

# 書籍メタデータ

表紙

# AIエージェント 開発は 仕様が9割

Vibe Codingで失敗しないための設計図





## 基本情報

項目	内容
タイトル	AIエージェント開発は仕様が9割
サブタイトル	Vibe Codingで失敗しないための設計図
著者	[著者名]
ステータス	draft
バージョン	0.1.0
作成日	2026-01-01
更新日	2026-01-01

## 想定読者

- AIコーディングツール（Claude Code, GitHub Copilot, Cursor等）を活用したいエンジニア
- AI活用で手戻りやブレに悩んでいる開発チーム
- ドキュメント駆動開発に興味があるテックリード・マネージャー

## 書籍のゴール

読者が「AI仕様駆動開発」の考え方を理解し、自分のプロジェクトで7文書構成を導入して、AIに安心して開発を任せられるようになること。

## 構成概要

部	タイトル	章数	狙い
第1部	なぜ「AI任せ」は失敗するのか	2章	問題提起と原因分析
第2部	結論：仕様が9割	3章	コアコンセプトの提示
第3部	実践ワークフロー	4章	具体的な導入・運用手順
第4部	現場FAQ	3章	懐疑派への回答
第5部	組織展開	2章	チーム・組織への適用

## 帯コピー案

- 「AIが書いたコード、誰がレビューする？」——その問い合わせに答える本。
- 仕様を先に書けば、AIは迷わない。手戻りゼロの開発フローがここに。
- 7つの文書で、コーディングを“人間の仕事”から解放する。

## 参考資料

- [AI仕様駆動開発 公式ドキュメント](#)
- [GitHub Spec-Kit](#)
- [Microsoft Spec-Driven Development](#)

# はじめに

## 「AIに任せる開発なんて無理」の正体

「AIにコードを書かせる？ 結局、手直しに時間がかかるって意味ないよ」

エンジニアの間でよく聞く声です。Claude Code、GitHub Copilot、Cursor——AIコーディングツールは確かに進化しています。しかし、多くの開発現場では、こんな悩みが絶えません。

- AIが生成したコードが、既存のアーキテクチャと全く合わない
- PRが巨大になりすぎて、レビューが不可能になる
- 「なぜこう実装したのか」が分からず、修正に余計な時間がかかる
- 何度も同じような間違いを繰り返す

これらの問題を経験すると、「やっぱりAIは信用できない」という結論に至りがちです。

しかし、**本当の問題はAIの能力ではありません。**

問題は、AIに渡している「入力」——つまりスコープの曖昧さ——にあります。

## 実際に効果はあるのか？——現場のデータ

「理屈はわかった。でも本当に効果があるのか？」

そう思う方のために、私が関わったプロジェクトでの計測結果を紹介します。

RAGチャットボットの機能実装において、Issue駆動でAIに開発を任せた結果です。

指標	結果
コード生産性	業界平均（50～200行/日）の <b>115～460倍</b>
PRマージ率	<b>**98%**</b> (321件中315件)
コードレビュー工数	<b>97%削減</b> (人間は全体の3.3%のみ担当)

\* これは開発工程全体ではなく、一部機能実装における統計データです。

ポイントは、**品質を落とさずに**この生産性を実現したことです。98%のPRがそのままマージされている事実が、それを証明しています。

詳細なレポートはこれら：[AI駆動開発レポート](#)

では、どうやってこの結果を出したのか？——それが本書で解説する「3つの戦略」です。

## 結論を先に言います：3つの戦略がすべて

私が実際にAI駆動開発を実践してきた経験から、効く戦略は3つだけです。

### 戦略1：Issueでスコープを絞る（コンテキストエンジニアリング）

これが最も重要です。

「ログイン機能を作って」と指示すると、AIは迷います。認証方式は？エラーハンドリングは？既存コードとの整合性は？——書かれていなことを推測し、ほぼ確実にズレた実装をします。

