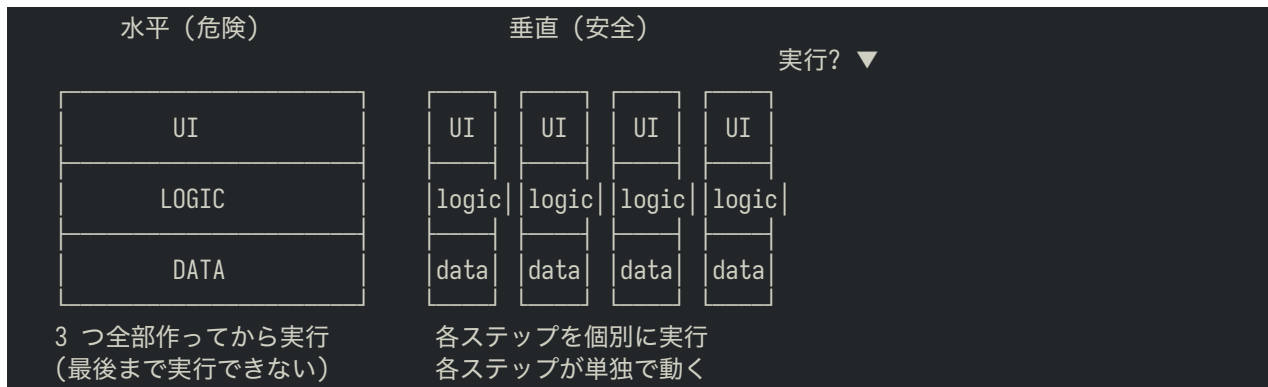
- データベーススキーマを丸ごと作る。
- 次にバックエンドのロジックをすべて作る。
- それから UI を丸ごと作る。

3 つすべてが終わるまで何も実行できないので、問題を早期に捕まえられません。

より良いアプローチ (垂直):

- ステップ 1: そのことの、考えられる限り最も単純なバージョンを、偽のデータでやるボタン。
- ステップ 2: それを本物のデータを使うようにする。
- ステップ 3: エッジケースを処理する。

違いは、並べて見ると一番よく分かります。水平は一度に一つの層をまるごと作り、最後まで何も実行しません。垂直は各ステップで、すべての層を貫く薄い実行可能な帯を作ります:



各スライスは実行可能です。それがアプリを常に動く状態に保つのです。

良いスライスを見極める便利なテスト: **デモできますか?** 画面を指さして「これを見て」と言えるなら、それは本物の垂直スライスです。正直なデモが「まあ、スキーマは整いました」だけなら、あなたは水平な層を作ったのであり、それが正しいかどうかはずっと後—たいていは最悪のタイミング—でしか分かりません。

実例: 「お気に入り」機能

小さな機能を端から端まで作ってみましょう。ユーザーがリスト内の項目をお気に入りとしてマークでき、いくつかお気に入りにしたかの件数を見られる、という機能です。

切り分ける

私ならこう垂直ステップに分割します。

1. 各項目の隣に星ボタンを追加する (まだ何もしない、ただ表示されるだけ)。
2. 星をクリックすると、オン/オフの見た目が切り替わる (状態のみ、保存はしない)。
3. トグルを反映した「☐ 3 favorites」のカウンターを表示する。
4. お気に入りを保存して、ページのリロード後も残るようにする。

各ステップが独立して実行可能で、目に見えることに注目してください。どのステップの後で止めても、動くアプリが手元に残ります。

ステップ 1: ボタン

良いプロンプトは、ファイル名、変更内容、そして触らないものを名指します。

ItemCard コンポーネント (src/components/ItemCard.jsx) で、各カードの右上隅に星ボタンを追加して。輪郭線の星アイコンをレンダリングして。まだクリックの挙動はつなげないで — ただ表示されるだけにして。カードの残りのレイアウトはそのままにして。

アプリを実行します。星が見えますか？ いいですね。コミットします。

```
git add -A && git commit -m "Add star button to item cards"
```

ステップ 2: 状態をトグルする

星ボタンをクリックしたら、塗りつぶしと輪郭線が切り替わるようにして。これをローカルなコンポーネント状態で管理して — まだどこにも保存しないで。塗りつぶされた星は、お気に入りという意味する。

状態 (state) はアプリの短期記憶だと考えてください—いまこの瞬間に何が真か、たとえば特定の星がいま塗りつぶされているか、といったことです。これであなたは**状態と UI を一緒に扱**っていますが、これは普通のことです。(UI、つまりユーザーインターフェースとは、人が見てクリックするアプリの部分のことです。) AI はおそらく次のようなものを追加するでしょう。

```
const [isFavorite, setIsFavorite] = useState(false);

<button onClick={() => setIsFavorite(!isFavorite)}>
  {isFavorite ? <StarFilled /> : <StarOutline />}
</button>
```

星をいくつかクリックしてみます。切り替わりますか？ コミット。

ステップ 3: カウンター

項目リストの先頭に「★ N favorites」のカウンターを追加して。現在星が塗りつぶされている項目がいくつあるかを数えて。カウンターから見えるように必要ならお気に入りの状態をリストコンポーネントへ引き上げてもいい。

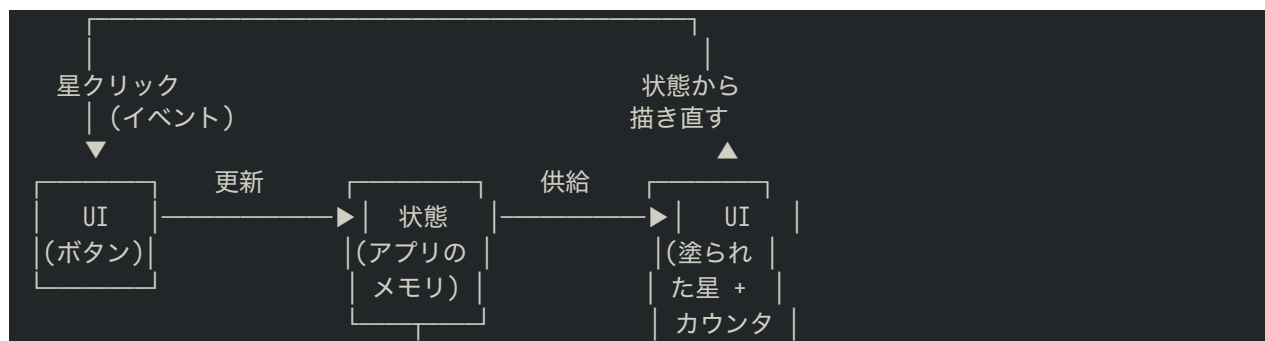
このステップでは、AI に**状態の置き場所を再構成**してもらう必要があるかもしれません—それは想定内で、まさにそれを独立した小さなステップとしてやる理由です。実行して、星をいくつか切り替え、数字が変わるのを見ます。コミット。

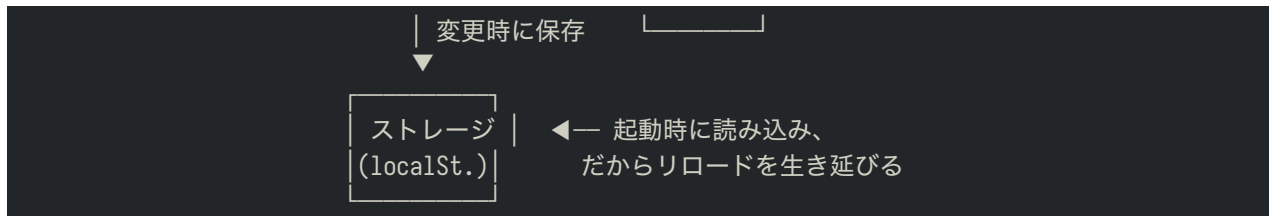
ステップ 4: 永続化する

ページのリロード後もお気に入りが残るように永続化して。今は localStorage を使って — お気に入りの項目 ID のリストが変わるたびに保存し、起動時に読み込んで。

(localStorage とは、ブラウザが訪問者本人のコンピュータに置いておく小さなメモ帳のようなもので、アプリが何かを書き留めておき、リロード後にもまた見つけられるようにするものです。) ページをリロードします。まだお気に入りのままですか？ あなたは今、実行可能なスライスをつずつ重ねて、機能を出荷したのです。

完成した機能の部品どうしがどう会話するかを見てみましょう。クリックは状態へと流れ下り、画面は状態から描き直され、状態はストレージに写し取られてリロードを生き延びます:





すべての行を理解せずに diff を読む

ループのステップ 4 は「変更をレビューする」であり、人々が最もごまかしやすいステップです。有用にレビューするのに、自分でコードを書ける必要はありません。小さな危険信号の一群に目を通せばいいのです。

AI が変更を手渡してきたら、一番すっきり見える方法は diff —追加され、削除された正確な行—です。

```
git diff
```

+ で始まる行は追加されたもの、- で始まる行は削除されたものです。コンパイラではなく、レビュアーのように読みましょう。あなたが探すのは、こういうものです。

- ・ **触る必要のないファイルに触れていないか?** カウンターを頼んだのに、なぜ `auth.js` が diff にあるのか?
- ・ **重要そうなものを削除していないか?** 見覚えのない削除行 (-) は、質問する価値があります。
- ・ **頼んでいない振る舞いをこっそり変えていないか?** 新しいデフォルト値、変わった条件、「親切な」おまけ。
- ・ **形が依頼と合っているか?** 一文の依頼が 200 行を生んだなら、見直す価値があります。

何かおかしければ、自分で直す必要はありません—ただ尋ねましょう。「なぜ `auth.js` を変えたの? 私はカウンターだけ頼んだのに。」半分くらいは、AI が「よく気づきましたね」と言って元に戻します。その習慣一つが、完全には理解していないコードをどれだけ睨みつけるよりも、多くのバグを捕まえます。

AI に自由に走らせるか、手動で操縦するか

すべてのステップが、ハンドルを同じだけ握る必要があるわけではありません。

AI に自由に走らせてよいのは:

- ・ ステップが小さく、明確に定義されているとき (「ボタンを追加して」)。
- ・ AI が明らかに何千回も見てきた、ありふれたパターンであるとき。
- ・ ミスが安価で明白なとき (アプリを実行した瞬間に気づけます)。

手動で操縦する—より小さなプロンプト、より綿密なレビュー—のは:

- ・ 変更が、簡単には復旧できないデータに触れるとき (レコードの削除、マイグレーション)。
- ・ AI が同じところを堂々巡りで「修正」し続けるとき。
- ・ 振る舞いが微妙で、画面を一瞥しただけでは現れないとき (金額の計算、権限、認証)。
- ・ 2 回試して、良くなるどころか悪くなっているとき。

信頼できる経験則: **影響範囲が大きいほど、ステップを小さく。**

アプリを常に動く状態に保つ

これが、他のすべてを可能にする規律です。

- ・ **各ステップの後に実行する。** 3 ステップごとではなく。毎回。
- ・ **動く状態だけをコミットする。** コミットは、戻ってこられるセーブポイントです。あるステップが、一、二分で直せない何かを壊したら、それを捨てて (`git restore` . または `revert`)、より明確な依頼で再プロンプトしましょう—壊れた土台の上に修正を積み上げてはいけません。
- ・ **1 ステップにつき 1 つの関心事。** 「ついでにあれもさっと直そう」と誘惑されても、こらえてください。今のスライスを終え、コミットし、それから新しいステップを始めます。

- ・ **バグを診断せず、記述する。** 何かが壊れたら、見たままを AI に正確に伝えましょう。「2 つ目の星をクリックした後、カウンターが 2 ではなく 1 を表示する。」原因は AI に見つけさせます。

小さな単位でコミットする

小さなコミットは、あなたのセーフティネットです。一つひとつが、首尾一貫した単一のステップであるべきです。

```
git commit -m "Add star button to item cards"
git commit -m "Toggle favorite state on star click"
git commit -m "Add favorites counter to list header"
git commit -m "Persist favorites in localStorage"
```

ステップ 4 が横道にそれでも、最初の 3 ステップは安全で、失うものはほとんどありません。大きく、まれなコミットこそが、人々が自分のコードに触れるのを恐れるようになる原因です。

スライスが大きすぎるとき

ときどき、「一つのステップ」を選んだのに、それでもうまくいかないことがあります—AI がアプリの半分を書き直し、diff は読めず、実行すると新しいバグが一度に 3 つ飛び出してくる。それはまず運の悪さではありません。スライスが大きすぎたという信号です。

直し方は、もっと薄く切ることです。「永続化付きのお気に入りを追加」が爆発したなら、それは一つのスライスではありませんでした。単一のプロンプトを変装にまとめた機能まるごとだったのです。最後の良いコミットまで戻って、分割しましょう。

これを一発で頼まない:

"カウンターとlocalStorage永続化付きのお気に入り機能を追加して。"

代わりにこう、コミットごとに一つずつ頼む:

"各項目カードに星ボタンを追加して。まだ振る舞いはなしで。"

"星がローカル状態で塗りつぶし/輪郭をトグルするようにして。"

"塗りつぶされた星を反映するカウンターを追加して。"

"お気に入りのIDをlocalStorageに永続化し、起動時に読み込んで。"

良い目安: 一つのステップが何をするかを短い一文で説明できないなら、それは二つのステップです。そして、あるステップの diff が 1 分で目を通すには長すぎるなら、次のステップはもっと小さくすべきです。切り分けは、直前のスライスがどこで厚すぎたかを感じ取りながら、だんだん上手くなっていくスキルです。

まとめ

機能は一つの巨大なプロンプトではありません—小さく、実行可能なスライスの積み重ねです。切り分け、一つのステップをプロンプトし、レビューし、実行し、コミットし、繰り返す。簡単なものは AI に走らせ、リスクのあるものの近くではしっかり操縦する。各チェックポイントでアプリを動く状態に保てば、あなたは常に、小さな取り消し一つで堅固な地盤に戻れます。

次の章では、ステップが計画どおりにいかなかったときにどうするか—Vibe によるデバッグを見ていきます。

おさらいと練習

要点

- ・ 機能は一つの巨大なプロンプトではなく、小さな垂直スライスの積み重ねです—一次に進む前に、各スライスを作り、実行し、動くことを自分の目で見て確認しましょう。
- ・ 毎回ビルドループを回しましょう: スライス □ 一つのステップを選ぶ □ プロンプト □ diff をレビュー □ アプリを実行 □ コミット □ 繰り返す。

- diff はコンパイラではなくレビュアーのように読みましょう—触るべきでないファイル、黙って変わった挙動、依頼よりはるかに大きい diff に注意します。
- 小さく、コストが低く、ありふれたステップは AI に走らせましょう。影響範囲が大きいところでは手動で操縦します (より小さなプロンプト、より綿密なレビュー)。
- 動く状態だけを小さな単位でコミットしましょう—スライスがおかしくなったら、壊れた土台の上に修正を積まず、捨てて再プロンプトします。

やってみよう

プロジェクトに入れたい小さな機能を一つ選び、3~5 個の垂直ステップに切り分けて、それぞれを短い一文で書きましょう。最初のステップだけを作ります: それをプロンプトし、git diff を実行して変更をざっと見て、アプリを実行して動作を確認し、一行のメッセージでコミットします。今はそこで止めましょう。目的は、ループが一度きれいに回るのを感じる事です—実行可能なスライス一つ、実際に読んだ diff 一つ、戻れるコミット一つ。

この機能を追加したい: [機能を 1~2 文で説明]。
まだコードは書かないで。まず、これを 3~5 個の垂直ステップに切り分けて。
各ステップは、私が作って、実行して、単独で動くのを見られるものにして。
各ステップの後でもアプリが動き続けるように順序を決めて。各ステップごとに一文の説明をつけ、最初に変更するファイル名を挙げて。それから待って —
一つずつ作るよう私が頼むから。

AI を使ったデバッグ



あなたが作るプログラムは、必ずどこかで壊れます。それはやり方が間違っているサインではなく、ソフトウェアにとってごく当たり前の状態です。バグは技量の失敗ではありません。本物のコードが本物のデータ、本物のユーザー、そして誰も想像しなかった本物のエッジケースと出会ったときに起こることです。イライラする午後と5分で済む修正の違いは、方法にあります。Vibe coding でデバッグするとき、スタックトレースをにらみつけて意味が分かるまで考え込むのはあなたではありません—スタックトレースとは、プログラムがクラッシュしたときに出力するエラー報告のことで、後ほど詳しく解きほぐします。あなたはディレクターです。問題を再現し、適切な証拠を集め、よく整理された手堅い「事案」を AI に渡して、重い作業を任せるのです。この章では、まさにそのための再現可能なプロセスを紹介します—バグが一行のタイプミスであっても、三層の深さにもつれた競合状態 (race condition) であっても通用するプロセスを。(競合状態とはタイミングのバグです。二つの出来事が、あなたの予想しなかった順序で起こること—最後の一席を二人が同時につかみ取るように。)

基本となる心構え

初心者が犯す最大の間違いは、「動かない、直して」とチャットに貼り付けて、あとは祈ることです。AI はあなたの画面も、データも、あなたが何を期待していたのかも見えません。AI が知っているのは、あなたが伝えたことだけです。あなたが何も与えなければ、AI にできる唯一のことをします。推測です—そして、たまたま間違っている自信満々の推測は、答えがないことよりも危険です。あなたがそれに基づいて行動してしまうからです。AI を使ったデバッグとは、ほとんどの場合 **適切なコンテキストを与えること** であり、それによって AI は推測ではなく推論ができるようになります。

良いバグ報告は、相手が人間 であれ AI であれ、次の 3 つの問いに答えています。

- **何をしたいのか?** (それを引き起こす手順)
- **何が起ると期待していたのか?**
- **実際には何が起こったのか?** (エラー、誤った出力、クラッシュ)

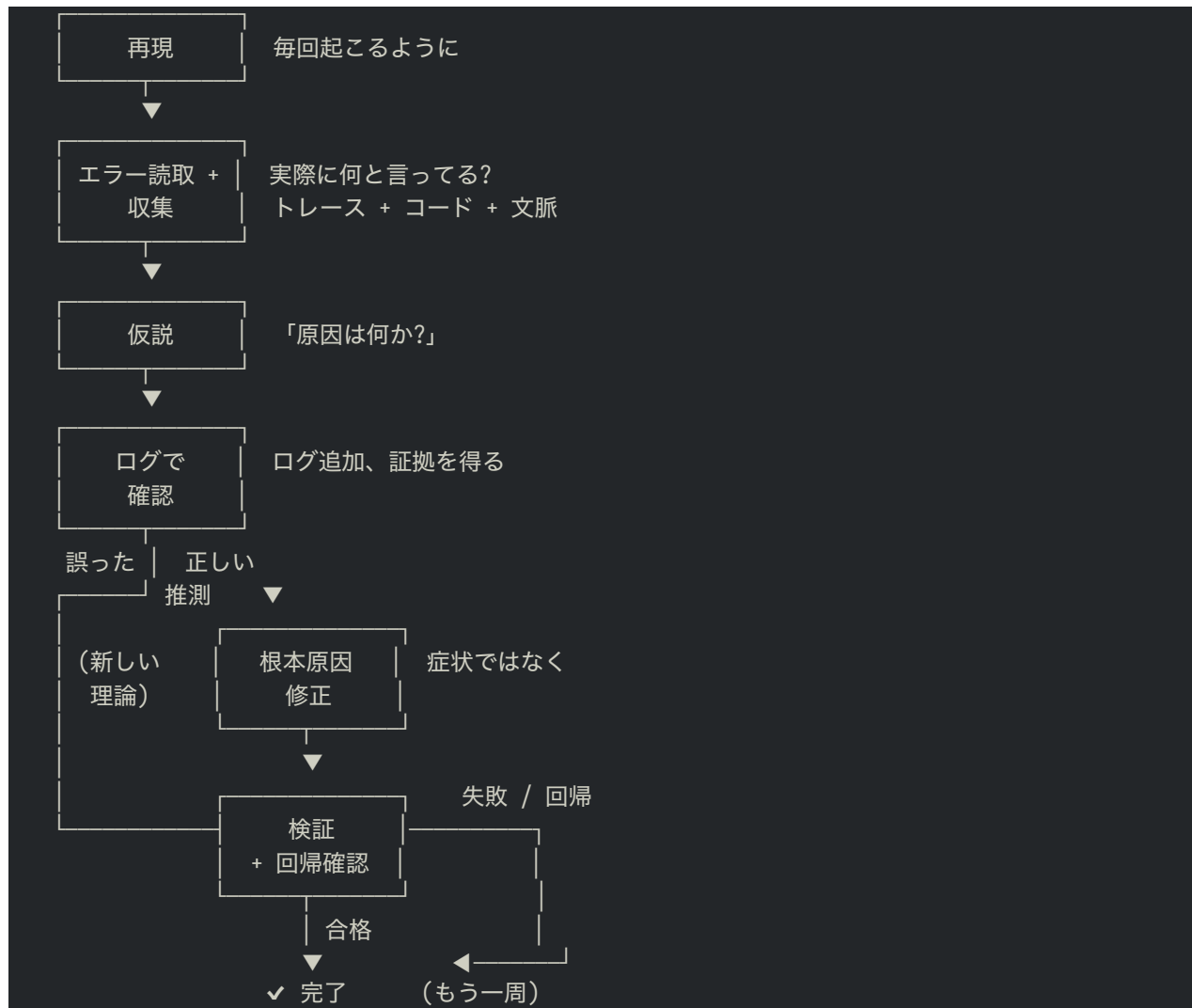
この 3 つに答えられるなら、あなたはすでに大半の人より一歩先を行っています。この 3 つが重要なのは、バグとは定義上、あなたが期待したことと実際に起こったこととの間の ギャップ だからです。AI は、両側が見えないギャップを見つけられません。「壊れている」は片側だけを説明します。「ようこそ画面を期待していたのに、空白のページとこのエラーが出た」は両側を説明します—するとギャップはほとんど自ら名乗り出ます。

デバッグのチェックリスト

これらを順番に進めてください。いきなり「直して」に飛びつくと、AI は堂々巡りに陥ります。

1. **確実に再現する。**バグが毎回起こる正確な手順を見つけます。再現できないバグは、直ったことを確認できないバグです—消えたかどうか永遠に推測するしかなくなります。断続的に起こるなら、それを切り替える変数を狩りましょう。特定の入力、ログイン状態かログアウト状態か、遅いネットワークなど。
2. **エラーそのものを読む。**流し読みしないこと。エラーメッセージとスタックトレースの 先頭 は、たいてい問題が起きたファイルと行を示しています。初心者は、画面上で最も有用な一文を、怖そうに見えるという理由で読み飛ばしてしまいがちです。
3. **コンテキストを集める。**エラーのテキスト、スタックトレース、関連するコード、そして期待していた挙動の一行説明を集めます。
4. **AI に全体像を渡す。**上記すべてを 1 つのメッセージに貼り付けます。AI に仮説を立てさせましょう。
5. **修正ではなく仮説を立てる。**「直して」より先に「これの原因は何か?」と尋ねます。間違った理論に基づく間違った修正は時間の無駄です—そして時には、最初のバグの上に 2 つ目の バグを積み上げます。
6. **ログを追加して仮説を検証する。**コードを変える前に、コードが実際に何をしているのかを確認します。
7. **問題を二分探索する。**怪しい範囲を繰り返し半分に分けて、どこで 壊れるのかを絞り込みます。
8. **症状ではなく根本原因を直す。**修正が、目に見える不具合だけでなく なぜ 壊れたのかに対処していることを確認します。
9. **修正を検証し、リグレッションをチェックする。**再現手順をもう一度実行します。それから、近くの機能がまだ動くことを確認します—リグレッションとは、修正がそれまで動いていた何かをこっそり壊してしまうことです。ある漏れをふさいだら、すぐ隣のパイプが破裂するように。

そのチェックリストは、実はバグが消えるまで回り続ける 1 つのループです。各ボックスが次に証拠を手渡し、検証に失敗すれば、学んだことを携えてもう一周します:



順番は恣意的ではありません。各ステップは、次のステップが必要とする証拠を生み出します。再現は読むものを与え、読むことは集めるコンテキストを与え、コンテキストは AI に仮説を立てる材料を与えます。ステップを飛ばせば、次のステップに穴を食わせることになります。

AI に適切なコンテキストを与える

ここがあなたの効きどころの大半です。バグを報告するときは、次のものを含めてください。

- ・ **エラーメッセージを一字一句そのまま**。正確にコピーすること。言い換えてはいけません。エラーを言い換えるのは、電話で写真を説明するようなものです。肝心だった唯一のディテールを取りこぼします。
- ・ **スタックトレース**、特に先頭の数行。
- ・ **関連するコード**—エラーを投げた関数と、それが呼び出すものすべて。
- ・ **期待していたことと、実際に起こったこと**。

強力なデバッグプロンプトの実例は、こんな感じです。

サインアップフォームを送信するとエラーが出ます。出力全体は次のとおりです：

```

TypeError: Cannot read properties of undefined (reading 'email')
    at validateUser (src/auth/validate.js:14:23)
  
```

```
at handleSignup (src/routes/signup.js:31:10)
at processTicksAndRejections (node:internal/process/task_queues:95:5)
```

再現手順: フォームに入力し、「Sign up」をクリックする。
期待: 「Welcome」メッセージ。
実際: ページが空白になり、上のエラーがコンソールに出る。

これが validate.js です (14 行目あたり):
[関数を貼り付け]

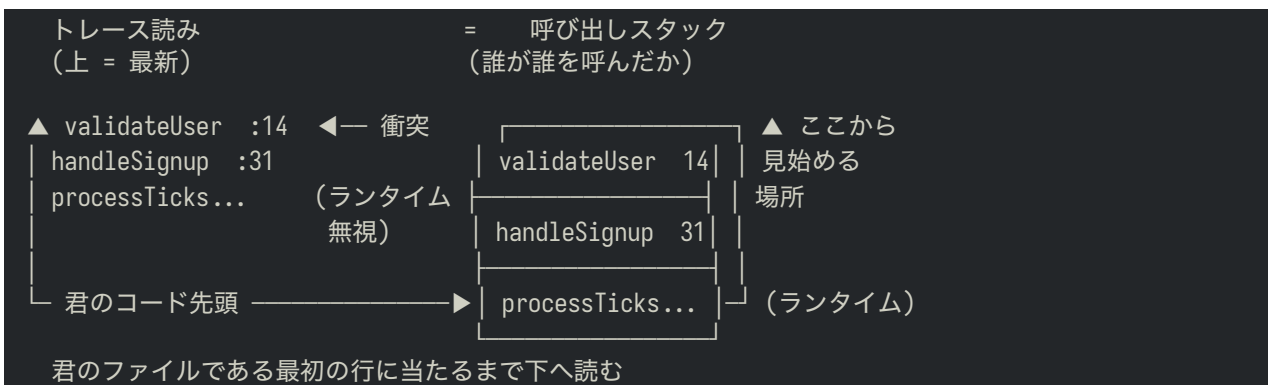
そしてこれが、それを呼び出す handleSignup 関数です:
[関数を貼り付け]

何が原因として考えられ、何かを変える前にどう確認すればいいですか?

