05

Claude Code: エージェント型コーディングのベストプラクティス

— Claude Code: Best practices for agentic coding —

公開日	2025-04-18
原題	Claude Code: Best practices for agentic coding
著者	Anthropic Engineering Team
原文	https://www.anthropic.com/engineering/claude-code-best-practices
翻訳	Claude（機械翻訳／Anthropic）
編集	2026-04-09

Claude Code: エージェント型コーディングのベストプラクティス

Claude Code はエージェント型のコーディング環境です。質問に答えて待機するだけのチャットボットとは異なり、Claude Code はあなたのファイルを読み、コマンドを実行し、変更を加え、そしてあなたが眺めていても、指示を変えても、席を離れても、自律的に問題を解き進めます。

これは仕事のやり方を変えます。自分でコードを書いて Claude にレビューしてもらう代わりに、何が欲しいかを記述し、Claude がどう作るかを考えます。Claude が探索し、計画し、実装します。

しかし、この自律性には学習曲線が伴います。Claude はあなたが理解しておくべきいくつかの制約のもとで動きます。

本ガイドでは、Anthropic の社内チーム、そして多様なコードベース・言語・環境で Claude Code を使うエンジニアに対して効果を発揮することが実証されたパターンを扱います。エージェント型ループが内部でどう動いているかは How Claude Code works を参照してください。

ベストプラクティスの多くは、1 つの制約に由来します。**Claude のコンテキストウィンドウはすぐ一杯になり、一杯になるにつれて性能は劣化する**、という制約です。

Claude のコンテキストウィンドウには、あなたの会話のすべて——メッセージも、Claude が読んだすべてのファイルも、すべてのコマンド出力も——が入っています。しかし、これはすぐ一杯になります。1 回のデバッグセッションやコードベースの探索だけで、数万トークンを生成・消費することもあります。

LLM の性能はコンテキストが埋まるにつれて劣化するため、これが効いてきます。コンテキストウィンドウが満杯に近づくと、Claude は前の指示を「忘れ」始めたり、ミスを増やしたりします。**コンテキストウィンドウは管理すべき最も重要なリソース**です。1 セッションでどう埋まっていくかを実際に知るには、起動時に何がロードされ、各ファイル読み取りがいくらかかるかを示すインタラクティブなウォークスルーを見てみてください。カスタムステータスラインで使用量を継続的に追跡できます。トークン使用量削減の戦略は Reduce token usage を参照してください。

Claude に自分の仕事を検証する手段を与える

テストやスクリーンショット、期待出力を含めて、Claude が自分で確認できるようにしましょう。これは、あなたができる単一の最高レバレッジな行動です。

Claude は、自分の仕事を自分で検証できる（テストを走らせる、スクリーンショットを比較する、出力を検証する）と、劇的に性能が上がります。

明確な成功基準がないと、正しく見えるのに実際には動かないものを作りかねません。あなた自身が唯一のフィードバックループになってしまい、ミスのたびにあなたの注意が必要になります。

戦略	Before	After
検証基準を提供する	*「メールアドレスを検証する関数を実装して」*	*「validateEmail 関数を書いて。テストケース例: user@example.com は true、invalid は false、user@.com は false。実装後にテストを実行」*
UI の変更は視覚的に検証する	*「ダッシュボードを見栄えよくして」*	*「[スクリーンショットを貼付] このデザインを実装して。結果のスクリーンショットを取って元画像と比較し、差分をリスト化して修正して」*
症状ではなく根本原因に対処する	*「ビルドが失敗する」*	*「ビルドがこのエラーで失敗する: [エラー貼付]。修正してビルド成功を確認して。根本原因に対処し、エラーを抑え込まないで」*

UI の変更は Claude in Chrome extension で検証できます。ブラウザに新しいタブを開いて UI をテストし、コードが動くまで繰り返します。

