

終わらないコードレビュー、
ドキュメント更新、
コミットメッセージの作成……。

自分専用の魔法を ターミナルに!

Claude Code カスタムコマンド自作実践ガイド

その定型業務、
AIに「丸投げ」する
レシピを作りませんか?

```
> Claude Code カスタムコマンド作成...  
Generating recipe...  
  
function functional_terminal() {  
  return {  
    "sir": "Claude Code as kedt thof  
    under:168  
  }  
  return true  
}  
>
```



ふう……
またこのプロジェクト
特有の命名規則の修正か。

AIツールは導入したけど、
毎回前提条件を
説明するのも
手間……

標準機能を使うだけでは、
ツールの真価は
半分も引き出せていない。

『もう一人の自分
(デジタル・アバター)』を
作り出すんだ。

【カスタムコマンド】
自分の開発スタイルやプロジェクトの
ルールをAIにインストールし、一言で
定型業務を完遂させる技術。

```
Terminal
import "code.h";
import "mario.termina";

public tside {
  near aetmoin state(String ago, args) {
    conet boss = null;
    const netan = null;
    const stage = nostn;
    const bask = nostn;
    const stger = nosin;
    const scpen = nusteer;
    tonaloc .1;
    System.out.log("SE-WE を用注 博計 設置器 ルール");
  }
  public facteeritoere {
    System.out.println(String$tenError));
  }
}
>>
```

君はまだ、
Claude Codeを単なる
単なる『チャット』としてしか
使っていないようだな。

もう一人の自分……？

でも、現場の『暗黙の了解』を
をAIに覚えさせるなんて
なんて無理じゃ……

26歳

そこで登場するのが、
プロジェクトの『憲法』とも呼べる
【CLAUDE.md】だ。

CLAUDE.md

```
# Claudeの動作を制御するためのファイル
# 以下の内容は、Claudeの動作を制御するためのファイル
# 以下の内容は、Claudeの動作を制御するためのファイル

# Claudeの動作を制御するためのファイル
# 以下の内容は、Claudeの動作を制御するためのファイル
# 以下の内容は、Claudeの動作を制御するためのファイル
```

これをルートディレクトリに置けば、
ビルド手順やコードスタイルといっ
『この現場の流儀』をClaudeが
Claudeに常握した状態になる。
毎回ゼロから説明する
時間の浪費は終わりだ。

指示が曖昧で意図を見逃される？

→【解決策】<rules></rules> のような XML タグを使って命令構造を視覚的に分離しろ！

Security Reviewer

(フロントマターと具体的な指示文)

ルールの適用が不安定？

→【解決策】抽象的な説明より、良いアウトプットの具体例を「例示 (Examples)」として入れろ！ AI はパターンの継続が得意だ。

スキルの核心は、
Markdown の冒頭に書く
『フロントマター (設定情報)』だ。

例えば、専用の
セキュリティ監査コマンド
/security-check を召喚
を召喚してみよう。

```
Terminal
import "code.tr";
import "nerve."veruine";

public helde {
  @var actwin state(String ago, arg1 {
    const hoes = null;
    const nengn = var1;
    const ctogo = null;
    const ctogz = necte;
    const stogr = nectn;
    const troon = nuctoer;
    @var ip;
    @var out.log("TE-GE 5555 5555 5555 5555 5555");
    @var fentatitorat {
      Sytles.out.petoEND(trilogErenEren);
    }
  }
}
```

上者 50歳

メインの会話スレッド (Main Thread)

その通り。
大きな目的を一つの
一つの会話だけで進めると、
コンテキストが膨大になり
重要な詳細が埋もれてしまう。

これを打破する概念が
「サブエージェント」
の活用である。

「サブエージェント
(Subagents)」の

なるほど！
でも、複数のファイル修正が
複雑なタスクを頼むと、
途中でAIが指示を忘れて
『迷子』になりませんか？



メインの会話

サブエージェント(専門家)

専門タスクの依頼

成果・レポートだけ
を持ち帰る

思考の『作業部屋』を隔離するんだ。
メインの会話を汚さず、
重たい作業を別室の専門家に任せ、
成果だけを受け取る。

主者(50歳)

