コードより先に、証拠を移行せよ

AIコーディングエージェント時代の言語選定と検証戦略

鈴垣美影 編

AIがコードを書く時代に、人間は何を学び、どの言語に賭け、どう既存資産を移すべきか。構文暗記の終わりから、Python、Rust、仕様マイニング、移行エンジンまでをつなぎ、「コードより先に証拠を移行せよ」という行動指針へ導く一冊。


証拠
検証
移行
コード

目次

Chapter 01

二つの実験結果が矛盾する朝

今回の問い: AIでコードを書く力が上がったのに、なぜ開発全体は必ずしも速くならないのか。

この本は、AIが書いたコードをどう信じるか、という問いから始まる。書く速度と、正しいと確認する速度は同じではない。

生成速度と検証速度を分ける

GitHub Copilotの実験では、単発のJavaScript HTTPサーバ実装タスクでAI支援群が速く完了した。一方、METRの実験では、経験豊富なOSS開発者が自分のリポジトリで作業したとき、AI利用で時間が長くなったと報告された。

この二つは単純な矛盾ではない。前者は「何を作るか」が狭く、正解判定も比較的はっきりしている。後者は文脈が深く、レビュー、修正、既存設計との整合が重い。AIが速くするのは主に生成であり、検証まで自動的に安くするわけではない。

ここでいう検証とは、コンパイルが通るかだけではない。既存仕様を壊していないか、性能が落ちていないか、運用上の癖を踏んでいないかまで含む。AI時代の開発では、この検証側が新しいボトルネックになる。

ただし、これらの数字はAI一般の永遠の評価ではない。METR自身も後続更新で測定設計を見直している。本書では数字を予言ではなく、生成と検証を分けるための観察窓として扱う。

この本の中心仮説

本書では、コードが正しいと判断するための材料を「証拠」と呼ぶ。型、テスト、仕様、ベンチマーク、静的解析、運用ログ。これらはすべて、AIが出したコードを受け入れるか棄却するための判断材料になる。

AIが安くしたのは、コードを書くことだ。まだ安くしていないのは、コードが正しいと知ることだ。だから、これから価値が上がるのは「書ける人」よりも、「正しさの証拠を設計できる人」である。

補助メモ: ここで焦って「AIは速い/遅い」と決めない。生成と検証を分けるのが、この本の入口。

AIが速くする領域と、検証コストが支配する領域は別に見る。

理解チェック

Q1. Copilot RCTとMETRの結果を並べて読むとき、最も大事な違いはどれか。

- 使ったAIモデルの名前

＊ タスクの文脈深度と検証コスト

- 参加者の国籍

解説: 単発課題と実リポジトリ作業では、検証の重さが違う。

Q2. この本でいう「証拠」に最も近いものはどれか。

- コードが短いこと

＊ コードが正しいと判断する材料

- 人気のある言語名

解説: 型、テスト、仕様、ベンチなどが証拠になる。

3行まとめ

・AIは生成を速くするが、検証を自動で消すわけではない。

・開発全体の速度は、証拠の濃さに左右される。

・この本の主役はコードではなく証拠である。

次へ: 次は、言語の価値を4層に分けて、AIがどの層を安くしたのかを見る。

Chapter 02

言語価値スタック

> 今回の問い: プログラミング言語の価値は、構文の覚えやすさだけで決まるのか。

言語の価値は一枚岩ではない。AIで安くなった層と、逆に値上がりした層を分ける。

4層で見る

第一層は構文とAPIの再生だ。for文の書き方、標準ライブラリの関数名、定型的な使い方。この層はAIが最も得意で、価値が下がりやすい。

第二層はエコシステムだ。ライブラリ、サンプル、トラブルシュート記事、既存事例。AIはこの層を学習し、参照し、再利用する。厚いエコシステムはAIの出力品質も支える。