検証はテストスイートでも、リンターでも、出力を確認する Bash コマンドでも構いません。検証の仕組みを磐石にすることに投資してください。

まず探索し、次に計画し、その後にコードを書く

誤った問題を解いてしまわないよう、調査・計画と実装を分離しましょう。

Claude にいきなりコーディングさせると、誤った問題を解くコードが生まれることがあります。Plan Mode を使って探索と実行を分離してください。

推奨ワークフローは 4 つのフェーズから成ります。

1. 探索（Explore）

Plan Mode に入ります。Claude は変更を加えずにファイルを読み、質問に答えます。

```
claude (Plan Mode)

read /src/auth and understand how we handle sessions and login.
also look at how we manage environment variables for secrets.
```

2. 計画（Plan）

Claude に詳細な実装計画を立てさせます。

```
claude (Plan Mode)

I want to add Google OAuth. What files need to change?
What's the session flow? Create a plan.
```

Claude が先に進む前に、`Ctrl+G` でテキストエディタに計画を開いて直接編集できます。

3. 実装（Implement）

通常モードに戻って、Claude に計画に沿ってコードを書かせ、計画と照らして検証します。

```
claude (Normal Mode)

implement the OAuth flow from your plan. write tests for the
callback handler, run the test suite and fix any failures.
```

4. コミット（Commit）

説明的なコミットメッセージで Claude にコミットさせ、PR を作成します。

```
claude (Normal Mode)

commit with a descriptive message and open a PR
```

Plan Mode は有用ですが、オーバーヘッドもあります。スコープが明確で修正が小さいタスク（タイポ修正、ログ行の追加、変数のリネームなど）には、Claude に直接やらせましょう。

計画が最も役立つのは、アプローチに確信がないとき、変更が複数ファイルに及ぶとき、あるいは修正対象のコードに不慣れなときです。差分を 1 文で説明できるなら、計画はスキップしてよいです。

プロンプトに具体的な文脈を与える

指示が精確であるほど、修正の回数は減ります。

Claude は意図を推測できますが、あなたの心を読むことはできません。特定のファイルを参照し、制約に言及し、例となるパターンを指し示しましょう。

戦略	Before	After
タスクをスコープする。どのファイル、どのシナリオ、どんなテストのやり方かを指定する	*「foo.py にテストを追加して」*	*「foo.py 用のテストを書いて、ユーザーがログアウトしているエッジケースをカバーして。モックは避けて」*
ソースを指す。質問に答えられるソースに Claude を向ける	*「ExecutionFactory の API はなぜこんなに奇妙なんだ?」*	*「ExecutionFactory の git 履歴を追いかけて、この API がどう生まれたかを要約して」*
既存パターンを参照する。自分のコードベースのパターンを Claude に指し示す	*「カレンダーウィジェットを追加して」*	*「ホームページの既存ウィジェットの実装を見てパターンを理解して。HotDogWidget.php が良い例。そのパターンに従って新しいカレンダーウィジェットを実装し、月を選び前後に年をページ送りできるようにして。コードベースで既に使われているもの以外のライブラリは使わずにゼロから作って」*
症状を記述する。症状、ありそうな場所、「直った」状態を提供する	*「ログインのバグを直して」*	*「セッションタイムアウト後にログインが失敗するとユーザー報告があった。src/auth/ 特にトークンリフレッシュの認証フローを確認して。問題を再現する失敗テストを書いてから修正して」*

探索中で軌道修正が許される状況では、曖昧なプロンプトも有用です。`「このファイルで改善すべき点は？」` のようなプロンプトは、自分では思いつかなかった観点を浮かび上がらせます。

リッチなコンテンツを与える

`@` でファイルを参照したり、スクリーンショットや画像を貼ったり、データを直接パイプで流し込んだりできます。

Claude にリッチなデータを与える方法はいくつかあります。

- **`@` でファイルを参照する。**どこにコードがあるかを説明する代わりに、`@` で参照する。Claude は応答前にそのファイルを読みます。
- **画像を直接貼り付ける。**プロンプトにコピー&ペーストかドラッグ&ドロップで画像を入れる。
- **ドキュメントや API リファレンスの URL を渡す。**`/permissions` で頻繁に使うドメインを許可リスト化する。
- **データをパイプで流す**: `cat error.log | claude` のように実行してファイル内容を直接渡す。
- **必要なものを Claude に取りに行かせる。**Bash コマンド、MCP ツール、ファイル読み取りを使って自分で文脈を取ってくるよう Claude に指示する。