これなら
クリーンな
環境が保てる！

プロ主人公詞
(26歳)



```
1 console.log('Exploring files...');
2 function exploreFiles(dir) {
3   return new Promise((resolve, reject) => {
4     fs.readdir(dir, (err, files) => {
5       if (err) reject(err);
6       resolve(files);
7     });
8   });
9 }
10 exploreFiles('./src').then(files => {
11   console.log('Files found:', files);
12 });
```

1 Explore (探索)
- 関連ファイルや
ドキュメントを読み、
現状を把握。

```
1 console.log('Exploring files...');
2 function exploreFiles(dir) {
3   return new Promise((resolve, reject) => {
4     fs.readdir(dir, (err, files) => {
5       if (err) reject(err);
6       resolve(files);
7     });
8   });
9 }
10 exploreFiles('./src').then(files => {
11   console.log('Files found:', files);
12 });
```

2 Plan (計画)
- 変更の影響範
囲を特定し、
設計図を描く。

この4ステップの『黄金律』をサブ
エージェントの指示に組み込むことで、
自動化の精度は劇的に向上する。



2 Plan (計画)
- 変更の影響範囲を特定し、
設計図を描く。

```
1 console.log('Exploring files...');
2 function exploreFiles(dir) {
3   return new Promise((resolve, reject) => {
4     fs.readdir(dir, (err, files) => {
5       if (err) reject(err);
6       resolve(files);
7     });
8   });
9 }
10 exploreFiles('./src').then(files => {
11   console.log('Files found:', files);
12 });
```

3 Code (実装)
- 実際にコードを書き、
テスト検証する。

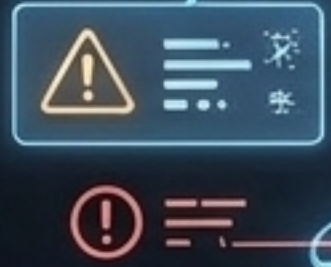
```
1 import { useState } from 'react';
2 import './App.css';
3 function App() {
4   const [count, setCount] = useState(0);
5   return (
6     <div>
7       <h1>Hello World</h1>
8       <button onClick={() => setCount(count + 1)}>
9         Click me
10       </button>
11     </div>
12   );
13 }
14 export default App;
```