第三層は検証可能性だ。型、所有権、コンパイラ診断、静的解析。AIが間違った候補を出したとき、機械的に弾けるかがここで決まる。

第四層は証拠だ。特定のコードベースに蓄積されたテスト、仕様、ベンチ、運用知識。これは組織固有の資産で、AIに最も渡しにくく、だからこそ価値が高い。

PythonとRustを同じ物差しで比べない

Pythonは第二層、エコシステムで強い。Rustは第三層、検証可能性で強い。だから「Pythonは遅い」「Rustは難しい」という単純比較では、何を買っているのかが見えない。

AI時代の言語選定は、どの層に投資するかの問題になる。構文暗記が安くなるほど、上の層、特に検証可能性と証拠の価値が上がる。

> 補助メモ: 言語名で殴り合う前に、どの層の話をしているか分ける。ここを混ぜると議論がすぐ雑になる。

AIは構文/API再生層を安くしたが、検証可能性と証拠の価値は上げた。

理解チェック

Q1. AIで最も価値が下がりやすい層はどれか。

＊ 構文/API再生

- 組織固有の証拠

- 運用上の暗黙仕様

解説: 暗記と定型再生はAIが得意な領域。

Q2. Rustの強みを価値スタックで見ると、主にどこにあるか。

＊ 検証可能性

- REPLの実験速度

- 自然言語の読みやすさ

解説: 型と所有権が検証器として働く。

3行まとめ

・言語価値は4層に分けて見る。

・AIは下層を安くし、上層の価値を上げた。

・PythonとRustは別の層で勝っている。

次へ: では、値上がりした証拠とは何か。次章で具体的に定義する。

Chapter 03

証拠生成という職能

今回の問い: コードが安く書ける時代に、人間は何を作るべきか。

答えはコードだけではない。人間が作るべきものは、コードを信じるための証拠である。

証拠の種類

型は、コンパイル時に確認できる証拠だ。テストは、実行時に期待動作を確認する証拠だ。仕様は、何を満たすべきかを記述する証拠だ。ベンチマークは、性能が落ちていないことを示す証拠だ。

これらは人間の安心材料であると同時に、AIの制御材料でもある。AIがコードを出し、型やテストやベンチがそれを弾く。この構図では、証拠はエージェントを動かすためのレールになる。

証拠はコードより長生きする

実装をPythonからRustへ移しても、「この関数は負の残高を返してはいけない」という性質は残る。コードは言語に縛られるが、性質やベンチの意図は言語を越えて持ち運べる。

だから移行プロジェクトで最初に移すべきものはコードではない。まず移すべきなのは、何を守るべきかを示す証拠である。

証拠が薄いとAIは遅くなる

AIが出した候補を、人間が毎回頭の中で検証するなら、AIは単なる作業割り込みになる。逆に、証拠がCIや型やテストに入っていれば、人間に届く前に多くの候補を機械が棄却できる。