環境を整える

いくつかのセットアップ手順で、Claude Code のあらゆるセッションが大幅に効果的になります。拡張機能の全体像と使い分けは Extend Claude Code を参照してください。

効果的な CLAUDE.md を書く

`/init` を実行すると、現在のプロジェクト構造に基づいたスターター CLAUDE.md が生成されます。そこから時間をかけて磨いていきます。

CLAUDE.md は、Claude が全会話の冒頭で読み込む特別なファイルです。Bash コマンド、コードスタイル、ワークフローのルールを含めましょう。これにより、コードだけからは推測できない永続的な文脈を Claude に与えられます。

`/init` コマンドはコードベースを解析してビルドシステム、テストフレームワーク、コードパターンを検出し、磨いていくための土台を提供します。

CLAUDE.md には必須フォーマットはありませんが、短く人間が読みやすく保ちましょう。例:

```
# Code style
- Use ES modules (import/export) syntax, not CommonJS (require)
- Destructure imports when possible (eg. import { foo } from 'bar')

# Workflow
- Be sure to typecheck when you're done making a series of code changes
- Prefer running single tests, and not the whole test suite, for performance
```

CLAUDE.md は毎セッション読み込まれるため、広く適用できるものだけを含めましょう。時々しか関係しないドメイン知識やワークフローは skills を使ってください。skills は必要に応じてオンデマンドでロードされ、全会話を膨張させません。

簡潔に保つこと。行ごとに自問: *「これを消したら Claude はミスを犯すか?」*。ノーなら消します。肥大化した CLAUDE.md は、Claude に「本当の指示」を無視させてしまいます。

✅ 含めるべき	❌ 含めるべきでない
Claude が推測できない Bash コマンド	Claude がコードを読めば分かるもの
デフォルトと異なるコードスタイルのルール	Claude が既に知っている標準的な言語慣習
テスト手順と好みのテストランナー	詳細な API ドキュメント(代わりにリンクを貼る)
リポジトリのエチケット(ブランチ命名、PR 慣習)	頻繁に変わる情報
プロジェクト固有のアーキテクチャ決定	長い解説やチュートリアル
開発環境のクセ(必要な環境変数など)	ファイルごとのコードベース説明
よくある落とし穴、非自明な振る舞い	「きれいなコードを書け」のような自明な慣行

ルールがあるのに Claude が反する振る舞いをし続けるなら、ファイルが長すぎてルールが埋もれている可能性が高いです。CLAUDE.md に答えが書いてあるのに Claude が質問してくるなら、表現が曖昧かもしれません。CLAUDE.md をコードのように扱ってください——何かおかしくなったらレビューし、定期的に剪定し、変更したら実際に Claude の振る舞いが変わるかで検証します。

「IMPORTANT」や「YOU MUST」といった強調を加えて遵守率を改善するようチューニングできます。チームで共有できるよう CLAUDE.md は git に入れてください。このファイルは時間とともに価値を蓄積します。

CLAUDE.md は `@path/to/import` 構文で追加ファイルをインポートできます:

```
See @README.md for project overview and @package.json for available npm commands.

# Additional Instructions
- Git workflow: @docs/git-instructions.md
- Personal overrides: @~/.claude/my-project-instructions.md
```

CLAUDE.md は以下のいくつかの場所に配置できます。

- **ホームフォルダ** (`~/.claude/CLAUDE.md`): 全 Claude セッションに適用。

- **プロジェクトルート**（`./CLAUDE.md`）: git に入れてチームと共有。
- **プロジェクトルート**（`./CLAUDE.local.md`）: 個人のプロジェクト固有メモ。チームと共有したくないので `.gitignore` に追加する。
- **親ディレクトリ**: モノレポで `root/CLAUDE.md` と `root/foo/CLAUDE.md` の両方が自動的に取り込まれるような用途に有用。
- **子ディレクトリ**: そのディレクトリ配下のファイルを扱うとき、Claude が子の CLAUDE.md をオンデマンドで取り込む。

