AI時代のソフトウェア開発を考える（2025/11版）

Nov 14, 2025 @ Agile Japan 2025

rev.9

Takuto WADA

id:t-wada 𝕏 @t_wada @twada @twada

📷🤯 𝕏 🤯

The End of Programming as We Know It

Tim O’Reilly

https://www.oreilly.com/radar/the-end-of-programming-as-we-know-it/

「Vibe Coding」の誕生

Andrej Karpathy
@karpathy

There's a new kind of coding I call "vibe coding", where you fully give in to the vibes, embrace exponentials, and forget that the code even exists. It's possible because the LLMs (e.g. Cursor Composer w Sonnet) are getting too good. Also I just talk to Composer with SuperWhisper so I barely even touch the keyboard. I ask for the dumbest things like "decrease the padding on the sidebar by half" because I'm too lazy to find it. I "Accept All" always, I don't read the diffs anymore. When I get error messages I just copy paste them in with no comment, usually that fixes it. The code grows beyond my usual comprehension, I'd have to really read through it for a while. Sometimes the LLMs can't fix a bug so I just work around it or ask for random changes until it goes away. It's not too bad for throwaway weekend projects, but still quite amusing. I'm building a project or webapp, but it's not really coding - I just see stuff, say stuff, run stuff, and copy paste stuff, and it mostly works.

8:17 AM · Feb 3, 2025 · **5M** Views

1.3K 5K 29K 15K

https://x.com/karpathy/status/1886192184808149383

2025年、世界は Vibe Coding の炎に包まれた

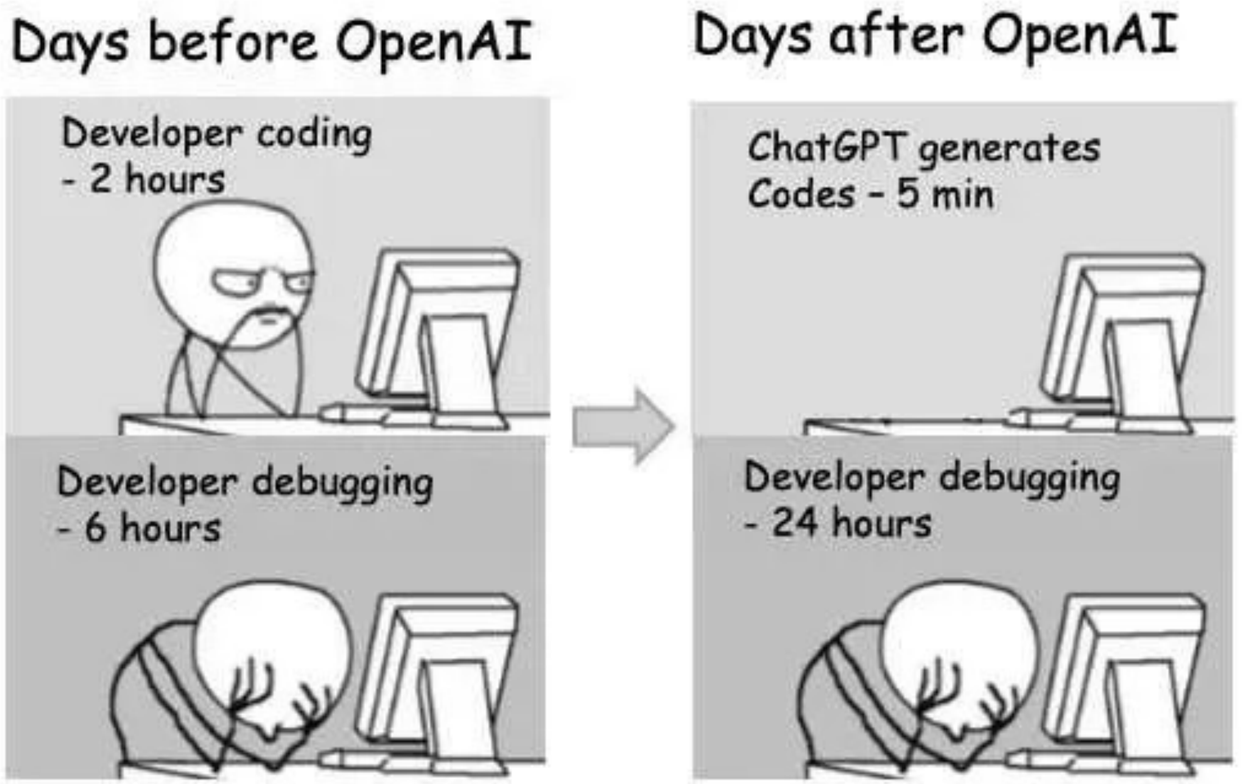
- 2025年、AI エージェントの進化、特に Vibe Coding の登場により、プログラミングの速度（物的生産性）は圧倒的に高速化した
  - 特に 2025年5月に Claude Max プランで（その後 Pro プランも）Claude Code が従量課金から定額制になったインパクトが大きかった
  - コスト構造とインセンティブを変え、投機的プログラミング（ガチャともいう）の世界をひらいた
- Vibe Coding をソフトウェアエンジニアリングの観点から表現するなら
  - 製品品質の中の機能適合性のみを実際の動作確認で検証することで、コードレビューを省略し、開発のスループットを最大化している状態

