

# コーディングエージェントの現在地とエンジニアの 仕事の変化について

大吉祥寺.pm 2025

# 自己紹介

- azukiazusa
- <https://azukiazusa.dev>
- FE (フロントエンド|ファイアーエムブレム)  
が好き



# アジェンダ - コーディングにおける AI との協業は避けられなくなっている

- 2025年現在ではどのような類型のコーディングエージェントが存在するのか？
- エンジニアの仕事はどうに変化し、そして変わらないものは何か？
- 急速な技術革新に伴いどのような課題が生じているのか

# AIコーディング支援の進化



## 補完型

コード補完やリファクタリングを支援

GitHub Copilot



## チャット型

自然言語での指示に基づくコード生成

ChatGPT



## エージェント型

自律的にタスクを完了するAI

Claude Code, Devin

# 補完型AI（GitHub Copilotなど）

```
function add // ← ここまで書くと...
```

```
function add(a: number, b: number): number {  
    return a + b;  
}
```

- 開発者が書き始めたコードを補完し Tab キーで提案を受け入れ
- ペアプログラミングのような体験
- 開発者が主導権を持つ

# 補完型AIの利点と制約

## 利点

- 定型的なコードの自動生成: テストのモックデータの生成のように決まり切ったコードを素早く生成
- 新しい言語やフレームワークの学習: ユーザーが知らない書き方を提案

## 制約

- ユーザーが何かを書き始めない限り、AIは何も提案してくれない
- 従来の「開発者がコードを書く」という基本的なワークフローは変わらない
- コードベース全体を理解していないため、コンテキストを無視した提案が多い

# チャット型AI (ChatGPTなど)

- 自然言語での指示が可能
- プログラミングの知識がなくても  
使える

The screenshot shows the ChatGPT interface with a dark theme. At the top, it says "ChatGPT" with a dropdown arrow. In the top right corner, there are "共有する" (Share) and three-dot menus. A search bar at the top right contains the placeholder "TypeScript で足し算を行う関数を書いて". Below the search bar, a message box displays the following text:

TypeScript で足し算を行う基本的な関数は以下のように書けます：

```
typescript
function add(a: number, b: number): number {
    return a + b;
}

// 使用例
const result = add(3, 5);
console.log(result); // 出力: 8
```

Below the code, there's an "解説:" (Explanation) section with the following bullet points:

- `function add(a: number, b: number): number`  
これは、2つの `number` 型の引数を受け取り、`number` 型の結果を返す関数です。
- `a + b` で足し算を実行しています。

At the bottom of the explanation section, there's a note: "もし別の型（例: 文字列との結合や BigInt）なども考慮したい場合は、その用途に合わせて調整可能です。" followed by a downward arrow icon.

In the bottom right corner of the message box, there are " COPY" and " EDIT" buttons. Below the message box, there's a footer with "質問してみましょう" (Ask me a question), a plus sign, "ツール" (Tools), and other icons.

# チャット型AIの利点と制約

## 利点

- 自然言語での指示が可能: プログラミングの知識がなくても使える
- コードの生成だけでなく、コードの説明やデバッグも可能

## 制約

- コードの品質やセキュリティに関する保証がない
  - コードをレビューするエンジニアのスキルが依然として重要
- チャット型の UI であるため、往復のやり取りが必要
- コードベース全体のアーキテクチャを理解し、適切なコードを生成することは難しい

# そして、コーディングエージェントの登場

## 自律的にタスクを完了する

- ユーザーからの指示に基づいて自律的にコードを生成し、実行するAI
- 人間の介入を最小限に抑えて、タスクの完了を目指す
- [Cursor](#) が登場したあたりから注目を集めた

日本では[CLINE](#)に全部賭けるの記事が出た頃から話題に

# AIエージェントの仕組み

- 複雑なタスクを最終的な解決に向けた論理的なステップの連続に区切る (Chain of Thought)
- 外部と対話するための「ツール」を使用してタスクを実行
  - ファイルの読み書き
  - シェルコマンドの実行
- ツールの実行結果からフィードバックを受け取り、次のステップを決定 (Self-Correction)
  - 例えばテストが失敗した場合、元の計画を修正しコードの修正を試みる
- 計画・実行・フィードバックのループを繰り返すことで、タスクの完了を目指す