権限を設定する

クラシファイアに承認を任せる auto モード、特定コマンドを許可リスト化する `/permissions`、OS レベル分離の `/sandbox` のどれかを使いましょう。どれも割り込みを減らしつつあなたの制御を維持します。

デフォルトでは、Claude Code はシステムを変更しうるアクション（ファイル書き込み、Bash コマンド、MCP ツールなど）に対して権限を要求します。安全ですが面倒です。10 回目の承認にもなればもうレビューしておらず、ただクリックしているだけでしょう。割り込みを減らす方法は 3 つあります。

- **Auto モード**: 別のクラシファイアモデルがコマンドをレビューし、スコープ拡大、未知のインフラ、敵対的コンテンツ主導のアクションのようなリスキーなものだけをブロックします。タスクの大筋には信頼があるけれど毎ステップクリックしたくないときに最適。
- **権限許可リスト**: `npm run lint` や `git commit` のように安全と分かっている特定ツールを許可。
- **サンドボックス**: OS レベルの分離を有効化し、ファイルシステムとネットワークアクセスを制限する。定義された境界内で Claude をより自由に動かせる。

permission modes、permission rules、sandboxing の詳細を参照してください。

CLI ツールを使う

`gh`、`aws`、`gcloud`、`sentry-cli` のような CLI ツールを外部サービスとのやり取りに使うよう Claude Code に指示しましょう。

CLI ツールは、外部サービスと対話するうえで最もコンテキスト効率が良い方法です。GitHub を使っているなら `gh` CLI をインストールしましょう。Claude は issue 作成、プルリクエストのオープン、コメントの読み取りで `gh` の使い方を知っています。`gh` がなくても Claude は GitHub API を使えますが、未認証リクエストはよくレートリミットに引っかかります。

Claude は知らない CLI ツールの習得も得意です。`Use 'foo-cli-tool --help' to learn about foo tool, then use it to solve A, B, C.` のようなプロンプトを試してみてください。

MCP サーバーを接続する

`claude mcp add` で Notion、Figma、あなたのデータベースのような外部ツールを接続しましょう。

MCP サーバーを使うと、issue トラッカーからの機能実装、データベースのクエリ、モニタリングデータの分析、Figma のデザイン統合、ワークフローの自動化を Claude に頼めます。

フックを設定する

例外なく毎回起こるべきアクションにはフックを使いましょう。

フックは Claude のワークフローの特定ポイントで自動的にスクリプトを実行します。助言的な CLAUDE.md 指示と異なり、フックは決定論的で必ず実行されます。

Claude 自身にフックを書かせることもできます。*「ファイル編集のたびに eslint を走らせるフックを書いて」*、*「migrations フォルダへの書き込みをブロックするフックを書いて」* のようなプロンプトを試してください。`.claude/settings.json` を直接編集してフックを手動設定したり、`/hooks` で設定内容を確認したりできます。

Skill を作る

`.claude/skills/` に `SKILL.md` を作れば、Claude にドメイン知識と再利用可能ワークフローを与えられます。

Skills は、プロジェクト・チーム・ドメインに特有な情報で Claude の知識を拡張します。関連があれば Claude が自動適用しますし、`/skill-name` で直接呼び出すこともできます。

`.claude/skills/` 配下に `SKILL.md` を含むディレクトリを追加すれば skill を作れます。

`.claude/skills/api-conventions/SKILL.md`:

```
---
name: api-conventions
description: REST API design conventions for our services
---
# API Conventions
- Use kebab-case for URL paths
- Use camelCase for JSON properties
- Always include pagination for list endpoints
- Version APIs in the URL path (/v1/, /v2/)
```