4 Commit (コミット)
- 変更内容を要約し、
ログを構成。

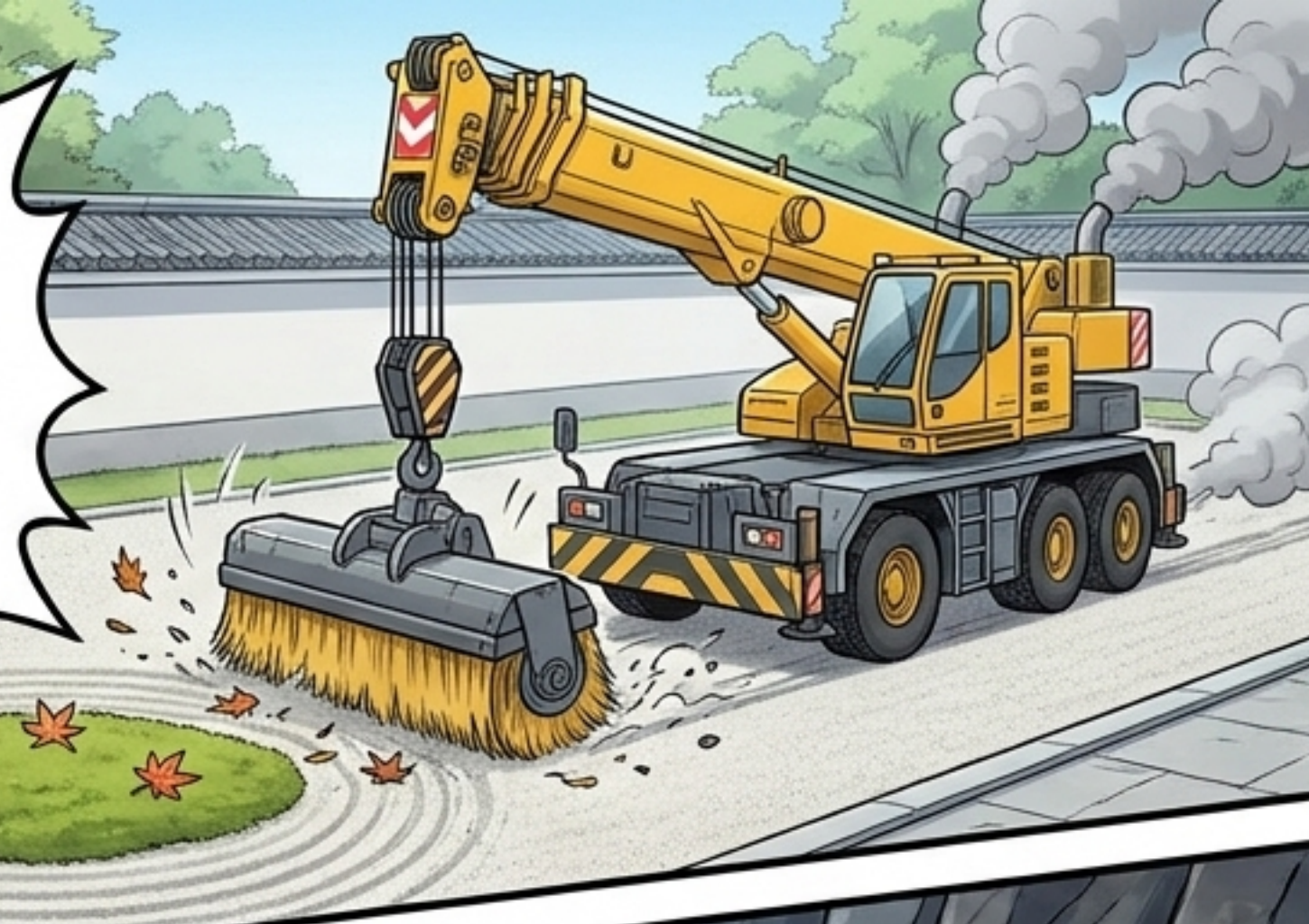


【サブエージェント運用の注意点】

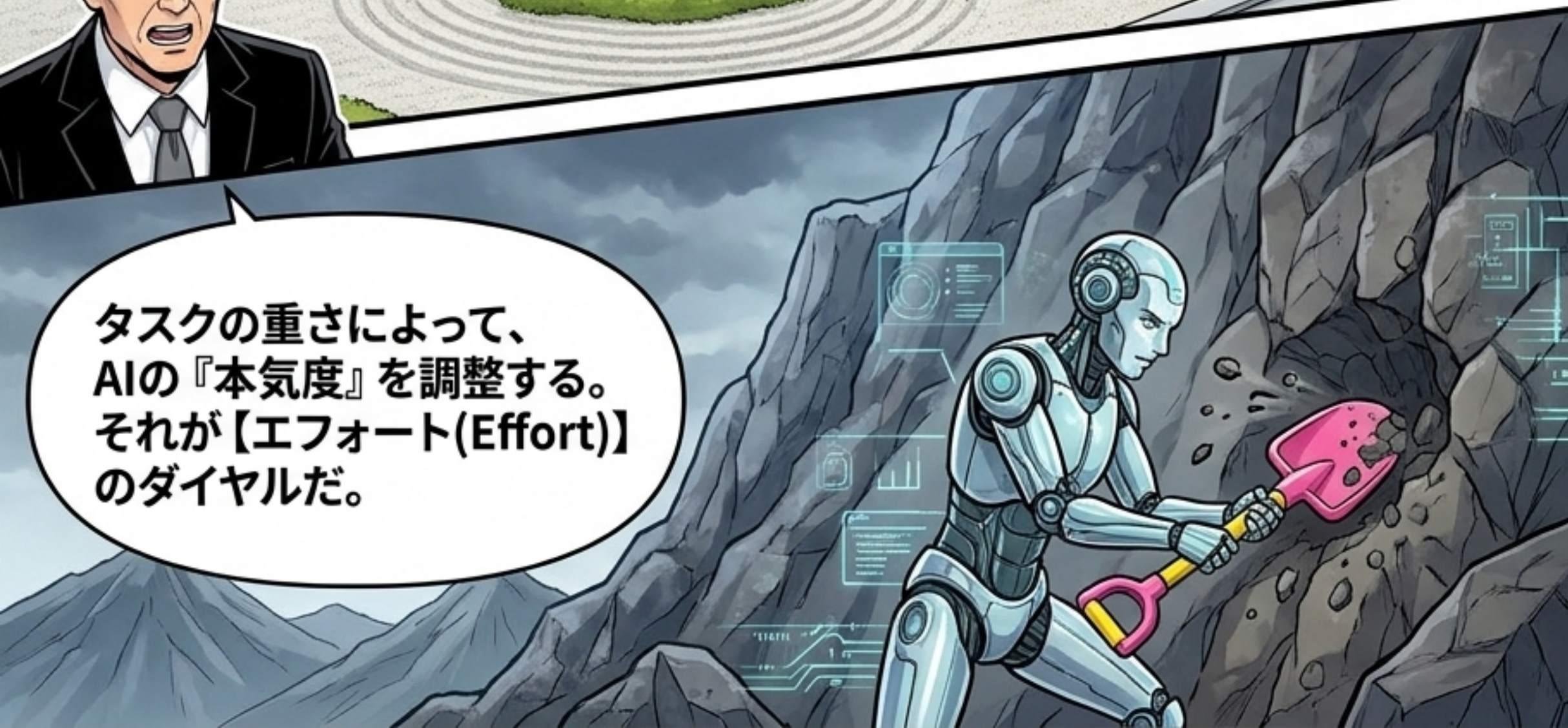
- **【注意1】**
終わり時を見失うのを防ぐため、必ず
「報告を終えたら作業を終了してく
ださい」と明示すること。
- **【注意2】**
サブエージェントはメインの記憶を
共有しない。「このファイルのこの関
数を…」と材料を明示的に手渡すこと。