これが「2025年の崖」の年に起こるとは……

Takuto Wada
@t_wada

Translate post

「2025年の崖」問題の勝負の年、人類は保守不能なシステムをより高速に構築する手段を得ていた……

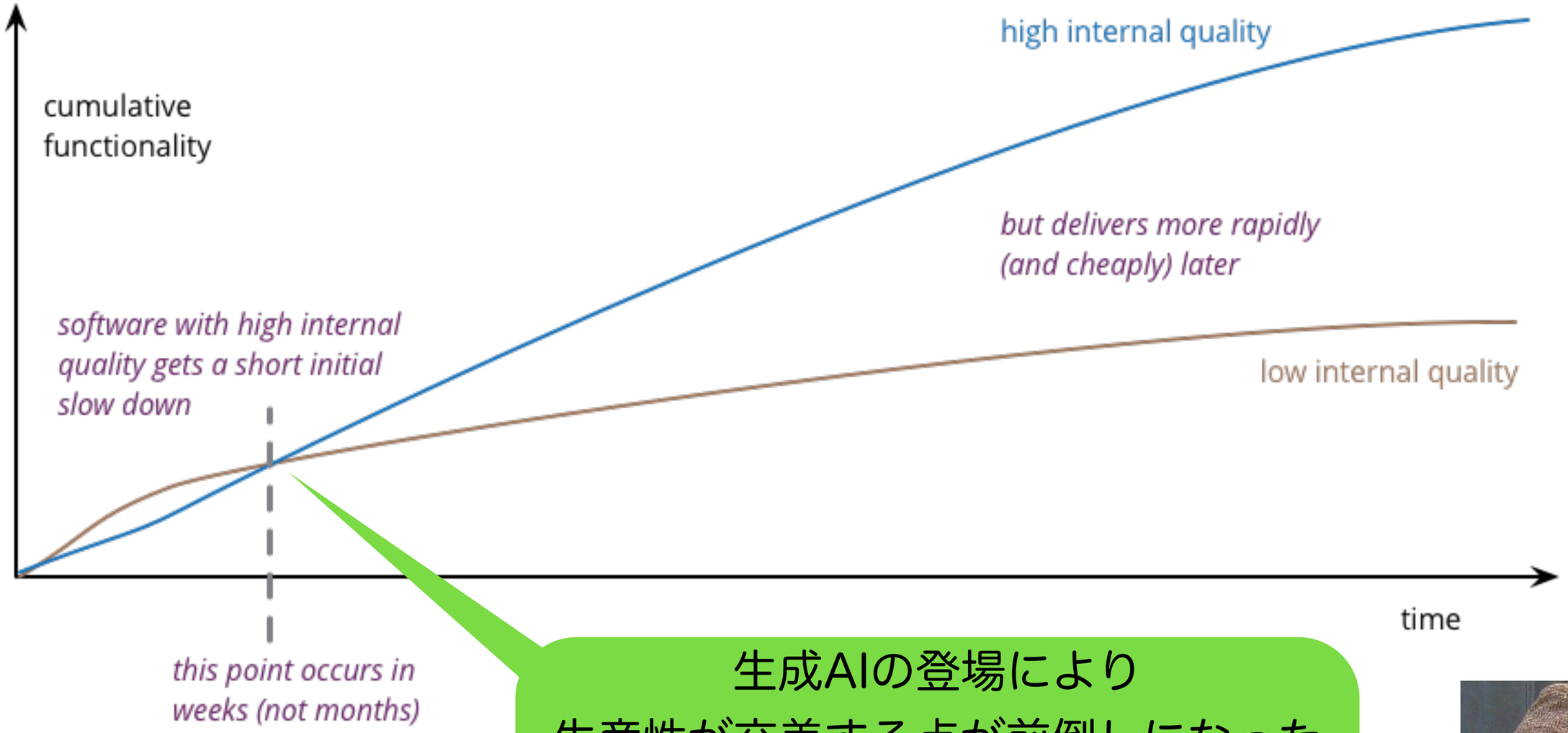


https://www.reddit.com/r/ProgrammerHumor/comments/zdvpwb/how_openai_chatgpt_helps_software_development/

そして時が加速した

- Vibe Coding をはじめとした AI 活用による開発生産性の高まりが原因で、開発規模が大きくなると発生する諸問題が、ごく短期間で発生するようになった
  - 技術的負債の高速な積み上げによる開発スループット低下
  - レビューできない分量のコード
  - Unknown-Unknown 領域の拡大
  - 増加するセキュリティリスク
- 内部品質への投資の損益分岐点（従来は1ヶ月程度）はどう移動したか

時が加速し、損益分岐点が移動した



生成AIの登場により
生産性が交差する点が前倒しになった
いまは数日から1週間くらいか

https://martinfowler.com/articles/is-quality-worth-cost.html

問題の構造は変わらず、圧倒的に顕在化が早まっただけ

- 技術的負債の高速な積み上げによる開発スループット低下は新たな課題か => 否
- 「レビューが課題となる」は新たな課題か => 否
- 「正しく適切なドキュメントを書けばコーディングは些細な問題」は新たな考えか
  - 否: ソフトウェアエンジニアリングの歴史にはずっと存在する
    （何度も墓からよみがえるタフなゾンビ）
- 落ち着いて見ると、問題の構造はあまり変わっていない。道具が変わり、顕在化するのがあまりにも早くなっただけ
- プログラミングを越えてソフトウェアエンジニアリングが必要になるタイミングが大幅に早まった

ソフトウェア
エンジニアリングと
問題解決の歴史

ソフトウェアエンジニアリングは時間で積分したプログラミング

我々が提案するのは、「ソフトウェアエンジニアリング」とは単にコードを書く行為のみならず、組織が時間の経過に応じてコードを構築し保守するために用いるツールとプロセス全てをも包含するということである。長期間にわたりコードを価値ある状態に保守することを最もうまく行うためのプラクティスとしてソフトウェア組織が導入できるのは、どのようなものか。エンジニアたちはどのようにして、コードベース（codebase）[†1]をさらに持続可能にしつつ、ソフトウェアエンジニアリングの規律自体を厳格化できるだろうか。我々はこれらの問いへの根本的な解を持ち合わせてはいないが、過去20年にわたるGoogleの集団的経験が、解の発見へ向かう道となりうるものを照らしてくれることを願っている。

本書が共有する重要な見識に、ソフトウェアエンジニアリングとは「時間で積分したプログラミング」とみなせる、というものがある。自分たちのコードを、着想し、導入し、保守し、廃止するまでのライフサイクルを通じて**持続可能**（sustainable）[†2]なものとするためにコードに導入できるのは、どんなプラクティスだろうか。

本書では、コードを設計し、コードのアーキテクチャーを定め、コードを書いていく際に、ソフトウェア組織が留意すべきと我々が感じる3つの根本的原則に重点が置かれている。