Skills は、直接呼び出す繰り返しワークフローを定義することもできます。

`.claude/skills/fix-issue/SKILL.md`:

```
---
name: fix-issue
description: Fix a GitHub issue
disable-model-invocation: true
---
Analyze and fix the GitHub issue: $ARGUMENTS.

1. Use `gh issue view` to get the issue details
2. Understand the problem described in the issue
3. Search the codebase for relevant files
4. Implement the necessary changes to fix the issue
5. Write and run tests to verify the fix
6. Ensure code passes linting and type checking
7. Create a descriptive commit message
8. Push and create a PR
```

`/fix-issue 1234` のように呼び出します。副作用のあるワークフローは手動でトリガーしたいので `disable-model-invocation: true` を使いましょう。

カスタムサブエージェントを作る

`.claude/agents/` に専門的なアシスタントを定義すれば、Claude が独立したタスクをそれに委譲できます。

Subagents は独自のコンテキストと独自の許可ツールセットで動きます。多くのファイルを読むタスクや、メイン会話を散らかさずに特化した集中を要するタスクに有用です。

`.claude/agents/security-reviewer.md`：

```
---
name: security-reviewer
description: Reviews code for security vulnerabilities
tools: Read, Grep, Glob, Bash
model: opus
---
You are a senior security engineer. Review code for:
- Injection vulnerabilities (SQL, XSS, command injection)
- Authentication and authorization flaws
- Secrets or credentials in code
- Insecure data handling

Provide specific line references and suggested fixes.
```

サブエージェントを使うよう Claude に明示的に指示しましょう: *「このコードのセキュリティ問題をレビューするためにサブエージェントを使って」*。

プラグインをインストールする

`/plugin` でマーケットプレースを見て回れます。プラグインは設定なしで skill、ツール、連携を追加できます。

プラグインは、コミュニティや Anthropic から提供される skill、フック、サブエージェント、MCP サーバーをひとつのインストール可能ユニットに束ねます。型付き言語を扱うなら、正確なシンボルナビゲーションと編集後の自動エラー検出を Claude に与えるコードインテリジェンスプラグインを入れましょう。

skill・サブエージェント・フック・MCP の選び方のガイダンスは Extend Claude Code を参照してください。

効果的に伝える

Claude Code との伝え方が結果の質に大きく影響します。

コードベースに関する質問をする

シニアエンジニアに聞くような質問を Claude に投げましょう。

新しいコードベースに入るときは、Claude Code を学習と探索に使いましょう。別のエンジニアに聞くのと同じような質問ができます。

- ログはどう動いている?
- 新しい API エンドポイントはどう追加する?
- `foo.rs` の 134 行目の `async move { ... }` は何をしている?
- `CustomerOnboardingFlowImpl` はどんなエッジケースを扱っている?
- なぜこのコードは 333 行目で `bar()` ではなく `foo()` を呼んでいる?

Claude Code をこう使うと効果的なオンボーディングワークフローになり、立ち上がり時間を改善し、他エンジニアの負荷を減らせます。特別なプロンプトは不要——直接質問してください。

Claude にインタビューさせる

大きな機能に取り組むときは、まず Claude にあなたをインタビューさせましょう。最小限のプロンプトから始め、`AskUserQuestion` ツールを使ってインタビューしてもらいます。

Claude は、まだ考えていないかもしれない点——技術実装、UI/UX、エッジケース、トレードオフ——について聞いてきます。

```
I want to build [brief description]. Interview me in detail using the AskUserQuestio
n tool.

Ask about technical implementation, UI/UX, edge cases, concerns, and tradeoffs. Do
n't ask obvious questions, dig into the hard parts I might not have considered.

Keep interviewing until we've covered everything, then write a complete spec to SPE
C.md.
```

仕様が出来上がったら、新しいセッションを立ち上げて実行します。新しいセッションは実装のみに集中したクリーンなコンテキストを持ち、参照できる仕様書も手元にあります。