馬鹿を言うな！
すべてのタスクに最大
リソースを割くのは、
庭の掃除にクレーン車を
持ち出すようなものだ。



タスクの重さによって、
AIの『本気度』を調整する。
それが【エフォート(Effort)】
のダイヤルだ。



じゃあ、全部のタスクを
最大パワー(Max)で
実行させれば
完璧ですね！



エフォート・レベル診断表

レベル	挙動	用途	コスト
Low (低)	読み込み最小限、 最速回答	誤字脱字修正、 コード整形、 コミットメッセージ構成	低
Medium (デフォルト)	コストと精度の バランス型、 自己検証含む	一般的な開発タスク、 関数の実装、 既存コード説明	中
High/Max (高)	大量ドキュメント読込 +生成と検証の 二重チェックループ	複数ファイルの 大規模機能追加、 未経験ライブラリ導入、 クリティカルなバグ修正	高

※Max設定は大量のトークンを消費するため、
セッション終了時に自動リセットされる安全装置付き。

これが『エフォート
(Effort)のダイヤル』の
具体的な使い分けだ。

なるほど、
タスクの重要度に応じて
切り替えるんですね。

エフォート設定が重複した場合、
この『優先順位』に従って
勝者が決まる。

1. 環境変数
(CLAUDE_CODE_EFFORT_LEVEL)
- 最も強力。他をすべて上書き。

2. スキル/サブエージェントの
フロントマター
- コマンド実行中のみ有効。

3. 現在のセッション
(/effort コマンド)

3. 現在のセッション
(/effort コマンド) -
その場での手動調整。

4. 設定ファイル (settings) -
プロジェクト全体のデフォルト設定。

5. モデルごとのデフォルト -
指定がない場合の状態。

序列を把握し、
意図しないリソース
消費消費を防ぐんだ。



憲法 (CLAUDE.md)、
防音室 (サブエージェント)、
火力調整 (Maxエフォート)……
すべて組み込みました！

```
> /security-check
```

