先達もすなる仕様駆動といふものを、オタクもしてみむとてするなり。

仕様駆動開発を実際にやってみて思ったこと

芳賀 雅樹 / @silasolla

May 20, 2026 — AI を紡ぐ者たち #1 (3-shake)

自己紹介

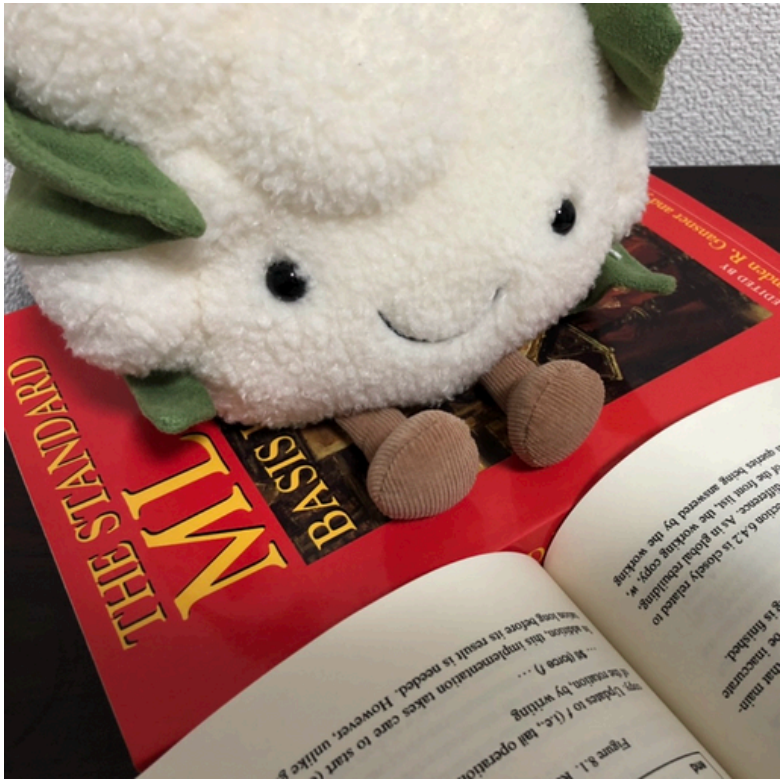
芳賀 雅樹 / @silasolla

開発プロセスのモダナイゼーションをやっています！

クラウドネイティブ技術 ／ 生成 AI ／ DevOps

お気に入りの言語は **Standard ML** です！

(Machine Learning ではなく Meta Language です……)

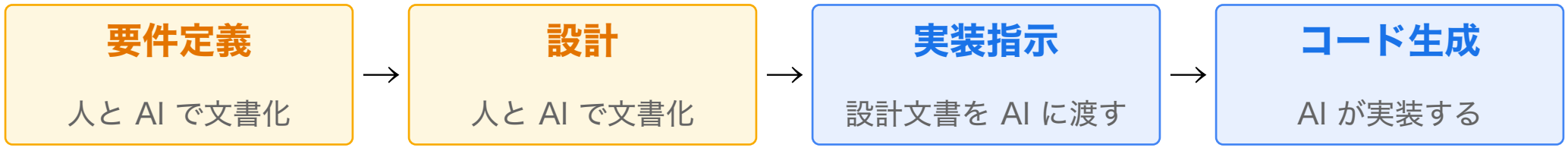


インターネット近影

やりたかったこと

エンジニアチームの開発フロー全体を **生成 AI で整備** したい

要件定義から実装までを一気通貫で AI に乗せられないか？



仕様を作るフェーズ

仕様をコードに変換するフェーズ

最初にやったこと

Gemini CLI + Conductor（Gemini CLI の公式拡張）

“Context-Driven Development” を掲げ Markdown で文脈を管理しながら開発を進める

3-phase の開発プロトコルを提供：

Context

`/conductor:setup`
製品方針／技術スタック／規約
を Markdown 化する

→

Spec & Plan

`/conductor:newTrack`
機能毎の仕様／実装の計画
を生成する

→

Implement

`/conductor:implement`
Plan を起点に仕様を参照して
AI が実装する

上流から下流まで単一ツールで（まだ Agent Skills / Plan Mode も GA していなかった）

ツールにコントロールを奪われた

ツールの定めるフローに縛られて **開発プロセスを柔軟に動かせない**

- 「ここは手動」や「別ツールに切り替え」など挟みにくい
  - Git の履歴とか勝手に触らせたくない
- 文書は Conductor が決めた **特定のファイル名／場所** に置く必要があった