時間と変化

コードがその存続期間にわたりどのように適応していかなければならないか

スケールと発展

進化するにつれて組織がどのように適応していかなければならないか

トレードオフとコスト

「時間と変化」、「スケールと発展」から得られる教訓に基づき、組織がどのように決定を行うべきか

『Googleのソフトウェアエンジニアリング』 p.x

https://www.oreilly.co.jp/books/9784873119656/

ソフトウェアエンジニアリングは時間で積分したプログラミング

- 「Googleだからそうなんでしょ」と思っていた規模の諸問題が自分たちのところにも短期間で降り注ぐようになった
- 「ソフトウェアエンジニアリングは時間で積分したプログラミング」
  - 8章　スタイルガイドとルール
  - 9章　コードレビュー
  - 10章　ドキュメンテーション
  - 11章　テスト概観
  - 12章　ユニットテスト
  - 13章　テストダブル
  - 14章　大規模テスト
  - 16章　バージョンコントロールとブランチ管理
  - 20章　静的解析
  - 22章　大規模変更
  - 23章　継続的インテグレーション
  - 24章　継続的デリバリー

https://www.oreilly.co.jp/books/9784873119656/

TDD から何を学んだか

- ソフトウェア開発は未知と既知の陣取りゲーム
- 設計に終わりはない
- 終わらない設計を自動テストとリファクタリングで支える
- 実装から設計へのフィードバックが必ずある

https://www.ohmsha.co.jp/book/9784274217883/

DDD から何を学んだか

- ドメインエキスパートとの対話と共通の語彙が大事
- 境界付けられたコンテクストの中で一貫した語彙を使う
- コードとモデルは互いにフィードバックしあう。一方通行ではない
- コードとドキュメントは互いにフィードバックしあう。一方通行ではない

https://www.shoeisha.co.jp/book/detail/9784798126708, https://www.oreilly.co.jp/books/9784814400737/

TOC から何を学んだか

- ボトルネックを解消しないとスループットは上がらない
- 制約理論で考える。リソース効率よりフロー効率
- コーディングがボトルネックだったことはない
  - レビュー待ちの PR がボトルネックであり「在庫」
  - コーディングがボトルネックではないならば、コーディング量を増やして PRをどれだけ積み上げても「在庫のムダ」
  - AI エージェントによる並列開発も同じ構図
- ペアプロやモブプロでレビュー、教育、フロー効率向上を手がけた

PdM から何を学んだか

- 我々はひたすら新機能追加に邁進してしまいがち
  - ビルドトラップ、忍び寄る機能主義（Creeping Featurism）
  - AI もこの傾向が強い。基本的に足し算指向
- 大事なのは Output よりも Outcome（成果）や Impact
- Outcome につながらないなら Output をどれだけ増やしても意味がない

https://www.oreilly.co.jp//books/9784873119250/

目指すべきは
AI とソフトウェア
エンジニアリングの融合

Vibe Coding の革新的なところ

Takuto Wada
@t_wada

Translate post

Vibe Coding は「自分だけが使う、自分がいちばん欲しいソフトウェアをだれでも作れるようになった」ことが革命的かつ真骨頂なのであって、他の人が使うソフトウェアを作ることには（少なくともまだ）向いてないと思います

12:46 PM · Jul 25, 2025 · **666.4K** Views

View post engagements

4　844　3.6K　674

https://x.com/t_wada/status/1948590698762305714

Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI

Ranjan Sapkota*‡, Konstantinos I. Roumeliotis†, Manoj Karkee*‡
*Cornell University, Department of Biological and Environmental Engineering, USA
†University of the Peloponnese, Department of Informatics and Telecommunications, Tripoli, Greece

‡Corresponding authors: rs2672@cornell.edu, mk2684@cornell.edu

***Abstract*—This review presents a comprehensive analysis of two emerging paradigms in AI-assisted software development: *vibe coding* and *agentic coding*. While both leverage large language models (LLMs), they differ fundamentally in autonomy, architectural design, and the role of the developer. Vibe coding emphasizes intuitive, human-in-the-loop interaction through prompt-based, conversational workflows that support ideation, experimentation, and creative exploration. In contrast, agentic coding enables autonomous software development through goal-driven agents capable of planning, executing, testing, and iterating tasks with minimal human intervention. We propose a detailed taxonomy spanning conceptual foundations, execution models, feedback loops, safety mechanisms, debugging strategies, and real-world tool ecosystems. Through comparative workflow analysis and 20 detailed use cases,**