セッションを管理する

会話は永続的かつ巻き戻し可能です。これを自分の有利に使いましょう！

早めに・頻繁に軌道修正する

Claude が脇道に逸れ始めたら、気づいた瞬間に修正しましょう。

最良の結果は緊密なフィードバックループから生まれます。Claude はたまに最初の試みで完璧に問題を解きますが、素早く軌道修正する方が概して良い解により速く辿り着きます。

- **`Esc`**：`Esc` キーで Claude を途中停止。コンテキストは保持されるので、再指示できる。
- **`Esc + Esc` または `/rewind`**：`Esc` を 2 回押すか `/rewind` を実行すると巻き戻しメニューが開き、前の会話・コード状態に戻すか、選んだメッセージから要約できる。
- **`"Undo that"`**：Claude に変更を取り消させる。
- **`/clear`**：関係ないタスク間でコンテキストをリセット。無関係なコンテキストを長く引きずるセッションは性能を下げる。

同じ問題について 1 セッションで 2 回以上修正しているなら、コンテキストは失敗アプローチで散らかっています。`/clear` でリセットし、学んだことを組み込んだより具体的なプロンプトで仕切り直しましょう。**「きれいなセッション + 良いプロンプト」は、ほぼ常に「長いセッション + 積み上がった修正」に勝ります。**

コンテキストを積極的に管理する

関係ないタスク間では `/clear` でコンテキストをリセットしましょう。

Claude Code は、コンテキスト上限に近づくと会話履歴を自動的に圧縮（compact）し、重要なコードと判断を保ちながら空き容量を作ります。

長いセッション中、Claude のコンテキストウィンドウは無関係な会話、ファイル内容、コマンドで埋まっていきます。これは性能を下げ、時に Claude の気を散らします。

- タスク間で頻繁に `/clear` を実行してコンテキストウィンドウを完全にリセットする
- 自動圧縮が走るとき、Claude はコードパターン、ファイル状態、主要な決定を含む「最も重要な部分」を要約する
- より制御したければ `/compact <instructions>` を使う（例: `/compact Focus on the API changes`）
- 会話の一部だけ圧縮するには `Esc + Esc` または `/rewind` でメッセージチェックポイントを選び「**ここから要約**」を選択。以前の文脈を保ったまま、その時点以降のメッセージを凝縮する
- `「圧縮時は常に修正したファイル一覧とテストコマンドを全て保持する」` のような指示を CLAUDE.md に入れて圧縮挙動をカスタマイズし、重要な文脈が要約で生き残るようにする
- コンテキストに残す必要のない簡単な質問には `/btw` を使う。回答は閉じられるオーバーレイに出て会話履歴には入らず、コンテキストを大きくせずに細部を確認できる

調査にサブエージェントを使う

`「use subagents to investigate X」` で調査を委譲しましょう。サブエージェントは別コンテキストで探索し、メイン会話を実装にきれいなまま保ちます。

コンテキストが根本的制約である以上、サブエージェントは使える最強のツールの 1 つです。Claude がコードベースを調査するとき、多数のファイルを読み、そのすべてがあなたのコンテキストを消費します。サブエージェントは別のコンテキストウィンドウで動き、要約を返してきます。

```
Use subagents to investigate how our authentication system handles token
refresh, and whether we have any existing OAuth utilities I should reuse.
```

サブエージェントがコードベースを探索し、関連ファイルを読み、知見を要約して返してくれる——この間、メイン会話は一切散らかりません。

Claude が何かを実装した後の検証にもサブエージェントを使えます。

```
use a subagent to review this code for edge cases
```

チェックポイントで巻き戻す

Claude のあらゆるアクションはチェックポイントを作ります。会話、コード、あるいはその両方を任意の以前のチェックポイントに復元できます。

Claude は変更前に自動でチェックポイントを作ります。`Escape` をダブルタップするか `/rewind` で巻き戻しメニューを開けます。会話のみ、コードのみ、両方、選んだメッセージからの要約、のいずれかに復元できます。詳細は Checkpointing を参照。