同じAIを使っても、証拠の厚いチームと薄いチームでは実効性能が変わる。AI導入の差は、モデルの差だけではなく、証拠基盤の差でもある。

補助メモ: テストを書くのは地味。でもAI時代には、地味な証拠がいちばん効く。ここ、ちゃんと価値が逆転してる。

証拠は検証時点と自動化度で整理できる。早く自動で落とせるほど安い。

理解チェック

Q1. 証拠が言語を越えて持ち運びやすい理由はどれか。

- 見た目が似ているから

＊ 守るべき性質は実装言語から独立しやすいから

- どの言語も同じ速度だから

解説: 性質や期待動作は、実装より寿命が長い。

Q2. AI導入前に整えるべきものとして最も近いのはどれか。

- ロゴ

＊ 証拠基盤

- キーボード配列

解説: AIを受け止める型・テスト・仕様が先に必要。

3行まとめ

・証拠はAI出力を受け入れるか決める材料である。

・コードより証拠の方が長く生きることがある。

・AIの実効性能は証拠の濃さに左右される。

次へ: 次は、この証拠を使ってAIを回すループ、Verifier-in-the-Loopを見る。

Chapter 04

Verifier-in-the-Loop

今回の問い: AIが出したコードを、どうやって人間の前でふるいにかけるのか。

AIを賢くするだけでは足りない。AIの隣に、強い検証器を置く必要がある。

生成器と検証器を分ける

LLMは候補を出す生成器である。検証器は、その候補が仕様を満たすかを判定する。候補が落ちたら、検証器は反例や診断を返し、生成器はそれをもとに修正する。

このループを本書では verifier-in-the-loop と呼ぶ。重要なのは、生成器の賢さだけではなく、検証器の強さ、速さ、診断の具体性がループ全体の性能を決めることだ。

人間は最後の検証器

構文チェック、型チェック、lint、ユニットテスト、プロパティベーステスト、ベンチマーク。これらを通した後に、人間レビューが来る。人間は最も高価で、最も文脈を読める検証器だ。

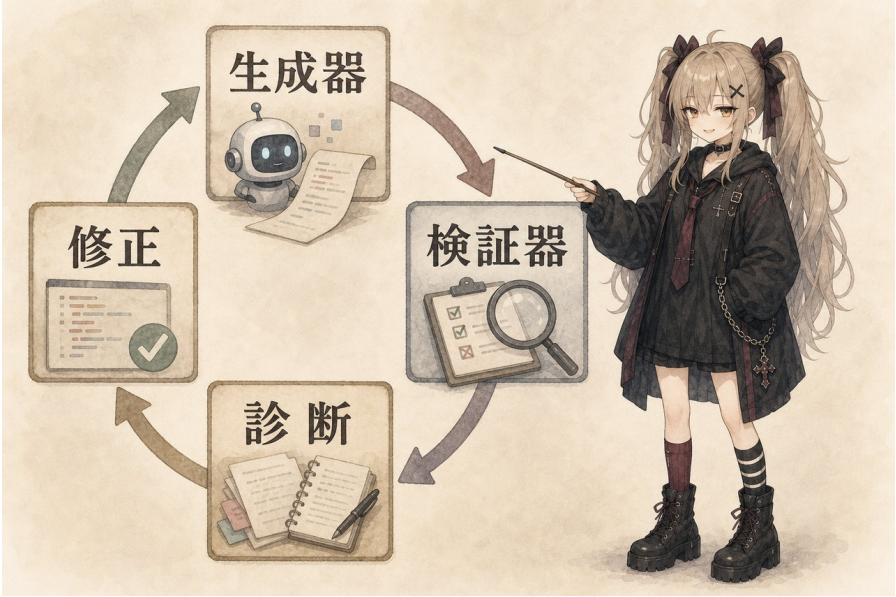
だから人間を最初の検証器にしてはいけない。人間の前に機械の漏斗を置く。これがAI時代のレビュー設計になる。