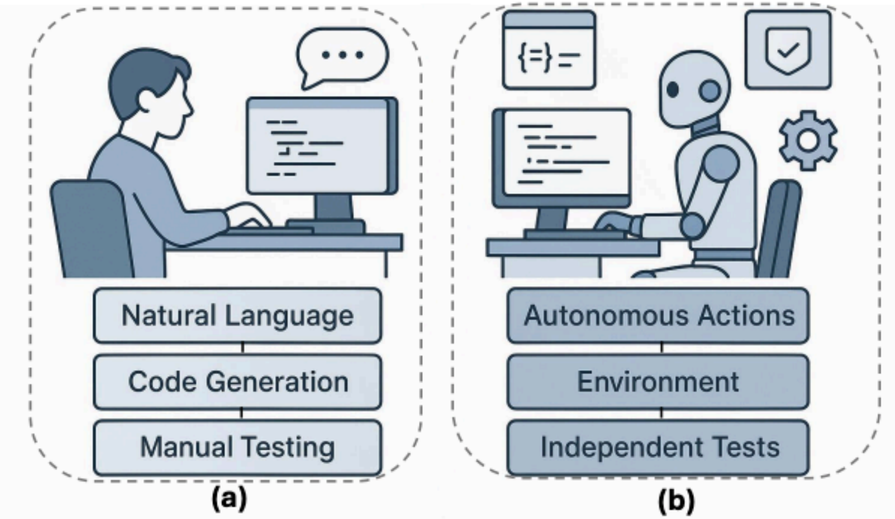
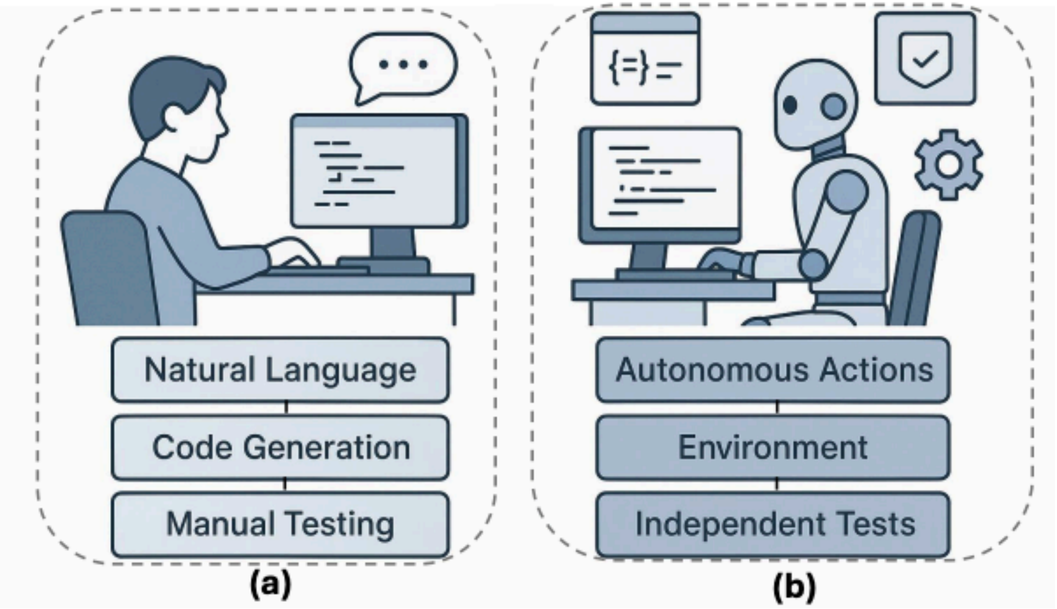
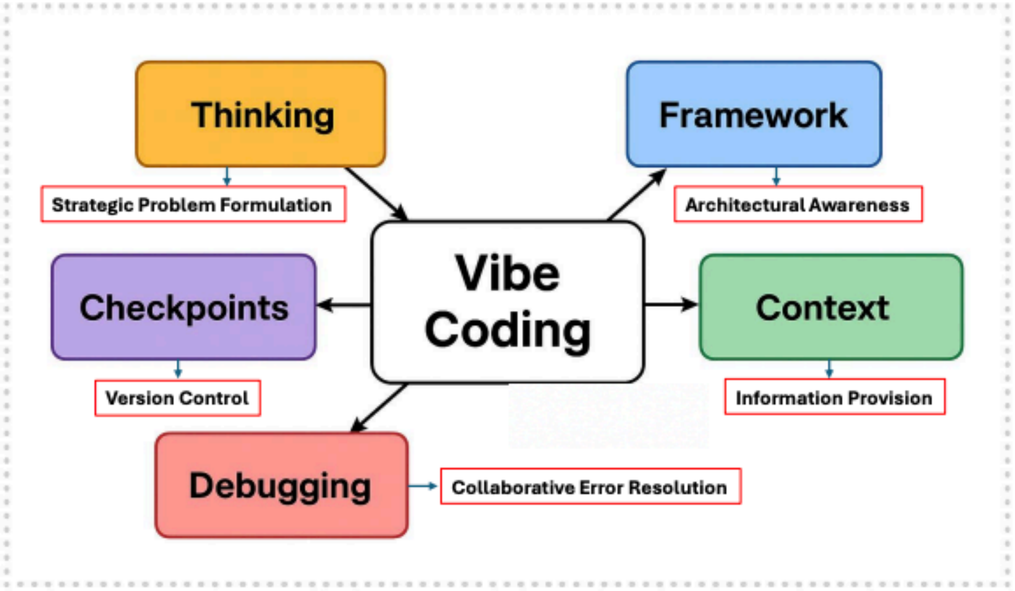
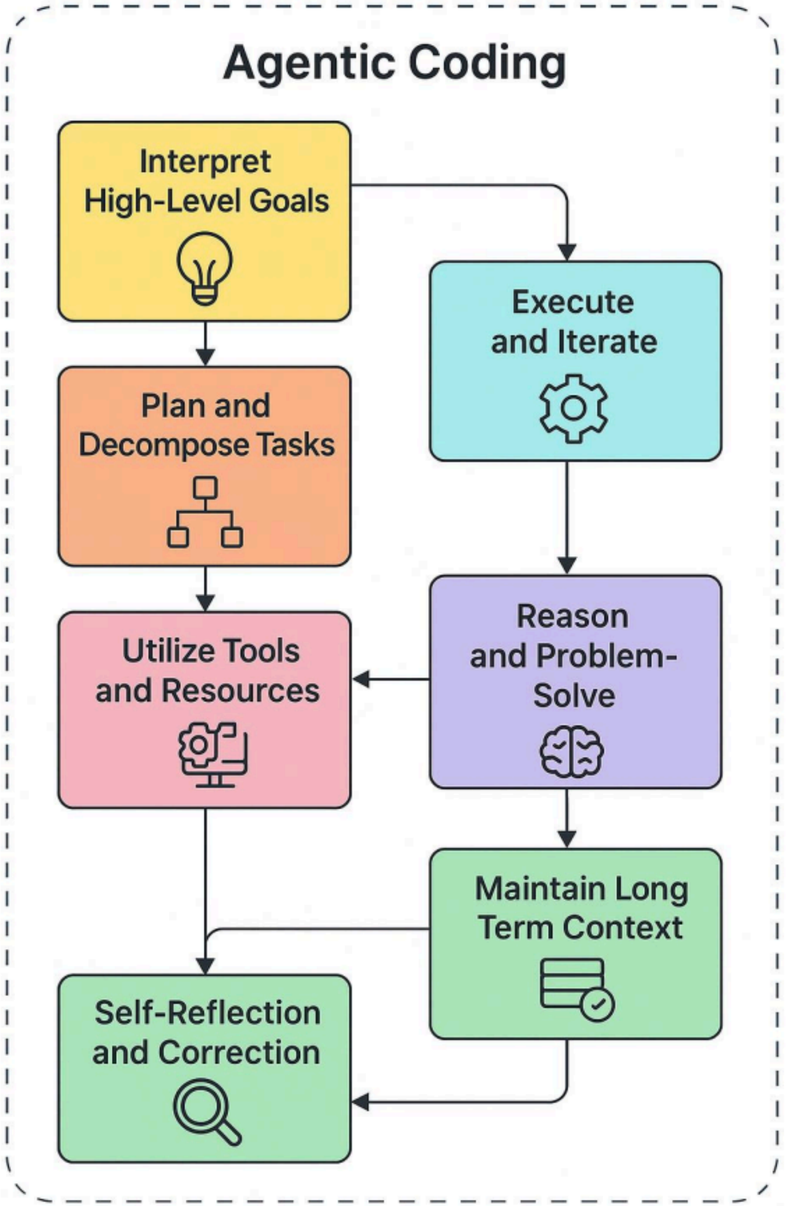