最後の一行に注目してください。あなたは即座の編集ではなく、仮説と確認のステップを求めています。これが AI を誠実に保ちます。

貼り付けるトレースを読む方法を知っておくと役立ちます。スタックトレースは新しい順に並んだ呼び出し履歴です。先頭の行はプログラムが実際に爆発した場所で、その下の各行はその上の関数を呼び出した関数です。だから例では、validateUser は email が undefined な何かから読み取られた場所で、handleSignup は validateUser を呼び出した側です。あなたのコードより下のフレーム—processTicksAndRejections のような—は言語ランタイムのもので、たいてい無視してかまいません。あなたのファイルを指す最初の行に当たるまで上から下へ読みます。そこがほぼ常に見始めるべき場所です。

同じトレースを呼び出しのスタックとして読むと、順序が一目瞭然になります。クラッシュが一番上にあり、その下の各フレームは、すぐ上のフレームを呼び出した側です:



分からないエラーを読む

遅かれ早かれ、アルファベットのアルファベットのようなエラーに出くわします—長い型名、フレームワーク内部だけのスタック、見たこともない専門用語のメッセージ。それで足を止めないでください。エラーはデータであって、あなたの能力に対する判決ではありません。メッセージ自体が不透明なときは、それを貼り付けて AI に直す前に翻訳してと頼みましょう。

このエラーが分からない。どんな修正も提案する前に、平易な言葉で説明して:
これが実際に何を言っているのか、どの行を指しているのか、
まさにこのエラーをふつう何が引き起こすのか。

[エラーとスタックトレースの全体を貼り付け]

平易な言葉での意味を先に尋ねることは、2 つのことをします。それはあなたにパターンを教え、次にそのエラーを見たとき自分で見分けられるようにします—そして数か月のうちに、これがあなたが実際に自分のスタック

を読めるようになる方法になります。さらにそれは、AI が汎用的なパッチに飛びつく代わりに、エラーが 言っていること に修正を根ざさせるよう強います。最もよくあるエラーの罠に注意してください。14 行目を指すエラーは、しばしばまったく別の場所で 引き起こされた ものです。トレースの行はプログラムが問題に気づいた場所であって、必ずしも問題が始まった場所ではありません。

仮説、ログ、二分探索

AI が原因を提示したら、ただ受け入れてはいけません。 **安く検証しましょう**。最も安いテストは、たいていログ行です。

AI に一時的なログを追加させましょう。

「何も変える前に、14 行目の直前で user の値を表示する console.log (または print) を追加して、本当に undefined なのか確認できるようにして。」

実行して、その出力を貼り戻します。これで理論ではなく 証拠 が手に入りました。user が undefined かもしれません—仮説が確認できます。あるいは問題なく、本当の犯人は一段上にあるのかもしれません。どちらにせよ、あなたは前進しています。調査中はログを惜しまず仕込みましょう。変数の値、そしてそれがどの変数で、ログがどこにあるのかを示すラベルまで。undefined を出す素の console.log(user) は、console.log("user at validate.js:14 =", user) よりはるかに役に立ちません。そしてバグが直ったら一時的なログを抜き取るのを忘れずに—残されたデバッグノイズは、それ自体が小さな散らかりです。

長いプロセスのどこかにバグが落ちていて、どこか分からないときは、 **二分探索** を使います。怪しい領域を半分に切ります。中間点にログを追加します。

考え方はこうです。各チェックが残りの半分の捨てるので、巨大な探索空間が一気に縮みます。ここではバグがステップ 1 とステップ 8 のあいだのどこかに落ちていて—3 回のチェックで追い詰めます：

```
段階:  1  2  3  4  5  6  7  8      バグはこの中のどこか
        ///////////////////////////////////////////////////

確認 1 — 4 にログ — OK —▶   バグは 4 の後、1-4 捨て
                                /////////////// (5 6 7 8 残り)

確認 2 — 6 にログ — BAD —▶   バグは 6 以下、7-8 捨て
                                /////////////// (5 6 残り)

確認 3 — 5 にログ — OK —▶   バグは段階 6   ✓ 発見

8 段階 → 3 回確認。  1000 段階 → 約 10 回確認。
```

そこでは値は正しく見えましたか？ ならバグはその 後ろ にあります—その半分を探します。すでにおかしく見えましたか？ バグは 前 にあります—その半分を探します。各ステップで探索空間が半分になるので、1000 行の経路でも約 10 回のチェックで絞り込めます。AI にはっきりこう伝えましょう。「これを二分探索したい—ログを仕込むべき中間点はどこ？」同じ技法はコードだけでなく履歴にも効きます。ある機能が先週は動いたのに今日壊れたなら、コミットを二分探索して、バグを持ち込んだ正確な変更を見つけましょう。

根本原因 対 症状

デバッグで最も誘惑的な罠は、原因を直さずにエラーを 消す パッチです。user が undefined なのを if (user) { ... } で「修正」すると、エラーは止まります—しかし今度はサインアップが黙って何もしなくなり、これはもっと悪い状態です。けたたましいクラッシュは少なくとも何かおかしいと教えてくれていたのに、今度は何も教えてくれません。本当の問いは なぜここで user は undefined なのか？ です。データベースの参照が失敗したのかもしれません。フィールド名が変わったのかもしれません。上流で await が抜けているのかもしれません。

AI が修正を提示したら、こう尋ねましょう。

- 「これは原因を直しているの、それとも症状を隠しているの？」

- ・「値が間違っていた 元々の 理由は何だったの?」
- ・「この同じ根本原因が、他に何か壊す可能性はある?」

根本原因の修正は、たいていコードを よりシンプル または より明快 にします。症状のパッチは、たいていガード節や、エラーを握りつづす try/catch、あるいは本当の問題を覆い隠すデフォルト値を追加します。後者の類いには疑いの目を向けてください。覚えておく価値のある兆候があります。ある修正が、正しいことをするのではなく クラッシュを防ぐ ことだけを役目とするコードを足すなら、あなたはおそらく症状をパッチしています。本当の修正は、不可能な状態をただ耐え抜くのではなく、まるごと取り除く傾向があります。

AI が堂々巡りになったとき

ときに AI が修正を提案し、それが効かず、別のを提案し、それも効かず、気づけば「代わりにこれを試して」のループに 3 周はまわっている、ということがあります。これは行き止まりではなく、シグナルです。こうしてください。

- ・ **立ち止まってコンテキストをリセットする。**これまで試したことすべてではなく、今 確実に分かっていること (ログで確認済みのこと) をまとめた新しいメッセージから始めます。失敗した試みで埋まった長いチャットは、AI の次の答えを積極的に汚染します—AI は自分の履歴の中の行き止まりにパターンマッチし続けます。
- ・ **除外したことを述べる。**「user が undefined で、DB クエリが null を返すことを確認した。クエリ文字列は正しそう。だから問題はクエリより上流にある。」これで AI が行き止まりを再提案するのを止められます。
- ・ **修正ではなく説明を求める。**「リクエストから 14 行目まで、user がどうやって値を得るのか、ステップごとに順を追って説明して。」説明を強いると、しばしば抜け穴が浮かび上がります。
- ・ **推測ではなく、もっとログを。**理論が尽きたら、証拠が前進の道です。
- ・ **問題を縮める。**それでもバグを再現できる最小のスニペットを、AI に手伝ってもらって作ります。ノイズが消えるとバグが一目瞭然になることが多く—もし小さい版で再現しないなら、原因は取り除いた部分にあると分かります。

残す前に修正を理解する

動くけれどもあなたが理解していない修正は、解決ではなく借金です—そして近くの何かが次に壊れたとき、利子の支払いがやってきます。AI がタイピングをするので、緑のチェックマークを受け入れて、なぜ コードが今緑なのかを一度も知らないまま先に進むのは簡単です。それに抗ってください。修正を残す前に、AI にそれを釈明させましょう。

- ・ **「なぜこれで直るのか、一、二文で説明して。」**説明が曖昧なら—「これでより堅牢になる」「これでエッジケースに対処できる」—押し返しましょう。本当の説明は、具体的な原因と、変更がそれをどう取り除くのかを名指しします。
- ・ **「そのバグは、原因と結果で言うと何だったの?」**あなたはその答えを自分の言葉で言い返せるべきです。できないなら、あなたはまだ自分のプログラムを理解していません。
- ・ **「このコードパスに触れる他に、何を再テストすべき?」**これは修正をリグレッションチェックに変えます。

これは、自分が出荷するものを理解せよという哲学を、その最も難しいケースに当てはめたものです。修正をゼロから 書ける 必要はありません—それは AI の仕事です。しかし、原因と結果の物語は頭の中に握っておく必要があります。それが再び壊れる午前 2 時にそこにいるのはあなたであり、そのとき唯一助けになるのは、その物が実際にどう動くのかを知っていることだからです。

まとめ

AI を使ったデバッグは、答えを知っていることではありません。規律あるプロセスを回すことです。再現し、コンテキストを集め、仮説を立て、ログで確認し、二分探索し、根本原因を直す—そしてその修正を、誠実に残せるだけ十分に理解する。ディレクターとしてのあなたの仕事は、調査を証拠に根ざしたものに保ち、ループが停滞したときにそれを見抜くことです。それができれば、ほとんどのバグは謎であることをやめ、本物のソフトウェアを作る上でのもう 1 つのステップになります。

おさらいと練習

要点

- バグとは、期待したことと実際に起きたことのあいだのギャップです—AI には両側と正確な再現手順を渡しましょう。決して「動かない」だけで済ませないこと。
- エラーは一字一句そのまま貼り付け、スタックトレースの先頭も一緒に渡します。あなたのファイルを指す最初の行が、ほぼ常に出発点です。
- 変える前に確認しましょう: 仮説を検証するラベル付きのログを 1 行追加し、(コードまたはコミット履歴で) 二分探索して各ステップで探索範囲を半分に減らします。
- 症状ではなく根本原因を直しましょう—クラッシュを止めるだけのガードや try/catch は、たいてい本当の問題を覆い隠します。
- AI が堂々巡りになったら、確実に分かっていることへコンテキストをリセットし、直すのではなく説明するよう求め、推測ではなく証拠を足しましょう。

やってみよう

次に何か壊れたら、修正を頼む前にバグレポートを書きましょう。新しいチャットで AI に 3 つを渡します: 再現手順、期待したこと、そして実際に起きたこと—エラーは一字一句コピーし、スタックトレースの先頭フレームを含めて。それから「直して」ではなく、仮説とそれを安く確認する方法 (たいていはログ 1 行) を求めます。その確認を実行して出力を貼り戻し、そのうえで初めて AI に修正を提案させましょう。「動かない、助けて」よりどれだけ道がまっすぐになるか感じてみてください。

バグがあります。どんな修正も提案する前に、仮説とそれを安く確認する方法を (ログ 1 行のような) ください —まだコードは編集しないで。

再現手順: [何をしたか]

期待: [何が起こるべきだったか]

実際: [何が起こったか]

エラーとスタックトレースの全体:

[一字一句そのまま貼り付け]

関連するコード:

[失敗した関数と、それが呼び出すものすべてを貼り付け]

何が原因として考えられ、何かを変える前に確認するには何をログに出せばいい?

サーバーインフラ



Vibe coding で、ノート PC 上では動くものを作りました。今度は他の人に使ってもらいたい。その隔たり—「自分のマシンでは動く」と「インターネット上で動く」のあいだのギャップ—こそがインフラの話です。良い知らせ: それを越えるのに DevOps のキャリアは要りません。必要なのは、機能する頭の中のモデルと、面倒な設定を AI に任せる意思です。

この章では、ホスティングの世界をわかりやすく解き明かし、トレードオフを平易な言葉で示し、個人開発者をプロトタイプから本番まで、選択肢に溺れることなく導くデフォルトの道筋を渡します。

コードが住める 5 つの場所

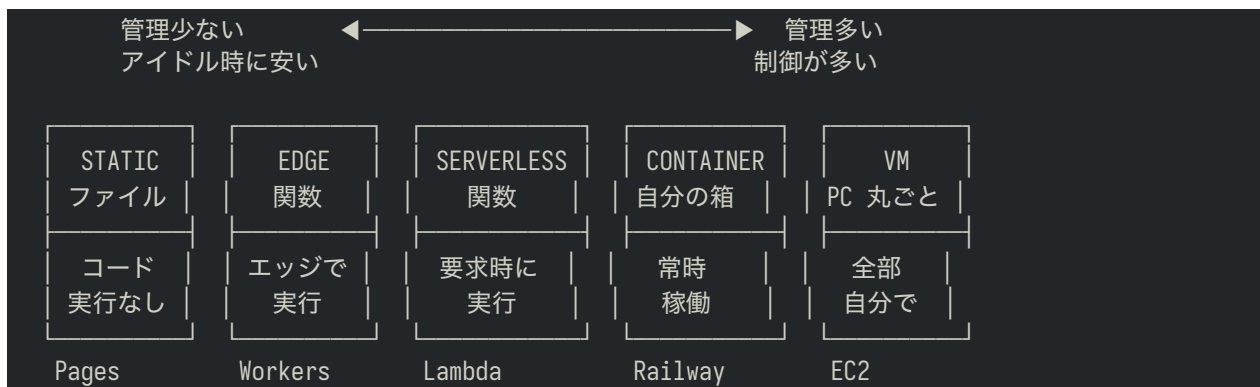
地球上のほぼすべてのホスティング製品は、この 5 つのカテゴリーのいずれかのバリエーションです。これらを覚えれば、マーケティングページが意味不明でなくなります。

- **静的ホスティング**—プレーンなファイル (HTML (HyperText Markup Language、ページの骨組み)、CSS (Cascading Style Sheets、見た目のスタイル)、JavaScript、画像) をブラウザにそのまま配信します。サーバーがあなたのコードを「実行」するのではなく、ただファイルを渡すだけです。インターネット上のフォルダだと考えてください。クラッシュするものも、パッチを当てるものもないので、これはインターネットに置けるもっとも信頼性の高いものです。例: GitHub Pages、Cloudflare Pages、Netlify、Vercel (静的モード)。
- **エッジ関数**—ユーザーに物理的に近いサーバー上で、世界中で実行される小さなコード片です。ほぼ瞬時に起動し、軽量なロジックに最適です。認証チェック、リダイレクト、小さな API (Application Programming Interface —あるプログラムが別のプログラムと話すための窓口) 呼び出しなど。落とし穴: 機能を絞った

環境で動くので、すべてのライブラリがそこで動くわけではありません。例: Cloudflare Workers、Vercel Edge Functions、Deno Deploy。

- **サーバーレス関数**—必要に応じて実行され、アイドル時には消えるコードです。課金は時間単位ではなくリクエスト単位です。あなたは関数を書き、プラットフォームがマシンを扱います。1 回のリクエストが数分走る必要が出てくるまでは最高です—たいていのプラットフォームはタイムアウト (多くは 10~60 秒) 後に関数を打ち切ります。例: AWS Lambda、Google Cloud Functions、Vercel Functions。
- **コンテナ**—あなたのアプリと、その環境一式 (適切な言語バージョン、ライブラリ、システムツール) を、コンテナと呼ばれる持ち運び可能な箱にまとめたものです。プラットフォームがその箱を動かし続けます。より多くの制御、わずかに多めのセットアップ。その箱を定義するただ 1 つのファイルが Dockerfile で、AI はそれをうまく書きます。例: Fly.io、Railway、Render、Google Cloud Run。
- **仮想マシン (VM)**—データセンターで借りる丸ごと 1 台のコンピューターです。すべて—OS、セキュリティ更新、ウェブサーバー、その全部—を自分でインストールし、管理します。最大の制御、最大の責任。例: AWS EC2、DigitalOcean Droplets、Hetzner。

5 つのカテゴリーを並べて見ると、下に行くほど 管理する量の少なさを 制御の多さ と引き換えにします:



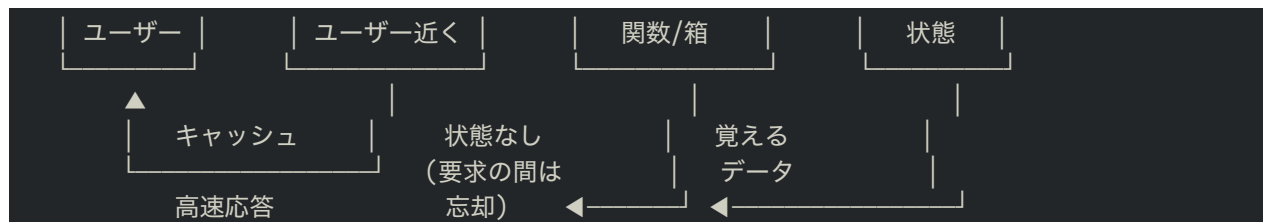
コストについてのおおまかな目安: プラットフォームがあなたのために管理する量が多いほど、誰もアプリを使っていないときは安くなり、みんなが使っているときの心配を多く肩代わりしてくれます。

いくつかの専門用語の解説

- **コールドスタート**—コードがしばらく実行されていないと、プラットフォームはそれを起こさなければなりません。その最初のリクエストは遅くなります (コンマ数秒から数秒)。エッジ関数ではほとんどありません。サーバーレス関数では時々あります。常時稼働のコンテナと VM にはありません。たいていのアプリでは目に見えませんが、キビキビしているべき決済フローでは効いてくることがあります。
- **スケーリング**—より多くのユーザーをさばいても倒れないこと。自動 スケーリングは、プラットフォームが代わりに容量を追加してくれます。手動 スケーリングは、あなたがボタンをクリックするか、さもなければ負荷で単に壊れるかです。「ゼロまでスケール (scales to zero)」もよく見かける関連表現です: 誰もアプリを使っていないとき、コストはゼロという意味です。
- **状態 (State)**—残しておく必要のあるデータ (データベース、アップロードされたファイル、ユーザーセッション)。静的層とサーバーレス層はしばしば「ステートレス」—リクエスト間ですべてを忘れる—なので、状態は別のデータベースやストレージサービスに住みます。ほぼすべての本物のアプリが 1 つの魔法の箱ではなく「ホスティング + どこかのデータベース」である理由がこれです。
- **リージョン (Region)**—あなたのコードとデータが物理的に置かれている世界の場所。速度 (近いほど速い) に、ときには法律 (一部のデータは特定の国に留めなければならない) に影響します。最初のプロジェクトなら、ユーザーにいちばん近いリージョンを選んで先に進みましょう。

リクエストがたどる旅の全体と、それぞれの部品がどこに住むかはこうです。ブラウザは近くのエッジと話し、エッジはキャッシュされたファイルを返すかあなたのコードを起こし、コードは覚えておくべきことがあればデータベースに手を伸ばします:





トレードオフ表

ホスティングタイプ	コスト (アイドル □ 多忙)	スケーリング	コールドスタート	複雑さ	最適な用途
静的	無料 □ 非常に安い	自動、手間いらず	なし	最低	マーケティングサイト、ドキュメント、ブログ、フロントエンド
エッジ関数	無料枠 □ 安い	自動、グローバル	無視できる程度	低	認証、リダイレクト、軽い API、パーソナライズ
サーバーレス	アイドル時 \$0 □ 利用に応じてスケール	自動	時々気になる	低～中	API、Webhook、急増/予測不能なトラフィック
コンテナ	稼働中は課金	ほぼ自動、設定可能	なし (ウォームに保てば)	中	本格的なバックエンド、長時間ジョブ、カスタムランタイム
VM	24 時間 365 日 課金	手動	なし	最高	特殊ハードウェア、レガシーソフトウェア、完全な制御

パターン: 表を下に進むほど、シンプルさと低いアイドルコストを 制御と柔軟性 と引き換えにします。たいいてい の vibe coder は、上の方に踏みとどまるよう努めるべきです。

どれをいつ選ぶか

- **ただのウェブサイトかフロントエンド?** 静的ホスティング。以上。無料で、速く、ほぼ壊しようがありません。
- **ちょっとしたバックエンドロジックが必要—お問い合わせフォーム、秘密にしたい API キー、認証チェック?** エッジ関数かサーバーレス関数。気前の良い無料枠に留まり、誰も使っていないときはゼロまでスケールダウンします。
- **データベース、バックグラウンドジョブ、または長時間稼働するサーバーを前提とするフレームワークを持つ本物のアプリ?** Railway、Render、Fly.io のようなコンテナプラットフォーム。これがたいいていの「実際の製品」プロジェクトのスイートスポットです。
- **特定の GPU、特別なシステムソフトウェア、または完全な制御が必要?** VM。上のどれかが本当に役目を果たせないときだけここに来ましょう。なぜなら、もう あなた がシステム管理者だからです。

役に立つルール: **役目を果たせる最もシンプルな選択肢を選び、本物の壁にぶつかったときだけ一段下に降りる**—想像上の壁ではなく。

「本物の壁」とは実際どんなものなのでしょうか。正直な例をいくつか。ビルドが動き続けるサーバーを生み出す (ライブ接続を保持するチャットのバックエンド) ので、静的では足りない。あるいは 1 つのリクエストが本物の仕事をする (動画のリサイズ、PDF 生成) ために関数がタイムアウトに当たり続けるので、コンテナに進級する。あ

るいは無料枠のリクエスト上限が本物のトラフィックで実際に使い切られる。これらの共通点に注目してください。どれも当てずっぽうではなく測定です。「いつか大きくなるかも」は壁ではありません。

AI がどうデプロイと設定を組み上げるか

ここが vibe coding の真骨頂です。デプロイ設定は、まさに AI が得意とする、正確で、よく文書化された、定型コードの多い作業です。あなたが指示し、AI が書きます。

効果的なプロンプトはこんな響きです。

- 「この Node アプリ用の Dockerfile と、それをデプロイする Railway の設定を追加して。」
- 「main へのプッシュごとに dist/ フォルダを Cloudflare Pages にデプロイする GitHub Actions のワークフローを書いて。」
- 「サーバーレス関数でコールドスタートのタイムアウトが出ている—何がおかしそうで、設定をどう直せばいい?」

AI があなたのために生成するファイルの一例です—コードをプッシュするたびに静的サイトを自動で出荷するデプロイワークフローです。

```
# .github/workflows/deploy.yml
name: Deploy
on:
  push:
    branches: [main]
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
      - run: npm ci && npm run build
      - name: Publish to Cloudflare Pages
        uses: cloudflare/wrangler-action@v3
        with:
          apiToken: ${ secrets.CF_API_TOKEN }
          command: pages deploy dist --project-name=my-app
```

この構文を暗記する必要はありません。必要なのは、これが何をするのかを認識することです。main へのプッシュごとに、コードをチェックアウトし、ビルドし、結果を公開します。何かが壊れたら、エラーを AI に貼り戻し、設定を直すよう頼みます。

多くのビルダーは、最初は GitHub Actions のファイルをまるごと飛ばして、自分のターミナルからコマンド 1 つでデプロイします。初日に何かをライブで見るには、それがいちばん速い道であることがよくあります。同じ Cloudflare Pages のデプロイを、コマンドラインからやるとこう見えます。

```
# install the CLI once
npm install -g wrangler

# build your site, then push the output folder live
npm run build
wrangler pages deploy dist --project-name=my-app
```

最初の実行ではブラウザ経由でのログインを求められ、その後はコマンド 1 つで済みます。トレードオフはシンプルです。ターミナルのコマンドは「今すぐ出荷」に最適で、GitHub Actions のファイルは「ずっと自動で出

荷」に最適です。たいていのプロジェクトは前者で始め、デプロイが日常になったら後者を足します。自分がどの段階にいるかに合うものを AI に頼みましょう。

本当の苦労を省くディレクションのコツをいくつか。

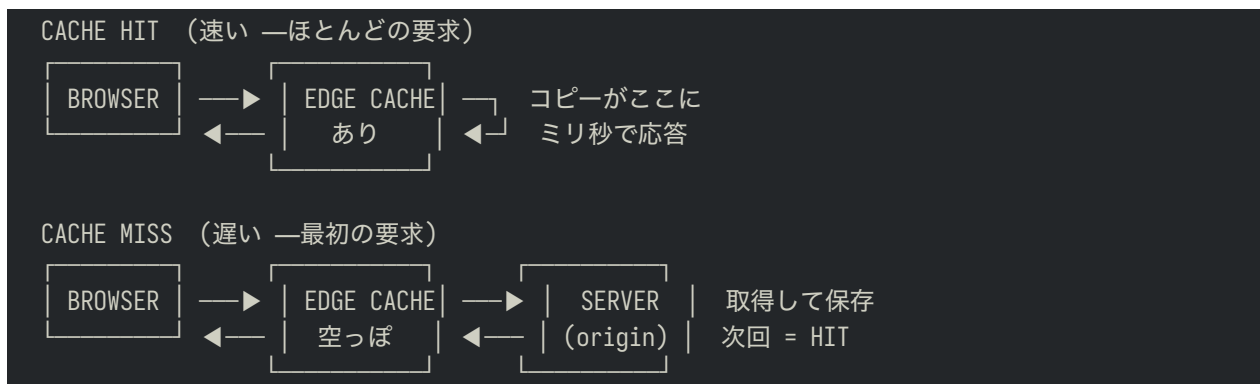
- **AI に、行動する前に説明させる。**「設定を書く前に、このプロジェクトにどのホスティングタイプが合うか、そしてなぜかを教えて。」これで過剰設計を早めに捕まえられます。
- **シークレットをコードの外に保つ。** API キーやパスワードはプラットフォームの環境変数 / シークレットストアに入れ、コミットするファイルには決して入れません。上の CF_API_TOKEN のように、AI にシークレット参照を使わせましょう。
- **早くデプロイ、頻繁にデプロイ。** 初日に「hello world」をライブにします。最初から動いているパイプラインは、ローンチ時に組み立てたものよりずっとデバッグしやすいです。
- **ロールバックの段取りを先に尋ねる。**「このデプロイが壊れたら、動いていたバージョンにどう戻せばいい？」良いプラットフォームは、以前のデプロイをクリック 1 つの距離に置いてくれます。元に戻すボタンがあることを必要になる前に知っておけば、午前 2 時のパニックが肩をすくめるだけで済みます。

個人開発者におすすめのデフォルトの道筋

数えきれないほどの一人プロジェクトで機能してきた道筋です。各ステップは、本当に必要になったときだけ踏みます。

1. **プロトタイプ: 静的 + マネージドなバックエンドサービス。** フロントエンドを静的ホスティング (Cloudflare Pages、Vercel、または Netlify) に置きます。データと認証には、Supabase や Firebase のようなマネージドバックエンドに頼り、自分でデータベースを運用するのを省きます。コスト: 実質ゼロ。
2. **初期の本番: エッジ関数かサーバーレス関数を追加する。** サーバーサイドのロジックが必要になったら—秘密の API 呼び出し、Webhook、カスタム認証—同じプラットフォームに関数を追加します。依然としてゼロまでスケールし、依然として安く、依然として 1 回のデプロイです。
3. **本物の製品: バックエンドをコンテナプラットフォームに移す。** アプリが関数では収まらなくなったら (長時間稼働するプロセス、永続的なサーバーを求めるフレームワーク、バックグラウンドジョブ)、コンテナ化して Railway、Render、または Fly.io にデプロイします。Dockerfile とプラットフォーム設定は AI に書かせましょう。
4. **スケールまたは特化: そのときだけ VM や AWS のようなクラウドプロバイダーを検討する。** ここで複雑さ (とあなたの請求額) が跳ね上がります。たいていの個人開発者はこのステップを必要とすることはなく、それは正しくやっている証であって—限界ではありません。

CDN (Content Delivery Network) はこの経路の前に座り、すでにコピーを持っているかどうかで速度を劇的に変えます。キャッシュ HIT はエッジから即座に答え、キャッシュ MISS はまずあなたのサーバーまで往復しなければなりません:



避けるべき間違いは、「本物のエンジニア」が使うとされているからという理由でステップ 4 から始めることです。真実は逆です。シンプルに始めるのがシニアの一手です。いつでもより多くの制御へと進級できますが、必要のなかったインフラと格闘して失う数か月は、簡単には取り戻せません。

「マネージド」が本当に意味すること (そしてあなたが依然として持つもの)

マネージドサービスとは、退屈で危険な部分—サーバー、セキュリティパッチ、スケーリング、バックアップ—をプラットフォームが代わりにこなしてくれるものです。「サーバーレス」「マネージドデータベース」「サービスとしてのプラットフォーム (PaaS)」は、どれも同じ取引の別の味です。あなたは運用を手放し、作り続けます。個人開発者にとって、これは怠惰ではなくテコです。Linux のセキュリティ更新を当てずに済んだ時間は、実際の製品に費やせる時間です。

しかし「マネージド」は「魔法」ではありません。プラットフォームがどれだけこなしても、いくつかはあなたの責任のまま残ります。

- **あなたのデータ。**プラットフォームはデータベースを動かし続けてくれますが、実際に復元をテストしたバックアップを持っているかは、あなた次第です。「エクスポート」ボタンを一度押して、そのファイルが空でないことを確かめましょう。
- **あなたのシークレット。**API キーはプラットフォームのシークレットストアに住みますが、漏れたものをローテーションすること、そして公開リポジトリやスクリーンショットに決して貼らないことは、あなたの仕事です。
- **あなたの請求。**オートスケーリングは、あなたの請求額をオートスケールするまでは素晴らしいものです。支出アラートと上限を設定しましょう—一次の節はまるごとこの話です。いちばん痛い失敗だからです。
- **あなたの理解。**真夜中に何かが壊れたとき、「プラットフォームが処理する」は途中までしか効きません。専門家である必要はありませんが、ログを読み、失敗を AI に明確に説明できるべきです。それが「自分が出荷するものを理解する」という哲学を一文に詰めたものです。

