「行数が増えるとAI開発は死ぬ」への反論

設計でどう潰しているか

ai-desk 方法論 — 沖井広行 · 2026-05

1 よくある批判

> **AI に適当に書かせる開発は、初期は速い。**
> **だが規模が大きくなると コンテキストを失って AI が壊し始める。**
> **5万行を超えたあたりで詰む。**

この批判は **普通の書き方をしていれば正しい。**
では「普通の書き方」をしないと何が起きるか。

2 普通のコードが規模で死ぬ仕組み

規模が増えると

- ✕ 関数が他の関数を呼ぶ → 連鎖的に context が必要
- ✕ ファイル間で依存 → 編集前に複数ファイル読む必要
- ✕ 共有ヘルパー (DRY) → 1 箇所の変更が広域に波及
- ✕ 抽象化・カプセル化 → 隠蔽された情報が AI から見えない
- ✕ 暗黙の前提 → 知らないと壊す

結果

- ✕ AI のスポットライト (~±300 行) を超える
- ✕ コンテキスト窓を食い尽くす
- ✕ 影響範囲が読み切れず壊す
- ✕ テスト落ちる、修正が連鎖、雪だるま式に詰む
- ✕ 「AI には限界がある」と結論付けられる

3 原因の原因

AI が壊れるのは **AI の限界ではなく、設計が AI 向けじゃないから。**

普通の設計原則 (DRY / 抽象化 / カプセル化) は、人間がコードを **読みやすくする** ために作られた。AI はそれと逆の最適化を必要とする:

- **ローカリティの極大化** — 1 箇所だけ読めば全部わかる
- **共有禁止** — 編集の影響範囲が固定
- **明示性** — 隠蔽せず展開して書く
- **1 ファイル完結** — ファイル間ジャンプを避ける

これが Bible §0.0「認知非対称性」。同じコードが、人間には読みにくく、AI には読みやすい。
普通の書き方こそが AI を殺している。

4 設計戦略 1 — Heavy Function

1 つの関数で完結させる。AI が「その関数だけ読めば全部わかる」状態を作る。

関数を切り刻まない

機能ごとに大きな関数を作る。データ加工・条件分岐・副作用トリガーを 1 つのスコープに inline する。

長さを恐れない

200-500 行の関数は問題ない。AI のスポットライトに収まれば良い。短く分割するほど他関数への依存ジャンプが増える。

普通の感覚と真逆。だが **1 関数完結 = 編集の影響範囲が関数内に閉じる**。スケールしても他関数を壊さない。

5 設計戦略２— 共有ヘルパー禁止

複数の関数から呼ばれる「共通ヘルパー」は作らない。重複を恐れない。

DRY (普通の常識)

- ✕ 同じコードは関数化して再利用
- ✕ 「美しい設計」と評価される
- ✕ しかし: 1 箇所変更で全呼び出し元に影響
- ✕ blast radius = O(callers)
- ✕ 規模が大きいほど破壊力増

AI-DESK: 共有禁止

- ✓ 似たコードは **各関数内に複製**
- ✓ 「重複は悪ではない、隠れた依存が悪」
- ✓ 1 箇所変更しても他は無傷
- ✓ blast radius = O(1)
- ✓ 規模が大きくなっても被害が広がらない

6 設計戦略 3 — Emblem 境界 + 局所読み込み

巨大ファイルを「物理分割せず仮想分割」して AI に局所読み込みさせる。

- `// [ai_s_emblem:#layer Name]` でセクション宣言
- `node ai-desk.js file.js skeleton` → 全 emblem の目次表示 (数十行)
- `node ai-desk.js file.js focus EmblemName` → 該当 emblem だけ抽出
- `node ai-desk.js file.js apply patch.js` → 原子的適用 (pre-flight 検証 + tag immutability)

> ファイルが **10 万行** でも、AI が読むのは該当 emblem の **数百行** だけ。
> コンテキスト消費は **ファイルサイズに依存しない**。

7 設計戦略 4 — 4 層 + REAL/SHADOW

データの流れと状態の責任を構造的に分離する。

4 層一方向フロー

L1 物理 → L2 意図 → L3 ロジック → L4 描画。データの流れが固定 → どこを編集すれば何に影響するかが構造的に決まる。

REAL は 1 つだけ書き換え可

L3 だけが `REAL_state` を更新できる。他層は read only。状態漏れバグが **原理的に発生しない**。

SHADOW は使い捨て

派生値は変数に保存しない。生成して即捨てる。「同期漏れ」バグが消える。

Bridge で層境界を明示

層を跨ぐ呼び出しは `// [ai_s_bridge:L3toL4]` でタグ付け。AI が「ここは認知ジャンプ点」と分かる。

8 設計戦略 5 — Constraint Folding

if/else ネストを「全可能世界 → filter」に置き換える。

- **普通の書き方:** 条件分岐ツリーを書く → 規模で組合せ爆発、テスト漏れ、AI が分岐迷子
- **ai-desk:** 全可能世界を **データ構造として列挙** → 制約で filter → 残った世界が答え
- 分岐ツリーが消える = AI の認知分裂が消える
- 逆引きが可能 (出力 → 入力)、テストも全網羅できる

最小実証: `constraint-janken.js` — 3 人ジャンケン 27 世界、if 文ゼロ。
実用例: `fighter-cancel.test.js` — 1920 世界全網羅、1 ファイル。

9 設計戦略 6 — Twin + Event Sourcing

改修可能性と検証可能性を構造で保証する。

Twin (検証双子)

GPU 実装と並走する CPU 純粋関数で計算を検算。バグが出ても「描画のバグか論理のバグか」を断定できる。

Event Sourcing

状態を上書き保存せず、Command の履歴を JSON で追記。各イベントは前ハッシュを含む直列ハッシュ。改ざん検知が数学的に保証される。

→ 過去のどの時点でも再現可能、改修時に「なぜこの状態になったか」が完全に追える。
規模が大きくなっても **履歴を遡って原因特定できる。**

10 設計が保証する数値

±300 行
AI が一度に読む量
(emblem 単位、ファイルサイズ非依存)

O(1)
編集の blast radius
(共有ヘルパー禁止)

O(変更単位)
検証範囲
(Twin / 網羅 per goal)

これらは 規模 (ファイル数 / 行数) に依存しない。
10 万行になっても、各数値は変わらない。

普通の codebase だと、これらは全部規模に比例して増える。**ai-desk は設計でそれを切断している。**

設計で潰した

「AI 開発は規模で死ぬ」は、
普通の書き方をした場合の話。
設計を AI 向けに作り直せば、死ぬ理由がなくなる。

Heavy Function · 共有禁止 · Emblem 局所読込 · 4 層 + REAL/SHADOW ·
Constraint Folding · Twin + Event Sourcing
—— これらが揃うと、**規模はもう破綻原因ではない。**

github.com/AoyamaRito/ai-desk

沖井広行 · 蒼山りと