Fig. 1: **Bird-eye-figure** showing a comparative illustration of **(a)** *Vibe Coding*, where a human developer uses

Vibe Coding から Agentic Coding へ





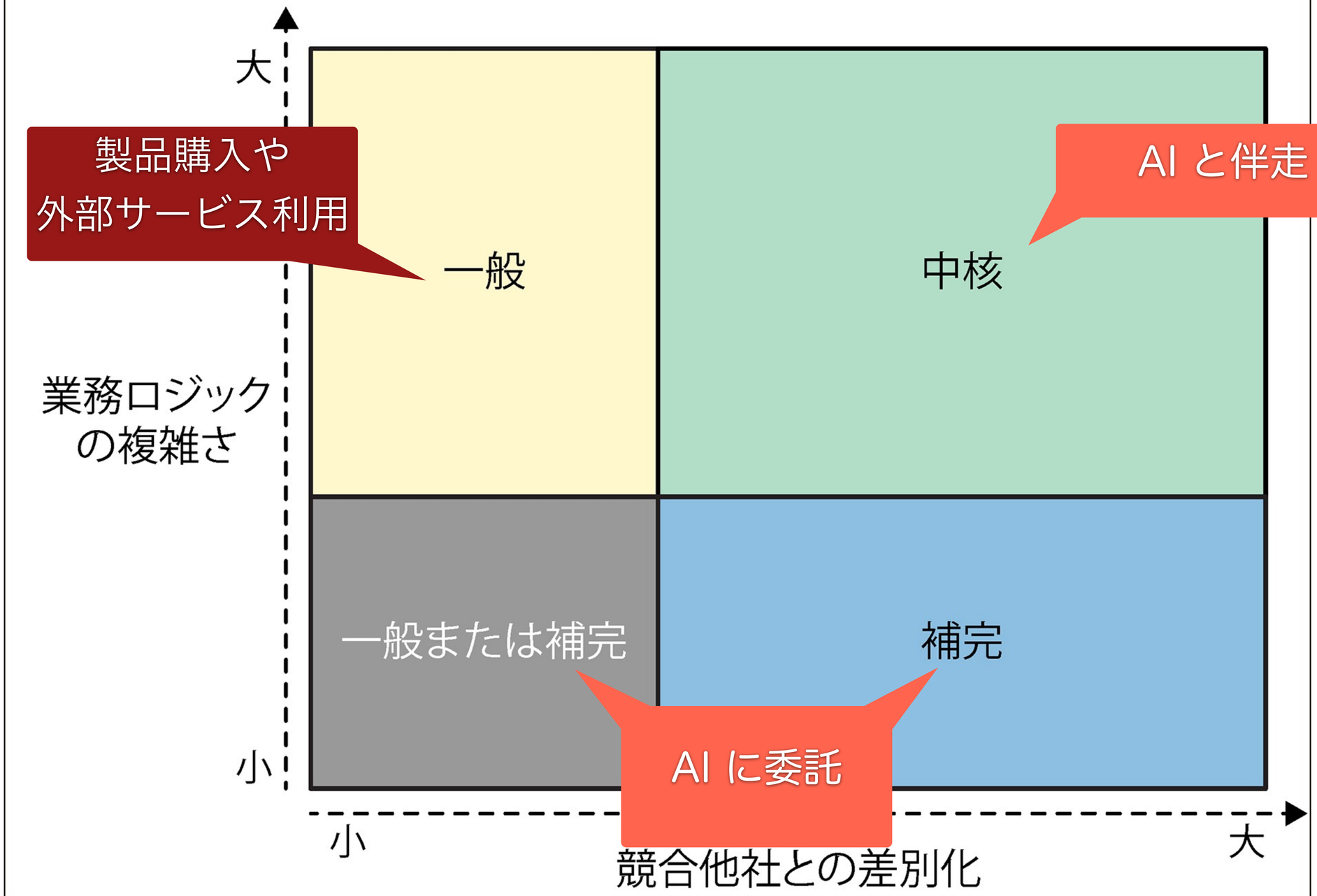


https://arxiv.org/pdf/2505.19443

AI との協業の2つのモード

- AI と伴走
  - AI と対話しながら直列開発
  - コードを書くスピードは（「AI に委託」に比べて）遅い
  - コントロールや状況把握の度合いが高い
  - traditional: "決定論的ではあるものの、人力であるためスケールしない"
- AI に委託
  - 自走する AI たちに任せて並列開発
  - コードが生成されるスピードは圧倒的に速い
  - コントロールや状況把握の度合いが低く、レビューが課題となる
  - emerging: "非決定論的で結果が確率的ではあるものの、非常によくスケールする"

どの領域にどう AI を使うか



https://www.oreilly.co.jp/books/9784814400737/

伴走のパターンと委託のパターン

- 「AI と伴走」のパターン
  - 教習車
  - 助手席
  - オートパイロット
  - 運転席（オーガニック・コーディング）
  - 0-1, 1-100
  - 根負けしない議論相手
  - リサーチアシスタント
  - 批判的レビュアー
- 「AI に委託」のパターン
  - 小人さん（夜のうちにやってくれる）
  - 分業制
  - コンペ