解決策はシンプルです。Issueを作り、スコープを絞る。

✖ 悪い例：「認証機能を実装して」

### 良い例：

Issue #42: JWTトークン検証ミドルウェアの実装

#### - 受け入れ基準：

- Authorization ヘッダーからBearerトークンを抽出
  - トークンの署名検証と有効期限チェック
  - 検証失敗時は401を返す
- 制約：
- 既存の authMiddleware.ts を拡張
  - エラーレスポンスは errors/auth.ts の形式に従う

Issueを書く行為は、コンテキストエンジニアリングそのものです。AIに「何をすべきか」「何をすべきでないか」を明示することで、推測の余地を排除します。

スコープが絞られていれば、AIは迷わない。迷わなければ、バグは生まれない。

## 戦略2：PRは70%の完成度でいい

完璧を求めてはいけません。

AIが書いたコードをPRに出す時点では、**70%の完成度**だと思ってください。これは妥協ではなく、**設計された戦略**です。

なぜか？

レビューを受けることで、スコープがさらに狭まります。

- 「この関数、責務が2つ混ざってるね」 → 分割
- 「エッジケースのテストが足りない」 → 追加
- 「命名がプロジェクトの慣習と違う」 → 修正

レビュー指摘に基づいてAIに再実装させると、指摘内容がそのままコンテキストになるため、より正確なコードが生成されます。

レビュー指摘：「validateUser関数、バリデーションと永続化が混ざってる」

↓

AIへの指示：「validateUser関数を、バリデーション専用のvalidateUserInput関数と、永続化専用のpersistUser関数に分割して」

↓

結果：明確な責務分離が実現

レビュー→再レビュー→完成 という反復が、スコープを収束させ、精度を上げていきます。

## 戦略3：指摘をナレッジ化する

同じ指摘を何度も受けいませんか？

- 「またエラーハンドリングが漏れてる」
- 「テストのモック設定が毎回違う」
- 「import文の順序がバラバラ」

これらは**AIに事前に教えておくべき知識**です。

指摘が多かった場合、そのナレッジをまとめて「次回どうやったら指摘されないか」を仕組み化します。

具体的な手段：

### 1. 静的解析の強化

- ESLint/Prettierルールの追加
- カスタムルールで頻出パターンを検出

### 2. pre-commit フックでAIレビュー

```
# .husky/pre-commit
claude-code review --staged --rules .review-rules.md
```

### 3. サブエージェントによる多視点レビュー

- セキュリティ視点のレビューエージェント
- パフォーマンス視点のレビューエージェント
- 既存コード整合性チェックのエージェント

指摘を受ける→ナレッジ化→自動チェックに組み込む→次回から指摘されない。

このサイクルを回すことで、AIの出力品質は**プロジェクト固有の文脈**を学習し、継続的に向上します。

## この3つを支える基盤：7文書構成

上記の3戦略を効果的に運用するには、**AIが参照できる仕様の基盤**が必要です。

それが、本書で紹介する**7文書構成**です。

文書	役割	戦略との関係
MASTER.md	プロジェクト索引	AIが迷子にならない地図
PROJECT.md	ビジョンと要件	Issueの背景を示す
ARCHITECTURE.md	システム設計	制約を明示する
DOMAIN.md	ビジネスロジック	ルールの唯一の置き場
PATTERNS.md	実装パターン	ナレッジの蓄積先
TESTING.md	テスト戦略	レビュー基準を定義
DEPLOYMENT.md	運用手順	リリース可能性を定義

Issueを書くとき、ARCHITECTURE.mdの制約を参照する。レビューで指摘されたパターンは、PATTERNS.mdに蓄積する。テストの書き方で揉めたら、TESTING.mdを更新する。