# コーディングエージェントの3つの類型<sup>(1)</sup>



## エディタ型

- 開発者が常に監視・介入可能
- 安全性重視
- よく使い慣れたエディタで動作

Cursor, Cline, Windsurf



## CLI型

# どのタイプのエージェントを使うべきか？

- タスクに応じて使い分けられるのが理想
- ユーザーの介入をほとんど必要としない自律型エージェントが最も効率的だが、タスクが複雑になると失敗する可能性が高い
- 人間の仕事の進め方に例えるならば、マイクロマネジメントと放任主義をどのように使い分けるか
- 大まかに以下の 3 つのタスクの分類で使い分けを考えると良い
  - 小規模なタスク
  - チャレンジングなタスク
  - 大規模なタスク

# 30分以内の小規模なタスク

## 自律型エージェントが最適

- 例：バグ修正（表示崩れ、ボタン位置調整など）、フィーチャーフラグの掃除など
  - 目的が明確かつ、影響範囲が小さいもの
- 複数タスクの並行処理
- 開発者のコンテキストスイッチの削減

## 活用例

Slackで「フィーチャーフラグxxxを削除して」と指示  
→ エージェントが自動でPR作成

# チャレンジングなタスク

CLI型 or エディタ型、状況により自律型

- **特徴:** 既存コードベースに参考がない、情報が不十分
- 自律型エージェントでは袋小路に入る可能性が高く、手戻り作業のほうが多くなる可能性がある

## CLI型を選ぶ場合

- 実装方針を AI と相談しながら決める
  - Plan モードなどが有効
- 間違った方向に進みそうであれば、隨時人間が介入する

## 自律型を選ぶ場合

- ミーティング中など介入できない時間帯
- 何もしないよりは60%程度の成果でも後で修正すれば効率的
  - 気に入らなければ捨ててしまってもいい
- 複数のエージェントを並行稼働させ、完了順にレビュー

# 大規模なタスク

## 人間によるタスク分解が必要

- 複数のサブタスクへの適切な分割が必要
  - [When to Use Devin - Devin Docs](#)
- 高度な知識と経験を持つテックリードやアーキテクトの判断が必要
- AI に限らず人間の開発者にタスクを任せると同じ

# エンジニアの仕事はこう変わった



## コードを書く

- コードを書く時間が多くの時間を占めていた
- コードの書き直しには相応のコストがかかる

→



## タスクを管理する

- コードはAIが高速で生成

# タスクを管理するスキル

## 適切なタスク分解

- プロジェクト全体の目標理解し、実現可能なサブタスクへの分解
- 各タスクの特性を見極めて、適切な実行者（エージェント）を選定

## 質の高い指示作成

- AIが理解しやすい明確な指示
- プロジェクトの設計や目的を記述したドキュメント

# AIを使うことは、もはやマネジメントである<sup>(1)</sup>

- マネージャーの仕事は作業分担と権限委譲
- 指示の言語化が重要
  - 上司が「あれやつといて」と言うだけでは不十分
  - AIに指示するプロンプトも同様

優れたマネージャー（対メンバー）	AIへのスタンス（対AI）
メンバー一人ひとりの特性を理解し、最適な仕事を割り振る。	各AIモデルの得意・不得意を理解し、タスクに応じて最適なAIを使い分ける。
仕事の目的やビジョンを共有し、メンバーの主体的な貢献を促す。	プロンプトを通じて目的や背景を明確に伝え、AIの「推測」の精度を高める。
期待するアウトプットのイメージを具体的に伝え、手戻りのない進行を管理する。	求める形式や制約条件を具体的に指示し、AIのアウトプットを望む方向へ導く。
定期的な対話を通じて進捗を把握し、的確なフィードバックで成長を支援する。	一度の指示で終わらず、対話を重ねてアウトプットの質を向上させる。

# 指示の言語化

## 曖昧な指示

カンバンボードのテストを書いてください

## 明確な指示

カンバンボードのタスク操作に関するテストケースを作成してください

- カンバンボードに表示されるタスクは、API から取得したデータに基づいていること
- 「新規追加」ボタンをクリックすると「新規追加ダイアログ」が表示されること
- タイトルを入力するまで submit ボタンは無効化されていること
- API のコールに失敗した場合、エラーメッセージがトーストで表示されること