健全な心構え: 運用 は外注しても、理解 は外注しないこと。プラットフォームにマシンを動かさせ、あなたはそのマシンが何をしているかを知る人であり続けましょう。

出荷前のクイックチェックリスト

人々にライブ URL (Uniform Resource Locator —ウェブアドレス) を訪れるよう告げる前に、このリストを通しましょう。どれも DevOps の専門知識は要りません—ほとんどは AI へのプロンプト 1 つか、ダッシュボードでのクリック 1 つです。

- **実際に読み込まれる。**ローカルのプレビューではなく、本物の URL を新しいブラウザ (またはスマホ) で開きましょう。「自分のマシンでは動く」ローンチが公の場で 404 を出す事例は、伝説的なほど多いです。
- **シークレットがコードに入っていない。**プッシュする前に、自分のキーをリポジトリで検索しましょう。AI に尋ねましょう: 「このプロジェクトでハードコードされた API キー、パスワード、トークンをスキャンして。」
- **支出上限が設定されている。**すべての有料 API にハードな上限、そしてホスティングアカウントに請求アラート。まず上限、お祝いは後で。
- **HTTPS がオンになっている。**HTTPS (HyperText Transfer Protocol Secure) はウェブの基本的な配送方式を暗号化したもので、アドレスバーに鍵マークを出してくれるものです。現代のプラットフォームはこれを自動でやりますが、小さな鍵アイコンがそこにあることを確かめましょう。ブラウザは暗号化されていないサイトに罰を与えます。
- **ロールバックの仕方を知っている。**何も燃えていない今のうちに、以前のデプロイのボタンを見つけておきましょう。
- **あなたに届く手段がある。**基本的なものでもいいので、ロギングやエラー報告のツールを置いて、問題の最初の兆しが怒ったユーザーにならないように。あなたのスタックにエラーロギングを組み込むよう AI に頼みましょう。

これは包括的な本番監査ではありません—恥ずかしく、高くつき、簡単に避けられる間違いを捕まえる短いリストです。反射になるまで、毎回のローンチで回しましょう。

まとめ

インフラは飛躍ではなく、はしごです。上に静的、下に VM、そしてあなたの仕事の大半は上段に幸せに住んでいます。

最も単純・安価・休止

STATIC ▲ ここから	ファイルのみ
EDGE / SERVERLESS	少しのロジック
CONTAINER	本物のバックエンド
VM ▼ 最後の手段	全部自分で運用

制御最大・作業最大

— 現実が迫った時だけ下へ —

5つのカテゴリーを知り、役目を果たす最もシンプルなものを選び、設定はAIに書かせ、できるだけ早く何かをライブに出荷しましょう。残りは、現実がそうさせるとき—そのときだけ—一段下に降りるだけのことです。

まとめと練習

重要なポイント

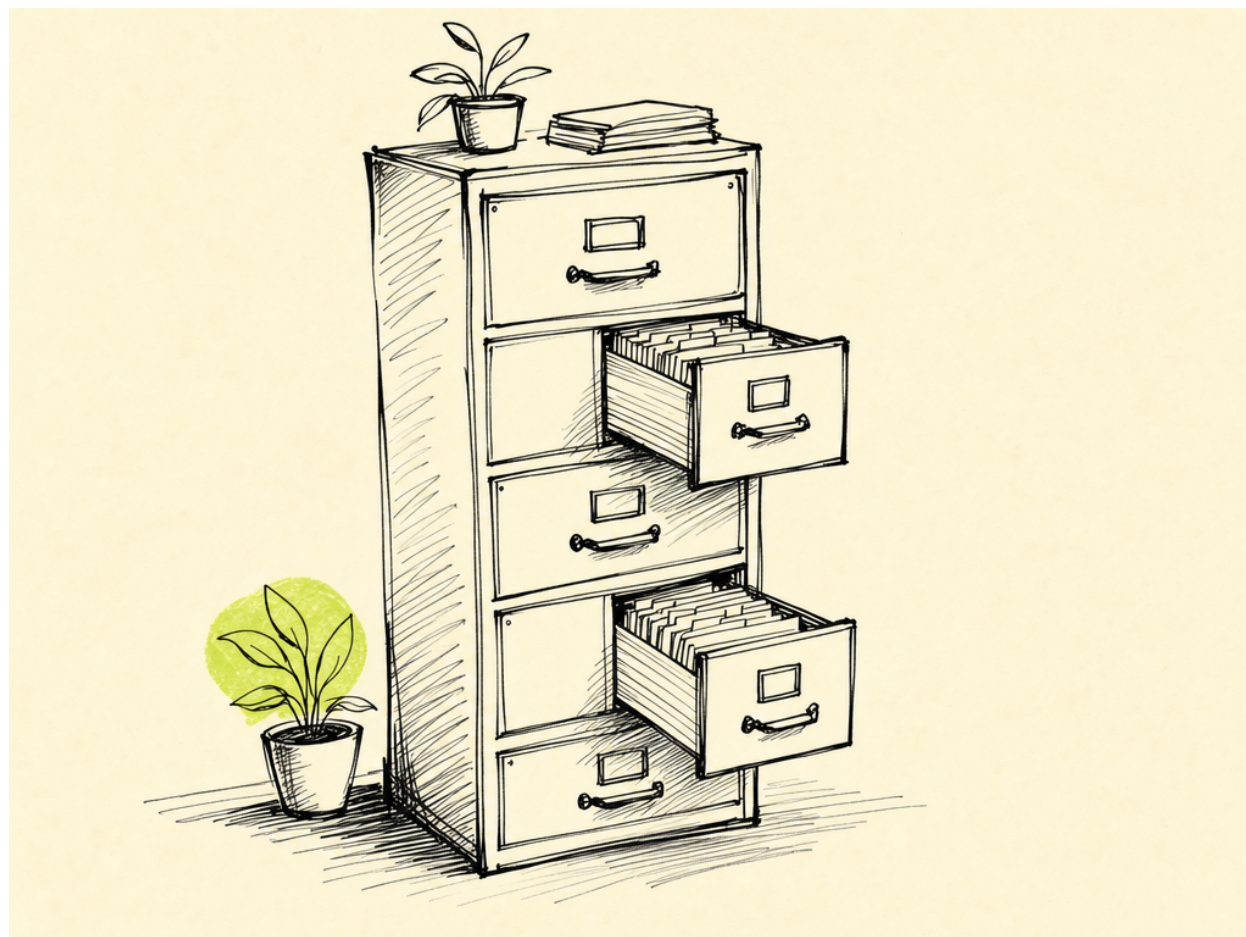
- インフラは5段のはしごだ—静的ホスティング、サーバーレス、PaaS、マネージドコンテナ、そして生のVM—そして仕事をこなせる最も単純な段を選ぶべきだ。
- **マネージド**サービスを優先せよ: 運用 (サーバー、パッチ、スケーリング、バックアップ) は委ね、理解は自分で持ち続ける。
- デプロイ設定とCI (Continuous Integration—コードをプッシュするたびにチェックを自動で走らせるロボット) ワークフローはAIに書かせてよいが、信頼する前にまず読むこと。
- 「マネージド」は「魔法」ではない—データ、シークレット、請求書、そして理解は依然としてあなたの責任だ。
- 公開する前に支出の上限を設定し、HTTPSを確認し、ロールバックの方法を知っておくこと。

やってみよう

すでに持っている小さなプロジェクト (HTML1 ページでもよい) を静的ホストにデプロイしてみよう。次にAIにpushのたびに再デプロイするGitHub Actionsワークフローを作らせ、生成されたYAMLを1行ずつ読み、ホストのダッシュボードで「前のデプロイにロールバック」ボタンを見つけられるか確認しよう。

```
I have a static site in this repo. Walk me through deploying it to a free static host, then generate a GitHub Actions workflow that rebuilds and redeploys on every push to main. Explain each step of the YAML, and tell me where in the dashboard I roll back a bad deploy.
```

データベースインフラ



本物のアプリはどれも、いずれ物事を覚えておく必要が出てきます—ユーザー、注文、メッセージ、設定。その記憶はデータベースに住みます。初心者にとって「データベース」は、専門家に守られたブラックボックスのように感じられるかもしれませんが、そんなことはありません。この章ではデータ層を解き明かし、あなたが良い選択をし、重い作業を安全に AI に任せて指示できるようにします。

早めに体に染み込ませるべきこと: コードは取り替えがききますが、データは普通そうではありません。バグのある関数は午後いっぱい書き直せますが、先週の火曜に壊してしまったカラム—あるいは一度も取らなかったバックアップ—は、永遠に失われるかもしれません。この非対称性こそが、データ層がスタックのほとんどの部分よりも、少しの先行理解に報いる理由です。

データストレージの 3 つのかたち

ほとんどのデータベースは 3 つの系統に分かれます。製品を暗記する必要はありません—自分の問題にどの かが合うかを見分けられればいいのです。

タイプ	例	最適な用途	トレードオフ
リレーショナル (SQL)	PostgreSQL、MySQL、SQLite	関係を持つ構造化データ。お金や正確さが重要なものすべて	先にスキーマを設計しなければならない。書き込みを非常に大きくスケールさせるのは難しい
ドキュメント / NoSQL	MongoDB、Firestore、DynamoDB	柔軟またはネストしたデータ、速いイテレーション、多様なレコード形状	一貫性のないデータを作りやすい。関係性やレポートが扱いにくくなる

タイプ	例	最適な用途	トレードオフ
キーバリュー	Redis、Memcached、Cloudflare KV	キャッシュ、セッション、カウンター、既知のキーによる高速な参照	内容による問い合わせができない。通常は信頼できる情報源 (source of truth) ではない

3 つのかたちは、データを本当に異なるやり方で保存します。並べて思い描いてみましょう:

リレーショナル (SQL)
行 + 列

id	email	name
1	a@...	Ana
2	b@...	Bob

テーブルが相互に
参照し合う

ドキュメント (NoSQL)
自己完結した塊

```

{
  id, email,
  name,
  posts: [ ... ]
}

```

すべて入れ子

キー・バリュー
1 キー → 1 バリュー

key	value
session	"x9f2"
count	"42"

中は検索不可 —
キーで引くだけ

リレーショナルデータベースは、互いを参照し合えるスプレッドシートの集まりだと考えてください。それと対話するときに使う言語が SQL (Structured Query Language、構造化問い合わせ言語) です。90% のプロジェクトに良いデフォルト: **リレーショナルデータベース (Postgres) から始めること**。成熟していて、予測可能で、構造を強制し、本物のアプリが必ず育てていく関係性を扱えます。データが本当に形を持たないときはドキュメントストアに手を伸ばし、キーバリューストアは主要なストアではなく 補助的な 層 (キャッシュ) として使います。

避けるべき罠は、雰囲気を選ぶことです。「NoSQL (SQL のテーブルを使わないデータベース—柔軟で自由な形のレコードを保存する) のほうがスケールする」が真なのは、ほとんどのアプリが決して到達しない狭い問題群においてだけで、しかもその代償は、データを信頼できるものにする一貫性保証を手放すことです。システム設計動画で最も格好よく聞こえるかたちではなく、自分の データ に合うかたちを選びましょう。迷ったらリレーショナルが安全な賭けです—キャッシュやドキュメントストアは後からいつでも足せますが、こんがらがった NoSQL データモデルをほどくのは苦痛です。

マネージド 対 セルフホスト

データベースは自分でサーバー上に運用することもできますし、**マネージド** なもの (Supabase、Neon、PlanetScale、RDS、Cloudflare D1) を借りることもできます。後者ではプロバイダーがバックアップ、パッチ適用、稼働時間を扱ってくれます。

Vibe coding では、**常にマネージドから始めましょう**。データベースをセルフホストするということは、バックアップ、セキュリティ更新、ディスク容量、そして午前 3 時の復旧に、もうあなたが責任を負うということです。それはものづくりから気を逸らす、本物の運用作業です。マネージドサービスは少しのお金がかかりますが、莫大な時間とリスクを節約します。具体的な理由とそれを裏打ちするスキルがあるときだけ、セルフホストへ進級しましょう。

マネージドプロバイダーを選ぶときは、自分が実際に何にお金を払っているのかを知る価値があります。マーケティングページはどれも似て見えるからです。本当に違う点がいくつかあります:

- **接続モデル**。サーバーレスのプラットフォーム (Cloudflare Workers、Vercel functions) はデータベース接続を絶えず開いては閉じるため、古典的な Postgres の接続プールを枯渇させかねません。Neon や Supabase のようなプロバイダーは、まさにこのために 接続プラー (PgBouncer または相当品) を備えています—サーバーレスのコードではプール済みの接続文字列を使いましょう。
- **ブランチとバックアップ**。Neon と PlanetScale はデータベースを Git のように「ブランチ」でき、マイグレーションをコピー上でテストできます。ポイントインタイムリカバリ (point-in-time recovery) が有効か確認し、どこまで遡れるかを把握しておきましょう。

- **ロックイン。** D1 と DynamoDB は優秀ですが、一つのクラウドにあなたを縛ります。素の Postgres (Supabase、Neon、RDS) は移植性があります—スキーマとクエリはどの Postgres ホストへも移ります。ほとんどの初心者には、移植可能なほうが長期的に安全な賭けです。

スキーマ設計の基本

スキーマ はあなたのデータの設計図です。どんなテーブルがあり、どんなカラムを持ち、それらがどう関係するか。良いスキーマ設計は、ほとんどが常識といくつかの習慣です。

- **「もの」1 つにつき 1 テーブル。** ユーザー、投稿、注文—それぞれが自分のテーブルを持ちます。
- **すべての行に一意的 ID が要る (主キー)。**
- **テーブルは外部キーでリンクする。** データをあちこちにコピーするのではなく。
- **正直な型を選ぶ。** お金は float ではなく decimal。日付はテキストではなくタイムスタンプ。
- **重複させない。** ユーザーのメールアドレスが 5 つのテーブルに住んでいたら、いずれ 5 つの異なるメールアドレスを持つことになります。

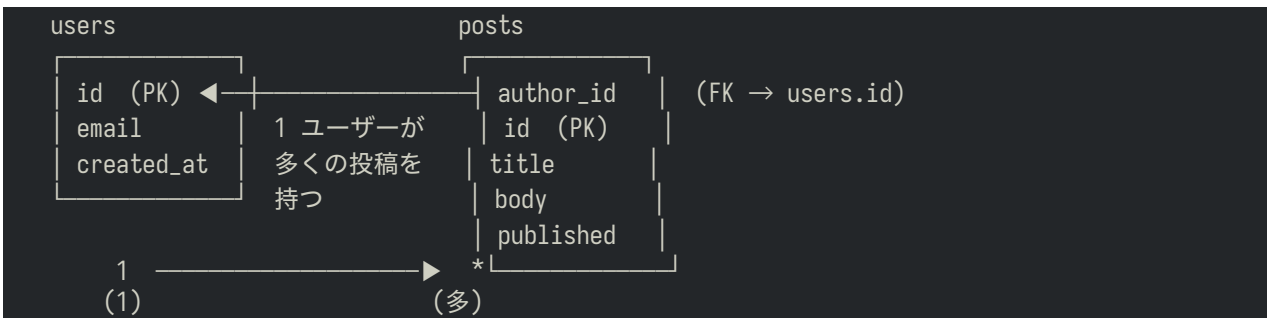
ブログ向けのシンプルなスキーマです。AI があなたのために生成する類いのものです。

```
CREATE TABLE users (
  id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  email       TEXT NOT NULL UNIQUE,
  created_at  TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE posts (
  id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  author_id   UUID NOT NULL REFERENCES users(id),
  title       TEXT NOT NULL,
  body        TEXT NOT NULL,
  published   BOOLEAN NOT NULL DEFAULT false,
  created_at  TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE INDEX idx_posts_author ON posts(author_id);
```

author_id ... REFERENCES users(id) の行が外部キーです。各投稿をちょうど 1 人の実在するユーザーに結びつけるので、データベース自身が有効な著者のない投稿を保存することを拒みます：



REFERENCES (投稿を実在するユーザーに結びつける外部キー) と、NOT NULL および UNIQUE 制約に注目してください。これらはデータベースがあなたのために強制してくれるガードレールで、コードにバグがあっても不正なデータが忍び込めないようにします。

意図的に加える価値のある制約は、親が消えたときに何が起こるかです。デフォルトでは、まだ投稿が参照しているユーザーを削除しようとする失敗します—それがしばしばあなたの望むことです。しかし明示的に決められます: REFERENCES users(id) ON DELETE CASCADE は投稿も一緒に削除し、ON DELETE RESTRICT は削除を阻止します。意図を持って選ぶほうが、本番でデフォルトを痛い目で知るよりましです。同じ「データベースに自

分を守らせる」本能は CHECK 制約 (CHECK (price >= 0)) にも当てはまります—すべてのコードパスが検証してくれると信じる代わりに、ばかげたデータを源泉で拒みます。

マイグレーション: スキーマを安全に変える

あなたのスキーマは変わっていきます—カラムを追加し、テーブル名を変え、インデックスを追加します。共有ドキュメントの変更履歴 (変更追跡) のようなものだと考えてください: すべての変更が順番に記録され、誰のコピーも同じやり方で最新に揃えられます。**マイグレーション** とは、そうした 1 つの変更を記述した、バージョン管理された、繰り返し実行できるスクリプトです。マイグレーションが重要なのは、あなたのノート PC、チームメイトのノート PC、そして本番のスキーマが、同一に保たなければならないからです。

マイグレーションを安全に保つルール:

- **本番でデータベースを手作業で編集しない。** マイグレーションを書きます。
- **マイグレーションは前へ進む。** それぞれが新しいファイルです。すでに実行された古いものを書き換えてはいけません。
- **破壊的な変更は慎重に。** カラムやテーブルを削除すると、データは永久に消えます。先にバックアップを取りましょう。
- **追加的な変更が最も安全。** NULL 許容のカラムを追加してもめったに何も壊れません。名前変更や削除は、稼働中のコードを壊しかねません。

マイグレーションは順序のある鎖を成します: それぞれが番号付きのステップで、データベースのすべてのコピーを、ある既知の状態から次の状態へ動かします。だからあなたのノート PC、チームメイトのもの、そして本番が、すべて同一に行き着きます:



Prisma、Drizzle、Rails migrations のようなツールが、これらを生成・追跡してくれます。任せましょう。

マイグレーションは 2 つの半分でできています: 適用する変更 (up) と、それを取り消す方法 (down) です。信頼できるツールは両方を生成するので、まずいデプロイがきれいにロールバックできます:

```
-- up: 任意の bio を追加。稼働中に適用しても安全
ALTER TABLE users ADD COLUMN bio TEXT;

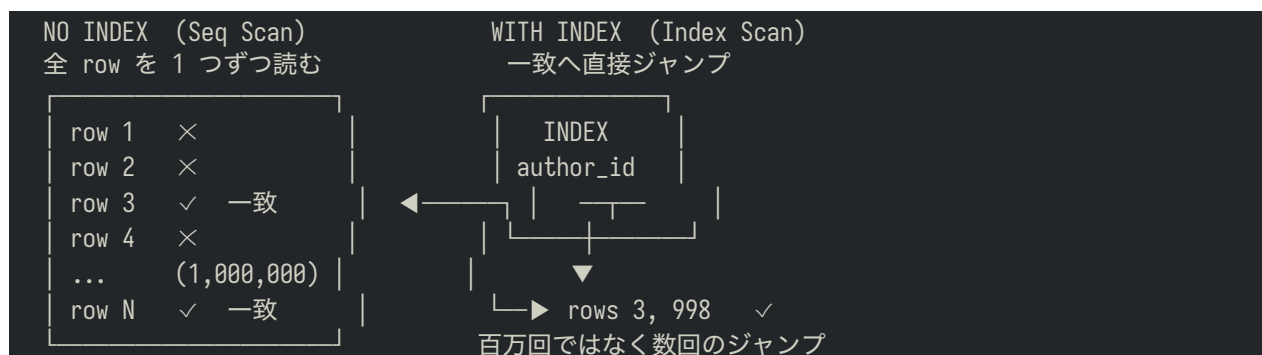
-- down: 取り消す
ALTER TABLE users DROP COLUMN bio;
```

本当に危険なマイグレーションは 名前変更 と 型変更 です。古いコードと新しいスキーマが一瞬食い違うからです。プロのパターンは、それらを追加的なステップに分割することです: 新しいカラムを追加し、バックフィルし、コードがそれを読むよう切り替え、何も参照しなくなった後のデプロイでようやく古いカラムを削除します。遅く感じますが、それが滑らかなリリースと、デプロイの最中にすべてのユーザーに 500 エラーが噴き出すこととの違いです。

インデックス

インデックス は本の巻末索引のようなものです。データベースがすべての行をスキャンする代わりに、一致する行へ直接ジャンプできるようにします。インデックスがなければ、データが増えるにつれてクエリは遅くなります—100 行なら問題なくとも、1,000,000 行では苦痛です。

インデックスがなければデータベースは 1 行ずつすべてを確認し、あればぴったりの行へ直接ジャンプします。本の索引が最初から最後まで読む代わりに正しいページへ送ってくれるのと同じです:



実践的な指針:

- 頻繁に **フィルタ** または **結合 (join)** するカラムにインデックスを張る (例: author_id、email)。
- 主キーは自動的にインデックスが張られます。
- **すべてにインデックスを張らない**—各インデックスは読み取りを速くしますが、書き込みを遅くし、容量を使います。
- 遅いクエリを見たときにインデックスを追加します。すべてのカラムに先回りして張るのではなく。

インデックスが効くかどうかを知る正直な方法は、当てずっぽうではなくデータベースに尋ねることです。どの SQL エンジンにも、使う計画を見せてくれる EXPLAIN (Postgres では EXPLAIN ANALYZE) があります:

```
EXPLAIN ANALYZE
SELECT * FROM posts WHERE author_id = '...';
```

大きなテーブルで出力が Seq Scan と言うなら、データベースはすべての行を読んでいます—インデックスが効くというサインです。1 つ追加すれば Index Scan に切り替わるはず。これは AI に渡すのに最適な作業です: 遅いクエリと EXPLAIN の出力を貼り付け、どのインデックスを追加すべきか、なぜかを尋ねましょう。あなたはインデックス理論を暗記するのではなく、答えを理解することで主導権を保ちます。

キャッシュ

キャッシュ は、高価な問いへの答えを保存しておき、二度と尋ねなくて済むようにします。トップページが訪問者ごとに同じ重いクエリを実行しているなら、結果を Redis に 60 秒キャッシュして、瞬時に配信しましょう。

キャッシュはパフォーマンスの最適化であって、信頼できる情報源ではありません。初心者の典型的な罠は、**キャッシュの無効化 (cache invalidation)** を忘れることです—キャッシュされたコピーが古くなり、ユーザーが古いデータを見ます。経験則: 本物のボトルネックを計測したときだけキャッシュし、そしてキャッシュされた値がいつ期限切れになる、あるいは更新されるかの明確なルールを常に持ちましょう。

きれいな戦略は 2 つあり、この 2 つを取り違えることがほとんどのキャッシュバグを生みます。**時間ベースの失効 (TTL —Time To Live、キャッシュ値に押された賞味期限のようなもの)** は「この値は 60 秒間有効、その後は捨てる」と言います—シンプルで、少し古いデータが無害なとき (閲覧数、トップページのフィード) には十分です。**イベントベースの無効化** は「元データが変わった瞬間にキャッシュ値を削除する」と言います—正確ですが、それをすべての書き込みパスで覚えておかねばならず、まさにそこで人は滑ります。迷ったら短い TTL を選びましょう: せいぜい 1 分古い値は、無効化を 1 つ忘れて永遠に古いままの値より、小さなバグです。

キュー

キュー は「今やる」の代わりに「これを後でやる」と言えるようにします。ユーザーがサインアップしたとき、ウェルカムメールを送りアバターをリサイズするあいだ、ユーザーを待たせたくありません。それらのジョブをキュー (RabbitMQ、BullMQ、Cloudflare Queues、SQS) に積みま。バックグラウンドのワーカーがそれらを拾い上げます。ユーザーは即座の応答を受け取ります。

遅い、失敗してリトライしうる、あるいはユーザーに応答する前に完了する必要のない、あらゆるタスクにキューを使いましょう。

ほとんどの初心者がぶつかる落とし穴は、キューに入れたジョブが2回以上実行されうることです—ネットワークの瞬断でワーカーがリトライし、ウェルカムメールが二度送られます。直し方は、ジョブを **冪等 (idempotent)** にすることです: 同じジョブを2回実行しても1回実行したのと同じ効果になるよう設計します (送る前に「もう送ったか?」を確認)。また、失敗し続けるジョブをどうするかを先に決めましょう。成熟したキューはそれをデッドレターキューに回し、毒入りメッセージ1つがその後ろに並ぶすべてを塞がないようにします。

初心者によくある間違い

- **モダンに聞こえるから NoSQL に手を伸ばす。** ほとんどのアプリは関係性を持ち、SQL はそれをうまく扱います。
- **お金を浮動小数点で保存する。** $0.1 + 0.2$ が 0.3 にならないのは有名です。decimal 型を使いましょう。
- **バックアップがない。** マネージドサービスはたいてい自動でバックアップします—確認しましょう、決めつけられないこと。
- **シークレットや巨大なファイルをデータベースに入れる。** ファイルはオブジェクトストレージ (S3、R2) に属します。データベースは **リンク** を保存します。
- **ユーザー入力を貼り付けた生クエリを AI に書かせる。** これは SQL インジェクションを引き起こします。パラメータ化クエリを使いましょう。
- **N+1 クエリ。** 項目ごとにクエリを1つ実行するループ (投稿を100件取得してから、著者の参照を別々に100回) はデータベースを叩きます。関連データを結合 (join) で1クエリにまとめて取得するか、参照をバッチにしましょう。
- **負荷テストを一切しない。** 自分のノート PC の50行で瞬時のクエリが、本番の500万行では違うことがあります。性能を信頼する前に、現実的な量のダミーデータをシードしましょう。

データ層を構築するために AI を指示する

AI はスキーマ設計とクエリ記述に優れています—ただし 設計の拠り所となる制約を与えればの話です。「データベースを作って」と言うてはいけません。ドメイン、関係性、そしてあなたの譲れない条件を記述しましょう。

強力なプロンプトはこんな感じです。

Design a PostgreSQL schema for a task-management app.

Entities and relationships:

- A user can own many projects.
- A project has many tasks.
- A task belongs to one project, has a status (todo / doing / done), and an optional due date.

Requirements:

- Use UUID primary keys and created_at timestamps.
- Enforce relationships with foreign keys.
- Add indexes for the columns we'll filter on.
- Give me the schema as SQL, plus a forward migration file.
- Explain any trade-offs you made.