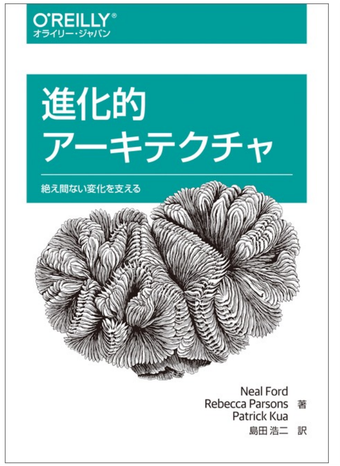
典型的な「AI と伴走」のパターン（2025初夏時点）

- 対話: LLMとの議論、LLMからの質問から設計を生む
  - 生まれた設計をプレーンテキストのドキュメント（Design Doc, ADR）に保存
  - 設計書のレビューを行い、必要であれば議論に戻る
- 設計書からタスクリスト（markdown）を生成し保存
  - タスクのレビューを行い、必要であれば議論に戻る
- タスクリストからタスクを1つ選び、サブタスクに分割
  - サブタスクの順番を TDD のワークフローにあわせて調整
- タスク毎にコーディングエージェントのセッションを立ち上げて（あるいは /compact して）実装
  - タスク毎に Git のブランチを作成（ダメだったらブランチを廃棄する）
  - TDD のワークフローを指定し、レビュー不能な量のコードが一度に生成されることを防ぐ
  - TDD のステップ毎に Conventional Commits のルールでコミットさせる
  - マージ前に全体レビュー。必要であれば人間が手直しし、ブランチをマージ
- 学びを反映させるために再び LLM と議論する

自動化から自働化へ

自動化（automation）から自働化（autonomation）へ

- Agentic Coding とは Reconciliation Loop である。望ましい状態を宣言的に定義し、評価関数（適応度関数）を与える。エージェントはその状態に収束するよう自律的に働く
- AI は自走するが暴走/迷走もする。ガードレール設計としてのソフトウェアエンジニアリングや技術の3本柱（バージョン管理、テスティング、自動化）の重要性がさらに増している



https://www.oreilly.co.jp/books/9784873118567/, https://www.oreilly.co.jp/books/9784814400607/

求める品質特性を定量的表現で AI に伝える

Characteristic	Description	Indicative qualitative measures	Indicative quantitative measures
Correct	It behaves as intended, with key workflows verified preferably through fast-running automated tests.	Edge cases are handled; no regressions during basic use.	Test pass rate near 100%; mutation score > 80%.
Testable	Its design supports meaningful unit, integration and end-to-end testing.	Tests are fast, focused and isolated; naming is consistent and purposeful.	Unit test coverage > 90%; no test flakiness.
Maintainable	The code is readable, modular and consistent enough for others to safely understand and change.	Idiomatic structure; easy onboarding for new contributors.	Low cognitive complexity; stable change rate in core modules.
Scalable	It's designed with non-functional concerns like performance, security and operational robustness in mind.	Design anticipates growth; avoids excessive coupling.	Baseline performance benchmarks; graceful degradation patterns.
Diagnosable	It provides enough instrumentation and structural clarity to support effective troubleshooting.	Logs are meaningful and context-rich; failures are traceable.	Presence of structured logs; alert coverage for key failure paths.
Disciplined	It follows sound engineering practices — version control, CI, static analysis, etc.	Frequent commits with clear messages; workflow is CI-compliant.	Commits gated by CI; clean lint runs; no critical SAST issues.

https://www.thoughtworks.com/insights/blog/generative-ai/can-vibe-coding-produce-production-grade-software

ツール/ライブラリ作者は LLM 親和性を考える時代

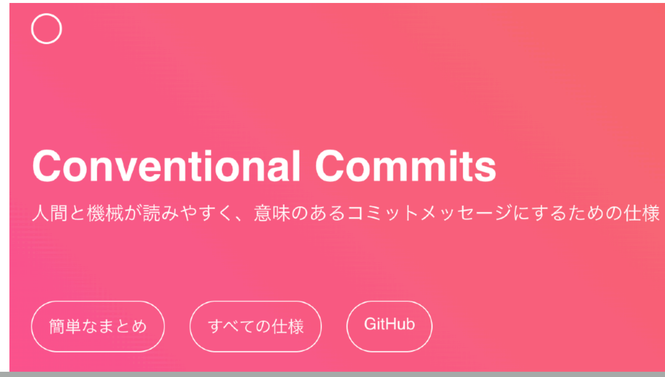
```
AssertionError [ERR_ASSERTION]:

# Human-readable format:
assert(scoreOf(rollsOf(frames)) === 132)
       ¦       ¦       ¦         ¦   ¦
       ¦       ¦       ¦         ¦   132
       ¦       ¦       ¦         false
       ¦       ¦       [[1,4],[4,5],[6,4],[5,5],[10],[0,1],[7,3],[6,4],[10],[2,8,6]]
       ¦       [1,4,4,5,6,4,5,5,10,0,1,7,3,6,4,10,2,8,6]
       133

# AI-readable format:
Assertion failed: assert(scoreOf(rollsOf(frames)) === 132)
=== arg:0 ===
Step 1: `frames` => [[1,4],[4,5],[6,4],[5,5],[10],[0,1],[7,3],[6,4],[10],[2,8,6]]
Step 2: `rollsOf(frames)` => [1,4,4,5,6,4,5,5,10,0,1,7,3,6,4,10,2,8,6]
Step 3: `scoreOf(rollsOf(frames))` => 133
Step 4: `132` => 132
Step 5: `scoreOf(rollsOf(frames)) === 132` => false
```

自動テストの重要性はより高まる

- 適応度関数、ガードレール技術の筆頭
- 自動テストの目的（2024版）
  - 信頼性の高い実行結果に短い時間で到達する状態を保つことで、開発者に根拠ある自信を与え、ソフトウェアの成長を持続可能にすること
- 自動テストの目的（2025版）
  - 信頼性の高い実行結果に短い時間で到達する状態を保つことで、AI に根拠ある判断基準を与え、ソフトウェアの成長を持続可能にすること