事前に整えていた資料

```
README.md
docs/architecture.md
docs/glossary.md
ADR ／ 設計メモ など
```

Conductor が要求する場所

```
conductor/product.md
conductor/tech-stack.md
conductor/workflow.md
conductor/tracks/.../spec.md など
```

→ プロダクト方針／技術スタックなど **同じ情報を 2 箇所に書く** 状態に
片方を更新すると片方が古くなる ／ AI が何を読んでいるか明らかでなくなる

資料を読ませすぎて混乱した

読ませたい文書がどんどん増える

さらに Conductor は `conductor/` 配下を **まとめて投げ込む** 設計

Context Rot

コンテキストが大きくなると
モデルの推論が崩れる

Lost in the Middle

長いプロンプトで最初と最後ばかり拾われ
中央付近の情報が抜け落ちる

SDD で指定したフェーズの順序やアーキテクチャ制約など
詰め込みすぎると逆に守ってくれなくなる

資料アクセスの階層化

いつ何を読ませるかを指定

— ちょうど “harness” という言葉が広く使われ始めたころ……

Tier 1 **プロジェクト全体の前提（常時効かせる）**
アーキテクチャや用語など横断的な原則 ／ コーディングで AI は書き換えない

Tier 2 **機能ごとの「仕様」（タスク分解などで効かせる）**
振る舞いや受入の条件など ／ コーディングで AI は書き換えない

Tier 3 **作業限りで使い切る「仕様」（作業時に効かせる）**
実装手順やタスク分解 ／ 人と AI で都度合意して作る

あくまでもポリシー宣言 ／ 強制はレビューや CI で掛ける

Skills を役割で分割する

ベースの指示（GEMINI.md など）は **最小限**
役割ごとの Skill を **オンデマンド** で読み込む
— 月並ですが，折角 Skills 使えるようになったので……

SDD / TDD のワークフロー
テスト → 実装 → リファクタの順序を保つ

レビュー観点の標準化
見るべき観点やアンチパターン

テスト戦略 ／ モック境界
いつモックし，いつしないか

PR 下書き ／ 簡単な Git 操作など雑務
常時 ON にせず必要時に呼ぶ

—— だいぶ良くなった感覚はあるが，正解とはいえないので試行錯誤（客観的な計測ではない）

そもそも「仕様」って何だよ

「仕様駆動」なる字面が先行して

「仕様」の意味を共有できていないがち：

何を「仕様」とよぶ？

要件？ 設計？ テスト？
振る舞いの定義？

使い捨て？ 保持？

実装して終わりの指示書？
継続して育てる文書？

凍結？ 生きている？

READ ONLY にする？
更新が前提？

cf. 鍋島 理人「仕様駆動開発への期待と誤解 〜『仕様』とは、結局何なのか〜」CodeZine (2026/04/23)
codezine.jp/article/detail/23908