それから、この章のチェックリストで **出力をレビュー** しましょう。正直な型、外部キーと制約の存在、妥当なインデックス、重複データのなさ。クエリを頼むときは、ユーザー入力が決して SQL に連結されないよう、**パラメータ化クエリ** を明示的に要求します。そして、何かを削除または名前変更するマイグレーションについては、先にバックアップできるよう、AI にそれを指摘させましょう。

データベースは、間違いが最も取り返しのつかないアプリの部分です—データは、いったん破損したり失われたりすると、いつも戻ってくるとは限りません。だからこそ、これらの基礎を知る価値があります。すべての行を

自分で書けるようになるためではなく、自信を持って AI を指示し、危険な動きが出荷される前に捕まえられるようにするためです。

まとめと練習

重要なポイント

- ほとんどのアプリは 3 つの保存形態—リレーショナル、ドキュメント、キーバリュー—のうち 1 つしか必要とせず、リレーショナルが安全なデフォルトだ。
- マネージドデータベースから始めよう。データとスキーマの判断はあなたが、サーバーの稼働はプラットフォームが担う。
- 良いスキーマ設計とは、正直な型、外部キーと制約、妥当なインデックス、そして重複のないデータを意味する。
- すべてのマイグレーションを一方通行の扉として扱え: drop やリネームの前にバックアップし、AI が書いたマイグレーションは入念にレビューする。
- ユーザー入力に SQL にそのまま連結されないよう、常にパラメータ化クエリを要求せよ。

やってみよう

小さなアプリのアイデアを 1 つ選び、そのエンティティと関係をまず普通の日本語で書き出そう。次にその説明を AI に渡してスキーマと前進マイグレーションを依頼し—何かを実行する前に、この章のチェックリストで結果をレビューしよう。

```
Design a relational schema for a personal bookmarks app.
```

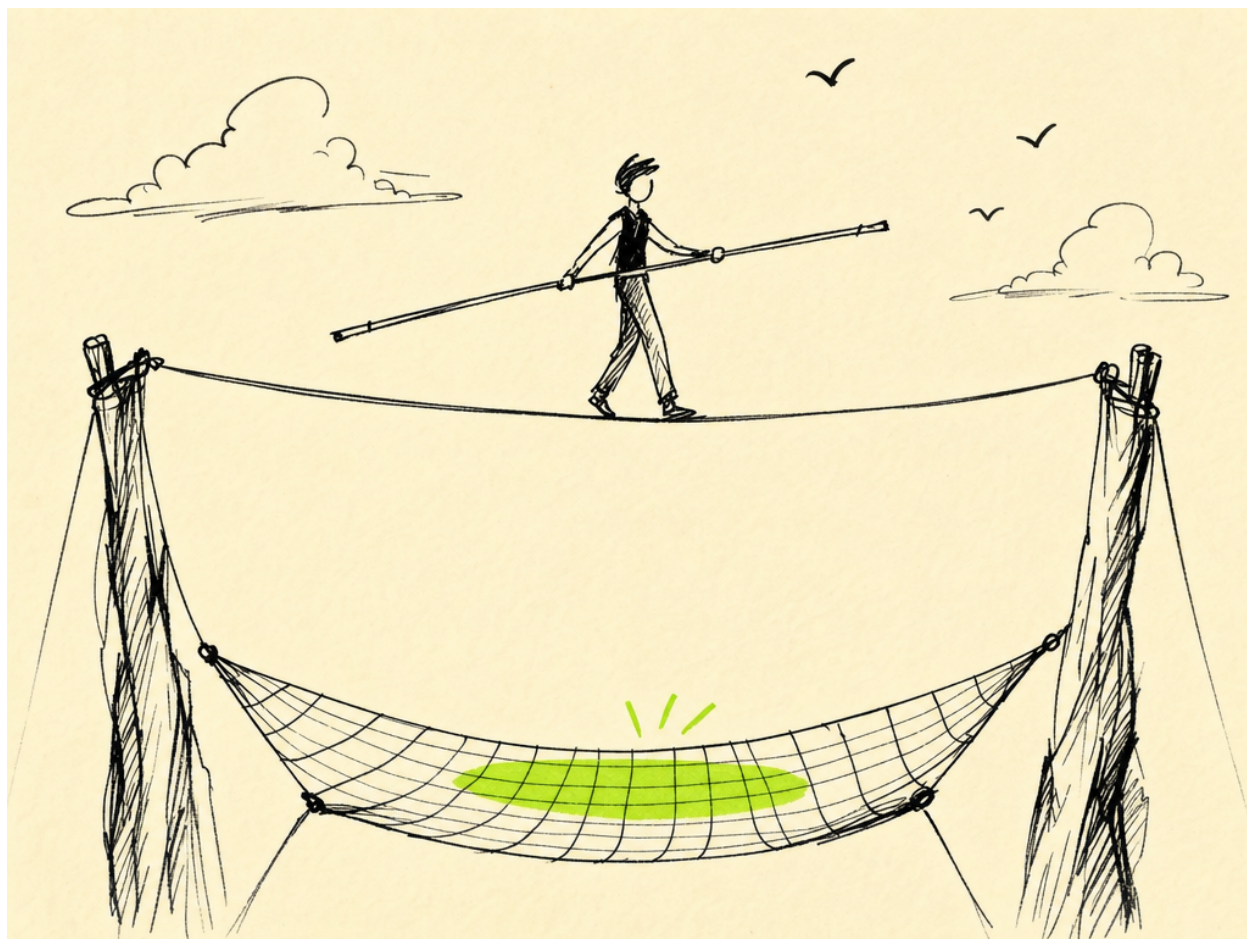
```
Entities and relationships:
```

- A user has many bookmarks.
- A bookmark belongs to one user, has a URL, a title, and many tags.
- A tag can apply to many bookmarks.

```
Requirements:
```

- Use UUID primary keys and created_at timestamps.
- Enforce relationships with foreign keys.
- Add indexes for the columns we'll filter on.
- Give me the schema as SQL plus a forward migration file.
- Flag any migration step that drops or renames so I can back up first.

ガードレール、テスト、レビュー



パイプコーディングは速く動けるようにしてくれます。危険なのは、間違った方向にも速く進んでしまうことです。しかも AI は自信ありげに振る舞うため、実際のユーザーの前で何かが壊れるまで気づかないかもしれません。ガードレールは、衝突を避けながらスピードを保つための仕組みです。

この章は、プロジェクトを官僚的な仕組みに変えることが目的ではありません。AI の (そしてあなた自身の) ミスが本番に届く前に捕まえる、いくつかの安価で自動的なチェックについての話です。一度設定すれば、あとはずっとバックグラウンドで働いてくれます。この章の哲学は一行に尽きます。**理解していないものは決して出荷しない**。ガードレールは、その「理解する」を、毎回必ずやれるくらい安価にしてくれるものです。

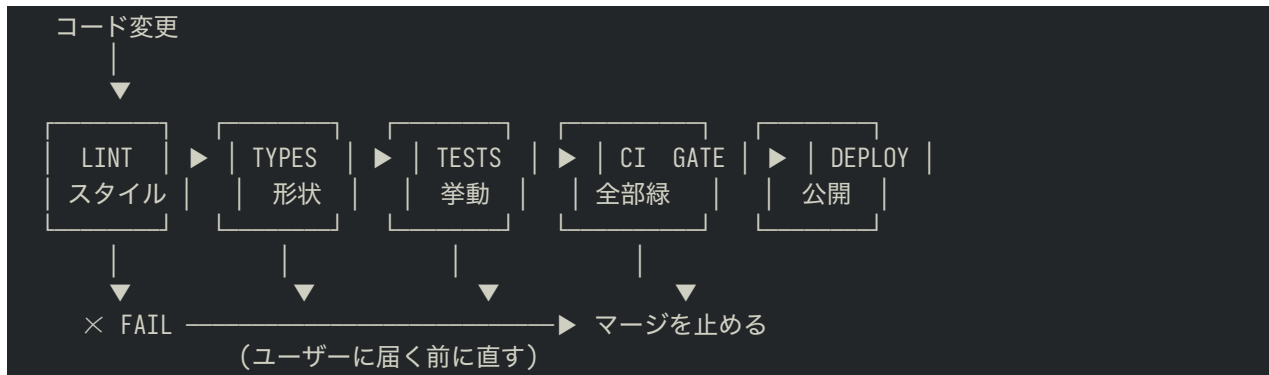
AI を指揮するときにガードレールが必要な理由

AI は、あなたのアプリが本来何をすべきかを知りません。もっともらしく見えるものにパターンマッチするだけです。つまり、AI は平然と次のようなことをやります。

- 理解できないチェックを「エラーを直すため」に削除する
- 1つのファイルを変えてと頼んだのに、3つのファイルをリファクタリングする
- 正常系では現れない微妙なバグを忍び込ませる
- 一度もコードを実行していないのに、自信たっぷりに「完了しました」と言う

ガードレールは「信じてください」を「証明してみせて」に変えます。目標は、偶然では物事を壊しにくいシステムを作ることです。そうすれば、息を止めることなく「とにかく試してみて」と言い続けられます。

ガードレールは、すべての変更が実際のユーザーに届く前に通り抜けなければならないパイプラインだと考えてください。各ゲートは安価で自動的、そしてそのどれか 1 つでも、まずい変更をぴしゃりと止められます：

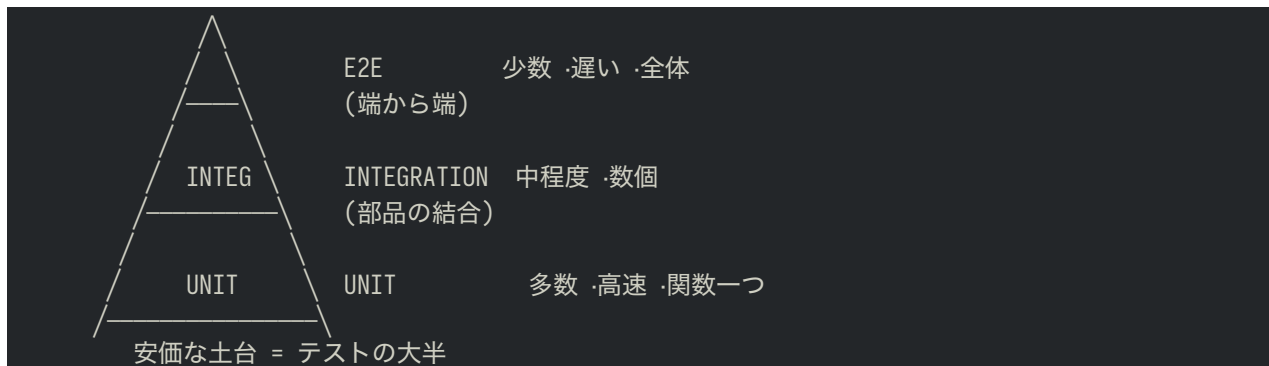


これが人間の同僚よりも AI において重要な、より深い理由があります。検証チェックを削除する人間は、たいてい自分がリスクを取っていることを知っていて、その重みを感じています。AI は何も感じません。AI はガードを取り除き、ファイルを書き換え、成功を報告します——その変更が見事だろうと破滅的だろうと、同じ陽気な口調で。あなたは AI の身振りを読めません。身振りというものがないからです。ガードレールこそがその身振りです——作業が実際に健全かどうかについて、あなたが得られる唯一の正直な信号なのです。

テスト: あなたのセーフティネット

テストとは、別のコードが期待どおりに動くかをチェックするコードです。一度書けば数秒で実行でき、何かが退行したらすぐに教えてくれます。

健全なテストスイートはピラミッドのような形をしています。土台には小さく速いユニットテストがたくさん、真ん中には統合テストが少し、頂上には遅いエンドツーエンドテストがほんの一握り。安全のほとんどは、土台から安価にやってきます:



カバレッジ 100% は必要ありません。壊れたときに最も痛い部分から始めましょう。ログイン、決済、再生成できないデータ、コアとなるビジネスロジックです。AI はテストを書くのが非常に得意で、しばしば機能を書くより上手です。期待される振る舞いが具体的だからです。

うまくいくプロンプトの例です。

src/pricing.js の calculateDiscount 関数のテストを書いて。
通常ケース、ゼロ/空ケース、最大割引の境界値、不正な入力を1つ、
これらをカバーして。このリポジトリの既存のテスト構成を使うこと
(他の *.test.js ファイルがどう構成されているか確認して)。
完了する前にテストを実行して、パスすることを見せて。

最後の一文が重要です。テストを書くだけでなく、必ず AI にテストを実行させましょう。一度も実行されていないテストは、ただの推測です。どのケースをカバーするかを具体的に伝えることにも価値があります。放っておくと、AI は正常系のバリエーションを 3 つ書いて「網羅した」と言いがちだからです。実際にバグを捕まえるのは、退屈なケースです。空の入力、境界値、例外を投げるべきもの。

AI が生成すべきものの簡単な例です。

```
import { calculateDiscount } from "../pricing.js";
import { describe, it, expect } from "vitest";

describe("calculateDiscount", () => {
  it("applies a normal percentage discount", () => {
    expect(calculateDiscount(100, 0.2)).toBe(80);
  });

  it("returns the full price when discount is zero", () => {
    expect(calculateDiscount(100, 0)).toBe(100);
  });

  it("never discounts below zero", () => {
    expect(calculateDiscount(100, 1.5)).toBe(0);
  });

  it("rejects a negative price", () => {
    expect(() => calculateDiscount(-10, 0.2)).toThrow();
  });
});
```

あとで機能を変更するときは、先にテストを実行しましょう。レッドになれば、ユーザーより先に壊れた箇所を見つけられます。AI がバグを直したら、古い振る舞いで失敗するテストを追加させましょう。そうすれば、そのバグがこっそり戻ってくることはありません。この習慣には名前があります。リグレーションテストです。あなたが遭遇したすべてのバグは、コードベースが代償を払って得た教訓です。リグレーションテストは、その教訓を保ち続ける方法です。

一つ警告を。テストもまた間違うことがあります。AI がバグのある振る舞いを正しいと主張するテストを書けば、そのテストは永遠にパスし続け、アプリは壊れたままになります。だから、緑のチェックマークだけでなく、アサーションを読みましょう。パスするテストは「コードがテストの言うとおりに動く」ことしか意味しません——そのテストが正しいことを言っているかは、あなたが確認しなければなりません。

型チェックとリント: タダで捕まえられるミス

ほとんど手間をかけずに、エラーのカテゴリ全体を捕まえてくれる 2 つのツールがあります。

- **型チェック** (TypeScript、mypy など) は、「数値を期待している箇所に文字列を渡した」「この値は undefined になりうる」といったミスを捕まえます。AI は、型があれば不可能になるはずのエッジケースを忘れがちです。
- **リント** (ESLint、Ruff など) は、未使用の変数、到達不能なコード、スタイルのブレを捕まえます。AI が触れたコードベースが、じわじわとぐちゃぐちゃになるのを防いでくれます。

これらは早めに設定し、AI に従うよう指示しましょう。「型チェッカーとリンターを実行して、完了と言う前にそれらが報告したものをすべて直して」。クリーンな型チェックを「完了」の定義の一部として扱しましょう。

ただし、安易な逃げ道に注意してください。型チェッカーが文句を言ったとき、本当に正しい修正は、それが見つけたケース——undefined の値、間違った型——を処理することです。速い修正は、`// @ts-ignore`、`any`、あるいは `# type: ignore` コメントでその文句を黙らせることで、AI は赤い文字が消えるからという理由で絶えずこれに手を伸ばします。それらはまた、チェックが助けようとしていたまさにその場所で、チェックを無価値にします。差分に抑制 (suppression) コメントが現れたら、それを解決策ではなく問いとして扱しましょう。「これはなぜ必要なの? 無視するのではなく、根本の型を直して」。

AI の変更範囲を絞る

AI による被害の最大の原因は、一度にあまりに多くを変更させてしまうことです。9 つのファイルにまたがる 400 行の差分はレビュー不可能で、バグを隠すのも簡単です。

変更は小さく、限定的に保ちましょう。

- 「ついでにこれも…」ではなく、**一度に 1 つの論理的な変更**を依頼する。
- 明示的に伝える。「src/auth.js だけを触って。それ以外は何もリファクタリングしないで」。
- 動作する状態をこまめにコミットして、いつでもロールバックできるクリーンなポイントを確保する。
- 大きな書き換えを提案してきたら、1 つずつレビューできるステップに分けるよう頼む。

範囲が小さければ、次のステップ (差分を読むこと) が実際に可能になります。ここには複利効果があります。小さなコミットは、被害範囲 (blast radius) も小さいのです。限定的な変更が間違っていたとわかれば、コミットを 1 つ戻して 10 分を失うだけです。広範な変更が間違っていれば、どの部分が壊したのかをほどくの 1 時間を使うか、午後の作業全体を捨てることになります。範囲の規律は、必要だとわかる前に買っておく安価な保険です。

差分を批判的に読む

差分とは、何が変わったかを正確に示すリストです。赤い行は削除、緑の行は追加。差分を読むことは、パイプコーディングで最もレバレッジの高い習慣です。なぜなら、頼んでいないことを AI がやっているのを捕まえられる場所だからです。

すべての行を理解する必要はありません。次のような危険信号をスキャンしましょう。

- **変わるはずのないファイル**。なぜ設定ファイルが触られたのか？
- **削除されたコード**。バリデーションチェック、エラーハンドラ、テストを「パスさせるため」に消していないか？
- **シークレットやハードコードされた値**。コードに入っているべきでない API (Application Programming Interface) キー、パスワード、URL (Uniform Resource Locator —ウェブアドレス)。
- **無効化されたチェック**。スキップされたテスト、コメントアウトされたガード、// @ts-ignore、緩められた権限。
- **スコープのはみ出し**。頼んだこととかけ離れた変更。

何かおかしく見えたら、こう尋ねましょう。「X を変えた理由を説明して。それは頼んでいない」。特に削除については、AI に正当化させましょう。「エラーの原因になっていたから」は、そのエラーを理解する理由であって、チェックを削除する理由ではありません。

習慣を身につける実践的な方法。変更を受け入れる前に差分を読み、決して後ではなく。コードが一度マージされ、アプリが動いているように見えたら、あなたは二度と戻って読みはしません——強制する摩擦がないからです。レビューの瞬間が、決定の瞬間でなければなりません。ここでツールが役立ちます。git diff --stat はファイルごとに 1 行のサマリーをくれるので、予想外のファイルがすぐに目に飛び込みます。プルリクエストのページでレビューすれば、エディタのあちこちに散らばる代わりに、変更全体をスクロール可能な一画面に表示してくれます。

レビューゲート: マージ前のチェックリスト

レビューゲートとは、変更が本番に出る前に満たされていなければならない事項の短いリストです。自分で確認するか、AI に各項目を確認させましょう。

- ☐ 差分を読み、何がなぜ変わったか理解した
- ☐ 変更は実際に頼んだ範囲に収まっている
- ☐ テストがパスする (新しい振る舞いにはテストがある)
- ☐ 型チェックとリンターがクリーン
- ☐ シークレット、デバッグログ、コメントアウトされたコードが残っていない
- ☐ 安全チェックがこっそり削除されていない
- ☐ 少なくとも正常系について、自分で変更を試した

チェックを入れられない項目があるなら、まだ完了ではありません。

このチェックリストが意図的に短いのは、あなたが飛ばすゲートはゲートではないからです。特に速く動いていて変更が「問題なさそう」に見えるとき、そのまま通したくなる誘惑があります。ユーザーに触れるものについては、それに抗いましょう。役立つコツは、AI にあなたの代わりに記入させることです——リストを貼り付けて、こう頼みます。「いま行った変更について、このレビューゲートの一つずつ通して。各項目について、PASS か FAIL を、一行の根拠とともに言って」。AI はテストを実行せずに正直に「テストはパス」とは書けませんし、行ごとに根拠を示させれば、曖昧な「よさそう」が監査可能な何かに変わります。

CI の基本: チェックを自動化する

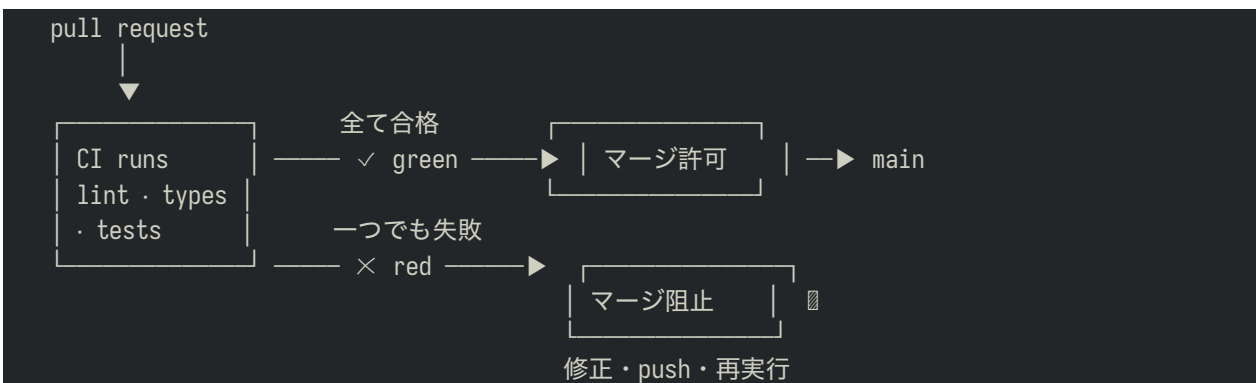
CI(継続的インテグレーション) は、コードをプッシュするたびにチェックを実行してくれるロボットです。だから実行を忘れることができません。たいていのプラットフォームでは、これは 1 つのファイルで済みます。最小限の GitHub Actions の例です。

```
# .github/workflows/ci.yml
name: CI
on: [push, pull_request]

jobs:
  check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
      - run: npm ci
      - run: npm run lint
      - run: npm run typecheck
      - run: npm test
```

これで、プッシュのたびに自動でリント、型チェック、テストが走ります。何か失敗すれば、コードがマージされる前に赤いマークが見えます。このファイルを自分のスタック向けに書いてもらうこともできます。「プルリクエストのたびにリント、型チェック、テストを実行する GitHub Actions のワークフローを追加して」。

ブランチ保護をオンにすると、赤いチェックは警告するだけでなく、ビルドが再びグリーンになるまでマージボタンを物理的にロックします:



CI の要点は、自動化それ自体ではなく、チェックをあなたの記憶からパイプラインへと移すことです——そこから、疲れた夜でも飛ばせません。基本がグリーンで回るようになったら、一度やっておく価値のあるアップグレードが 2 つあります。一つ目は、チェックをメインブランチの必須ステータス (GitHub の「ブランチ保護ルール」) にして、赤いビルドが無視できる警告ではなく、文字どおりマージボタンを止めるようにすること。二つ目

は、CI が失敗したら、失敗ログをそのまま AI に貼り付けること。「この実行で CI が失敗した、これが出力——診断して直して」。ログは精密な、機械が生成したバグ報告であり、それこそ AI が最も得意とする種類の入力なのです。

デフォルトで安全な習慣

これらを、恐れずに速く動くための反射的な習慣としてまとめましょう。

- ・ **本番に直接ではなく、ブランチで作業する。** ブランチはタダだが、壊れた本番はタダではない。
- ・ **小さく動作する状態をこまめにコミットする。** 頻繁なコミットでロールバックが簡単になる。
- ・ **AI が書いたものは必ず実行させる。** テスト、アプリ、型チェッカーなど。
- ・ **AI 用のターミナルと自分用のターミナルを分ける。** 独立して検証できるように。
- ・ **自分が読んでいない変更を絶対にリリースさせない。** スピードは良いガードレールから来るもので、レビューを省くことから来るものではない。

ガードレールはあなたを遅くしません。胃をきりきりさせることなく「よし、リリースしよう」と言えるようにしてくれるものです。一度設定すれば、AI は全力疾走でき、セーフティネットが転倒を受け止めてくれます。

ガードレールが悲鳴を上げたら: 直す前にトリアージする

ガードレールは、あなたがうまく反応したときにのみ役立ちます。そして失敗のパターンは予測可能です。ビルドが赤くなり、あなたは少し慌て、エラーを「直して」と添えて AI に貼り付ける。ときにはそれでうまくいきます。しかし多くの場合、それは実際に何が間違っていたかに対処せず、赤い文字だけを消す修正を生み出します——AI がテストを弱め、型を広げ、失敗する呼び出しを問題を飲み込む try/catch で包んでしまうのです。

AI に何かを「直させる」前に、10 秒をトリアージに使いましょう。ガードレールの失敗には、実質的に 3 種類しかありません。

- ・ **あなたがいま入れてしまった本物のバグ。** チェックは仕事をしています。チェックではなく、コードを直しましょう。
- ・ **いまや古くなったテスト。** あなたは意図的に振る舞いを変え、古いアサーションがもう一致しなくなった。新しい正しい振る舞いをアサートするよう、テストを更新しましょう——反射的にではなく、意図的に。
- ・ **不安定な、または間違ったチェック。** 本当に稀です。ここでは自分を疑いましょう。これは人々が本物のバグを放り込むバケツです。先に進めるのを許してくれる唯一のバケツだからです。

AI を正直に保つ指示は、何かを変える前にまず診断させることです。

「never discounts below zero」テストが失敗している。
まず、なぜ失敗するのか教えて——いまコードが何をしているか 対
テストが何を期待しているか。次に、どちらが正しいか教えて。
コードか、テストか。私が確認するまで、何も変えないで。

その一度の立ち止まりが、あなたを守るガードレールと、AI をこっそり「無力化するよう仕込んでしまった」ガードレールの違いです。赤いチェックは情報です。あなたができる最悪のことは、それが何を告げていたかを読まずに、緑にするための代償を払うことです。

まとめと練習

重要なポイント

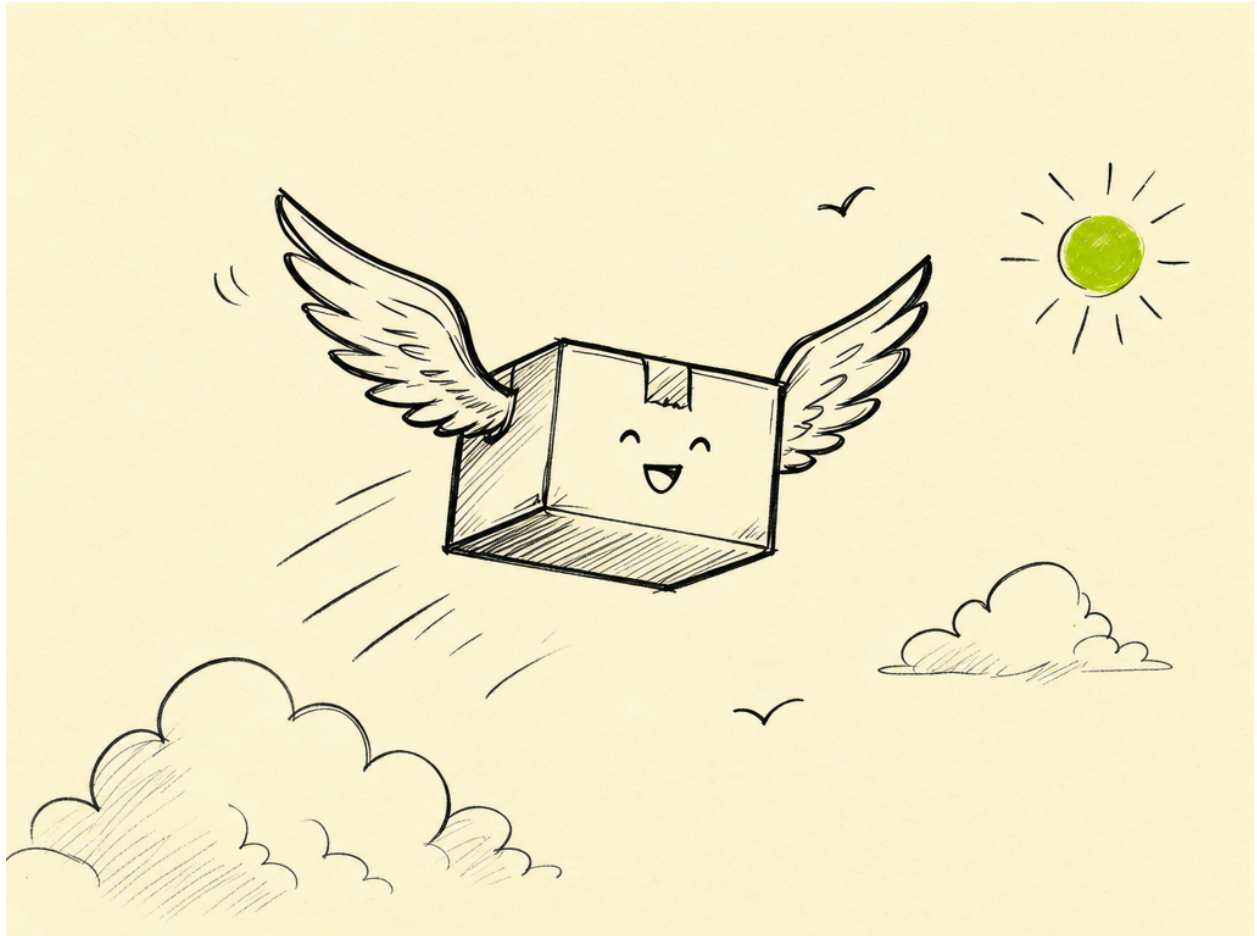
- ・ ガードレールは、安全網が落下を受け止めている間に AI が全力疾走できるようにする仕組みだ—遅くするのではなく、速くする。
- ・ テスト、型チェック、リントは無料でミスを捕まえてくれる。CI に組み込んで検査が自動で走るようにしましょう。
- ・ AI の変更は範囲を狭く限定し、マージ前にすべての diff を批判的に読むこと。
- ・ 赤い検査は情報だ—AI に「直して」と言う前に分類せよ (本物のバグ / 古くなったテスト / 不安定な検査)。
- ・ 読んでいない変更は決して公開しないこと。