検証器にも穴がある

テストを通すことだけを目標にすると、AIはテストに過適合した退化解を出すかもしれない。ベンチだけを見れば、正しいが危険なコードを通すかもしれない。

対策は、証拠を一種類にしないことだ。型、テスト、プロパティ、ベンチ、運用観測を重ねる。単独の検証器に頼らず、複数の証拠で面を作る。

補助メモ: AIに丸投げするんじゃなくて、AIが外したときに機械が殴り返す構造を作る。ここが実務の強さ。



生成器Gと検証器Vのループ。診断が具体的なほど、次の候補は良くなる。

理解チェック

Q1. verifier-in-the-loopで検証器が返すべきものは何か。

- 合否だけで十分

＊ 合否と診断

- 雑談

解説: 診断があるとAIの次の修正が改善しやすい。

Q2. 人間レビューの位置づけとして近いものはどれか。

- 最初の安い検証器

＊ 最後の高価な検証器

- 不要な儀式

解説: 機械で落とせるものを落としてから人間が見る。

3行まとめ

・AIは生成器であり、証明器ではない。

・強い検証器がAIの実用性を決める。

・人間レビューは最後に置く高価な検証器である。

次へ: このループを現実の言語で見る。まず、なぜPythonがまだ強いのか。

Chapter 05

Pythonはなぜ死なないか

今回の問い: Pythonは遅くて動的なのに、なぜAI時代にも強いのか。

Pythonの強さは、言語仕様だけでは説明できない。資産と実験速度が、Pythonの重力を作っている。

エコシステム重力

Pythonには、PyPI、NumPy、pandas、PyTorch、Jupyterなどの巨大な資産がある。これは単なる便利ツールの集まりではない。他人が何度も使い、壊し、直してきた振る舞いの蓄積である。

AIはこの蓄積から学ぶ。既存のサンプルが多く、失敗例も多く、解説も多い言語では、AIも補完しやすい。つまり、エコシステムの厚さは人間だけでなくAIにも効く。

実験速度という証拠生成

Pythonは静的検証を犠牲にする代わりに、実行して確かめる速度を高めた。ノートブックで仮説を書き、実行し、出力を見る。機械学習のように正しさが実験で決まる領域では、この速度が非常に強い。

これは「Pythonは検証が弱い」という話だけではない。Pythonは、実行という形の証拠を素早く作る言語でもある。

mypyは建て替えではなく補強

型ヒントとmypyは、Python資産を捨てずに検証層を後付けする方法である。Dropboxの大規模なPython型付け事例は、既存資産に証拠を重ねる発想の実例として読める。

ここで大事なのは、全面移行ではないことだ。既存の動く資産を壊さず、型という機械可読な部分仕様を上に積む。これは、後で見る移行戦略の

原型でもある。

補助メモ: Pythonを「遅いからダメ」で片づけるのは浅い。資産と実験速度まで見ないと、なぜ勝つのか読めない。

Pythonの重力は、資産が新しい資産を呼ぶ自己強化ループでできている。

理解チェック

Q1. Pythonの強さとして本章が重視するものはどれか。

＊ エコシステムと実験速度

- 必ず最速であること

- 型が一切不要なこと

解説: Pythonの価値は資産と試行の速さにある。

Q2. mypy導入は何の例として読めるか。

＊ 証拠の後付け

- 全面書き換え

- 言語を捨てる戦略

解説: 既存コードに型という証拠を重ねる。

3行まとめ

・Pythonはエコシステム重力で強い。

・実行して確かめる速度も証拠生成の一種である。

・型ヒントは既存資産への証拠の後付けである。

次へ: 次は反対に、検証を最初から強く持つRustを見る。

Chapter 06

Rustという賭け

今回の問い: Rustの難しさは、AI時代には弱点なのか、それとも検証器としての強みなのか。

Rustのコンパイラは、ただコードを拒否するだけではない。反例と修正の方向を返す検証器として読める。

コンパイラを検証器として雇う

Rustの所有権と借用検査は、use-after-freeやデータ競合のような問題を実行前に弾く。これは、テストで見つけにくい種類のバグをコンパイル時に前倒しで検出する仕組みである。