包括的な構成管理

- しばらくはモノレポが有利か
- 開発に関わる全てをリポジトリに入れる
- ドキュメントはプレーンテキストで、コードの近くに置かないと腐る
- 「AI へ委託」は AI の速さを殺さない非同期並列開発
  - マイクロマネジメントできないので、後からの監査が大事になる
  - 規律ある Git 利用による変更トレーサビリティを確保
    - Conventional Commits
    - Keep a changelog



https://www.conventionalcommits.org/ja/v1.0.0/

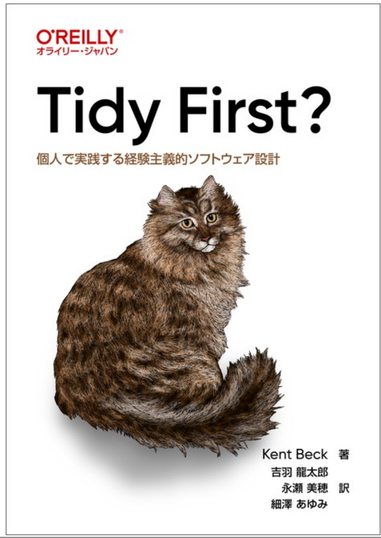
MTBF から MTTR へ

- 確信、把握の度合いを落として開発スループットを上げる選択をしている
- バグゼロのゼロリスク信仰から離れなければならない
- ソフトウェア開発はふたたび Unknown-Unknown 領域に戻ろうとしている
- Known-Unknown や Unknown-Unknown と戦う武器を増やす。 Property-based Testingや、オブザーバビリティの向上で戦う

https://www.lambdanote.com/products/proper, https://www.oreilly.co.jp/books/9784814400126/

レビュー負荷軽減の工夫

- diff 最小化バイアスがジワジワとシステムのエントロピーを上げる
- 『Tidy First?』の知見を活用する
  - Behavior Change と Structure Change を分ける
    - 2つのレビューコストは異なる
  - Behavior Change は不可逆変更。きちんとレビューする
  - Structure Change は可逆変更。
    ソフトウェアエンジニアリングの観点で仕組み化して
    レビューコストを 0 に近づけたい



https://www.oreilly.co.jp/books/9784814400911/

現実を見つめる

正直に現実を見つめよう

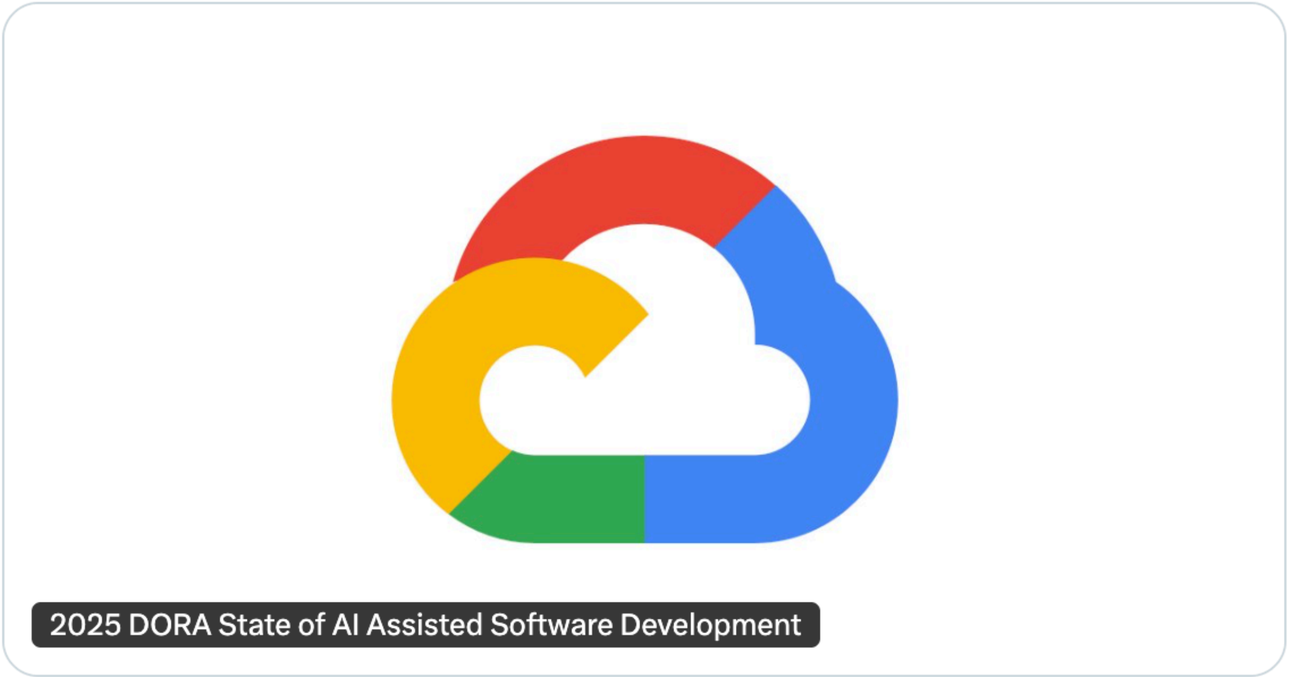
- 我々は最初から正しい設計をすることはできない
  - コードを書き始め、そのとき初めて問題を理解することばかり
  - 実装から設計へのフィードバックが必ずある
- あるとき正しい設計だったものも、時間が経つと正しくなくなる
- システム設計とは、ある時点の合目的的な設計から、次の時点の合目的的な設計までの状態遷移、その意思決定の連続である

DORAレポート2025: AIは増幅器であり、組織の能力を映す鏡

Takuto Wada
@t_wada

Translate post

DORAレポートの2025年版が出た。AIは増幅器であり、組織の能力を映す鏡。高パフォーマンス組織の強みを増幅し、より速く革新的にする。低パフォーマンス組織の機能不全を増幅し、さらなる混乱と不安定さを生み出す。AI導入の成否はツールではなくシステム（組織）の問題。