やってみよう

あなたのプロジェクトで、まだ持っていない自動検査を 1 つーテスト 1 つ、リンター、または型チェックの工程ー設定し、push のたびに走るよう CI ワークフローに追加しよう。次にわざと小さなものを壊し、直す前に検査が赤くなることを確認しよう。

```
Add a CI workflow to this repo that runs the test suite, the linter,
and the type checker on every push and pull request. Generate the
config, explain each step, and tell me how to read the output when a
check fails so I can tell a real bug from an out-of-date test.
```

本番へのリリース



あなたはずっと作り続けてきました。アプリは自分のマシンで動き、デモは友人を感動させ、これを世に出したいという衝動は本物です。多くの個人開発者が足踏みするのがこの瞬間です。コードが壊れているからではなく、「公開する」ことが独自の闇の魔術 (DNS (ドメイン・ネーム・システム。yourdomain.com のような名前を、サーバーが置かれている番号に変換してくれるインターネットの住所録です)、サーバー、シークレット、決済の Webhook) を持つ別の領域のように感じられるからです。朗報があります。これこそ、機械的でよく文書化された、AI が最も得意とする種類の作業です。あなたの仕事は指揮すること。AI の仕事はコマンドを打つことです。

ただし「指揮する」という言葉が肝心です。本書全体のテーマは自分が出荷するものを理解することです。フラグを暗記する必要はありませんが、各ステップが何のためのものなのかは知っておく必要があります。Webhook が何からあなたを守るのか、なぜシークレットが絶対にリポジトリに入ってはいけないのか、支出上限が何を檻に入れているのか。その形が分かっているれば、AI の自信たっぷりの答えがこっそり間違っているときに気づけま

すし、最初のデプロイが爆発したときに立て直せます。ローンチで火傷を負う開発者は、ほぼ必ず、自分が読めないコマンドを貼り付けた人たちです。

この章では、動作するローカルアプリから、人々が実際に支払いをできる本物の URL (ユニフォーム・リソース・ロケーター。ブラウザに打ち込むウェブアドレスのことです) まで、最後のマイルすべてを歩きます。

ホストを選んでデプロイする

たいていのパイプコーディングされたアプリには、自分で世話をするサーバーは必要ありません。現代のプラットフォームは、あなたのコードを受け取って数分でライブの URL を返してくれます。よくある選択肢です。

- ・ **静的サイトとフロントエンド:** Cloudflare Pages、Vercel、Netlify、GitHub Pages。
- ・ **フルスタックアプリと API (アプリケーション・プログラミング・インターフェース。あるプログラムが別のプログラムと話すために使う出入り口です):** Cloudflare Workers、Vercel、Render、Fly.io、Railway。
- ・ **より重いバックエンド:** コンテナプラットフォーム、または小さな VPS (バーチャル・プライベート・サーバー。データセンターにある、常時稼働するコンピューターのうち、あなたが借りた自分専用の一区画です)。

正しいやり方は、何を作ったかを AI に伝えて推薦させることです。試してみましょう。「これは *Postgres* データベースを使った *Next.js* アプリです。寛大な無料枠を持つ、最も安く信頼できるホストが欲しい。デプロイ手順と設定ファイルを教えて」。選択を実際に左右するものについて具体的に伝えましょう。フレームワーク、データベースの有無、バックグラウンドジョブやファイルアップロードが必要かどうか、そして予算です。静的なマーケティングページと、キューを持つフルスタックアプリでは、住む場所がまったく違います。

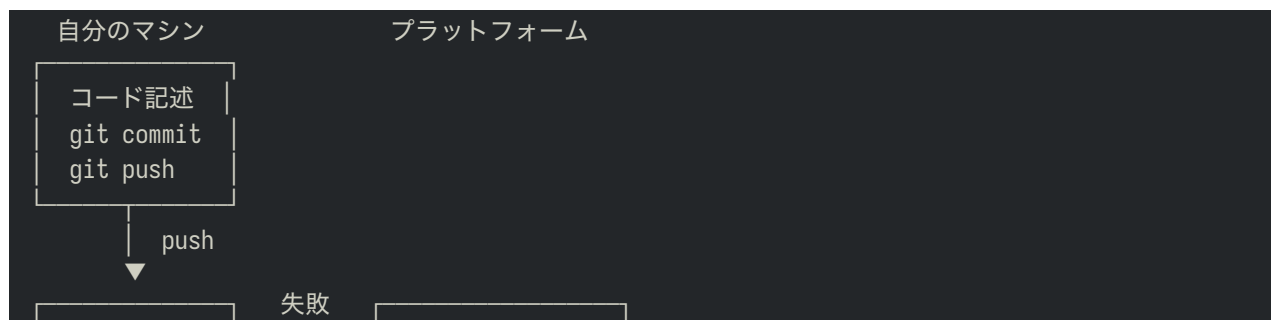
1 つ選んだら、デプロイはたいてい数個のコマンドで済みます。

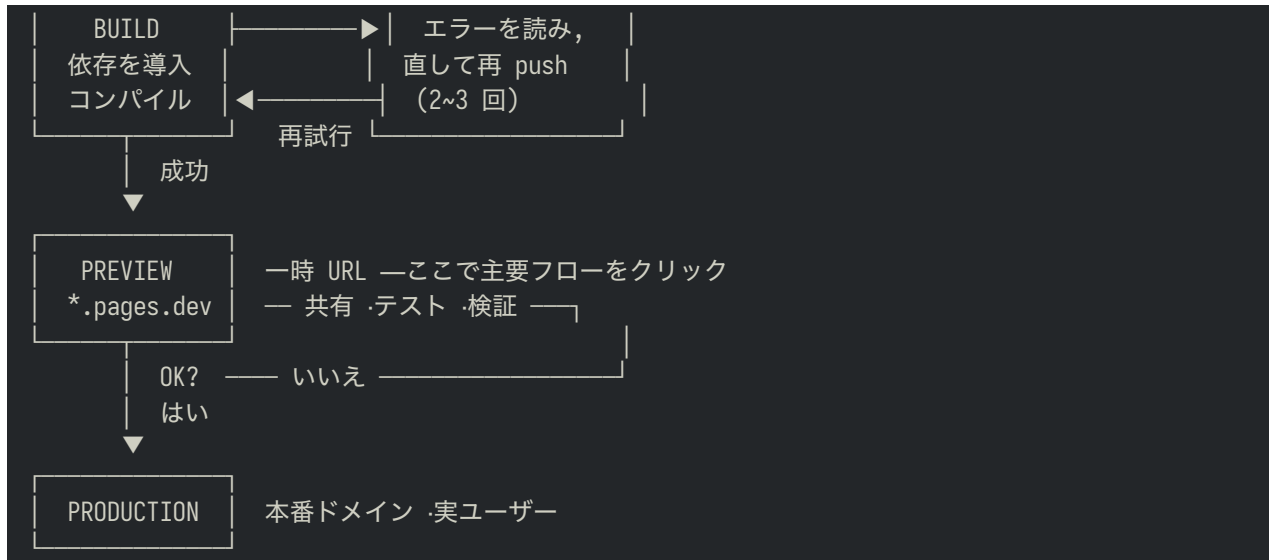
```
# 例: Wrangler 経由の Cloudflare Workers
npm install -g wrangler
wrangler login
wrangler deploy

# 例: Vercel
npm install -g vercel
vercel --prod
```

プラットフォームの設定 (*wrangler.toml*、*vercel.json* など) は AI に生成させ、最初のデプロイで出るエラーを説明させましょう。最初のデプロイはほぼ必ず何かしらのエラーを出します。ビルドコマンドの不足、出力ディレクトリの間違い、Node バージョンの不一致、ビルドが必要としているのに見つからない環境変数 (コードの外側からアプリに渡される、名前付きの設定値のことです) などです。どれもあなたが馬鹿なことをしたという意味ではなく、プラットフォームにもう一つ伝える必要があるという意味です。エラーをそのまま貼り戻して修正を頼み、さらになぜ起きたかも尋ねて、次は驚かされないようにしましょう。2、3 回のやり取りで、一時的な URL でライブになります。祝う前にその URL を開いて、コアのフローをクリックしてみましょう。「デプロイが成功した」と「アプリが動く」は同じ主張ではありません。

すべてのデプロイは同じパイプラインを歩きます。プレビューのステップがあなたのセーフティネットです。すべての変更が、本物のユーザーに触れる前に、クリックして確認できる自分専用の URL を持つのです。





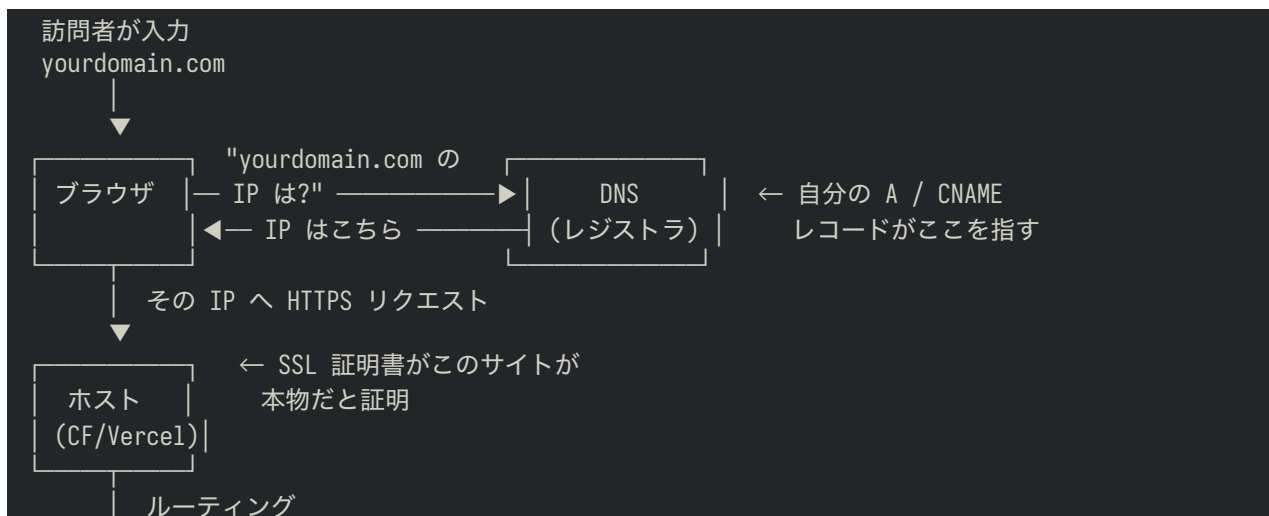
カスタムドメインと DNS

*.workers.dev や *.vercel.app の URL はテストには十分ですが、自分の名前が欲しくなります。流れは次のとおりです。

1. ドメインを買う (Namecheap、Cloudflare Registrar、Porkbun のいずれか)。
2. その DNS をホストに向ける。これはたいてい CNAME か A レコードを追加するか、ドメインのネームサーバーそのものを変更することを意味します。
3. ホストのダッシュボードでドメインを追加し、SSL 証明書 (そのサイトが本当にあなたのものだと証明し、鍵マークと HTTPS (ハイパーテキスト転送プロトコルの安全な暗号化版です) を有効にしてくれるデジタルの身分証です) を自動発行させる。

レコードが魔法のように感じられなくなるよう、手早いメンタルモデルを持ちましょう。**A レコード**は名前を IP アドレスへ直接向けます。**CNAME** は一つの名前を別の名前へ向けます (ホストの IP が知らぬ間に変わりうる時に便利です)。そして**ネームサーバー**はドメイン全体をプロバイダーに引き渡し、すべてのレコードを代わりに管理させます。たいていの現代のホストは CNAME か完全なネームサーバー切り替えを好みます。SSL 証明書とエッジルーティングを自分で管理したいからです。

訪問者のリクエストがたどる鎖全体と、あなたが設定した各部品がその中のどこに住んでいるかは、次のとおりです。



▼

アプリ

ページを配信

DNS は初心者にとって最も混乱する部分です。その大半は専門用語と反映の遅延のせいです。AI に翻訳させましょう。こう尋ねます。「私のドメインは *Namecheap* で登録していて、*Cloudflare Pages* でホストしています。どのレコードをどんな値で追加するか、そしてうまくいったかどうか確認するかを正確に手取り足取り教えて」。それから、ブラウザを更新するだけでなく、自分でも手早くチェックして結果を確認しましょう。

```
# 世界が実際に見ているレコードを調べる
dig yourdomain.com
nslookup yourdomain.com

# 証明書が有効で期限切れでないか確認する
curl -I https://yourdomain.com
```

DNS の変更が反映されるまでには、数分から 1 日かかることがあり、その中で最大の嘘つきはあなた自身のブラウザのキャッシュです。インターネットの他の部分がとくに更新された後も、平気で古い答えを見せ続けます。すぐにライブにならなくても、それは正常です。何かが壊れたと決めつける前に、携帯のモバイル回線で確認するか、オンラインの「DNS 伝播」チェッカーを使ってみましょう。

環境変数とシークレット

あなたのコードは API キー、データベース URL、トークンを必要とします。これらは**絶対に**リポジトリにコミットしてはいけません。シークレットがいったん git 履歴に入ると、事実上永遠に公開されたものになります。後のコミットでその行を消しても履歴からは消えませんし、ポットは新しい公開リポジトリを数分以内に漁って漏れた鍵を探します。2 つのルールが大いに助けてくれます。

- ・シークレットはローカルでは `.env` ファイルに保管し、`.env` が `.gitignore` に入っていることを確認する。
- ・本番用には、同じ値をホストのダッシュボード (または CLI (コマンドライン・インターフェース。クリックする代わりにコマンドを打ち込む、テキスト入力画面です) 経由) で環境変数として設定する。

```
# .env (ローカル専用 —絶対にコミットしないこと)
DATABASE_URL=postgres://localhost:5432/app
STRIPE_SECRET_KEY=sk_test_xxx
SESSION_SECRET=change-me

# ホスト側で本番の同等物を設定する
wrangler secret put STRIPE_SECRET_KEY
vercel env add STRIPE_SECRET_KEY production
```

すべての変数の名前 (値ではない) を列挙した `.env.example` をコミットしておきましょう。そうすれば未来の自分と AI の両方が、アプリが何を期待しているかを知れます。上の `sk_test_` という接頭辞に注目してください。テスト用キーと本番用キーをはっきり区別できるようにし、本番用キーをチャットに貼りかねないローカルの `.env` には決して置かないこと。もしチャットにシークレットを貼ったり、誤ってコミットしたりしたら、漏洩したものとして扱い、即座にローテーションしましょう。プロバイダーのダッシュボードで新しい鍵を生成し、あらゆる場所で更新し、古い鍵を無効化します。ローテーションには 5 分かかりますが、漏れたデータベース URL はデータベースの中身すべてを失わせかねません。

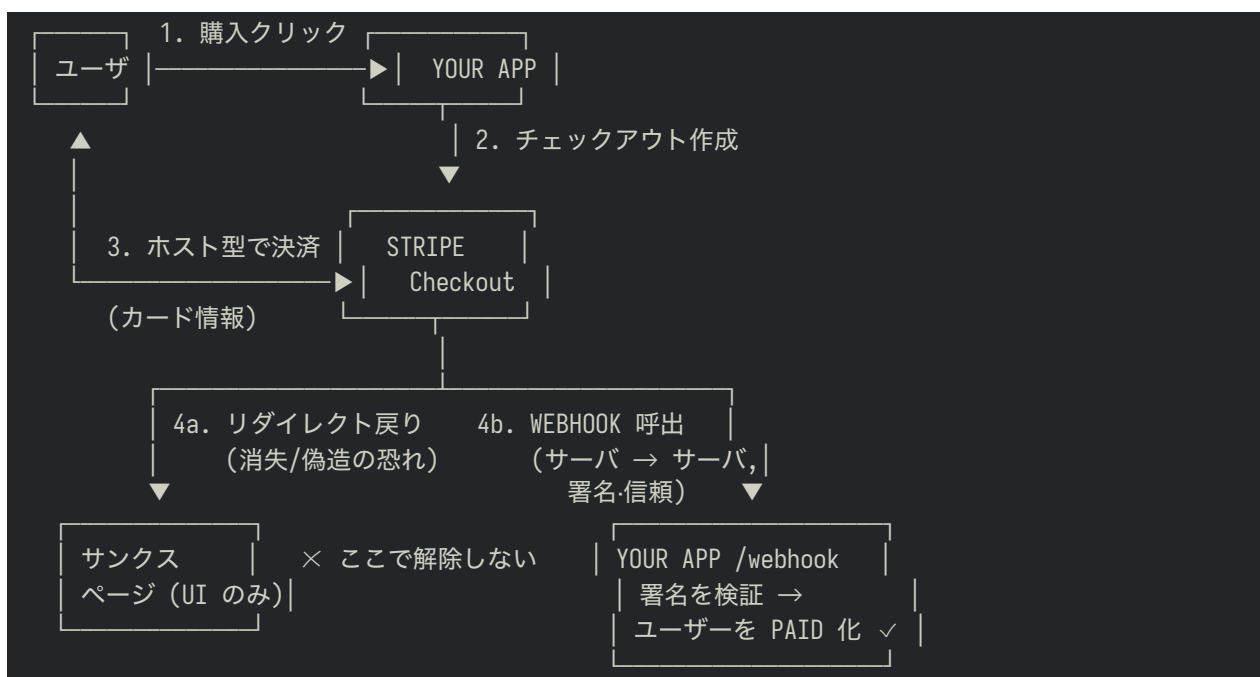
決済の接続

お金を受け取りたいなら、個人開発者にとっての標準的な道は Stripe です (売上税を代わりに処理してほしいなら Lemon Squeezy / Paddle。登録代行業者 (merchant of record) として、取り分は大きくなりますが税の頭痛

を消してくれます)。仕組みは反復的でよく文書化されているので、委任するのに理想的です。必要な部品は次のとおりです。

- **チェックアウトフロー**。Stripe Checkout が最速で、ホストされた決済ページを渡してくれるので、生のカード情報に触れずに済みます。
- Webhook は、あなたのアプリが「もう支払いは通った?」と何度も尋ね続ける代わりに、売上が確定した瞬間に決済会社のほうからアプリに電話をかけてくれるようなものだと考えてください。**Webhook** はまさにそれで、決済プロバイダーが呼び出すエンドポイント (あなたのサーバー上の URL) であり、決済が実際に成功したことをアプリが知るためのものです。ここは初心者が省いてしまい、なぜ注文が処理されないのかと不思議がる部分です。あなたのサイトに戻るリダイレクトは閉じられたり、ブロックされたり、偽装されたりします。サーバー対サーバーの真実の源は Webhook です。リダイレクトだけを信用してはいけません。Webhook を信用しましょう。
- **権限を保存する方法**。データベースでユーザーを「支払い済み」とマークします。そうすればリロードしても新しい端末でも、買ったものがちゃんと見えます。

肝心なのはどの矢印を信用するかです。あなたのサイトへ戻るユーザーのリダイレクトは失われたり偽装されたりします。製品を実際にアンロックするのは、サーバー対サーバーの線である Webhook です。



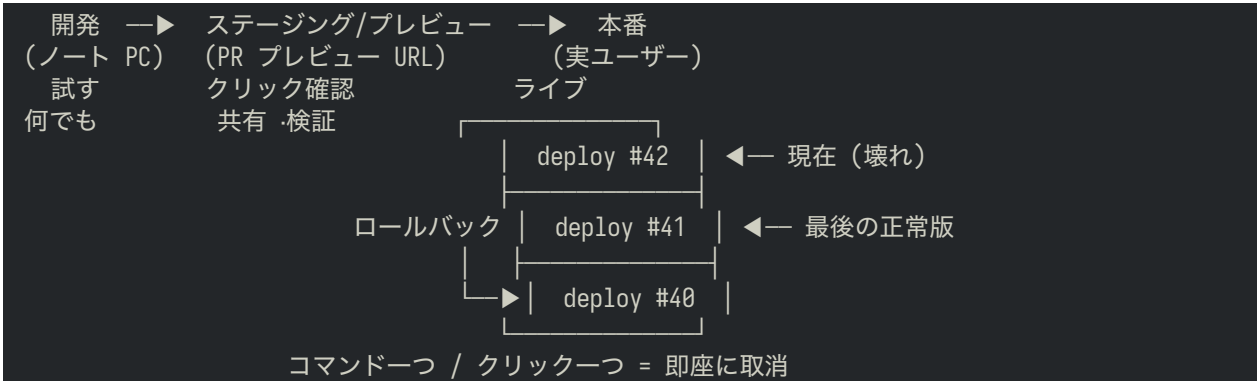
この3つすべてをAIにスキュフォールドさせ、Webhookの署名検証を説明させましょう。そこが唯一のセキュリティに敏感なステップだからです。それがなければ、あなたのWebhook URLを見つけた誰もが「決済成功」イベントを偽装して、製品をタダで解放できてしまいます。本番キーに切り替える前に、Stripeのテストモードでテスト用カード番号(4242 4242 4242 4242が有名です)を使ってすべてをテストし、Stripe CLIを使ってWebhookイベントをローカルマシンに転送し、発火する様子を見守りましょう。テストモードでの予行演習を省いてはいけません。アプリを通る最初の本物のお金は、すでに何十回も動くのを見た取引であるべきです。

ステージングとロールバック

優雅に立て直せるローンチと、深夜のパニックに変わるローンチを分けるのは、二つの習慣です。変更を本物のユーザーに届ける前に安全な場所でテストすること、そして悪いデブロイを即座に取り消せること。

ステージング環境とは、アプリの二つ目のコピーにすぎません。同じコード、別のURL、別の(あるいは使い捨ての)データベースで、本番の前に試す場所です。たいていのホストはこれをほぼ無料にしてくれます。すべてのプルリクエストが自動で自分専用の**プレビュー URL**を得るので、マージ前に変更をクリックして確認し、誰かと共有できます。データベースのマイグレーションのように簡単には戻せないものには特に使いましょう。

三つの環境を、変更がよじ登るはしごとして思い描いてください。ロールバックは、本番を最後の正常な段へ一気に落とすロープです。



ロールバックはアプリ全体に対する元に戻すボタンです。これらのプラットフォームは過去のすべてのデブロイを保持するので、最後の正常なバージョンに戻すのはたいていコマンド一つかダッシュボードのクリック一つです。

```
# Cloudflare Pages: 正常と分かっているビルドを再デブロイする
wrangler pages deployment list
wrangler pages deployment rollback <DEPLOYMENT_ID>

# Vercel: 以前のデブロイを本番へ昇格させる
vercel rollback <DEPLOYMENT_URL>
```

このコマンドは、必要になったときではなく、必要になる前に知っておきましょう。パイプコーディングの速さの肝は、出荷が安いことです。しかし安い出荷は、出荷の取り消しも同じくらい安いときにだけ安全に感じられます。

モニタリングとエラートラッキング

人々があなたのアプリを使い始めたら、いつ壊れたかを知る必要があります。理想的には、彼らに言われる前です。軽量なスターターキットです。

- **エラートラッキング:** Sentry は例外をスタックトレースとともに捕捉してグループ化し、どのユーザーのどの行が各エラーを引き起こしたかを教えてくれます。インストールは数行です。
- **稼働監視:** 無料のピンガー (UptimeRobot、Better Stack) が数分おきにサイトを叩き、ダウンしたときにメールで知らせてくれます。
- **ログ:** ホストがログをどこに置いているか (wrangler tail、Vercel ダッシュボードなど) を把握して、何かおかしいときに調査できるようにする。

AI に Sentry を追加させ、基本的なヘルスチェックのエンドポイントを接続させましょう。200 OK を返してデータベースに到達できるか確認する /health のような小さなルートで、稼働監視がそこを ping できます。目標はエンタープライズ級の可観測性ではありません。単に、怒りのツイートで障害を知らされないようにすることです。

パフォーマンスとキャッシュ

遅いアプリはユーザーを失い、検索でも順位が下がります。初日からマイクロ最適化する必要はありませんが、安価な勝ち手は掴んでおきましょう。

- 自分のファイルのコピーを世界じゅうの倉庫に置いておき、訪問者それぞれに最も近い倉庫から届けるようなものを想像してください。それが **CDN** (コンテンツ・デリバリー・ネットワーク) です。静的アセットはこれ経由で配信しましょう (たいていのホストは自動でやってくれます)。
- 頻繁には変わらない、コストの高いレスポンスやデータベースクエリを**キャッシュ**する。

- 画像を圧縮して遅延読み込みする。JavaScript は少なく配信する。
- Chrome DevTools で **Lighthouse** を実行し、レポートを AI に渡して優先順位付きの修正をもらう。

こう尋ねましょう。「これが私の *Lighthouse* レポートです。最も労力が少なく、最も影響の大きい変更を 3 つ教えて」。何を最適化するかは、当て推量ではなく計測に基づかせましょう。遅く感じるという理由で間違ったものを一日中最適化してしまうのは本当によくあることです。本当のコストが、インデックスのないクエリーつや、圧縮し忘れた 4 MB のヒーロー画像だったときに。

ローンチ前チェックリスト

何かを発表する前に、このリストを歩きましょう。記憶を頼らず、各項目を AI に検証してもらいましょう。

- ☐ アプリがクリーンにデプロイされ、本番 URL が読み込まれる
- ☐ カスタムドメインが有効な HTTPS で解決される
- ☐ すべてのシークレットが本番で設定済み。リポジトリにはひとつもコミットされていない
- ☐ `.env` が `gitignore` され、`.env.example` が最新である
- ☐ 決済をテストモードでエンドツーエンド (Webhook を含む) でテストした
- ☐ サインアップ、ログイン、コアの正常系がすべて本番で動く
- ☐ エラートラッキングと稼働監視が稼働している
- ☐ ロールバックの仕方を知っていて、一度テストした
- ☐ モバイルレイアウトを実機で確認した
- ☐ 基本的なプライバシーポリシーと利用規約のページが存在する
- ☐ データベースのバックアップが有効になっている
- ☐ ユーザーがあなたに連絡する明確な手段 (メールまたはフォーム) がある

ローンチした後

リリースは始まりであって、終わりではありません。最初のバージョンは仮説です。実際のユーザーが実験です。エラーダッシュボードを見守り、すべてのメッセージを読み、小さな修正を素早くリリースしましょう。最初の摩擦のサインですべてを作り直したくなる衝動に抵抗しましょう。尖った角を直し、人々が実際に使うものに集中投資し、残りは待たせておきましょう。

ここで最も強力な習慣は、ぐるっと締まったループです。問題やアイデアに気づき、それを AI に説明し、変更をリリースし、結果を見守る。機械的な作業が委任されているので、何週間もではなく何時間単位で反復できます。そのスピードこそが (どんな単一の機能でもなく) バイブで作ることの真の優位性です。あなたはライブになりました。さあ、進み続けましょう。

まとめと練習

重要なポイント

- 出荷は一瞬ではなくワークフローだ。デプロイし、観察し、修正し、繰り返す—そしてロールバックは一度テスト済みのものにしておく。
- 何かを告知する前に出荷前チェックリストを通し、記憶に頼らず AI に各項目を検証させる。
- 計測に最適化を導かせよう—「遅い気がする」という勘より、Lighthouse レポートが常に勝る。
- シークレットはプロダクション設定に置き、リポジトリには絶対置かない。`.env` は `gitignore` し、`.env.example` は最新に保つ。
- 最初のバージョンは仮説であり、実ユーザーが実験だ。小さく速い修正で反復しよう。