仕様が整っていれば、3つの戦略はスムーズに回ります。

## 本書の読み方

本書は5部構成になっています。

**第1部「なぜAI任せは失敗するのか」** では、AI活用が失敗する本当の原因を分析します。「AIが悪い」のではなく「スコープが曖昧」であることを理解していただきます。

**第2部「結論：仕様が9割」** では、7文書構成の全体像を提示します。各文書の役割と、3つの戦略との接続を解説します。

**第3部「実践ワークフロー」** では、Issueベースの開発フロー、70%完成度でのPR運用、ナレッジ蓄積の具体的な手順を示します。

**第4部「現場FAQ」** では、懐疑的なエンジニアからの質問に答えます。「結局人間が全部書くのでは？」、「品質は担保できるのか？」

**第5部「組織に展開する」** では、個人の取り組みをチーム標準に広げる方法を解説します。

## さあ、始めましょう

AIコーディングツールは、正しく使えば圧倒的な生産性をもたらします。

しかし、「正しく使う」とは何か？

スコープを絞り、70%で出し、指摘をナレッジ化する。

この3つの戦略と、それを支える7文書構成。

「AIに任せる開発なんて無理」を「AIに任せたら、本当に楽になった」に変える。

そのための旅を、一緒に始めましょう。

## 目次

### AIエージェント開発は仕様が9割

Vibe Codingで失敗しないための設計図

---

#### はじめに

- 「AIに任せる開発なんて無理」の正体
- 生成AIは"自律"ではなく"条件反射"
- この本で約束すること

→ [00\\_preface.md](#)

---

### 第1部 なぜ「AI任せ」は失敗するのか（そして、なぜ誤解なのか）

#### 第1章 AIに全部任せようとして事故る典型パターン

- "vibe coding"が破綻する瞬間
- PRが巨大化する／レビュー不能になる／仕様が消える
- 「AIが賢くない」のではなく「入力が仕様になっていない」

→ [part1\\_why-ai-fails/01\\_typical-failure-patterns.md](#)

#### 第2章 AIが苦手なのは"コーディング"ではなく"心を読むこと"

- LLMが強い領域：既知パターンの組み立て
- LLMが弱い領域：未記述要件の補完
- 解法：仮定を排除して、仕様を"書いてから"渡す

→ [part1\\_why-ai-fails/02\\_ai-weakness.md](#)

---

### 第2部 結論：AIに任せる開発は「仕様」が9割

#### 第3章 仕様を"生きた成果物"にする

- 仕様=合意の置き場所
- 変更は「差分」ではなく「影響度」で扱う
- "仕様→設計→テスト→運用"の連鎖を切らない

→ [part2\\_spec-is-90percent/03\\_living-spec.md](#)

#### 第4章 7文書が「AI任せ」を成立させる最小構成

- 7文書の全体像
- MASTER.md / PROJECT.md / ARCHITECTURE.md
- DOMAIN.md / PATTERNS.md / TESTING.md / DEPLOYMENT.md

→ [part2\\_spec-is-90percent/04\\_seven-documents.md](#)

## 第5章 「7文書」を回すための最低限のルール

- Frontmatterでメタデータを揃える
- 変更時は影響度評価→バージョン更新→Changelogまで一気通貫
- コミット前に検証チェック

→ [part2\\_spec-is-90percent/05\\_minimum-rules.md](#)

---

## 第3部 実践：AI仕様駆動開発のワークフロー

### 第6章 導入手順：既存プロジェクト／新規プロジェクト

- 新規：docs構造を作り、最初の7文書を生成
- 既存：既存設計を"吸い上げ"て、欠けた文書を補完
- 「最初から完璧」を捨てる

→ [part3\\_practice/06\\_introduction.md](#)

### 第7章 日々の開発フロー：AIに"タスク"を渡す前にやること

- 仕様の粒度：受け入れ基準が書けているか
- 設計の粒度：アーキテクチャ制約が明示されているか
- テストの粒度：テストが仕様を代替していないか

→ [part3\\_practice/07\\_daily-workflow.md](#)

### 第8章 文書追加の意思決定（Decision Matrix）

- 「PROJECT ? DOMAIN ? ARCHITECTURE ?」の判断
- 例：DB設計、認証、権限、監査ログ、SLO
- MASTER.mdの索引更新を必須タスクに

→ [part3\\_practice/08\\_decision-matrix.md](#)