AIが書いたRustコードがコンパイラに怒られる。これは失敗ではなく、検証器が反例を出したということだ。診断が具体的なら、AIはそのエラーを読んで修正できる。

借用検査器は診断オラクル

オラクルとは、テストで正解を判定する装置のことだ。Rustの借用検査器は、値がどこで移動し、どこで借用され、どこで使えなくなったかを示す。

人間には学習コストになるこの厳しさが、AIにとってはフィードバック信号になる。Rustの難しさは、AI時代には検証ループの強さとして再評価できる。

unsafeは穴ではなく台帳

Rustにもunsafeはあるだから完全ではない。しかしunsafeは、検証器が証明できない場所を明示する。つまり、人間が監査すべき場所を台帳化する。

安全性を完全に機械に任せるのではなく、機械が見られる場所と人間が見る場所を分ける。この境界が見えること自体が、証拠の品質を上げる。

補助メモ: Rustは面倒。でも、その面倒の一部は「あとで本番で泣く」より先に支払う税金でもある。

Rustコンパイラは、AIに反例と修正方向を返す診断オラクルとして働く。

理解チェック

Q1. RustのborrowcheckerをAI時代の文脈で見ると何に近いか。

＊ 診断オラクル

- 飾り

- 単なる速度最適化

解説: 合否だけでなく修正の手がかりを返す。

Q2. unsafeの扱いとして本章に近い見方はどれか。

- 隠しておけばよい

＊ 監査対象の台帳

- Rustが無意味である証拠

解説: 機械が保証できない場所を明示する。

3行まとめ

・Rustは検証をコンパイル時に前倒しする。

・コンパイラ診断はAIへの強いフィードバックになる。

・unsafeは監査対象を明示する台帳でもある。

次へ: 理屈だけでは足りない。次はAndroidの大規模データを見る。

Chapter 07

Androidの実証

今回の問い: 言語選択は、本当に脆弱性統計を動かすのか。

Androidの事例は、全面書き換えではなく新規コードの言語転換が効く、という重要なヒントをくれる。

メモリ安全性の数字

Googleは、Androidでメモリ安全性に関わる脆弱性の比率が、2019年の76%から2024年の24%へ下がったと報告している。これは、RustやKotlinなどのメモリ安全な言語を新規コードに使う戦略と結びつけて説明されている。

この数字は、単にRustが好きという話ではない。言語選定が、セキュリティ上の失敗の流入経路に影響しうるという実運用の証拠である。

古いコードを全部書き換えない

Androidの戦略で面白いのは、既存C/C++を全面的に書き換えるのではなく、新しく書くコードから変えた点である。古いコードには長年の運用によるエビデンスが溜まっている。全面書き換えは、そのエビデンスを捨てる行為にもなる。

危険なのは、まだ運用で叩かれていない新しいコードだ。そこにメモリ安全な言語を使うと、脆弱性の流入を抑えられる。これは漸進的移行の強い根拠になる。

一般化の限界

ただしAndroidは巨大なセキュリティ投資を持つ特殊な組織であり、同じ数字がすべての現場に当てはまるわけではない。Webアプリや業務システムでは、ボトルネックがメモリ安全ではないことも多い。

それでも、「新規コードから変える」「古いコードの運用実績を尊重する」という発想は、ほかの移行にも応用できる。

補助メモ: 全面書き換えしたくなる気持ちはわかる。でも、古いコードには古いなりの証拠が溜まってる。そこを捨てるのは高い。

古いコードの運用実績を尊重し、新規コードから安全な言語へ切り替える。

理解チェック

Q1. Android事例から得られる移行戦略のヒントはどれか。

- まず全部書き換える

＊ 新規コードから安全な言語へ寄せる

- 言語選定は無意味

解説: 新規コードの流入経路を変える発想が中心。

Q2. 古いコードを書き換えるリスクとして本章が重視するものはどれか。

＊ 運用実績という証拠を捨てること

- ファイル名が長いこと

- コメントが多いこと

解説: 動いてきた実績も一種の証拠。

3行まとめ

・Androidではメモリ安全性の数字が大きく改善したと報告されている。

・全面書き換えより、新規コードから変える戦略が効いた。

・古いコードの運用実績も証拠として扱う。

次へ: では、既存言語ではなくAI専用の新言語を作る案はどうか。次章で誘惑と罠を見る。

Chapter 08

「AI専用言語」の誘惑と罠

今回の問い: AIが書くなら、人間向けではない新言語を作った方がいいのか。

発想は魅力的だ。だが、新言語にはコーパスと検証器のコールドスタートがある。

AI専用言語の魅力

AIが主な書き手になるなら、構文は人間の覚えやすさではなく、機械の生成しやすさや検証しやすさに合わせればよい。正準形、効果システム、契約、性能予算を言語の中心に置けば、AIが間違いにくい言語を作れるかもしれない。