テストの記述は以下の要件に従ってください

- API のモックは `msw` を使用すること
- テストを書く際の注意事項は `docs/test-guidelines.md` を参照すること

## 指示の言語化

### 暗黙な指示

カンバンボードのテストを書いてください。

### 明確な指示

カンバンカードに「テストを書いてください」と書かれています。

AI時代も人間が設計・技術選定する点は  
変わらない。コードに書き起こす手段が  
変わっただけ。

- ・ ルールによって決まるルールは、そのルールが適用したデータに基づいていること
- ・ 「新規登録」ボタンをクリックすると、「新規登録ガイドログ」が表示されること
- ・ タイトルを入力するまで、submitボタンは無効化されていること
- ・ API のコードに実装した場合、エラーメッセージがトーストで表示されること

テストの記述は以下の要件に従ってください

- ・ API のモックは「new」を使用すること
- ・ テストを書く際の注意事項は「<https://test-guidelines.net>」を参照すること

# ドキュメントを育てる

- AI コーディングエージェントには長期記憶（メモリ）を管理する手段が用意されている
  - Claude Code: CLAUDE.md
  - Cursor: .cursor/rules/
- プロジェクトの設計やコーディング規約などを記述する
- メモリはセッションの開始時に自動でコンテキストに取り込まれる
- よく整備されたドキュメントは新規参入した人間にも有用

# プロンプトエンジニアリングの学習

AI エージェントを効果的に活用するためには、プロンプトエンジニアリングの理解が重要  
→ LLM の仕組みを理解し、適切なプロンプトを設計する

- [Prompt Engineering Guide](#)
- プロンプトエンジニアリングの概要 - [Anthropic](#)
- [ChatGPT プロンプトエンジニアリング](#)
- AI エージェント OSS のプロンプトを読むのもおすすめ



# テストや Lint の重要性

- AI エージェントを十分に活用するためには、テストや Lint の整備が不可欠
  - AI エージェントはフィードバックを受けて自己内省するため、テストや Lint の結果が重要な情報源となる
- AI がコードを書く速度に比べてテストや Lint の実行時間はボトルネックになりがち
  - CI の高速化が生産性の向上に直結する
- プロジェクトのコーディング規約を自作 Lint ルールとしてつくるのも有効

# コードレビュー中心の仕事

- AIが素早くコードを生成するため、開発者の仕事はコードレビュー中心にシフト
- 自分が出すプルリクエストも知らないコードとして、自己レビューする必要がある
- レビュー比率の増加: コードを書く楽しさの減少
  - 簡単なタスクはAIがすべて処理してしまい、人間には重要な判断だけが残される
- AI生成コードの速度: 人間がボトルネックに

# コードレビューがボトルネックにならないために

- レビューの自動化: Lint や自動テストを活用
- プルリクエストを適切なサイズに分割
  - 適切にモジュール化されているコードベースでは、AI エージェントが生成するコードも小さなモジュールに分割されるため、レビューの負担が軽減される
- AI によるコードレビュー
  - 論理的な誤りやセキュリティの問題を検出するのは得意

Claude Codeによる生産性向上の限界 | すていお

# 従来の開発生産性が高い組織がそのまま AIエージェントを活用できる可能性が高い

- ドキュメントの整備
- テストや Lint の整備
- コードを小さなモジュールに分割
- コードレビューの文化
- これらの文化が根付いている組織は、AIエージェントを活用する際にもスムーズに移行できる

従来の開発生産性が高い組織がそのまま AIエージェントを活用できる可能性が高い

- ・ドキュメントの整理
  - ・テストや CI の実行
  - ・コードの修正
  - ・コードの生成
  - ・これらの文化が根付いている組織は、AIエージェントを活用する際にもスムーズに移行できる
- AIがコードを書くようになつても、根本の部分は変わらない。

コーディングエージェントに期待しすぎない

# 「始めの一歩」のハードルを下げる

## ✗ 期待しすぎてはいけないこと

- 何も介入せずに完璧なコードが仕上がる
- 複雑な要求も一発で正確に実装してくれる
- プロジェクト全体の設計を任せられる

## ✓ 実際の価値

- コードを書き始めるハードルを下げる
- 最初の足がかりを提供してくれる
- そこから人間が改善・修正していく

# 「机に5分座る」効果



## 勉強のやる気を出すテクニック

「とりあえず机に5分座る」  
**作業興奮 (Arbeitsrausch)**

「やる気が出るから行動する」のではなく、「行動するからやる気が出る」



## コーディングでも同じ

「とりあえずAIに骨格コードを生成してもらう」  
→ 何かができてしまえば、そこから改善していくやる気が湧く  
動くコードが目の前にあると「もう少し改善しよう」という気持ちが自然に湧く