### 第9章 変更に強い運用：影響度評価で手戻りを消す

- 変更の種類：文言修正／概念追加／概念再定義
- HIGH変更のチェックリスト
- 仕様変更→AI再実装を"安全に繰り返す"設計

→ [part3\\_practice/09\\_change-impact.md](#)

---

## 第4部 現場で揉めるポイントへの回答

### 第10章 「それ、結局エンジニアが全部書くのでは？」への答え

- 役割分担の再設計
- "書く"ではなく"編集する"に寄せる

→ [part4\\_faq/10\\_engineer-role.md](#)

### 第11章 品質・セキュリティ・責任の所在

- 「AIが書いたコード」の責任
- テスト戦略を先に固める意味
- 監査可能性

→ [part4\\_faq/11\\_quality-security.md](#)

### 第12章 ツール実装：Claude Code / GitHub Copilotで"仕様駆動"を自動化する

- Claude Code Skills / pr-review-toolkit

- GitHub Copilot Agents（6種のテンプレート）
- ツール間の機能比較

→ [part4\\_faq/12\\_tool-implementation.md](#)

---

## 第5部 組織に展開する

### 第13章 チーム標準化：レビューの中心を「コード」から「仕様」へ

- 仕様レビュー→タスク→実装レビューの順番
- 「MASTERが更新されていないPRは受け付けない」ルール

→ [part5\\_organization/13\\_team-standardization.md](#)

### 第14章 ロードマップとナレッジ蓄積

- 成長フェーズで増やす文書
- "知見"をAIに食わせられる形で残す

→ [part5\\_organization/14\\_roadmap-knowledge.md](#)

---

## おわりに

- AIに任せるために必要なのは「仕様というOS」
- 次にやること

→ [99\\_afterword.md](#)

---

## 付録

### 付録：AIエージェント設定ファイル一覧

- AGENTS.mdとは（オープンスタンダード）
- ツール別設定ファイル（Claude Code, GitHub Copilot, Cursor等）
- 7文書との連携テンプレート

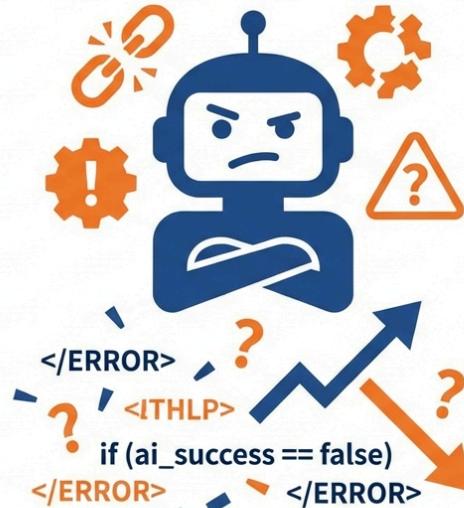
→ [appendix\\_agent-config.md](#)

## 第1部 なぜ「AI任せ」は失敗するのか

## Why AI Fails

# 失敗パターン

## 第1部：なぜAI任せは失敗するのか



「AIに任せれば楽になる」——その期待が裏切られる原因を解説します。vibe codingの限界、PRの肥大化、そしてAIが本当に苦手なこと。失敗パターンを知ることが、成功への第一歩です。

## 第1章 AIに全部任せようとして事故る典型パターン

### 1.1 この章で学ぶこと

- "vibe coding"が破綻する瞬間を理解する
- PRが巨大化・レビュー不能になるメカニズム
- 「AIが賢くない」のではなく「入力が仕様になっていない」という本質

### 1.2 "vibe coding"が破綻する瞬間

#### 1.2.1 「なんとなく」で始まる開発

AIコーディングツールを使い始めたエンジニアが最初にやりがちのが、 "vibe coding" です。

「なんかいい感じにユーザー認証作って」

「このAPIをRESTfulにして」

「エラーハンドリングをいい感じに」

「いい感じ」「なんとなく」——この曖昧な指示でも、AIは何かを生成します。そして、その生成されたコードは、一見すると動いているように見えます。

ここが罠です。

#### 1.2.2 暗黙知の推測ゲーム

vibe codingが破綻するのは、AIが暗黙知を推測し始める瞬間 です。

あなたのプロジェクトには、明文化されていないルールがたくさんあります。

- ・ 「認証はJWTを使う」（どこにも書いていない）
- ・ 「エラーレスポンスはこの形式」（過去のコードを見ればわかる）
- ・ 「この関数は非同期で呼ばれる前提」（設計時の暗黙の了解）
- ・ 「テストはこのパターンで書く」（チームの慣習）