試してみよう

デプロイ済みのアプリ (または今すぐ小さなものをデプロイ) を選び、出荷前チェックリストを一行ずつ照合する。正直にチェックできない項目ごとに、失敗した項目を AI に貼り付け、その差を埋める最小の変更を求める。最後に、ホームページではなくライブサイトのディープリンクをクリックし、プロダクションが開発環境と同じように動くか確認して締めくくる。

この章のプロンプト

```
Here is my deployed app and its stack: [describe app + hosting + services].  
Act as my launch reviewer. Walk this pre-launch checklist item by item  
and, for each, tell me how to verify it on the LIVE site (not in dev):  
- production URL loads over valid HTTPS on the custom domain  
- all secrets set in production; none committed to the repo  
- core happy path (sign-up, login, main action) works in production  
- payments tested end-to-end including the webhook (if any)  
- error tracking and uptime monitoring are live  
- I can roll back, and have tested it once  
- mobile layout works on a real phone  
- database backups are enabled  
For every item that fails, give me the SMALLEST change that fixes it.  
Do not assume anything passes — make me prove each one.
```

ケーススタディ: 0 からローンチまで



理論は役に立ちますが、バンプコーディングを教えてくれるのは、それが起きるのを目の当たりにすることに勝るものではありません。この章では、AI を指揮してエンドツーエンドで作りリリースした、小さいけれど本物の 4 つのプロジェクトを歩きます。どれもおもちゃではありません。それぞれにユーザーがいて、ドメイン名があり、お金か時間がかかっています。

それぞれについて、アイデア、AI に渡した仕様、選んだスタック、実際に局面を動かしたプロンプト、危うく頓挫しかけた障害、そしてどうやってライブになったかを見ていきます。これらはハイライト集ではなく、実行手順書 (runbook) として読んでください。重要なのは、これらの正確なプロジェクトそのものではなく、その手が繰り返されることです。4 つ目あたりになると、リズムが見えてきます。短い仕様、マネージドなスタック、証拠に基づくデバッグ、そして現実に対して試すローンチ。

ケーススタディ 1: 多言語のランディング + 決済ページ

アイデア

ある友人が、韓国へ移住する駐在員向けに 29 ドルの PDF ガイドを売っています。彼女は、世界中で速く読み込まれ、英語・韓国語・中国語に対応し、サーバーに一切触れずにカード決済ができる 1 枚のページが欲しいと考えていました。

仕様

コードを書く前に、仕様は 1 段落に収めました。

29 ドルのデジタル商品向けの 1 ページのマーケティングサイトを作って。
3 言語 (en/ko/zh)。ブラウザから自動検出するが、トグルで切り替え可能にする。1 つの「購入」ボタンで Stripe Checkout を開き、成功時にサンクスページへリダイレクトする。私が保守するバックエンドサーバーは無し。どこからでも速く読み込めること。

仕様が何を言っていないかに注目してください。フレームワーク無し、デザインシステム無し、アナリティクス無し、メール収集無し。そのどれもがもっともな機能だったでしょうし、そのどれもが最初のローンチを遅らせたでしょう。1 段落の仕様という規律は、その大部分が何かを削ぎ落とす規律です。

スタック

「私が保守するバックエンドは無し」と「世界中で速く」という条件は、エッジ CDN (コンテンツ・デリバリー・ネットワーク。あなたのファイルのコピーを世界じゅうのデータセンターに置いておき、訪問者それぞれが近くの一つから読み込めるようにする仕組みです) 上の静的ホスティングと、唯一シークレットを必要とすること (Stripe セッションの作成) のための小さなサーバーレス関数を、まっすぐ指示していました。私たちは Cloudflare Pages 上の静的サイトと、1 つの Pages Function を選びました。アーキテクチャ全体が一文に収まります—1 つの関数を呼び、その関数が Stripe と話す静的ページ—そしてその単純さこそが、後でデバッグ可能にしたものでした。

システム全体は三つの箱で、シークレットに触れるのは真ん中の一つだけです。



となったプロンプト

広い指示から始め、AI に構造を提案させました。

静的なランディングページ (素の HTML/CSS/JS、フレームワーク無し) を en/ko/zh の言語トグル付きでスキヤフォールドして。文言は言語をキーとした 1 つの JS オブジェクトに格納する。デフォルトは navigator.language から検出する。index.html 1 つと main.js 1 つに収めること。

それから、シークレットが存在する決済パスです。

```
/api/checkout に Cloudflare Pages Function を追加して。29 ドルの単一商品の Stripe Checkout セッションを作成し、リダイレクト URL を返す。STRIPE_SECRET_KEY は環境変数から読み、絶対にハードコードしない。購入ボタンはこの関数に POST し、返ってきた URL へ window.location する。
```

「絶対にハードコードしない」という一句は、その居場所をきちんと得ています。デフォルトに任せると、AI は例を「動く」ように見せるためにプレースホルダーのキーを平気でインラインに置き、プレースホルダーはやがて本物のキーとなって出荷される傾向があります。プロンプトの中で一度声に出して言うほうが、レビューで捕まえるより安上がりです。

障害

最初のライブテストは失敗しました。購入ボタンは何も起きず、ブラウザのコンソールに CORS (クロスオリジン・リソース・シェアリング。どのサイトがどのサイトを呼び出してよいかについてのブラウザのルールです) エラーが出ました。私たちは推測しませんでした。正確なエラーを貼り戻したのです。

```
購入をクリックすると次のログが出る: 「Access to fetch at '/api/checkout' blocked by CORS policy.」 関数とページは同じドメインにある。これが関数のコード: [貼り付け]。実際の原因は何?
```

AI は即座に見抜きました。関数は Stripe の URL を JSON (ジャバスクリプト・オブジェクト・ノーテーション。プログラム同士が構造化データをやり取りするのに使う、読みやすい簡単なテキスト形式です) で返していましたが、関数の OPTIONS プリフライトが処理される前に fetch が行われていたのです。なぜなら、私たちが静的ページと関数を誤って 2 つの別々の Pages プロジェクトにデプロイしていたからでした。本当の修正はコードではなく、デプロイのトポロジーでした。関数を同じプロジェクトの /functions ディレクトリに移動すると、エラーは消えました。

この教訓は噛みしめる価値があります。症状はコードにありましたが (fetch から出た CORS エラー)、原因はインフラにありました (1 つではなく 2 つのプロジェクト)。もし構成を貼り付けずに AI に「CORS エラーを直して」と任せていたら、AI は Access-Control-Allow-Origin ヘッダーを付け足したでしょう—コンパイルは通り、症状を覆い隠し、本当のバグを生かしたままにする修正です。文字どおりのエラーとともに周辺のコンテキストを貼り付け、自分の構成に合わない最初のもっともらしい説明を鵜呑みにしないこと。

ローンチ

私たちは本番の Stripe キーを暗号化された環境変数として Pages のダッシュボードに追加し (リポジトリには絶対に入れない)、彼女のドメインの DNS (ドメイン・ネーム・システム。名前をサーバーに対応づけるインターネットの住所録です) を Pages に向け、カードで本物の 29 ドルのテスト購入を 1 回実行してから返金しました。失敗パスもわざとクリックしてみました—拒否されるテストカード—彼女が課金されず、空白の画面ではなくまともなメッセージを見られることを確認するためです。午後のうちにライブになりました。彼女はその晩に最初の売上を上げました。

ケーススタディ 2: 認証 + データベース付きの小さな SaaS ツール

アイデア

あるフリーランスのデザイナーが、クライアントごとに請求対象の時間を記録し、月次で CSV (カンマ・セパレーテッド・バリューズ。どんな表計算ソフトでも開ける、プレーンテキストの表計算ファイルです) をエクスポートできる、プライベートなダッシュボードを欲しがっていました。凝ったものではありませんが、アカウント (彼女のデータが彼女だけのものになるように) と永続化が必要でした。

仕様

認証とデータベースはリスクを高めるので、仕様は境界についてより具体的になりました。

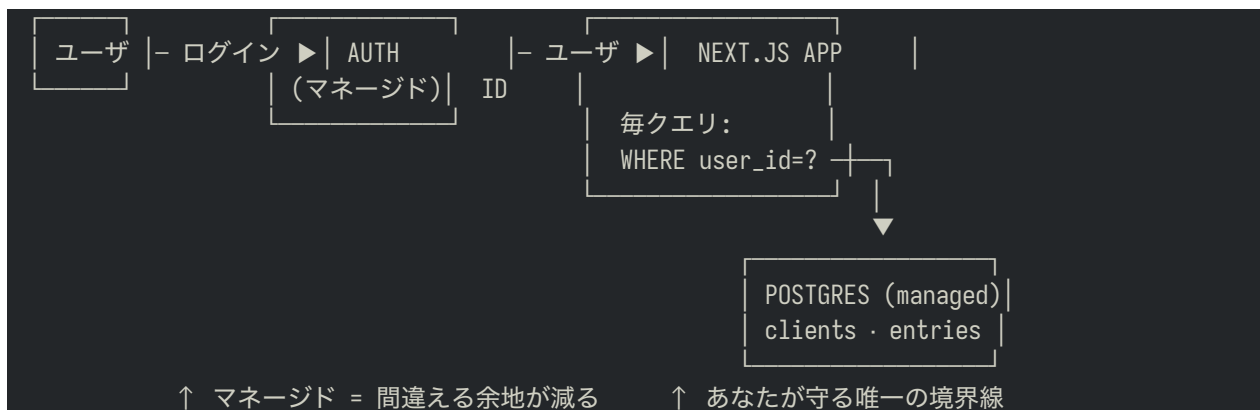
ログイン後の Web アプリで、ユーザーは次のことができる：メールでサインアップ/ログインする、クライアントを作成する、時間エントリ（日付、クライアント、時間数、メモ）を記録する、月で絞り込んだエントリのテーブルを見る、その月を CSV としてダウンロードする。各ユーザーは自分のデータしか決して見られない。モバイル対応。

「各ユーザーは自分のデータしか決して見られない」は UX の一行に見えます。それは実際には、アプリ全体のセキュリティモデルを十数語に圧縮したものです。仕様でそれに名前を付けておいたおかげで、AI が逸れるたびにそこへ立ち返って指し示すことができました。

スタック

個人開発者にとっての勝ち筋は、認証とデータベースが自分で書くコードではなくマネージドサービスであるスタックです。私たちはサーバーレスホストにデプロイする Next.js アプリを選び、ホストされた Postgres データベースと、メールログイン・セッション・パスワードリセットを代わりに処理してくれるドロップイン型の認証プロバイダーを組み合わせました。間違えうるコードが少ないほど、AI が私たちに代わって間違えうるコードも少なくなります。とりわけ認証は、AI と一緒に自前で組みたいとは思わない部類です。失敗の仕方が静かで、被害範囲が全員のアカウントに及び、マネージドプロバイダーは何百万回ものログインでエッジケースを叩かれてきたからです。

マネージドの部品は外側にあり、あなたが所有する唯一のルールは真ん中にあります—すべてのクエリが user_id でフィルタするので、あるユーザーが別のユーザーの行を読むことは決してできません。



となったプロンプト

重い処理は認証プロバイダー自身のテンプレートに任せ、その上にドメインロジックを重ねるよう AI を指揮しました。

私たちは「認証プロバイダー」の Next.js スターターを使っている。
2 つのテーブルを持つ Postgres スキーマを追加して: clients (id, user_id, name) と time_entries (id, user_id, client_id, date, hours, note)。すべてのクエリはセッションのログインユーザーの id で絞り込まなければならない (MUST)。マイグレーションと型付きのデータアクセス関数を生成して。

「user_id で絞り込まなければならない (MUST)」の一行は、プロジェクト全体で最も重要な文でした。私たちはデータに触れるほぼすべてのプロンプトでこの制約を繰り返しました。マルチユーザーアプリで最も恐ろしいバグは、あるユーザーが別のユーザーの行を見てしまうことだからです。打ち込んでいるときは反復が冗長に感じられます。しかしまさにその冗長さがあなたを救うのです。モデルは別々のプロンプトをまたいで、その制約がどれほど重要かを覚えていないからです。

エクスポートについては次のとおりです。

```
/api/export ルートを追加して。月（YYYY-MM）を受け取り、ログイン  
ユーザーのその月の time_entries をクライアント名と結合して取得し、  
CSV ダウンロードをストリームする。有効なセッションが無ければ  
リクエストを拒否する。
```

障害

テスト中、私たちは 2 つのアカウントを作り、アカウント B がドロップダウンでアカウント A のクライアントを見られることを発見しました。これこそ私たちが恐れていたバグです。AI に「直して」と頼むのではなく、まず問題を証明させました。

```
アカウント B がアカウント A のクライアントを見ている。コードベースの  
中で clients テーブルを読むすべてのデータベースクエリを見せて。  
そして各クエリについて、セッションの user_id で絞り込んでいるか  
どうか教えて。まだ何も直さないで — 監査だけして。
```

監査により、1 つのクエリ (ドロップダウンのローダー) が浮かび上がりました。それは制約を追加する前に書かれていて、すり抜けていたのです。私たちは欠けていた絞り込みを追加させ、それからガードを頼みました。

```
すべての読み取りが通る単一のヘルパーを追加して。セッションを受け取り、  
user_id の絞り込みを注入することで、今後どのクエリも忘れられない  
ようにする。既存のクエリをそれを使うようリファクタリングして。
```

これにより、一度きりの修正が構造的な保証に変わりました。それから私たちはその保証をテスト可能にしました。検証できないガードはただの願望にすぎないからです。

```
2 人のユーザーを作り、それぞれにクライアントを 1 つ作らせてから、  
ユーザー A のセッションがどのデータアクセス関数を通してても  
ユーザー B のクライアントを決して読めないことをアサートする  
テストを書いて。
```

教訓: AI がセキュリティバグを混入させたら、その実例にパッチを当てるだけでなく、その種類のミスを取り除くよう指揮すること。そして、誰かがそれを再び開けば大声で失敗するテストで、その種類を施錠してしまうこと。

ローンチ

私たちはテスト用に 1 か月分のデータを投入し、CSV をエクスポートし、表計算ソフトで開いて数値とエンコーディングが正しいことを確認してから、強力なデータベースパスワードを設定し、認証情報をローカルファイルからローテーションして取り除きました。サーバーレスホストにデプロイし、そのダッシュボードで本番の環境変数を追加し、彼女に URL を渡しました。彼女は本物のサインアップで自分でオンボーディングしました。ビルド全体が週末 1 回分でした。

ケーススタディ 3: 個人用の自動化ユーティリティ

アイデア

純粹に利己的なものです。毎朝その日のカレンダーの予定と天気を取得し、1 行の要約を整形して、個人の Telegram チャットに送るスクリプト。狙いは、コーヒーの前に 3 つのアプリを開くのをやめることでした。

仕様

```
私のタイムゾーンで朝 7 時に走るスケジュールジョブ。私のカレンダーから  
今日の予定を読み、私の街の予報を取得し、こんな Telegram メッセージを  
1 通送る: 「今日は 3 件、最初は 9:30。最高 24C、16 時以降に雨。」
```

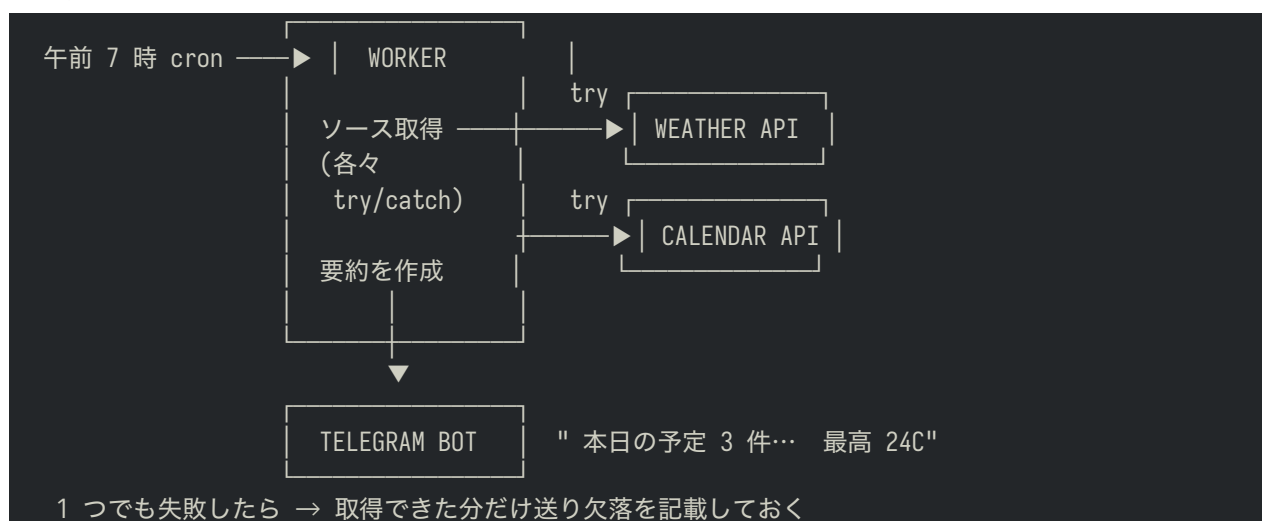
どれかのソースが失敗しても、手元にあるものは送り、何が欠けているかを
書き添える。

最後の一文 (クラッシュせずに優雅に劣化する) は、明示的に頼まなければならない種類の振る舞いです。AI はそれを前提してくれません。言わずにおくと、モデルが手を伸ばすデフォルトは「エラーで throw」であり、朝 7 時の cron ジョブにとってそれは、静かな朝と、なぜそうなったのかの手がかりの不在を意味します。

スタック

単一のスケジュール実行されるサーバーレス関数 (Cron でトリガーされる Worker)。動かし続けるマシンが不要だからです。全体は 1 つのファイルとスケジュール 1 つです。個人用ユーティリティでは、これは聞こえる以上に重要です。世話をするサーバーがあるものは何であれ、いずれあなたが支払いを忘れてしまい、プロジェクトは失敗ではなく放置で死ぬのです。

一つのトリガーが二つのソースへ広がり、一つのメッセージへ集まります。各ソースを包む try/catch が、一度の失敗が実行全体を殺す代わりに優雅に劣化させてくれます。



となったプロンプト

Asia/Seoul の朝 7 時に cron でトリガーされる Cloudflare Worker を書いて。天気 API とカレンダー API(両方のトークンはシークレットとして渡す) を呼び、1 行の要約を組み立て、Telegram Bot API に POST する。1 つの失敗がメッセージを潰さないよう、各外部呼び出しを try/catch で囲む。すべてのトークンは env から読む。

朝 7 時を待たずに文言を反復したくなったときは、こうしました。

手動トリガーを追加して: Worker が GET リクエストで ?test=1 というクエリパラメータ付きで叩かれたら、同じロジックを即座に実行し、送信する代わりにメッセージのテキストをレスポンスで返す。これをシークレットトークンの背後に置いて、私だけがトリガーできるようにする。

そのテストトリガーは、過酷なフィードバックループから私たちを救ってくれました。1 日 1 回ではなく、必要となときに出力を見られたのです。これは盗む価値のある一般的な手です。コードがスケジュールで走るときはいつでも、時計を待つと決める前に、今オンデマンドで走らせる方法を自分用に用意しておくこと。

障害

Telegram メッセージは届きましたが、時刻が間違っていました。予定が UTC で表示され、韓国時間ではなかったのです。天気は問題ありませんでした。私たちは原因を切り分けました。

カレンダーの時刻が 9 時間ずれている — Asia/Seoul ではなく UTC で表示されている。天気時刻（「16 時以降に雨」）は正しい。両方をこう整形している：[貼り付け]。カレンダーの整形だけを直して、なぜ天気のパスは最初から正しかったのかを教えてください。

AI は、天気 API（アプリケーション・プログラミング・インターフェース。あるサービスが、別のプログラムにデータを取得させるために公開している出入り口です）はすでにローカライズされた文字列を返していたのに対し、カレンダー API は UTC のタイムスタンプを返していて、私たちがそれをそのまま出力していた、と説明しました。AI はカレンダーの整形ステップにタイムゾーン変換を 1 つ追加しました。修正を「カレンダーのパスだけ」に絞ったことで、動いている天気のコードを書き換えるのを止められました。AI の修正が、問題なかったものを壊してしまうのはよくあるパターンです。そのプロンプトの後半—「なぜ天気のパスは最初から正しかったのかを教えてください」—は二重の役目を果たしました。推測ではなく実際の根本原因を教えてください。AI にその違いを言語化させることで、修正が 2 つのパスを 1 つの壊れたものへうっかり「統一」しないようにしたのです。

ローンチ

ここでの「ローンチ」は、単に Worker をデプロイし、cron スケジュールを設定し、API トークンを暗号化されたシークレットとして追加することを意味しただけでした。私たちは ?test=1 エンドポイントを何度か叩いてメッセージがうまく読めることを確認し、それから放置しました。以来、毎朝走っています—そして優雅な劣化の一行は、1 か月後に天気 API が障害を起こしたときその真価を証明しました。メッセージはやはり届き、ただ予報だけが欠けていたのです。

ケーススタディ 4: 手をかけずに済むコンテンツサイト

アイデア

ある趣味人が、Markdown ノートのフォルダ—何年もの旅行で書いたレストランレビューを、ログインする CMS も月額請求も無しに、検索可能な公開サイトに変えたいと考えていました。ノートを追加して、プッシュして、おしまい。

仕様

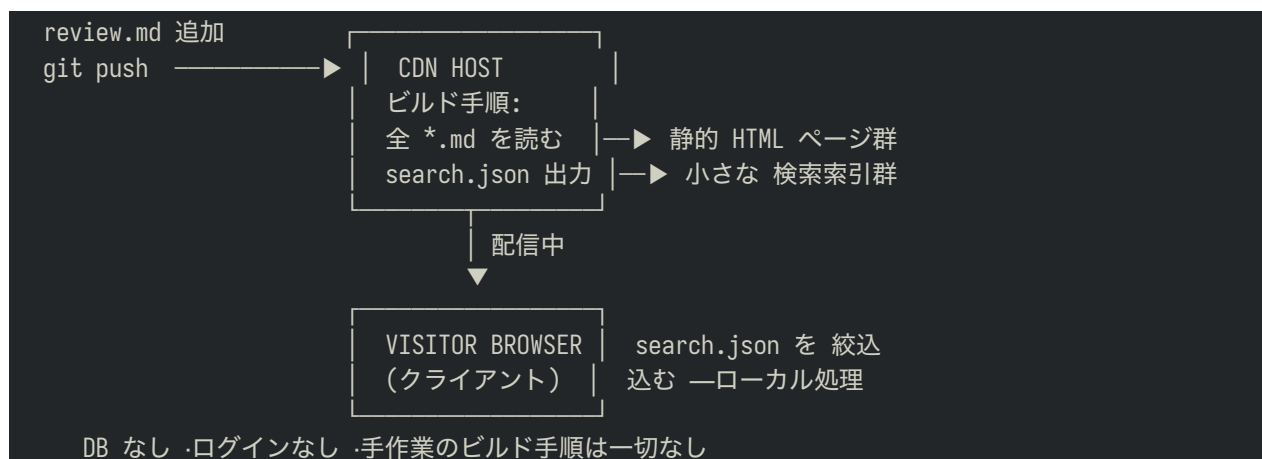
Markdown ファイルのフォルダから生成される静的サイト。レビュー 1 件につき 1 ファイル（フロントmatter に title, city, rating, body）。レビューを新しい順に並べるホームページ、レビューごとのページ、そして city か name でのクライアントサイド検索を作って。データベース無し、ログイン無し、私が手で実行するビルド手順無し。新しい .md ファイルをプッシュしたら、それが公開されること。

隠れた要件は最後の一文にあります。「私が手で実行するビルド手順無し」は、この仕様が出力そのものだけでなく、本当はワークフローに関するものだということを意味します—そしてワークフロー要件は、名前を付けなければ AI が飛ばす部類です。動いているアプリには現れないからです。

スタック

ビルド時に Markdown を読む静的サイトジェネレーターを、Git トリガーのビルドを備えた CDN ホストにデプロイしました。リポジトリにプッシュすると、ホストが再ビルドし、CDN が配信します。検索はビルド時に生成される小さな JSON インデックス上でクライアントサイドに行うので、サーバーも無くクエリのコストも無い—レビュー数百件では妥当なトレードオフであり、見つけたものではなく私たちが名前を付けた意図的な選択です。

ワークフロー全体はたった一本の矢印です。プッシュが、公開するビルドをトリガーします。訪問者自身のブラウザがインデックスをフィルタする以外、リクエスト時には何も動きません。



となったプロンプト

私たちはどんな UI よりも先に、AI をデータの形に錨で固定しました。

/reviews 内のすべての .md ファイルを読む静的サイトをセットアップして。各ファイルのフロントマターには title, city, rating(1-5), date がある。ビルド時にそれらをすべてパースして並んだリスト（新しい順）にし、生成して：それらを並べるホームページと、/reviews/<slug> にレビューごとのページ。どんなスタイリングよりも先に、データ読み込みのコードを見せて。

「どんなスタイリングよりも先に」は意図的なレバーです。一度に全部を頼めば、信用できないデータに配線された美しいページが手に入ります。データ層を先に頼めば、たった 1 ピクセルが注意を逸らす前に土台を検証できます。それから、安く保つよう範囲を絞った検索です。

ビルド時に、すべてのレビューについて {title, city, slug} を持つ search.json を生成して。ホームページに検索ボックスを追加し、見えているリストを city か title のマッチでクライアントサイドに絞り込む — ネットワーク呼び出し無しで、すでに読み込まれたインデックスをただ絞り込むだけ。JS は 50 行未満に収める。

障害

ビルドはローカルでは通りましたが、デプロイされたサイトは個々のレビューページすべてで 404 を返しました。ホームページは動き、レビューごとのページは動きませんでした。私たちは理屈をこねるより、症状と設定を貼り付けました。

ホームページはライブで動くのに、/reviews/<slug> のページはローカルの dev では動くのに本番では全部 404 になる。これが私のビルド出力ディレクトリの一覧と、ホストのデプロイ設定：[貼り付け]。なぜローカルの dev はこれらのページを配信するのに本番はしない？

AI はそれを、dev サーバーのルーティング（パスを動的に解決する）と静的ホスティング（物理的に存在するファイルだけを配信する）の違いに突き止めました。ビルドがレビューページを間違った出力フォルダに生成していたので、それらは一度もアップロードされなかったのです — dev は、本番にはできないルートをルーターが偽装することで

バグを隠していました。

私たちは出力ディレクトリをホストが実際にデプロイするものに合わせ、ページが現れ、ローンチのチェックリストに一行を加えました。ホームページだけでなく、ライブサイトでディープリンクをクリックすること。より広い教訓は、「dev で動く」と「デプロイして動く」は別の主張であり、静的エクスポートこそその 2 つが分かれる地点だということです—dev サーバーは、実際のファイルベースのホスティングが決してそうでない仕方、ルートに対して寛容なのです。

ローンチ

私たちは本物のレビューを 1 つプッシュし、ホストが再ビルドするのを見守り、その特定のレビューへのディープリンクをライブのドメインで読み込みました—dev が嘘をついていたまさにそのケースです。それからインデックスがそれを拾ったか確認するため、その city を検索しました。次のレビューを追加するのは一行のコミットでした。それがすべての要点でした。ローンチは一瞬の出来事ではなく、私たち抜きで動き続けるワークフローだったのです。

4 つに共通すること

- ・ **仕様が最初に来て、それは短かった。** 文書ではなく 1 段落。しかし重要な制約に名前を付けていた。言語、「自分のデータしか見られない」、「手元にあるものは送る」、「ファイルをプッシュしたら公開される」—動いているアプリには決して現れないワークフローの制約も含めて。
- ・ **シークレットはリポジトリに一切触れなかった。** どのプロジェクトも、キーをコードではなくダッシュボードで設定した環境変数や暗号化されたシークレットから読んだ。私たちは「絶対にハードコードしない」と声に出して言った。デフォルトはその逆だからだ。
- ・ **スタックは、間違えうるものが少なくなるように選んだ。** マネージドな認証、マネージドなデータベース、静的ホスティング、サーバーレス関数—どの選択も、AI が私たちに代わって壊しうるコードの一種類と、私たちが保守しなければならないものの一種類を取り除いた。
- ・ **障害は、AI に正確な証拠を与えることで解決された。** 文字どおりのエラー、実際のコード、デプロイ設定、本物の症状を与え、修正を狭く絞ることで、動いているコードを動いたままにした。二度、症状はコードにあったが原因はインフラにあった。
- ・ **ローンチは現実に対する本物のテストを意味した。** 実際の購入 (と拒否された購入)、2 つ目のユーザーアカウント、オンデマンドのトリガー、デプロイされたサイトのディープリンク。「コンパイルが通った」ではない。