すべての動きを慎重に計画する代わりに、リスクのある試みを Claude にやらせて構いません。うまくいかなければ巻き戻して別のアプローチを試せばよいのです。チェックポイントはセッションを跨いで残るので、ターミナルを閉じても後で巻き戻せます。

チェックポイントが追跡するのは *Claude による* 変更だけで、外部プロセスによるものは追跡しません。これは git の代替ではありません。

会話を再開する

`claude --continue` で中断した所から再開、`--resume` で最近のセッションから選択できます。

Claude Code は会話をローカルに保存します。タスクが複数セッションにまたがるとき、文脈を再説明する必要はありません。

```
claude --continue    # 最新の会話を再開
claude --resume      # 最近の会話から選択
```

`/rename` でセッションに `"oauth-migration"` や `"debugging-memory-leak"` のような分かりやすい名前を付ければ、後で見つけやすくなります。セッションをブランチのように扱ってください——異なる作業の流れは別々の永続的コンテキストを持てます。

自動化とスケーリング

1 人の Claude で効果的に動けるようになったら、並列セッション、非対話モード、fan-out パターンで出力を掛け算しましょう。

ここまではすべて、1 人の人間、1 人の Claude、1 つの会話を前提にしてきました。しかし Claude Code は水平スケールします。このセクションのテクニックは、もっと多くを片付ける方法を示します。

非対話モードで実行する

CI、プリコミットフック、スクリプトでは `claude -p "prompt"` を使いましょう。ストリーミング JSON 出力なら `--output-format stream-json` を付けます。

`claude -p "your prompt"` で、セッション無しで非対話的に Claude を実行できます。非対話モードは、Claude を CI パイプライン、プリコミットフック、あらゆる自動ワークフローに統合する方法です。出力フォーマットは結果をプログラム的にパースできるよう、プレーンテキスト、JSON、ストリーミング JSON から選べます。

```
# ワンショットクエリ
claude -p "Explain what this project does"

# スクリプト向け構造化出力
claude -p "List all API endpoints" --output-format json

# リアルタイム処理向けストリーミング
claude -p "Analyze this log file" --output-format stream-json
```

複数の Claude セッションを並列で動かす

複数の Claude セッションを並列実行して開発を加速したり、独立した実験を走らせたり、複雑なワークフローを開始したりできます。

並列セッションの主要な 3 つの走らせ方があります。

- Claude Code デスクトップアプリ: 複数のローカルセッションをビジュアルに管理。各セッションが独自の隔離された worktree を持つ。
- Web 上の Claude Code: Anthropic のセキュアなクラウドインフラ上の隔離 VM で動かす。
- Agent teams: 共有タスク、メッセージング、チームリードによる複数セッションの自動協調。

作業の並列化に加え、複数セッションは品質重視のワークフローを可能にします。書いたばかりのコードに対してバイアスのない形でコードレビューできるため、新鮮なコンテキストはレビュー品質を改善します。

たとえば Writer/Reviewer パターンを使ってみましょう。

セッション A (Writer)	セッション B (Reviewer)
`API エンドポイントのレートリミッターを実装して`	
	`@src/middleware/rateLimiter.ts のレートリミッター実装をレビューして。エッジケース、レースコンディション、既存ミドルウェアパターンとの一貫性を見て。`
`レビューフィードバックはこれ: [セッション B の出力]。これに対応して。`	

テストでも似たことができます。Claude A にテストを書かせ、Claude B にそれを通すコードを書かせるのです。

ファイル横断で fan-out する

タスクをループで回して、各呼び出しで `claude -p` を実行します。バッチ操作の権限を絞るには `--allowedTools` を使います。

大規模な移行や分析では、多数の並列 Claude 呼び出しに作業を分配できます。

1. タスクリストを生成

Claude に移行対象ファイル全部を列挙させます（例: `list all 2,000 Python files that need migrating`）。

2. リストを回すスクリプトを書く

```
for file in $(cat files.txt); do
  claude -p "Migrate $file from React to Vue. Return OK or FAIL." \
    --allowedTools "Edit,Bash(git commit *)"
done
```