AIはこれらを知りません。

知らないから、推測します。推測は、プロジェクトの文脈と乖離します。乖離したコードは、レビューで弾かれるか、より悪いケースでは本番で事故を起こします。

### 1.2.3 破綻のシグナル

vibe codingが限界に達したときに現れるシグナル：

#### 1. 「なんでこうなった？」が増える

- コードの意図が読めない
- AIの判断根拠が不明
- 修正しようにも、どこを直せばいいかわからない

#### 2. レビューコメントが膨大になる

- 「ここ、うちのプロジェクトではこう書く」の連発
- 同じ指摘が何度も繰り返される

#### 3. 「もう一度書いて」が頻発する

- 微調整では済まず、全面書き直しになる
- 結局、手で書いた方が早かった

#### 1.2.3.1 実際に起きた失敗事例

あるチームで、「決済画面のUIを改善して」という指示でAIにコードを生成させました。AIは見た目の良いUIを生成しましたが、3つの問題が発生しました。

1つ目は、既存のデザインシステムを無視したことです。プロジェクトにはコンポーネントライブラリがありましたが、AIはそれを知らず、独自のスタイルを生成しました。結果、他の画面と統一感のないUIになりました。

2つ目は、アクセシビリティ要件を満たしていないかったです。このプロジェクトではWCAG 2.1 AAに準拠する必要がありましたが、AIが生成したコードはコントラスト比やキーボードナビゲーションの要件を満たしていませんでした。

3つ目は、エラーハンドリングが不十分だったことです。決済エラー時の表示が「エラーが発生しました」という汎用メッセージだけで、カード番号の誤りなのか、残高不足なのか、ネットワークエラーなのか、ユーザーには区別できませんでした。

これらの問題は、「UIを改善」という曖昧な指示に、デザインシステムの参照、アクセシビリティ要件、エラーハンドリング方針が含まれていなかったことが原因です。

## 1.3 PRが巨大化する／レビュー不能になる／仕様が消える

### 1.3.1 なぜPRは巨大化するのか

「ログイン機能を作って」——この一言で、AIは何を作るでしょうか？

- ・ ログインフォームのUI
- ・ バリデーションロジック
- ・ API呼び出し
- ・ 認証トークンの保存
- ・ エラーハンドリング
- ・ 成功時のリダイレクト
- ・ ログアウト機能も「あった方がいいだろう」と追加
- ・ パスワードリセット機能も「普通はあるよね」と追加