ちなみに「厳密な仕様」と言うと

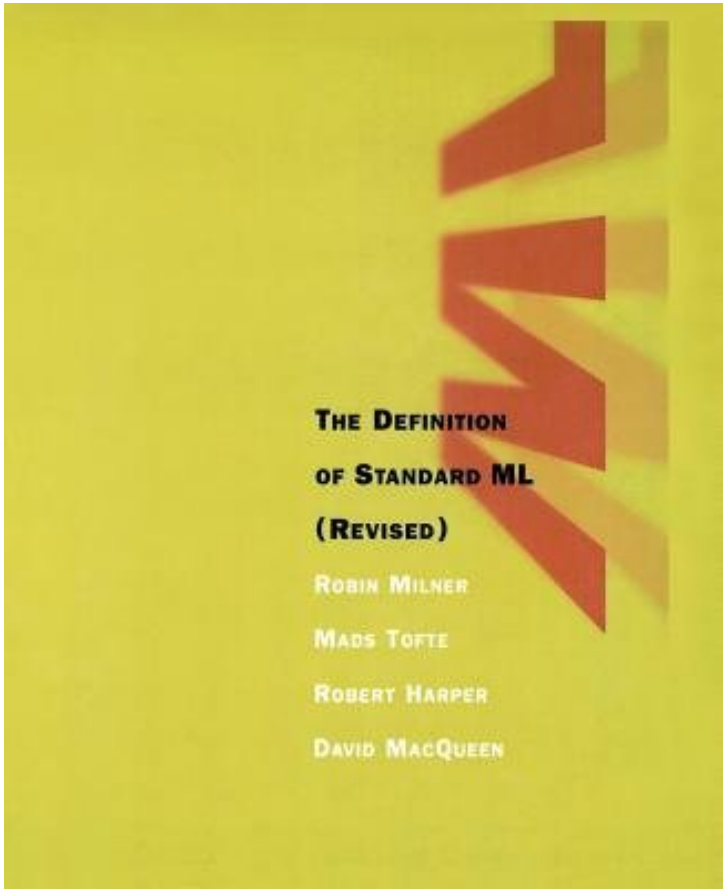
操作的意味論（これは自然意味論）の
推論規則として 仕様を定義する界隈もある

—— 関数型まつり 2026 で話します……！

こういう定義式が並んでいる：

$$\frac{C \vdash \mathrm{exp} \Rightarrow \tau' \to \tau \quad C \vdash \mathrm{atexp} \Rightarrow \tau'}{C \vdash \mathrm{exp}\,\mathrm{atexp} \Rightarrow \tau}$$

—— 極北（普通の SDD はもっと気楽……）



The Definition of Standard ML (Revised), 1997

「仕様駆動」という言葉が指すもの

Spec-first

仕様を揃えて実装

その後は **手放す**

Spec-anchored

仕様を起点に実装

完了後も **保持／更新**

Spec-as-source

仕様が **正**

コードはそれに従属

Kiro / SpecKit は **Spec-first 相当** だが……
→ **Spec-anchored / Spec-as-source** を思い描く人もいて話が噛み合わなくなる

cf. Birgitta Böckeler “Understanding Spec-Driven-Development: Kiro, spec-kit, and Tessl”
martinfowler.com/articles/exploring-gen-ai/sdd-3-tools.html

ツールの比較（参考）

	プロジェクトレベル 永続資料寄り	**実装する機能ごと** 使い捨て寄り
Kiro	`.kiro/steering/` product / tech / structure	`.kiro/specs/<feature>/` requirements / design / tasks
SpecKit	`.specify/memory/constitution.md`	`specs/<feature>/` spec / plan / tasks
Conductor	`conductor/product.md` など product / tech-stack / workflow	`conductor/tracks/<track>/` spec / plan
Claude Code	`CLAUDE.md` `.claude/skills/<skill>/`	Plan Mode 適当な Markdown に書き出して残せる

→ プロジェクトレベルに **Tier 1** ／ 機能ごとの層を **Tier 2** と **Tier 3** 分割してやってみている

ドキュメントから仕様をオフロード

仕様（プロンプト）を厚くすると
Context Rot / **Lost in the Middle** のリスクが増す

→ **決定論的なゲート** を併用して負荷を下げる

TDD

テストで振る舞いを固定

Linter / Formatter

構文レベルの品質を担保

型検査 / 静的解析

意味レベルのミスを検出

あるいは —— 仕様を **手放してコードを正にする**

Code Wiki でドキュメントを生成したり，OpenAPI を実装から導出したり

人はすぐタイパを求める

生成 AI 時代における工数見積もりわからん（特に他の人の工数）
生成 AI で早くできたらラッキーだね、で終わるだろうから多めに見積もりたい
世界が慣れて甘い見積もりじゃ納得してもらえない世界に

人はすぐタイパを求めるので，せっかく AI で視野が拡張されても「はやい！やすい！もろい！」になる.

そ、そんな〜

Nano Banana 2 の考える「タイパ食堂」

設計や運用を丁寧にやりたい

コーディング自体は AI に任せて素早くなったが……

→ タイパばかりでなく **品質担保に重心を寄せてもよい** のでは

⚠ アドホックなコードが増え
レビューしたくなくなる → 形骸化

⚠ 「取得 → 更新」と雑に書いて
ロック必要な並行性の文脈を無視

✓ **見通しの良い設計** を先に用意
→ DRY 原則（用法用量を守って）

✓ **レビュー観点** を整理（AI 指示でも）
→ 標準化された知識として資産に

AI が実装するなら責任は誰が取るのか？

国民は

- 99 % の働かない **〈消費者〉**
- 働く 1 % のエリート **〈生産者〉**

に分類された世界

「責任を取って辞職する」専門の職業まで現れる

『未来職安』柞刈湯葉

まとめ

AI の馬力だけに頼らず **ソフトウェア設計に重心** を寄せたい

結局，責任を負うのは人間

テクノロジーの発展とは切り分けて考えたい