From cloud.google.com

11:13 AM · Oct 8, 2025 · **90.9K** Views

https://dora.dev/research/2025/

個人と組織が能力を上げなければならない

- 「AI は知識の代替ではなく増幅器」 by Tomohisa Takaokaさん
  - AI（Artificial Intelligence）は IA（Intelligence Amplifier）
- 「高度なAIは自分の鏡みたいなもので、AIから引き出せる性能は、自分の能力にそのまま比例する」 by mizchi さん
  - 高度なAIは組織の鏡みたいなもので、AIから引き出せる性能は、組織の能力にそのまま比例する
- 労力は外注できるが、能力は外注できない
- 個人と組織が共に能力を上げなければならない

わからないものはレビューもデバッグもできない

また、破壊的な技術が登場すると、「これさえあればすべてが何とかなる」と極端な思考に陥る人がいることを指摘し、「プログラミングを勉強しなくても生成AIがやってくれるから大丈夫だろう」と考える人もいるかもしれないとした上で、生成AIは省力化の役には立つが、能力不足を補うものではないと指摘した。和田氏は現場で「労力は外注できるが、能力は外注できない」と言っているそうだ。

最後に和田氏は、ソフトウェアエンジニアはスキルアップから逃れることはできないが、生成AIによってスキルアップのスピードを上げることができる。これでイテレーションが圧倒的に早くなる。今後数年はこのような形で付き合っていくべきではないかと語り、セッションを締めくくった。

https://x.com/t_wada/status/1859756507115684348

能力向上のためにも AI を使う

- Output を出すためだけでなく、自分たちの能力を上げるためにも AI を使う
  - 批判的レビュアーとして使う
  - 根負けしない議論相手として使う
  - 新しい言語を学ぶために使う
- あえてのオーガニック・コーディングも良い
  - アウトプット最小、フィードバック最大になる
  - コード差分からドキュメントを生成する手法もある
- 能力差を埋めるツールを活用する
  - 例: SDD 系ツール（Kiro, spec-kit 等）

学びにも追い風が吹いている

Takuto Wada
@t_wada

1. 何回同じことを聞いても怒られず即時フィードバックを得られる
2. フワッとした曖昧な質問から専門用語（≒検索キーワード）に辿り着ける

この2点は（特に初心者の）ソフトウェアエンジニアの教育あるいは独学においてChatGPT等の対話型生成AIが果たした画期的な進化だと考えています。

Translate post

8:41 AM · Jan 17, 2025 · **343.3K** Views

https://x.com/t_wada/status/1880037737392402496

変化を抱擁せよ

「賭ける」べきか否か

- 結局のところ「 code = ai(docs) 」はほとんどの場合成り立つのだろうか？
  - 既存コードを直すより、ドキュメントから生成し直す方がはやくかつ良くなるのだろうか？
  - 結局のところ抽象度が一段上がるのだろうか？ コードはアセンブラ相当になり、自然言語がコード相当になるのだろうか？
    - アセンブラは確かに一方通行だ
- ともすると「問題は将来のAIが解決してくれるはず」と考えがち
- コーディングは競争力を失うのだろうか？
  - 我々はコーディングをやめるべきなのだろうか？

「〜べきか否か」という問いの立て方は誤っている

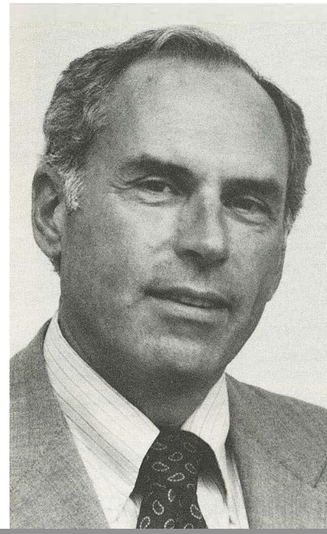
- 我々はいま生成 AI の光に目を焼かれて視野狭窄を起こし、バイアスの下にいる
- 『決定力! :正解を導く4つのプロセス』を読むとバイアスとの戦い方がわかる
  - 「〜べきか否か」という問いの立て方は失敗に導かれやすい
    - "whether or not" ではなく "compare and contrast"
    - XOR ではなく AND
- 決定を遅延すれば良い。選択肢を広げれば良い。
  賭けるのではなく「両にらみ」で行けば良い
- 不確実な状況下において選択肢を狭めるのは危険でしかない

https://www.amazon.co.jp/dp/B00HS9IP26

アマラの法則

We tend to overestimate the effect of a technology in the short run and underestimate the effect in the long run.

私たちは技術の短期的な効果を過大評価し、
長期的な効果を過小評価する傾向がある



https://en.wikipedia.org/wiki/Roy_Amara

「変化を抱擁せよ」

- 私たちは技術の短期的な効果を過大評価し、長期的な効果を過小評価する傾向がある
- 世界は思ったよりはゆっくりと、だが確実に大きく変わる
- 賭けなくていい。可能性を並べ、手を動かして評価し、変化を楽しもう

ご清聴ありがとうございました

- AIで時が加速したが、問題の構造、根本的原因は同じ。歴史から学んだ知見が今こそ重要
- プログラミングからソフトウェアエンジニアリングへ。Vibe Coding から Agentic Coding へ
- リソース効率重視の「委託」と、フロー効率重視の「伴走」。状況に応じて適切なモードを選択
- 望ましい状態を宣言的に定義し、評価関数を与え、AIが自律的に収束する仕組みが自働化には重要
- AIから引き出せる性能は、個人と組織の能力に比例する。能力向上への投資が不可欠
- 現実を直視する。最初から正しい設計はできない。わからないものはレビューもできない。この前提で設計プロセスを組み立てる
- 「賭ける」のではなく「両にらみ」。XORではなくAND。決定を遅延し、選択肢を広げ、可能性を並べて手を動かして評価する
- 我々は短期的効果を過大評価し、長期的効果を過小評価しがち。世界は思ったよりはゆっくり、だが確実に変わる