この考えは軽く扱うべきではない。AI時代の言語設計として、意味論の濃度を上げる方向は本物の論点である。

でも、学習データはどこにあるのか

新言語には学習コーパスがない。GitHub上の実例も少なく、失敗例も少なく、Stack Overflowの回答も少ない。LLMは過去のコードと解説から学ぶため、新言語はAIにとっても苦手な言語として始まる。

さらに、新言語には成熟したコンパイラや標準ライブラリ、デバッガ、プロファイラ、検証ツールもない。検証器を信じるための信頼計算基盤 TCB も、ゼロから育てる必要がある。

勝ち筋は構文ではなく意味論

だから最初に作るべきものは、まったく新しい構文ではない。既存言語の上に、正準フォーマット、契約、効果の注釈、性能ベンチゲートを重ねる方が現実的だ。

AI専用言語の本体は、実は新しい見た目ではなく、機械が検証できる意味論である。そこは既存言語の上にも積める。

補助メモ: 新言語、ロマンはある。でもロマンだけでTCBは育たない。まず既存の厚みを使う方が強い。

AI時代に投資すべきは構文刷新ではなく、検証可能性を上げる意味論の層。

理解チェック

Q1. AI専用新言語の最大の初期リスクはどれか。

＊ コーパスと検証器の不足

- 名前が短いこと

- 色が地味なこと

解説: AIも学習例と検証環境に依存する。

Q2. 本章が優先する投資先はどれか。

- 新しい構文だけ

＊ 意味論と検証可能性

- ロゴデザイン

解説: 正準形、契約、効果、性能予算が中心。

3行まとめ

・AI専用言語の発想は魅力的だが、コールドスタートが重い。

・新言語はTCBも育て直す必要がある。

・まず既存言語上に検証可能性を増築する。

次へ: では、その検証可能性を増築する材料、つまり仕様はどこから得るのか。次は仕様マイニングへ進む。

Chapter 09

仕様マイニング

今回の問い: 仕様書が古い、または存在しないとき、何を正しさの基準にするのか。

答えは、動いているシステムの振る舞いである。仕様は書類だけでなく、本番の入出力やログの中にもある。

仕様は地層である

仕様には三つの層がある。入力に対する出力を決める機能仕様。常に成り立つべき不変条件を表す性質仕様。レイテンシ、エラー率、バッチ完了時刻のような運用仕様。

移行で壊れやすいのは、しばしば運用仕様だ。機能的には同じでも、p99レイテンシが悪化すれば本番では失敗である。

旧実装をオラクルにする

絶対の正解がわからなくても、新旧実装の差分は検出できる。差分テストは、同じ入力を旧実装と新実装に投げ、出力の違いを調べる。

旧実装が完全に正しいとは限らない。それでも、既存ユーザーが依存している振る舞いを守るという意味では、旧実装は重要なオラクルになる。

本番入出力を扱うときは、個人情報、規制、季節性の偏り、書き込み副作用を無視してはいけない。仕様マイニングはログを乱暴に集める話ではなく、観測できる振る舞いを安全に証拠化する設計である。

PBT、シャドートラフィック、翻訳検証

プロパティベーステストは、少数の例ではなく法則を書く。シャドートラフィックは、本番リクエストを新実装にも流し、応答差分だけを見る。翻訳検証は、変換前後の意味が保たれているかを可能な範囲で機械確認