始めるのに大きなアイデアは要りません。必要なのは小さなアイデア、1 段落の仕様、そしてタイプするのではなく指揮しようとする意志です—そして、完成だと呼ぶ前にそれを現実に対して試す忍耐です。

まとめと練習

重要なポイント

- ・ 仕様が先で、しかも短かった—動いているアプリには決して現れないワークフロー制約を含め、重要な制約を挙げた一段落だった。
- ・ シークレットはリポジトリに触れなかった。どのプロジェクトも環境変数か、ダッシュボードで設定した暗号化済み設定からキーを読んだ。
- ・ スタックは間違えにくいように選ばれた—マネージド認証、マネージドデータベース、静的ホスティング、サーバーレス関数。
- ・ 障害は AI に正確な証拠を与えて解決した—実際のエラー、実際のコード、デプロイ設定—そして修正範囲を狭く絞った。
- ・ 出荷は「コンパイルできた」ではなく、現実に対する本物のテストを意味した: 実際の購入、2 つ目のアカウント、デプロイ済みサイトのディープリンク。

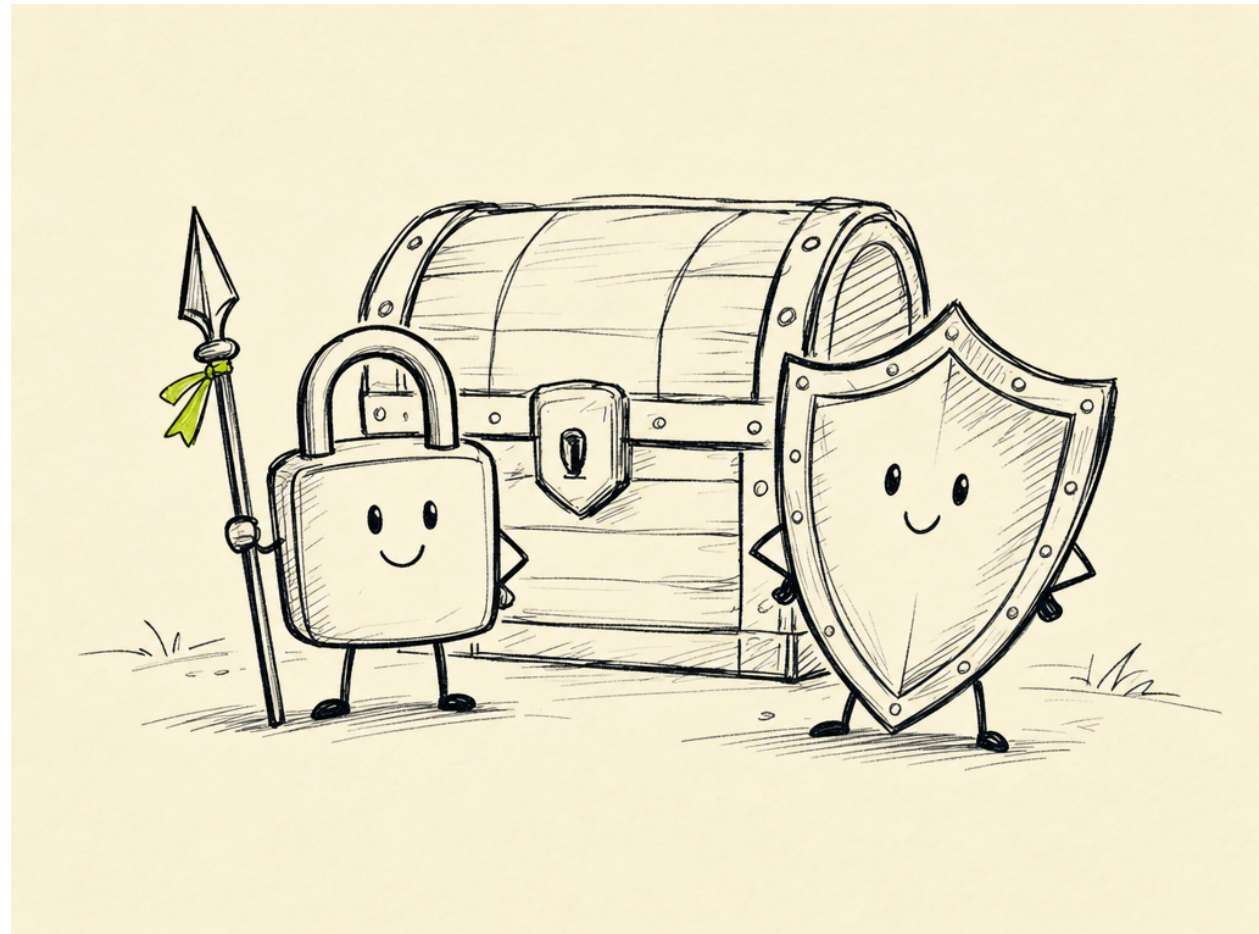
試してみよう

自分のアイデアを一つ選び、この 4 例のように一段落の仕様を書こう: 中心となる動作、各ユーザーが見られるデータ、そして少なくとも一つのワークフロー制約 (コンテンツがどう公開されるか、誰が何をできるか) を挙げる。次に、壊れるカスタムコードを減らすために頼るマネージドの構成部品を列挙する。一段落と短い箇条書き程度にとどめよう—それを超えて膨らんだら、過剰に仕様化している。

この章のプロンプト

```
I want to build: [one-paragraph description of the app].
Before any code, help me tighten this into a buildable spec.
1. Restate the single core action a user performs.
2. List the data each user can see — and what they must NOT see.
3. Name the workflow constraints that won't show up in the UI
   (e.g. "pushing a file should publish it", "only their own data").
4. Recommend a stack that MINIMIZES custom code: managed auth,
   managed database, static or serverless hosting where possible.
5. Flag the one part most likely to go wrong, and how we'll test it
   against reality (a real purchase, a second account, a deep link).
Keep the spec to a paragraph plus bullets. Push back if it's too big.
```

バイブコーダーのためのセキュリティ



セキュリティは、バンプコーディングの中でも静かに牙をむく部分だ。壊れた機能はすぐに表面化する——ボタンが動かない、だから直す。だがセキュリティの穴は、誰かがそれを見つけてあなたのユーザーのデータをフォーラムにばらまくその日まで、決して表面化しない。その頃には、そのバグを書いた AI はとっくになくなって、なぜ自分のパスワードが漏れたのかを千人に説明するはめになるのはあなただ。

この章は恐怖を煽るためのものではない。安全な小さなアプリを出荷するために、ペネトレーションテスターになる必要も、OWASP Top 10 を暗記する必要もない。必要なのは、ソフトウェアが侵入される数少ない手口を理解すること、AI が繰り返し生み出すパターンを見分けられること、そして出荷前にレビューのゲートを 1 つ追加することだ。哲学はこの本の他の部分と同じだ。**理解していないものは決して出荷しない**——そしてセキュリティは、理解していないことの代償が最も高くつく場所である。

AI が生成するコードがデフォルトで安全でない理由

AI はあなたが与えたプロンプトに最適化する。そしてあなたのプロンプトはほぼ必ず「動くようにして」と言っていて、「安全にして」とは決して言っていない。だから AI は、動く機能までの最短経路を生み出す。最短経路はたいてい安全でない経路だ——文字列連結で組み立てたクエリ、権限チェックのないエンドポイント (アプリへの番地付きの扉の一つ。アプリがリクエストに応答する単一の URL のことだ)、学習したサンプルがそうだったというだけの理由でそのまま貼り付けられたシークレット。

さらに悪いことに、AI はインターネット全体から学習している——その中には、それ自体が安全でなかった、10 年分のチュートリアルや放置されたりポジトリも含まれる。安全なコードは書くのが難しく、投稿されることも稀なので、安全でないコードのほうが学習データの中ではよりありふれている。「ログインフォーム」を頼めば、統計的にはインターネット上の平均的なログインフォームが返ってくる可能性が高く、そしてインターネット上の平均的なログインフォームには問題がある。

AI はまた、脅威の感覚をまったく持たない。ファイルアップロード機能を作る人間なら、ふと不安がよぎる——誰かが厄介なものをアップロードしたらどうしよう？ AI は何も感じない。それが鉄壁だろうと穴だらけだろうと、同じ自信でハンドラを書く。ためらいもなければ、「うーん、この部分はちょっと不安だな」もない。その不安は、あなた自身が供給しなければならない。

ここから得られる教訓は「AI のコードは危険だから使わない」ではない。「動く」と「安全だ」は別の 2 つの間いだという、そして AI に 2 つ目を答えさせようと強制しない限り、AI は 1 つ目しか答ええないということだ。

インジェクション: 入力がコードになるとき

インジェクションは最も古く、いまだに最もありふれた深刻なバグだ。ユーザー由来のデータが、データではなく命令として扱われたときに起こる。ほとんどのアプリにとって重要なのは 2 種類だ。

SQL injection ——SQL(ストラクチャード・クエリ・ランゲージ。アプリがデータベースにデータを問い合わせるために使う言語だ)——は、ユーザー入力がデータベースクエリに紛れ込むことだ。典型的な脆弱パターンはこうだ。

```
// VULNERABLE: ユーザーの入力がそのままクエリに貼り付けられている
app.get("/user", (req, res) => {
  const name = req.query.name;
  db.query(`SELECT * FROM users WHERE name = '${name}'`);
});
// もし誰かが name = '; DROP TABLE users; -- を渡すと
// クエリはテーブルを削除するコマンドになってしまう。
```

修正方法は**パラメータ化クエリ** (プリペアドステートメントとも呼ばれる) だ。クエリとデータを別々に送るので、データベースが一方を他方と取り違えることがない。

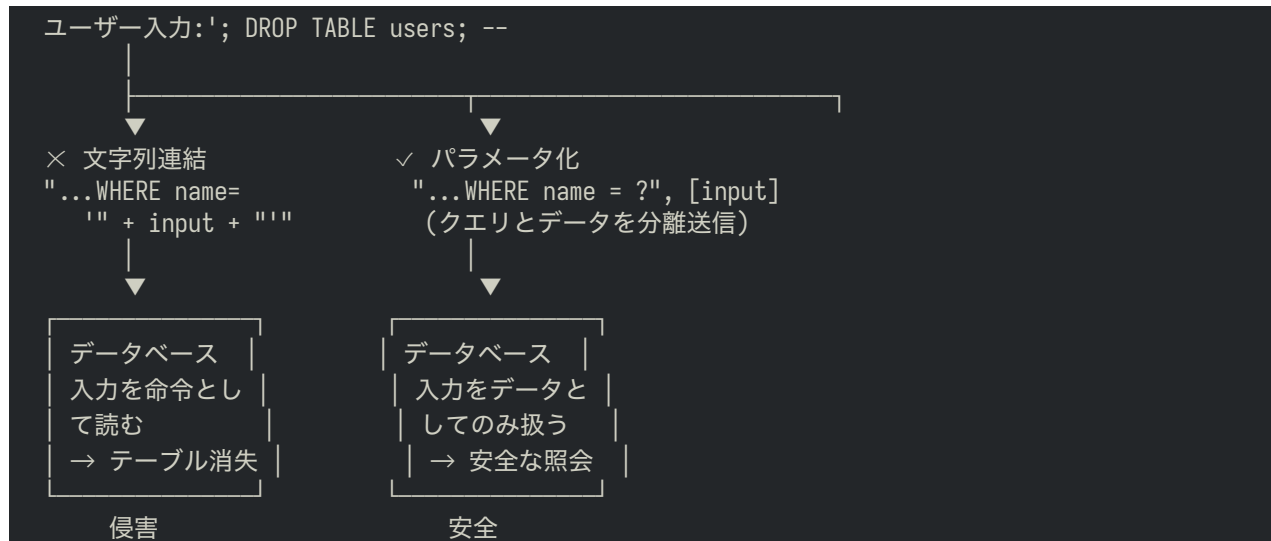
```
// SAFE: 値はパラメータとして渡され、決してコードとして扱われない
app.get("/user", (req, res) => {
  const name = req.query.name;
```



```
db.query("SELECT * FROM users WHERE name = ?", [name]);
});
```

ルールはこうだ。**クエリを文字列連結で組み立ててはいけない**。diff の中で、変数の周りに SQL を組み立てるバッククォートや + を見かけたら、手を止めてパラメータ化版を要求すること。

二つの経路は同じユーザー入力をデータベースに入れますが、データベースがその入力をコマンドと取り違えないようにするのは、片方だけです。



XSS (cross-site scripting) は、ブラウザにおける同じ発想だ。ユーザー入力を受け取って、それを生の HTML(ハイパーテキスト・マークアップ・ランゲージ。ブラウザに何を表示するかを伝えるコードだ) としてページに落とし込むと、攻撃者は <script> タグを注入でき、それがあなたの他のユーザーのブラウザで実行される——たとえばセッションを盗むために。修正方法はエスケープだ。ユーザーのコンテンツを HTML ではなくテキストとしてレンダリングする。React のようなモダンなフレームワークはデフォルトでエスケープしてくれる。これは素晴らしいことだが、AI が「動くようにする」ために dangerouslySetInnerHTML や innerHTML のような抜け道に手を伸ばすまでの話だ。それらは保護をバイパスする。diff の中でそうした呼び出しを見かけたら、疑うべきものとして扱うこと。

あらゆるインジェクションバグの背後にある統一原理はこうだ。**データとコードを分離し続けること**。ユーザー入力が入力クエリ・コマンド・テンプレート・HTML に渡るときは必ず、何かがそれをエスケープするかパラメータ化しなければならない。

シークレットと API キー: 露出したキーの罠

シークレットとは、アクセス権を与えるもの全般を指す。API(アプリケーション・プログラミング・インターフェース) キー、データベースパスワード、決済プロバイダのトークン、署名鍵。最もありふれていて——そして最も高くつく——パイプコーディングのミスは、これを漏らすことだ。

人々を絶えず捕らえる罠が 2 つある。

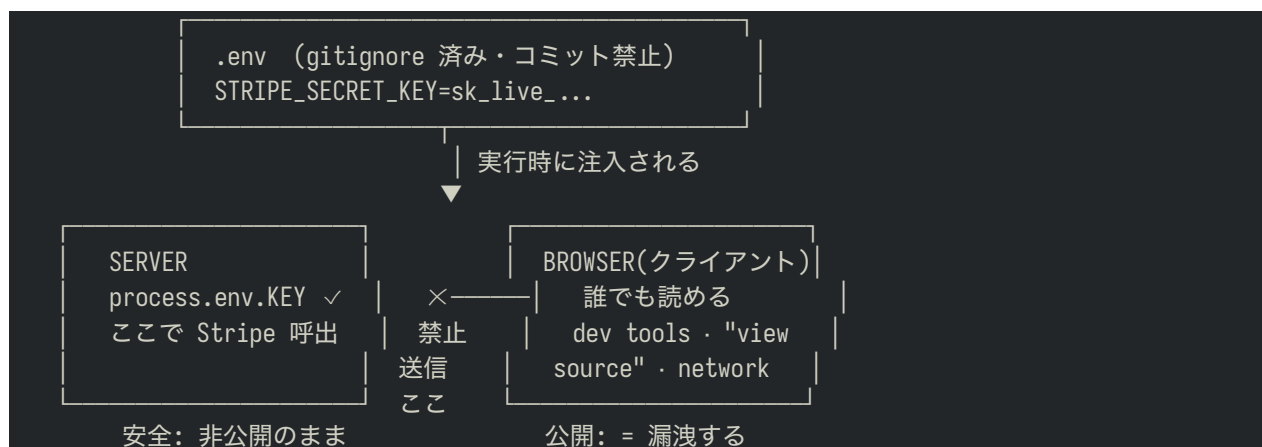
- **クライアントコードの中のシークレット**。フロントエンド (ブラウザで動く JavaScript) にあるものはすべて公開だ。ユーザーは dev tools を開いて読める。React コンポーネントから API を呼んでと頼まれた AI は、喜んでそこにあなたのシークレットキーを貼り付ける——そして今や、あなたのサイトを訪れた誰もがそれをコピーして請求額をつり上げられる。シークレットキーはサーバーに置くべきものであり、ブラウザに配信されるコードには決して置いてはいけない。
- **リポジトリの中のシークレット**。ファイルにハードコードされたキーは git にコミットされる。あとで削除しても、それは git の履歴に永遠に残り、ボットは push から数分以内にまさにこれを狙って公開リポジトリをスキャンする。

シークレットの正しい置き場所は**環境変数**だ——実行時にアプリに供給される値で、コードからは完全に切り離されている。

```
// VULNERABLE: キーがソースの中にあり、git にコミットされてしまう
const stripe = new Stripe("sk_live_51H8xQ2eZvK...");

// SAFE: キーは環境から読まれ、コードには決して書かれない
const stripe = new Stripe(process.env.STRIPE_SECRET_KEY);
```

シークレットには安全な住み家がただ一つあります——サーバー側で、環境から読み込むこと。ブラウザは公共の場です。そこへ出荷されたものは、開発者ツールを開く誰にでも読めます。



シークレットを1つでも書く前に、`.env`(および`.env.local`など)を`.gitignore`に追加すること。もしキーがコードや履歴に一度でも入り込んだら、それは焼かれたものとして扱う。ローテーションする(新しいものを生成して古いものを失効させる)——行を削除するだけでは不十分だ。古いキーはまだ有効で、まだ世に出回っているのだから。

認証と認可: 誰でも呼べるエンドポイント

この2つの言葉は似て聞こえるが、その違いこそが本当の侵害が潜む場所だ。

- **認証 (Authentication)** はあなたは誰か?——ログインし、身元を証明すること。
- **認可 (Authorization)** はあなたは何をしてよいか?——このログイン済みのユーザーが、このデータに対してこの操作を行ってよいかどうか。

AIは認証はそこそこ得意だ。ライブラリがほとんどを処理してくれる。AIが日常的に失敗するのは認可のほうだ。認可はあなたのアプリのルールに固有のもので、AIはそれを知らないからだ。教科書どおりの惨事はこうだ。

```
// VULNERABLE: ログイン済みかはチェックするが、誰なのかはチェックしない
app.get("/admin/export-all-users", requireLogin, (req, res) => {
  res.json(db.getAllUsers()); // ログイン済みなら誰でもここを叩ける
});
```

このエンドポイントは「保護されている」——ログインしていなければならない。だがどんなログイン済みユーザーでも、30秒前にサインアップした人を含めて、これを読んで全ユーザーのデータをダウンロードできる。認証はあり、認可はない。修正方法は、操作そのものに対して権限をチェックすることだ。

```
// SAFE: このユーザーが本当に管理者かを確認する
app.get("/admin/export-all-users", requireLogin, (req, res) => {
  if (!req.user.isAdmin) return res.status(403).send("Forbidden");
  res.json(db.getAllUsers());
});
```

```

要求 → [ 認証 (authn)
          " ログイン済?" ] → [ 認可 (authz)
                               " このユーザは
                               この操作可?" ] → 実行
                                              する

      ✓ 多くが実装
        する部分
      × よく欠落
        → ここが侵害
  
```

例 — `/admin/export-all-users`

```

ログインユーザ → [ auth ✓ ] → [ authz 検査なし ] → 全ユーザが流出
ログインユーザ → [ auth ✓ ] → [ isAdmin? → 403 ] → ブロック ✓
  
```

ファイルの取り扱いとアップロード

- ・ **型とサイズを検証する。**制限がなければ、誰かが 10GB のファイルや実行可能ファイルをアップロードする。誰でもリネームできる拡張子だけでなく、実際の中身をチェックすること。
- ・ **ファイル名を決して信用しない。** ../../etc/passwd のようなファイル名は、素朴なハンドラに意図したフォルダの外へ書き込ませてしまう (これを path traversal という)。提供された名前を使う代わりに、自前の安全な名前を生成すること。
- ・ **アップロードをアプリと同じオリジンから配信しないこと、そしてアップロードされたファイルがコードとして実行されないようにすること。**サーバーが実行するフォルダにユーザー提供のファイルが落ちることは、乗っ取りへの直通路だ。

依存関係のリスク: AI が発明したパッケージ

第二に、**未検証の依存関係全般**。追加するパッケージはすべて、あなたのアプリの全権限で動くコードだ。依存関係が増えるほど、バグとサプライチェーン攻撃の両方の攻撃面が増える。無名なものの長い尾よりも、数少ない、よく知られたライブラリを優先し、1 つ追加する前に「これに本当にパッケージが必要か、それとも 10 行で済むか?」と問うこと。

エコシステムの audit ツールを定期的に行うこと (npm audit、pip-audit など)——それはすでにインストール済みのものの中の既知の脆弱性を指摘してくれて、AI はそれが報告したものを直すのが得意だ。

セキュリティレビューのゲート

ここまでのすべてを「不安」から「プロセス」へと変える、たった 1 つの習慣がある。出荷する前に、**AI に自分のコードを攻撃させる**ことだ。その機能を書いたモデルは、たいていその中の穴を見つけられる——ただ、頼まれない限りそうしないだけだ。AI を「作る側」から「敵対する側」へ反転させよう。

あなたはこのエンドポイントを書いた。今度は、それを破ろうとする攻撃者として振る舞ってほしい。悪意あるユーザーが次のことをできるあらゆる方法を挙げて：

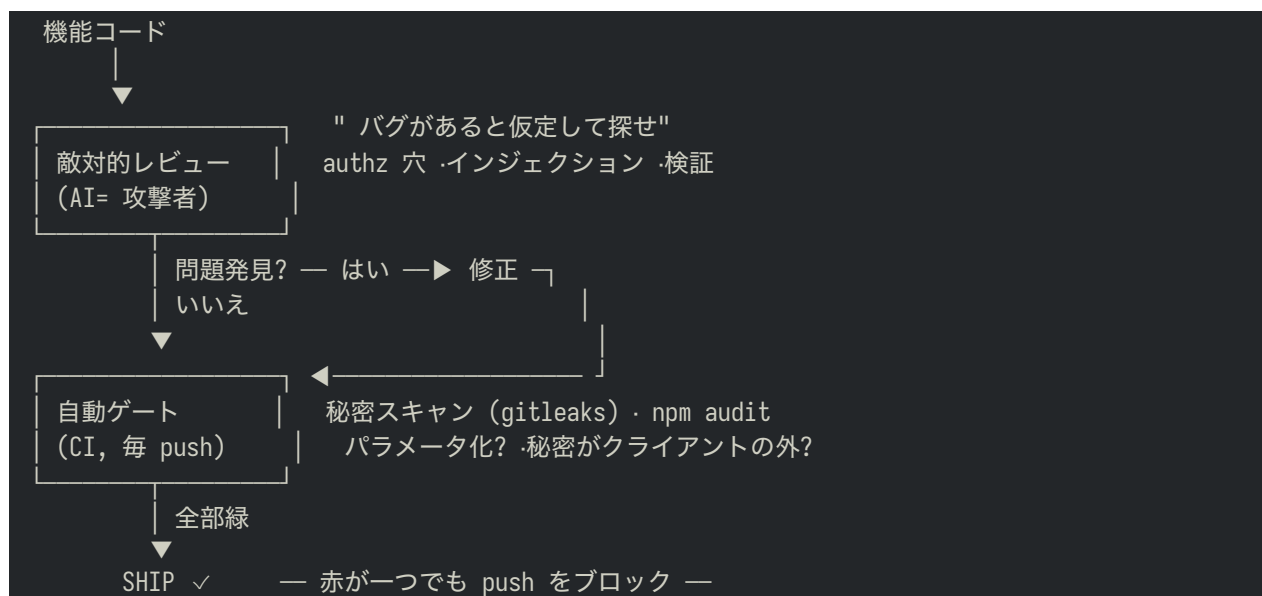
- 本来読んだり変更したりできないはずのデータを読む・変更する (認可の穴)
- 入力経路でコードを注入する (SQL injection, XSS, command injection)
- 検証やレート制限の欠如を悪用する

それぞれについて、悪用する正確なリクエストを示し、それから修正を示して。私を安心させようとしなくて——脆弱性は「ある」と仮定して、それを見つけて。

最後の一行が重要だ。中立のままにしておくと、AI は「安全に見えます!」と言いがちだ。欠陥が存在すると仮定するよう指示されると、AI は実際に探しに行く。この敵対的なパスを、ユーザーに触れるものすべてに対して実行する短い出荷前チェックリストと組み合わせよう。

- すべてのエンドポイントが、ユーザーがログイン済みかだけでなく認可をチェックしている
- すべてのデータベースクエリがパラメータ化されている——文字列で組み立てた SQL がない
- ページにレンダリングされるユーザー入力がエスケープされている (生の HTML 注入がない)
- クライアントコードにシークレットがなく、リポジトリにコミットされたものもない
- .env が gitignore されている。漏れたキーはローテーション済み
- ファイルアップロードが型とサイズを検証し、生成した名前を使っている
- 新しい依存関係が、実在と評判について目視確認されている

通らなければ何も出荷されないシーケンスとして、このゲートを思い描いてください。AI が作り手から攻撃者へと帽子をかぶり替え、その後、自動スキャンが人間のパスが見逃しうるものを下支えします。



そして push する前に**シークレットスキャナ**を実行すること——gitleaks のようなツール (あるいはあなたのプラットフォームの組み込みスキャン) が、キーのような形をしたものをコードと履歴から grep する。これはこのリストで最も高くつくミスに対する、ワンコマンドのセーフティネット、AI に CI (コンティニュアス・インテグレーション。コードを push するたびに、あなたのチェックを自動で走らせてくれるロボットだ) へ組み込ませて毎回の push で走るようにできる。

自動化への慢心、セキュリティ編

最も危険な瞬間は初日ではない——90日目だ。初日のあなたは慎重で、すべての diff を読み、攻撃者プロンプトを実行し、アップロードハンドラをチェックする。やがて 100 の変更が出荷されても何も悪いことは起きず、チェックが儀式のように感じられ始める。小さな変更で「今回だけ」とレビューゲートを飛ばす。その小さな変更は、認可チェックのないエンドポイントを追加する。しばらくは何も起きない——何も起きないことこそ、内側から見たセキュリティの穴の姿そのものだからだ。

これは自動化への慢心がセキュリティに向けられたもので、ここでは他のどこよりもたちが悪い。セキュリティの失敗は静かで、遅れてやってくるからだ。壊れた機能は数分であなたを罰する。壊れた権限チェックは、侵害が起きるその瞬間まで、見かけ上の成功であなたに報いる。災害が起きないことは安全の証拠ではない——それは、まだ見つかっていないだけの脆弱なアプリの、通常の状態にすぎない。

防御策は、ゲートを飛ばさないくらい安価にし、飛ばせないくらい自動にすることだ。シークレットスキャンと audit を CI に置いて、あなたの記憶に頼らず走るようにし、攻撃者プロンプトを、auth・データ・アップロードに触れるものすべてに対していつでも使えるようにしておく。1本の線を守ること。ユーザーデータや権限を扱う変更は、誰か——AI の正直な助けを借りたあなた——が能動的にそれを破ろうとするまで、出荷しない。パイプコーディングにおけるスピードは、良いゲートから生まれるのであって、静けさが万事順調を意味すると信じることから生まれるのではない。

まとめと練習

重要なポイント

- AI が生成したコードはデフォルトで安全でない。きれいな第一印象を信じず、脆弱性があると仮定して探しにいこう。
- 大きなカテゴリは繰り返す: インジェクション、露出したシークレット、欠落した認可、安全でないファイルアップロード、そして発明された依存関係。
- 認可はログイン単位ではなくリクエスト単位だ——どのエンドポイントも、ただログインしているかではなく、このユーザーがこの操作をしてよいかを確認しなければならない。
- プッシュのたびに gitLeaks のようなシークレットスキャナを走らせ、一度でも漏れたキーはローテーションする。スキップできないようスキャンを CI に組み込もう。
- 90 日目は初日より危険だ——セキュリティのゲートを、スキップしないほど安く、スキップできないほど自動にしよう。