1つの指示から、関連しそうなものの全部を生成しようとなります。

結果、PRは何百行、何千行にもなります。

### 1.3.2 レビュー不能のメカニズム

巨大なPRがレビュー不能になる理由は単純です。

人間がレビューできる限界は、1回あたり200～400行程度と言われています。

それを超えると：

- 細部を見落とす
- 「まあ動いてるからいいか」という妥協が生まれる
- セキュリティホールやパフォーマンス問題が見逃される

また、巨大なPRは変更の意図を追跡できなくなります。

「なぜこのファイルが変更されているのか」「この変更は本当に必要だったのか」「別の方がはなかったのか」

これらの問い合わせに答えられないPRは、レビューしようがありません。

### 1.3.3 仕様が消える

最も深刻な問題は、仕様が消えることです。

AIに「ログイン機能を作って」と言ったとき、その仕様はどこにありますか？

- チャットログの中？（1週間後には埋もれる）
- あなたの頭の中？（共有されていない）
- なんとなくの合意？（明文化されていない）

仕様が明文化されていなければ、何が正しいのか判断できません。

- このバリデーションルールは正しいのか？
- エラーメッセージの文言は意図通りか？
- リダイレクト先はこれでよかったのか？

レビューも、実装者も、後から見た人も、誰も判断できない。

これが「仕様が消える」という状態です。

#### 1.3.3.1 仕様消失がもたらす具体的なコスト

仕様が消えることの影響を、ある開発チームの実測値で示します。

このチームでは、仕様が明文化されていない機能についてバグ報告があったとき、平均して以下の時間がかかっていました。

- 仕様の確認：45分（Slackの過去ログを検索、関係者に確認）
- 「正しい動作」の合意形成：30分（複数人で議論）
- 修正の実装：20分（実際のコード変更）
- レビュー：15分

つまり、修正自体は20分で終わるのに、「何が正しいか」を確認するだけで75分かかっていました。修正時間の3.75倍です。

さらに深刻だったのは、同じ機能について異なる解釈で実装されたケースです。AさんとBさんが別のPRで同じ機能に触れ、それぞれ異なる「正しい動作」を想定して実装しました。結果、本番で矛盾した動作が発生し、緊急対応に4時間を要しました。

仕様を明文化するのに必要な時間は、1機能あたり30分程度です。この30分を惜しんで、何倍もの時間を無駄にしていたことになります。

## 1.4 「AIが賢くない」のではなく「入力が仕様になっていない」

### 1.4.1 責任の所在を正しく理解する

ここまで問題を見て、「やっぱりAIはまだ使えない」と結論づけるのは早計です。

問題の構造を整理しましょう。

現象	よくある誤解	本当の原因
意図と違うコードが生成される	AIの理解力が低い	意図が明示されていない
PRが巨大になる	AIが勝手に拡張する	スコープが定義されていない
仕様と合わない	AIがプロジェクトを知らない	仕様がAIに渡されていない
同じミスを繰り返す	AIが学習しない	ナレッジがフィードバックされていない

すべての原因是、「**入力**」の問題です。

#### 1.4.2 LLMの動作原理を思い出す

生成AIは、与えられた入力から「最も確率の高い続き」を生成します。

入力が曖昧なら、出力も曖昧になります。入力がブレれば、出力もブレます。入力に含まれていない情報は、推測するしかありません。

これはAIの限界ではなく、**AIの仕様**です。

#### 1.4.3 入力を「仕様」にする

解決策は明確です。

**AIへの入力を「仕様」にする。**

仕様とは何か？

- 何を作るのか（要件）
- どう作るのか（設計制約）
- 何を作らないのか（スコープ外）
- 何が正しいのか（受け入れ基準）
- どこに置くのか（既存コードとの関係）

これらが明示されていれば、AIは推測する必要がありません。

推測が不要なら、ブレません。ブレなければ、レビューも楽になります。レビューが楽なら、品質も上がります。

---

### 1.5 章末チェックリスト

この章の内容を実践に移すためのチェックリスト：

- 自分の指示が「vibe coding」になっていないか振り返る
- 最近のPRサイズを確認し、200行を超えているものを特定する
- 仕様がチャットログにしか残っていないケースを洗い出す
- 「AIが悪い」と思った場面を、「入力が不十分だった」視点で再評価する