3. 最初の数ファイルでテスト、その後スケール実行

最初の 2〜3 ファイルの結果を見てプロンプトを磨いてから全セットで走らせます。`--allowedTools` フラグは Claude ができることを制限し、無人実行時に重要です。

既存のデータ／処理パイプラインに Claude を統合することもできます。

```
claude -p "<your prompt>" --output-format json | your_command
```

開発中は `--verbose` を付けてデバッグし、本番では外しましょう。

Auto モードで自律実行する

バックグラウンドの安全チェック付きで割り込みなく実行するには auto モード を使いましょう。クラシファイアモデルがコマンドを実行前にレビューし、スコープ拡大、未知のインフラ、敵対的コンテンツ主導のアクションをブロックしつつ、ルーチンの作業はプロンプトなしで進めます。

```
claude --permission-mode auto -p "fix all lint errors"
```

`-p` フラグの非対話実行では、クラシファイアが繰り返しアクションをブロックすると auto モードは中止します（フォールバック先のユーザーがいないため）。しきい値は when auto mode falls back を参照してください。

よくある失敗パターンを避ける

よくあるミスを紹介します。早く気づくほど時間を節約できます。

- **キッチンシンクセッション。**1 つのタスクで始め、Claude に無関係なことを聞き、元のタスクに戻る。コンテキストが関係ない情報で埋まっている。

  > **修正**: 関係ないタスク間では `/clear` する。

- **繰り返しの修正。**Claude が間違える、修正する、まだ間違っている、また修正する。コンテキストが失敗アプローチで汚染されている。

  > **修正**: 2 回修正しても失敗したら、`/clear` して、学んだことを組み込んだより良い初期プロンプトを書く。

- **過剰に指定された CLAUDE.md。**CLAUDE.md が長すぎると、Claude は半分を無視する。重要なルールがノイズに埋もれる。

  > **修正**: 容赦なく剪定する。指示なしでも Claude が正しく動くなら削除するかフックに変える。

- **信頼して検証ギャップ。**Claude がそれっぽく見えるがエッジケースを処理しない実装を出す。

> **修正**: 必ず検証（テスト、スクリプト、スクリーンショット）を用意する。検証できないものはリリースしない。

- **無限の探索**。「調べて」と範囲を切らずに頼む。Claude が数百のファイルを読み、コンテキストを埋める。

> **修正**: 調査は狭くスコープするか、サブエージェントに任せて探索がメインコンテキストを食わないようにする。

自分の直感を育てる

このガイドのパターンは石のように固定されたものではありません。一般的にうまく動く出発点ですが、あらゆる状況で最適とは限りません。

1 つの複雑な問題に深く潜っているとき、履歴そのものに価値があるなら、*あえて* コンテキストを積み上げるべきこともあります。探索的なタスクでは、計画を飛ばして Claude に任せるべきこともあります。問題を制約する前にどう解釈するかを見るために、わざと曖昧なプロンプトが正解になることもあります。

何が効くかに注意を払いましょう。Claude が素晴らしい出力を出したとき、あなたが何をしたかに気づいてください——プロンプトの構造、与えた文脈、どのモードだったか。Claude が苦戦したときは「なぜか？」を問いましょう。文脈がノイズだらけだったか？ プロンプトが曖昧すぎたか？ タスクが 1 回には大きすぎたか？

時間とともに、どんなガイドにも書ききれない直感が育ちます。いつ具体的であるべきか、いつ開放的であるべきか、いつ計画しいつ探索すべきか、いつコンテキストをクリアしいつ積み上げるべきか——それが分かるようになります。

関連リソース

- How Claude Code works: エージェント型ループ、ツール、コンテキスト管理
- Extend Claude Code: skill、フック、MCP、サブエージェント、プラグイン
- Common workflows: デバッグ、テスト、PR その他のステップバイステップレシピ
- CLAUDE.md: プロジェクト慣習と永続文脈を保存する