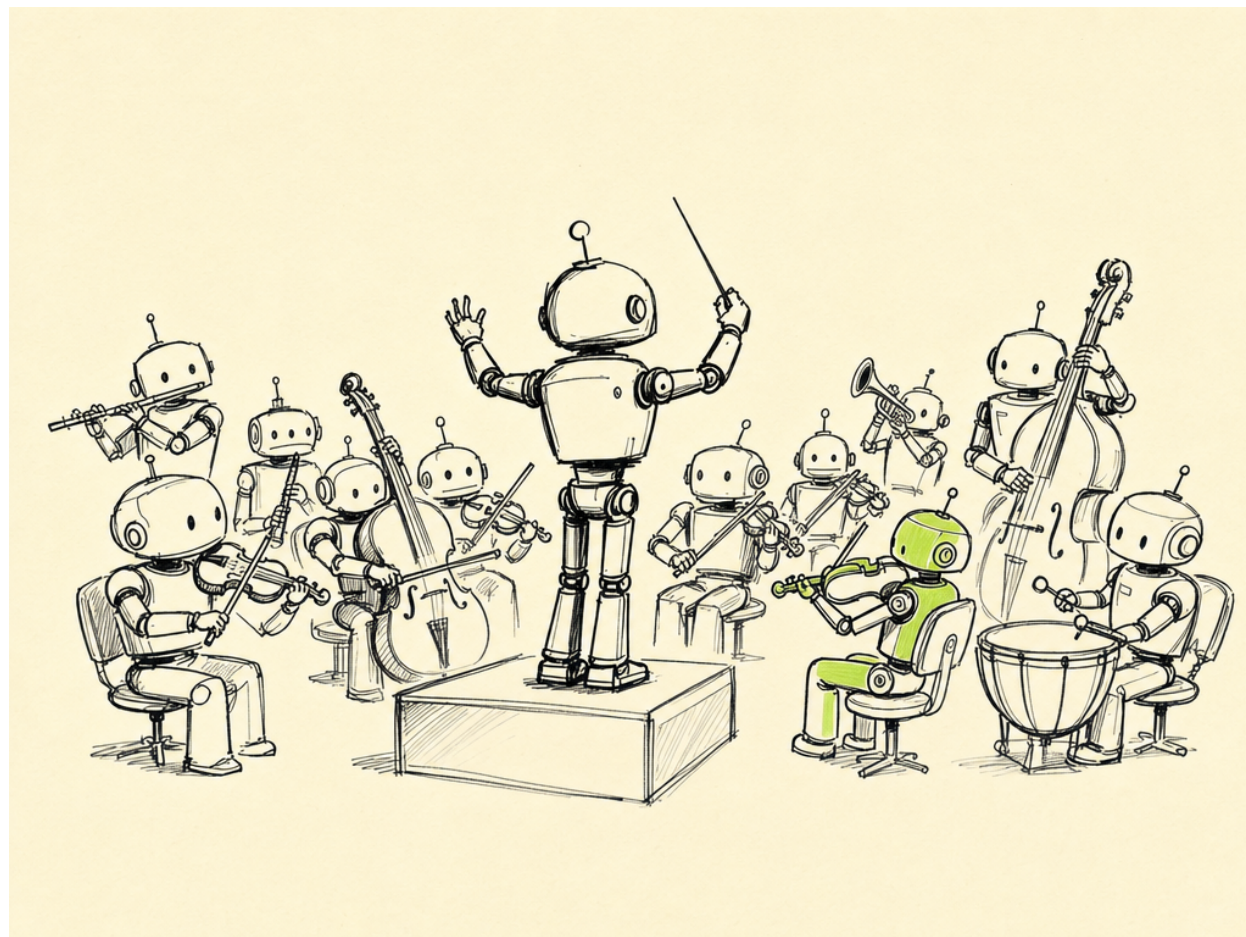
試してみよう

認証・ユーザーデータ・アップロードを扱う自分の機能を一つ選び、下の敵対的レビュー用プロンプトを走らせる。AI が報告する脆弱性ごとに、それを悪用する正確なリクエストと最小限の修正を求め、修正を適用してからプロンプトを再実行する。最後にリポジトリと履歴全体に gitLeaks(またはプラットフォームのシークレットスキャン)を実行して締めくくる。

この章のプロンプト

```
Here is the code for a feature that handles [auth / user data / uploads]:
[paste the code].
Act as an attacker reviewing this for security holes. Specifically check:
- broken authorization (can a user act on another user's data?)
- injection via input (SQL injection, XSS, command injection)
- exposed secrets or keys in code or responses
- unsafe file handling (type, size, path, generated names)
- missing validation or rate limits
For EACH issue, show the exact request that exploits it, then the fix.
Do NOT reassure me — assume there IS a vulnerability and find it.
```


エージェント・オーケストレーション



本書のほとんどにおいて、AI はあなたが手に握る道具だった。あなたが尋ね、AI が答え、あなたが結果を確認する。たいていの場合、これは今でも正しいメンタルモデルだ。だが道具たちは第二のモードを身につけた——ステップの代わりにゴールを手渡すと、AI があなたのもとへ戻ってくるまで数分 (時にはもっと長く) 働くモードだ。

この章はその転換について、そしてそれが起きたときに筋を見失わないことについてだ。「エージェント」を「もう注意を払わなくていい」と読みたくなる誘惑がある。それこそが人々を厄介に陥れる読み方だ。エージェントはあなたがすでに理解している仕事に対するてこであって——もはや目を通さなくなったコードを出荷していいという許可証ではない。

三つの層

AI 支援を三つの層として見ると役立つ。それぞれがあなたに少しずつ多くの手綱を渡してくる。

- **エディタ内補完。**あなたがタイプする間、AI が次の行やブロックを提案する。あなたが完全に制御する。各提案を受け入れるか拒否する。作業の単位は数行で、カーソルのすぐ下に現れるので既定ではすべて確認する。
- **リポジトリ単位のアシスタント。**プロジェクト全体を見られて、複数のファイルを一度に編集し、コマンドを実行し、出力を読むアシスタントと対話する。本書の作業のほとんどはここで起きる。作業の単位はステップか小さな機能であり、毎ターンごとに diff を確認する。
- **自律エージェント。**ゴールを与える——「テストスイートを通せ」「このモジュールを新しい API(アプリケーション・プログラミング・インターフェース。あるソフトウェアが他のソフトウェアのために公開している一連のコマンドのことだ) へ移行せよ」——するとエージェントが計画し、編集し、実行し、エラー

を読み、多くのステップにわたって再試行する。あなたは個々の行動一つひとつではなく結果を確認する。作業の単位はタスク全体だ。

各層はあなたにより多くのロープを手渡します。作業の単位が大きくなり、連続的な操縦が減り、レビューする時点が後ろにずれます。

ティア	作業単位	確認対象	制御
補完 ▼	数行	各提案	 厳格
レポ補助 ▼	工程/機能	差分/ターン	
自律 エージェント	タスク全体	その結果	 緩い

より多くの綱 ▼ — 継続的に操縦 → 放して結果を確認

重要な跳躍は最後のものだ。最初の二つの層ではあなたは絶え間なく操縦する。自律エージェントでは放ってから結果を確認する。それは別の技術であり、この章の残りはそれについてだ。

いつ走らせ、いつ手動で操縦するか

エージェントを無人で走らせるかどうかの決定は、三つの問いに帰着する——その場を離れる前に回すチェックリストだと思ってほしい。

- **爆発半径。**これが失敗したら、どれだけ壊れるか？ コードベース全体のフォーマット処理は爆発半径が小さい。データベースのマイグレーションや認証ロジックの変更は大きい。半径が大きいほど、手で操縦せよ。
- **取り消せるか。**安価に元へ戻せるか？ クリーンな作業ツリー上の新しい git ブランチでの作業は取り消せる——git restore . で消える。データを削除し、メールを送り、カードに請求し、本番へプッシュする作業はそうではない。取り戻せないものの上でエージェントを無人で走らせるな。
- **検証できるか。**結果が正しいか分かるか？ 「完了」が緑のテストスイートや目に見えて動くページを意味するなら、数秒で検証でき、エージェントはうってつけた。「完了」が微妙な判断——金の計算、セキュリティ規則、競合状態——に依存するなら、手早い確認でゲートを掛けられないので、近くに留まれ。

これらを組み合わせる単純な方法: **作業が取り消せて結果を安価に検証できるときは走らせ、そうでないときは手で操縦せよ。** 機能づくりの章の規則を持ち越そう——爆発半径が大きいほどステップは小さく——そしてその兄弟を加えよ: 爆発半径が大きいほど手綱は短く。

さまよえないようにタスクの範囲を定める

自律エージェントは特定の仕方ですぐ失敗する。漂流するのだ。一つのことを頼んだのに、「親切に」別の三つをリファクタリングし、一度も言及していないファイルに触れ、読めない diff を携えて戻ってくる。直しは賢いエージェントではなく——きつめのブリーフだ。

良い自律タスクは四つの部分からなる: 明確な**ゴール**、明示的な**制約**、具体的な**完了基準**、そしてエージェント自身が回せる**ガードレール**。ガードレールが最も重要な部分だ、なぜならそれがエージェントがあなたに気づかれずには乗り越えられない柵だからだ。

Goal: Make the failing tests in tests/checkout/ pass.

Constraints:

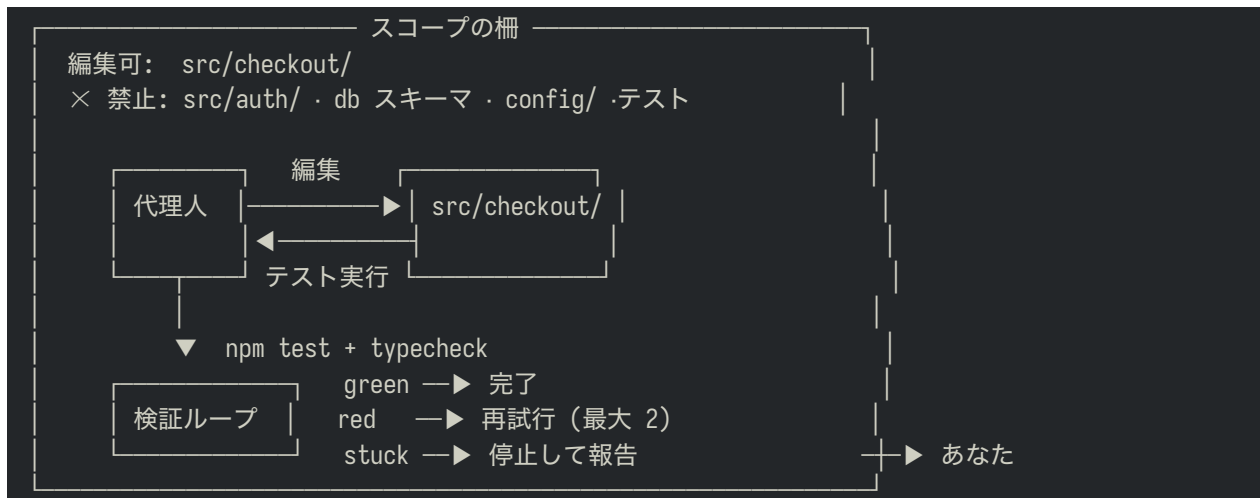
- Only edit files under src/checkout/. Do not touch src/auth/, the database schema, or anything in config/.
- Do not add new dependencies.
- Do not change the tests themselves — the tests define correct behavior.

Done when:

- `npm test tests/checkout/` is fully green.
- `npm run typecheck` passes.
- `git diff --stat` shows changes only under src/checkout/.

If you get stuck after two real attempts, stop and report what you tried and what's still failing. Don't paper over a failure to make the suite go green.

このブリーフが効くのは、エージェントを箱に閉じ込めるからです。制約が触れてよいものに壁を引き、完了基準はエージェントが回せるコマンドであり、検証ループがそれを正直に保ちます。



三つのことがこのブリーフからさまよい出るのを難しくしている。制約はエージェントが触れてよいファイルの周りに壁を引く。完了基準は雰囲気ではなく誰でも回せるコマンドだ。そして最後の段落は、もがく代わりに立ち止まって尋ねる許可を与える——エージェントが 20 分間「修正し」続けてツリーを見つけたときより悪く残していくスローモーションの惨事を防ぐのが、まさにこれだ。

本物の柵はテストだ。自分のテストを回せるエージェントはきつく正直なフィードバックループを持ち、たいいていその中に留まる。自分の仕事を確かめる手段のないエージェントは、壊れたものを自信たっぷりに手渡してくる。この章から習慣の一つを持ち帰るなら：**エージェントに自分を検証する手段を与え、それを通すことを完了の定義とせよ。**

バックグラウンドおよび並列エージェント

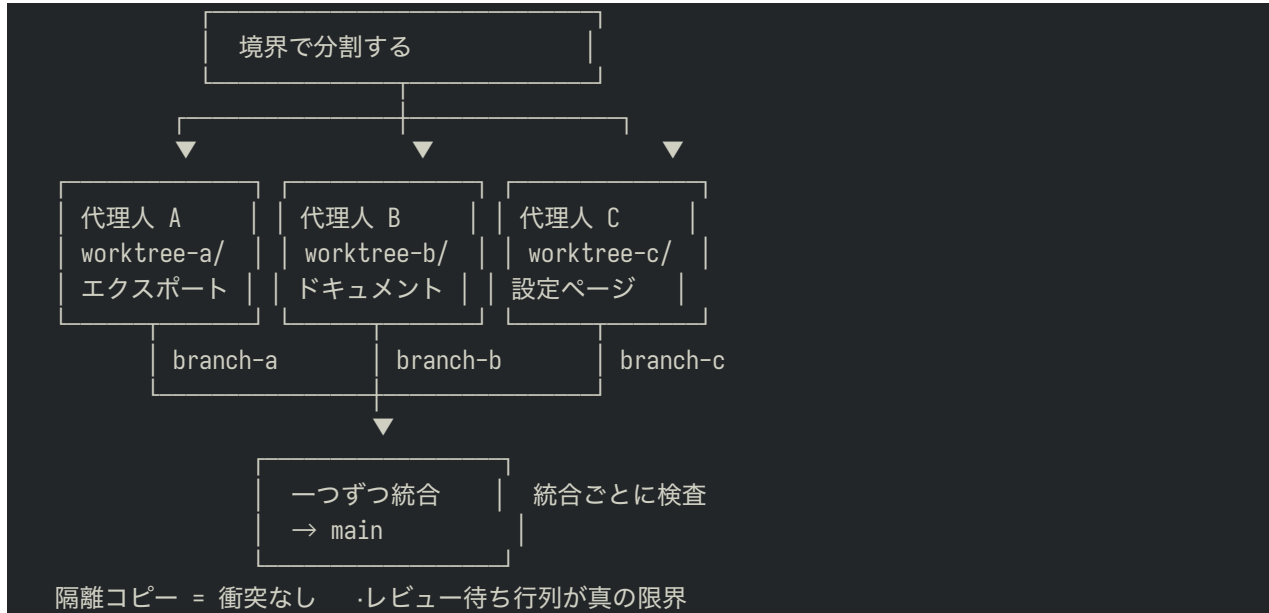
範囲を定めたタスクでエージェントを信頼できるようになると、次の一手は明らかだ: 複数を一度に走らせること。現代の道具はバックグラウンドエージェントを——それぞれ自分のブランチやワークツリー (プロジェクトの別々の作業用コピーのことで、あるエージェントの編集が別のエージェントの編集と決してぶつからないようにする) で——立ち上げられるようにし、あなたが別のことをしている間に三つか四つの範囲を定めたタスクが進む。

これは本物の高速化だが、独立した仕事に限ってのことだ。あなたが引き換えにするコストは調整だ。同じファイルを編集する二つのエージェントは競合する diff を生み、それを手でマージするのは節約した分より高くつぎうる。

- **行数ではなく境界で分けよ。** 各エージェントに他と重ならない塊を与えよ——別々のモジュール、別々の機能、別々のファイル。「エージェント A はエクスポート機能、エージェント B はドキュメント」はきれいだ。「両方のエージェントが app.js を編集」はマージの頭痛の予約だ。
- **作業空間を隔離せよ。** 各エージェントを自分の git ブランチやワークツリーで走らせ、互いの未コミットの変更を踏まないようにせよ。一度に一つずつマージし直し、マージのたびに検査を回せ。

- **数を正直に保て。**それでも全部の結果を確認せねばならない。四つのエージェントが走っているとは、読むべき四つの diff と信頼すべき四組のテストだ。ひと握りを超えると、エージェントではなくレビューの待ち行列があなたのボトルネックになる。

各エージェントは自分専用の隔離されたワークツリーで作業するので、彼らの編集が衝突することはありません。あなたは一度に一つずつマージし直し、各ゲートでチェックを回します。



間違いは並列性を無料と扱うことだ。そうではない——あなたは仕事を書くことからレビューと統合へ移したのだ。しばしば素晴らしい取引だが、あなたが実際にレビューをする場合に限る。

マルチエージェント: ファンアウト、敵対者、統合

別々のタスクにまたがってではなく一つのタスクの中で回る「多数のエージェント」の第二の種類がある。一つのエージェントがすべてをやる代わりに、役割を分ける:

- **幅のためのファンアウト。**複数のエージェントが同じ問題を異なる角度から攻める——あるいは大きなコードベースの異なる部分を探索する——そして報告する。網羅を欲するときに有用だ:「この非推奨関数と呼んでいるすべての場所を見つけよ」はよく並列化する。
- **敵対的レビュアー。**一つのエージェントが書く。二つ目のエージェントの唯一の仕事は、一つ目の出力を攻撃することだ——バグ、抜けたエッジケース、未処理のエラーを見つける。「これは間違っていると仮定して、なぜか突き止めよ」という指示で diff に向けられたレビュアーは、作者エージェントが自分を納得させてしまったものを捕まえる。
- **統合。**最後のパスが発見をまとめ、一つの一貫した結果を生む——動くコードと、レビュアーが何を指摘しどう解決したかの短いメモ。

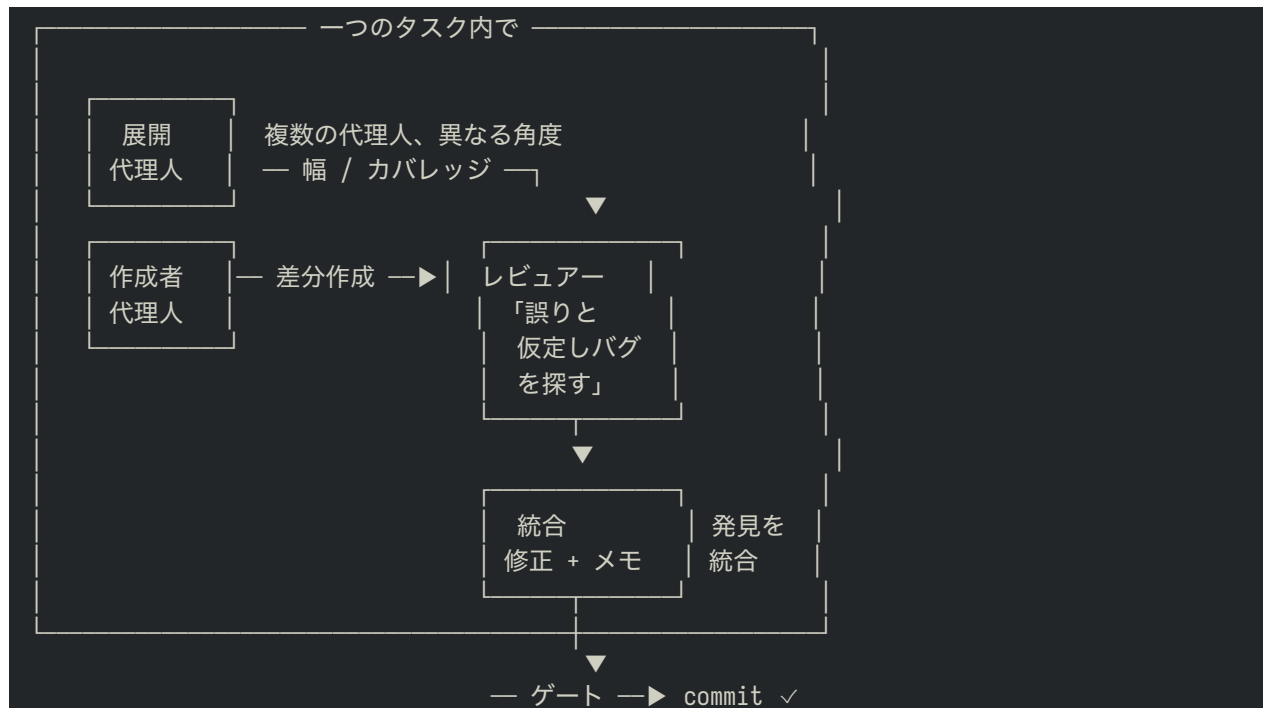
Reviewer agent brief:

You are reviewing a diff written by another agent. Assume it contains at least one bug and find it. Specifically check:

- Does it actually satisfy the original goal, or just pass the tests?
- Edge cases: empty input, nulls, the largest realistic input.
- Anything deleted or changed that the task didn't ask for.

Report concrete problems with file and line. Do not rewrite the code — just report. If you find nothing after a genuine look, say so plainly.

三つの役割は、末尾にゲートを持つパイプラインを形づくります。敵対的レビューアがパスを終え、統合が見つかったものを解決するまで、何もコミットには届きません。



敵対的レビューアは、エージェントの艦隊を持たない一人きりの場合でも整える価値がある——なぜなら変更の作者は、人間であれ AI であれ、その欠陥を見つけるのに最も不向きな者だからだ。敵対的なプロンプトでもう一度見直すことは安価な保険だ。

規模に合わせて出力をレビューする

ここに正直な問題がある: エージェントが複数のタスクにわたって数百行を書いているとき、あなたは十行の diff をレビューするように全行を読むことはできない。読めるふりをする、ゴム印 (見ずに承認) か麻痺 (何も信頼しない) のどちらかに至る。どちらも出荷しない。

抜ける道は、すべてを読む代わりにいくつかの層でゲートを掛けることだ:

- **テストは床だ。** 関連するテストが緑で型チェッカーがきれいであれば、どのエージェント出力もマージされない。これは交渉の余地がなく自動だ——この部分は機械がやる。
- **危険な部分をスポットレビューせよ。** 全部は読めない、爆発半径の大きい部分を読み: 認証、金、データ削除、外部呼び出しに触れるもの何でも。残りは機能づくりの章の危険信号を拾い読みせよ——触れるべきでなかったファイル、ひそかに削除されたもの、小さな依頼が生んだ巨大な diff。
- **コードベースではなく diff を読め。** `git diff --stat` は何がどれだけ変わったかを数秒で教えてくれる。一つのモジュールに範囲を定めたタスクが六つで変更を見せるなら、一行を読む前に、もっとよく見よという信号だ。
- **走らせよ。** 緑のテストスイートは動くアプリではない。それをクリックせよ。最後の確認はいつも他のどの章とも同じものだ: あなたの目の前で、頼んだとおりに実際に動くか?

ここで範囲定めは二度報われる。きつく範囲を定めたタスクは実際にレビューできる diff を生む。放ったタスクはそうできない diff を生む。レビューの問題はほとんど上流で、ブリーフで解決される。

コストと暴走ループ

自律エージェントはループを回して働く: 試し、確かめ、再び試す。そのループこそ要点のすべてだ——そしてそれはまた、ひそかに請求書を膨らませたり永遠に回り続けたりしうるものでもある。

これは前に出たコスト安全の考えと結びつく：**定額のサブスクリプション道具は止まることで失敗し、従量制の有料 API エージェントは課金することで失敗する。**サブスクリプションのコーディングアシスタントが上限に達すると、窓がリセットされるまで単にそれ以上やらないだけだ。あなたが有料 API に向けて書いたスクリプトが再試行ループに嵌まると、呼び続ける——そして呼ぶたびに金がかかる——あなたが気づくかカードの上限が埋まるまで。

だから規則は単刀直入だ：

- **有料 API に向けて制限なしのエージェントループを決して走らせるな。** LLM(ラージ・ランゲージ・モデル。これらのコーディング道具の背後にいる種類の AI だ)API を呼ぶ自分のループを書いたなら、必ずハードキャップがなければならない——最大反復回数、支出上限、タイムアウト。例外なし、「見張っておくよ」もなし。見張らないのだから。
- **すべてのループに出口を与えよ。**「N 回試したら止まれ」「テストがもう二度失敗したら止まれ」「10 分走ったら止まれ」。停止条件のないエージェントは機能ではなくバグだ。
- **最初の数回を見張れ。** エージェントをバックグラウンドで走らせると信頼する前に、いくつかのフォアグラウンド実行を初めから終わりまで見張れ。あなたはそれが収束するかを確かめている——同じ修正の周りで詰まらず、完了に近づいているか。
- **ガードレール付きの道具を選べ。** 内蔵の上限がある管理されたアシスタントは、真夜中に書いた生の API ループより走らせ放しにしても安全だ。閉じて失敗するサブスクリプションは機能であり、開いて失敗する API は危険要素だ。

正直な限界

エージェントは本書で最も強力な形のものであり、まさにそれゆえに、それが何であって何でないかについて澄んだ目を持つ価値がある。

それはあなたが理解をやめてよい自律性ではない。「このコードは何をすべきで、間違っていたら自分はどうかかるか」にもはや答えられなくなった瞬間、あなたはてこを使うことから運を当てにすることへ越えてしまっている。この章のすべての層——範囲定め、ガードレール、テスト、スポットレビュー、コスト上限——は、エージェントにタイピングをはるかに多く任せながらも、あなたをその線の理解する側に留めるために存在する。

技術の天井は本書の残りと同じだ：欲しいものを正確に言い、それを得たときに見分けるあなたの能力。エージェントはその能力を掛け算する。置き換えはしない。火傷を負うのはエージェントを使っただけの人ではなく——エージェントを使って考えるのをやめた人だ。そうなるな。仕事は手渡せ、判断は握っておけ。

まとめと練習

重要なポイント

- エージェントは、望むものを正確に言い、それを得られたかを見分ける能力を倍加させるだけで、置き換えはしない。
- スコープ設定は二度報われる：狭く絞ったタスクは実際にレビューできる diff を生み、緩いタスクはレビューできない diff を生む。
- コードベースではなく diff をレビューしよう—`git diff --stat` は数秒でスコープの逸脱を示し、緑のテストスイートが実際に動かすことの代わりになることはない。
- 有料 API に対してスロットルなしのエージェントループを絶対に回さない。どのループにもハードキャップを—最大反復回数、支出上限、またはタイムアウトを設けよう。
- バックグラウンド実行を信じる前に、最初の実行をフォアグラウンドで見守り、エージェントが収束するか確認しよう。

試してみよう

小さく境界の明確なタスクを一つ選び、明示的な停止条件（「3 回試行したら停止」または「テストがあと 2 回失敗したら停止」）とともにエージェントに渡す。フォアグラウンドで実行を最後まで見守り、それから `git diff --stat` に続けて `git diff` で diff だけをレビューする。変更がスコープに定めたより多くのファイルに触れていたなら、それがブリーフを締め直して再実行する合図だ。

この章のプロンプト

```
I'm giving you an autonomous task. Obey these guardrails exactly:
- SCOPE: only touch [list the files/modules]. If the fix needs more,
  STOP and tell me instead of expanding scope.
- GOAL: [one concrete, checkable outcome].
- STOP CONDITIONS: stop after 3 attempts, OR if tests fail twice in a
  row, OR if you've made changes outside the scope above.
- COST: do not call any paid API in a loop without a hard cap.
- REPORT: when you stop, show me `git diff --stat`, the test result,
  and one sentence on whether you converged or got stuck.
Begin, and narrate each attempt so I can watch you converge.
```