する。

どれか一つで完全になるわけではない。強いが狭い証拠、広いが粗い証拠、本番分布に強い証拠を積み重ねる。

> 補助メモ: 仕様書がないなら終わり、じゃない。動いているものから掘る。ここがこの本でいちばん実務っぽいところ。

仕様は機能・性質・運用の三層で掘る。層ごとに使う道具が違う。

理解チェック

Q1. 仕様マイニングで掘る対象として最も近いものはどれか。

＊ 本番の振る舞い

- ファイル名

- 作者の好み

解説: 入出力、ログ、レイテンシなどが素材になる。

Q2. 差分テストの実務的な強みはどれか。

＊ 絶対正解がなくても不一致を検出できる

- 必ず形式証明できる

- テストが不要になる

解説: 旧実装を一時的なオラクルとして使える。

3行まとめ

・仕様は本番の振る舞いから採掘できる。

・差分テストは旧実装をオラクルとして使う。

・複数の証拠生成器を積み重ねる。

次へ: 証拠が掘れたら、次はそれを使って実際に移行する。移行を一回限りの作業ではなくエンジンにする。

Chapter 10

漸進的移行の設計

今回の問い: AIで大量移行できる時代でも、なぜ一気に全部書き換えない方がよいのか。

問題はコードを書く速度ではない。移行中に検証フィードバックを得られる構造を作れるかである。

FFI境界は無料ではない

PythonからRustを呼ぶ、C++からRustを呼ぶ、Goから別サービスへ切り出す。こうした境界にはコストがある。データ変換、エラー表現、文字列、null/None/Option、認知負荷。境界は少なく、粗く、計測可能にする必要がある。

細かい関数をたくさん移すと、言語境界の通過コストが本体より重くなることがある。移行単位は、コールグラフやデータ境界を見て切る。

移行エンジンの五段階

第一に仕様採掘。第二にAIによる変換。第三に型、テスト、ベンチ、差分テストによる検証。第四にシャドー配備。第五に切替と旧コードの退役。

この流れは一回だけのプロジェクトではなく、繰り返し使うパイプラインにする。失敗したら反証例を前段に戻す。そうすると、移行エンジンは経験を蓄積する。

人間が決めるレーン

AIに任せてよい領域と、人間が決める領域を分ける必要がある。未定義動作への依存、並行性の意味論差、浮動小数点の再現性、外部観測可能なタイミングは、保存すべき意味が一意でないことがある。