---

### 1.6 次章への橋渡し

この章では、AI活用が失敗する典型パターンを見てきました。

次章では、なぜAIが「心を読めない」のかをより深く理解します。LLMの得意・不得意を正確に把握することで、「何を明示すべきか」がクリアになります。

## 第2章 AIが苦手なのは"コーディング"ではなく"心を読むこと"

### 2.1 この章で学ぶこと

- LLMが得意な領域と苦手な領域を正確に理解する
  - 「未記述要件の補完」がなぜ危険なのか
  - 仮定を排除して仕様を渡す方法論
- 

### 2.2 LLMが強い領域：既知パターンの組み立て

#### 2.2.1 コーディング能力は本物

まず明確にしておきたいのは、LLMのコーディング能力は非常に高いということです。

以下のようなタスクでは、人間を凌駕するスピードと正確さを発揮します：

##### 1. 既知のパターンの実装

- CRUD操作
- 認証・認可の標準的な実装
- デザインパターンの適用
- ライブラリの標準的な使い方

##### 2. コードの変換・リファクタリング

- 言語間の変換 (Python → TypeScript)
- フレームワーク間の移行 (Express → Fastify)
- レガシーコードのモダナイズ

##### 3. ボイラープレートの生成

- 設定ファイル
- テストの雛形
- CIパイプラインの設定

##### 4. 明確な仕様に基づく実装

- 「この入力でこの出力を返す関数」
- 「このAPIスキーマに従ったエンドポイント」
- 「このテストケースをパスする実装」

#### 2.2.2 なぜ強いのか

LLMは、膨大な量のコードを学習しています。

GitHub上の何百万ものリポジトリ、Stack Overflowの質問と回答、技術ドキュメント、ブログ記事——これらすべてから「こういう要件にはこういう実装」というパターンを学習しています。

そのため、すでに世の中に存在するパターンであれば、高い精度で再現できます。

---

### 2.3 LLMが弱い領域：未記述要件の補完

#### 2.3.1 「書かれていないこと」への対処

LLMが苦手なのは、書かれていない要件を正しく補完することです。

例を見てみましょう。

指示：「ユーザー登録APIを作って」

AIが推測しなければならないこと：

- パスワードの最小文字数は？（8文字？12文字？）
- メールアドレスの重複チェックは？（どのタイミングで？）
- ユーザー名の使用可能文字は？（英数字のみ？日本語OK？）
- 登録後に確認メールを送る？（即時アクティブ？）
- ディレクトリーコードを返す？（400？409？422？）
- レスポンスに何を含める？（ID？作成日時？トークン？）

これらの情報が指示に含まれていなければ、AIは「一般的にはこうだろう」という推測で実装します。

その推測が、あなたのプロジェクトの要件と一致する保証はどこにもありません。

### 2.3.2 勝手な仮定の危険性

AIが置く「勝手な仮定」は、しばしば発見が遅れます。

```
// AIが書いたコード
async function registerUser(email: string, password: string) {
    // AIの仮定：パスワードは8文字以上でよいだろう
    if (password.length < 8) {
        throw new Error('Password must be at least 8 characters');
    }

    // AIの仮定：メールの重複チェックは先にやるべきだろう
    const existing = await db.user.findByEmail(email);
    if (existing) {
        throw new Error('Email already exists');
    }

    // AIの仮定：パスワードはbcryptでハッシュ化するだろう
    const hashed = await bcrypt.hash(password, 10);

    // AIの仮定：作成日時は自動で入るだろう
    return db.user.create({ email, password: hashed });
}
```

このコードは動きます。一見、問題ないように見えます。

しかし：

- 会社のセキュリティポリシーは「12文字以上、英数字記号混在」かもしれない
- 既存システムではArgon2を使っているかもしれない
- ユーザー名も必須かもしれない
- 監査ログを書く要件があるかもしれない

動くコードが、正しいコードとは限らないのです。

### 2.3.3 推測の連鎖

一つの推測が、次の推測を呼びます。

「認証API」 → JWT使うだろう → RS256だろう → 公開鍵はどこに置く？ → 環境変数だろう  
→ トークンの有効期限は？ → 1時間だろう → リフレッシュトークンは？ → いるだろう  
→ どこに保存する？ → Cookieだろう → httpOnlyだろう → SameSiteは？ → Strictだろう

すべてが推測の上に推測を重ねた構造になります。

この推測の連鎖のどこかがプロジェクトの要件と違っていれば、大きな手戻りが発生します。

#### 2.3.3.1 推測連鎖の可視化

推測の連鎖は、実際のコードレビューで発覚するまで見えにくいものです。しかし、AIに「あなたが置いた仮定を列挙して」と聞くことで可視化できます。

あるプロジェクトで試してみた結果を紹介します。「ユーザー認証機能を実装して」という指示に対して、AIが置いた仮定を確認したところ、17個の仮定が見つかりました。

- 認証方式に関する仮定 (JWT、有効期限、リフレッシュトークン) : 5個
- セキュリティに関する仮定 (ハッシュアルゴリズム、ソルト、CSRF対策) : 4個
- ストレージに関する仮定 (トークンの保存場所、セッション管理) : 3個
- エラーハンドリングに関する仮定 (レスポンス形式、ログ出力) : 3個
- UIに関する仮定 (リダイレクト先、エラー表示) : 2個