3つの秘伝が、
今一つに統合される——。



CLAUDE.md

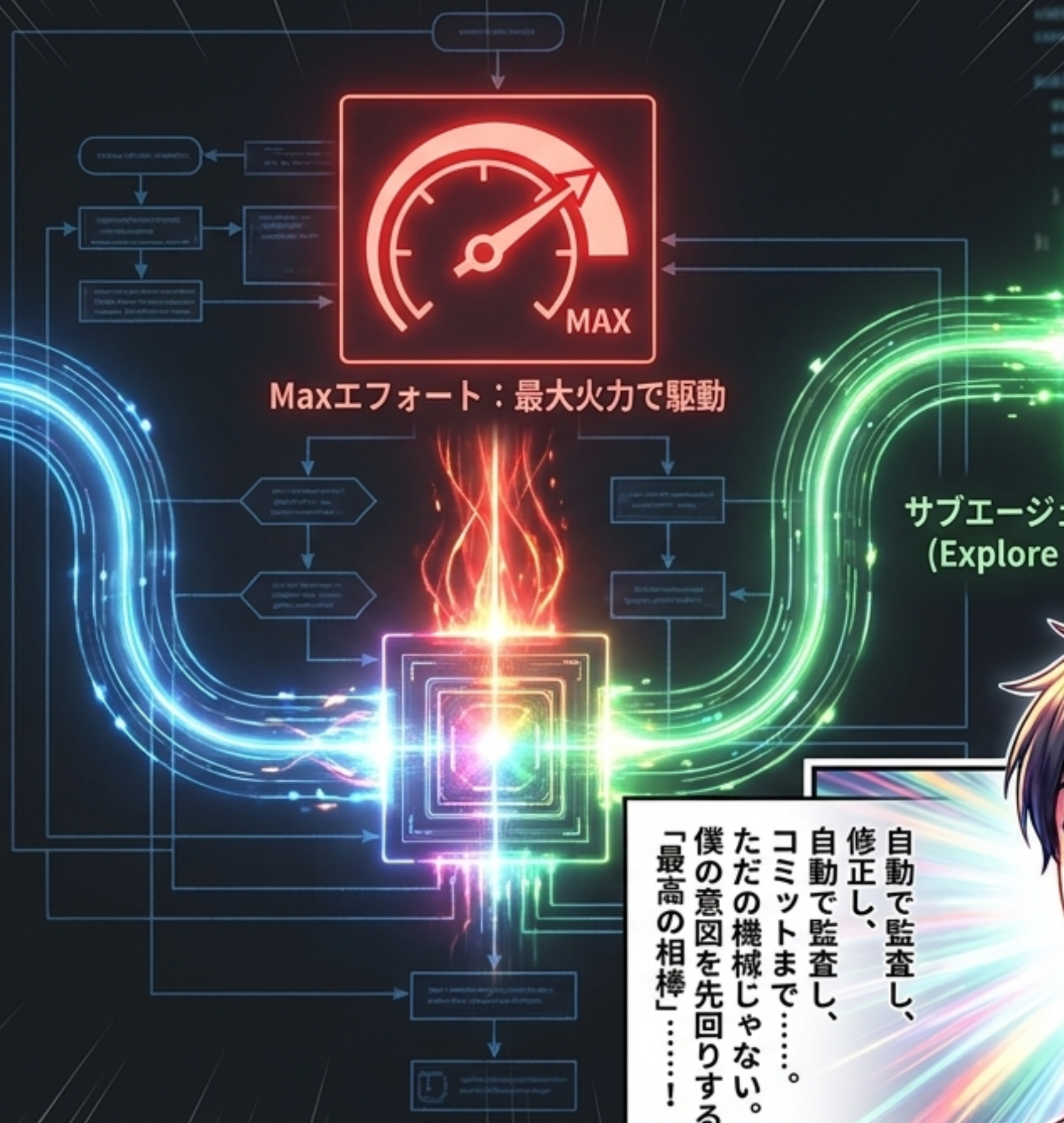
プロジェクトの掟を読み込み中...

```
// 以下のコードを実行して入力する
let input = document.getElementById("input");

let output = document.getElementById("output");

function handleSubmit() {
    let inputText = input.value;
    let outputText = "入力されたテキスト: " + inputText;
    output.value = outputText;
}

// ボタンをクリックして実行
document.getElementById("button").addEventListener("click", handleSubmit);
```



サブエージェント：別室にて検証・修正開始
(Explore → Plan → Code → Commit)



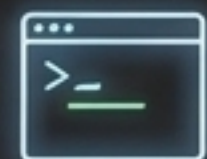
ターミナルに、
もう一人の自分が
誕生しました……!!

自動で監査し、
修正し、
自動で監査し、
コミットまで……。
ただの機械じゃない。
僕の意図を先回りする
「最高の相棒」……!!

今日から始める3つのステップ



3. 鏡合わせの対話 -
/claude を起動し、「私の開発スタイル
についてどう把握してる?」と尋ね、
AIの認識を検証する。



2. レシピの記録 -
そのフレーズを CLAUDE.md や新規
スキルとして記述する。



1. 現状の把握 -
一番頻繁にAIに頼む「お願いの
フレーズ」を一つ特定する。

魔法のレシピを
作るのは、
君の専門知識と美学だ。

AIとの協働において、
タクトを振るのは常にあなたです。
さあ、今すぐターミナルを開き、
あなただけの第一歩を
刻みなさい!