ここはAIの能力不足ではなく、そもそも決めるべき仕様が曖昧な領域だ。だから人間レーンに送る。AIに流す前に、分水嶺を設計する。

補助メモ: 漸進移行は弱気な妥協じゃない。検証ループを移行にも適用するための設計だよ。

移行エンジンは、仕様採掘から退役までを反証例でつなぐパイプラインである。

理解チェック

Q1. 移行エンジンで失敗結果はどう使うべきか。

- 捨てる

＊ 反証例として前段に戻す

- 隠す

解説: 差し戻しが次の変換の改善材料になる。

Q2. AIに任せにくい領域として近いものはどれか。

＊ 保存すべき意味が一意でない領域

- 単純なデータ整形

- 型が明確な純粋関数

解説: 人間が仕様を決める必要がある。

3行まとめ

・移行の難所は共存期間と境界にある。

・移行は再実行可能なエンジンとして設計する。

・AIレーンと人間レーンを分ける。

次へ: 最後に、ここまでの議論を一つの命題にまとめる。コードはキャッシュだ。証拠が資本だ。

Chapter 11

コードより先に証拠を移行せよ

今回の問い: AI時代のプログラミング言語戦略を、一文にまとめるなら何か。

この本の結論は、どの言語が勝つかではない。何を言語の壁を越えて持ち運ぶべきかである。

コードはキャッシュだ

コードは、ある時点の実行環境に合わせて証拠を具現化したものだ。実装言語が変わればコードは変わる。しかし「何を守るべきか」を示す証拠は残せる。

レガシー問題の正体は、コードが古いことではない。コードしか残っていないことだ。仕様、性質、ベンチ、差分履歴が残っていれば、コードは再生成できる。

未来の言語は発見される

未来の言語は、机上の設計図だけから発明されるのではない。証拠生成コストが低く、AI生成と検証ループが濃く回る場所へ、コードと知識と人が集まる。

その集中がコーパスを厚くし、AI性能を上げ、さらに集中を呼ぶ。言語の勝者は、構文の美しさだけではなく、証拠の重力によって発見される。

月曜日にやること

自分のコードベースで、まず証拠を棚卸しする。型はどこまであるか。テストは何を守っているか。ベンチはあるか。本番のログから仕様を掘れるか。AIを導入する前に、AIを受け止める証拠を作る。

次に、移行対象を探す。性能や安全性のコストが高く、境界が切りやすく、検証しやすい場所から始める。コードを移す前に、証拠を移す。これが本書の行動指針である。

補助メモ: 最後はここ。言語を当てに行くんじゃなくて、どの言語でも持ち運べる証拠を積む。これが強い。

コードは再生成可能なキャッシュであり、証拠ベースこそが長く残る資本である。

理解チェック

Q1. 本書の最終結論に最も近いものはどれか。

＊ コードより先に証拠を移行せよ

- 人気言語だけ追えばよい

- AIを使えば検証は不要

解説: 証拠を先に持ち運ぶことが中心。

Q2. 未来の言語が「発見される」とはどういう意味か。

＊ 証拠生成が濃く回る場所へ自然に集中する

- 誰も設計しない

- 偶然名前が決まる

解説: 検証・コーパス・移行の重力が言語の形を決める。

3行まとめ

・コードはキャッシュ、証拠が資本である。

・未来の言語は証拠の重力から発見される。

・月曜日にやることは、証拠の棚卸しと小さな移行実験である。

次へ: ここから先は、あなたのコードベースで証拠を掘る番だ。

出典メモ

Peng et al., The Impact of AI on Developer Productivity - https://arxiv.org/abs/2302.06590 / Copilot RCTで単発課題の完了時間が短くなったという序章の対比に使用。

METR, Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity - https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/ / 経験豊富なOSS開発者の実作業でAI利用が遅くなったという対比に使用。

METR, We are Changing our Developer Productivity Experiment Design - https://metr.org/blog/2026-02-24-uplift-update/ / 2025年実験結果の扱いと測定設計の更新に関する温度管理に使用。

Google Security Blog, Eliminating Memory Safety Vulnerabilities at the Source - https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html / Androidにおけるメモリ安全性脆弱性比率の推移と新規コード戦略に使用。

Android Open Source Project, Memory safety - https://source.android.com/docs/security/test/memory-safety / Androidにおけるメモリ安全性問題の背景説明に使用。

Dropbox Tech Blog, Our journey to type checking 4 million lines of Python - https://dropbox.tech/application/our-journey-to-type-checking-4-million-lines-of-python / Python資産に型という証拠を後付けする事例として使用。

PEP 484, Type Hints - https://peps.python.org/pep-0484/ / Pythonの型ヒントと漸進的型付けの背景に使用。

RustBelt project - https://plv.mpi-sws.org/rustbelt/popl18/ / Rustの安全性保証を語る際の範囲と注意点の参照先。

fish shell, Fish 4.0: The Fish Of Theseus - https://fishshell.com/blog/rustport/ / C++からRustへの段階的移植事例として参考。

Discord Blog, Why Discord is switching from Go to Rust - https://discord.com/blog/why-discord-is-switching-from-go-to-rust / 性能上の意味論的制約を計測してRustへ移した事例として参考。

GitHub Blog, Scientist: Measure Twice, Cut Once - https://github.blog/developer-skills/application-development/scientist/ / 本番挙動を比較しながら置換する考え方の参考。

X Engineering, Diffy: Testing services without writing tests - https://blog.x.com/engineering/en_us/a/2015/diffy-testing-services-without-writing-tests / 旧実装と新実装の差分比較による検証事例として参考。