# ジュニアエンジニアの役割はどうなる？

- 2027年までにジュニアエンジニアが行っているタスクの約26%が、AIによって拡張または完全に自動化されると予測されている
  - [The impact of AI on junior application developer skills](#)
- ジュニアエンジニアが不要になるのではないかとの懸念もある

# AIに精通したジュニアエンジニアが活躍するチャンス

- 新しいAI時代のエンジニアは、AIを活用したうえでプログラミングの学習を始めることが予測される
  - AIは学習コーチとして適切な存在であり、わからないことを自然言語で質問したり、知らない知識を教えてくれたりする
- AIネイティブの世代が登場した場合、私達が思いも寄らない観点での問題解決や新しいアイデアを提供してくれる可能性がある

<https://github.blog/ai-and-ml/generative-ai/junior-developers-arent-obsolete-heres-how-to-thrive-in-the-age-of-ai/>

# 生成 AI との付き合い方

- 学習でのAI活用法
- 生成AIの盲信

# 学習でのAI活用法

- 学習目的ではAIに頼らずに書く
  - すぐに答えを求めるのではなく、自分で考える
- プログラミングでは**写経**と呼ばれる学習方法が広く行われてきた
- 自ら手を動かした経験は記憶に残りやすい
- **自分の言葉でAIに質問する**
  - 知らなかった構文や書き方を学ぶためにAIを活用
  - AIをパーソナルコーチのように活用することで、学習の効率を高める

基礎力 + 瞬発力

# 基礎力 → プログラミングの基礎的な知識やスキル

- AI はあくまでツールであり、最終的な判断は人間に委ねられる
- AI が生成したコードが正しいかどうかを判断するためには、プログラミングの基礎的な知識やスキルが必要
- 基礎的な知識は普遍的であり、長期間にわたって役立つ
- AI とのチャットでのやり取りでは体系的に学習することは難しい  
→ 書籍での学習をおすすめ <https://azukiazusa.dev/blog/computer-it-books/>

# 瞬発力 → 新しい技術やツールに対して迅速に適応できる能力

- 日々新しいAIツールが登場している
- 新しいツールが出たらとりあえず触ってみよう
  - いち早く試すことで優位に立てる
  - 新しいツールに触れることで、異なるアプローチや設計思想を学べる
- サンクコストを捨てる
- 子どものような好奇心旺盛な姿勢で新しい技術に触れる人が強い

# 生成AIの盲信

AIが生成したコードを無批判に受け入れてしまう危険性

批判的思考力を身につけて、AIを適切に活用しよう

# 劇場のイドラ (Theater of Idols)

フランシス・ベーコンの「4つのイドラ」より

権威や流行に盲従してしまう認知バイアス

## AI時代における「劇場のイドラ」

- **AI の権威性**：「AI が言うから正しい」
- **技術の流行性**：「最新のAIツールなら間違いない」
- **自動化への過信**：「機械の方が人間より正確」



権威への盲従は  
判断力を奪う

# 批判的思考力の重要性

批判的思考：与えられた情報や状況を鵜呑みにせず、多角的な視点から分析し、論理的に考察して、客観的な判断や評価を行うための思考力

## AI時代に必要な批判的思考

- **出力の検証**：生成されたコードの妥当性確認
- **複数の視点**：異なるアプローチの検討
- **根拠の確認**：なぜその解決策なのかを自分の言葉で説明できるように

### 思考力改善ドリル

批判的  
思考 から  
科学的  
思考へ



植原亮

クイズ感覚で問題を解いて  
クリティカル・シンキングの力を養い、  
科学リテラシーがぐんぐん身につく！  
考える力を磨くための27章。

keigo-shobo

# 時代の転換点を楽しもう

人類の歴史は技術革新による仕事の変化の連続

狩猟採集 → 農耕社会 → 産業革命 → 情報化社会 → AI時代

- 歴史の転換点に立ち会える貴重な体験
- 数年後に「2025年は変化に戸惑っていたけど、確かに時代の転換点だったね」と言える日が来る

# まとめ

- 自律型エージェントの 3 つの類型
- エンジニアの役割は「コードを書く」仕事から「タスクを指示し、レビューする」仕事へシフト
- しかし、開発の根本の部分は変わらない
- 生成 AI との付き合い方のポイントは、基礎力と瞬発力、そして批判的思考力