このうち、プロジェクトの要件と一致していたのは9個でした。残りの8個は修正が必要でした。

推測の可視化を習慣にすることで、「どの程度の仕様を明示すれば推測を減らせるか」が感覚としてわかるようになります。最初は面倒に感じますが、手戻りを大幅に減らせるため、結果的には時間の節約になります。

## 2.4 技術バージョン：もう一つの「未記述要件」

### 2.4.1 AIの学習データには締め切りがある

LLMにはもう一つ、見落とされがちな弱点があります。

それは学習データのカットオフ（知識の締め切り日）です。

LLMは特定の日付までのデータで学習されています。

- Claude Sonnet 4 : 2025年3月頃
- Claude Sonnet 4.5 : 2025年7月頃（信頼性が高いのは2025年1月まで）
- GPT-4o : 2024年6月頃
- GPT-5 : 2024年10月頃
- Gemini 2.5 Pro : 2025年1月末頃

つまり、カットオフ以降にリリースされた技術については、古い情報しか持っていない可能性があります。

### 2.4.2 バージョンを書かないとどうなるか

指示：「Next.jsでApp Routerを使ってAPIルートを作って」

AIが推測すること：

- Next.jsのバージョンは？（13.4？14.0？15.0？）
- 各バージョンでAPIルートの書き方が微妙に違う
- App Routerはいつから安定版？

バージョンを明記しなければ、AIは「学習時点で一般的だったバージョン」を想定します。

結果として：

- 非推奨APIを使ったコードが生成される
- 新しいバージョンでは動かない書き方が提案される
- セキュリティパッチが適用される前の脆弱な実装パターンが使われる

### 2.4.3 実際に起きる問題

Next.jsを例を見てみましょう。

```
// AIが書いたコード (Next.js 13.x想定)
// pages/api/users.ts ← Pages Routerの書き方
export default function handler(req, res) {
  res.status(200).json({ users: [] });
}
```

```
// 実際のプロジェクト (Next.js 15.x) では
// app/api/users/route.ts ← App Routerの書き方
export async function GET() {
  return Response.json({ users: [] });
}
```

ディレクトリ構造も、エクスポート形式も、レスポンスの返し方も、まったく異なります。

AIが古い書き方で実装すると、プロジェクトの構成と合わせず、すべて書き直しになります。

#### 2.4.4 Reactでも同様の問題

```
// AIが書いたコード (React 18想定)
// クライアントコンポーネントとして全体を実装
'use client';
import { useState, useEffect } from 'react';

export default function UserList() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch('/api/users').then(r => r.json()).then(setUsers);
  }, []);

  return <ul>{users.map(u => <li key={u.id}>{u.name}</li>)}</ul>;
}
```

```
// React 19 + Next.js 15では
// Server Componentとしてシンプルに実装
export default async function UserList() {
  const users = await fetch('/api/users').then(r => r.json());
  return <ul>{users.map(u => <li key={u.id}>{u.name}</li>)}</ul>;
}
```

Server Componentsを知らないAIは、不要な 'use client' を付け、useEffect で無理やりデータを取得します。

#### 2.4.5 解決策：技術スタックにバージョンを明記する

MASTER.mdやARCHITECTURE.mdで、使用する技術とバージョンを明示します。

## ## 2.5 技術スタック

技術   バージョン   AIへの注意点
----- ----- -----
Next.js   16.1.x   App Router使用 (Pages Router禁止)
React   19.2.x   Server Componentsがデフォルト
TypeScript   5.9.x   strict mode有効

| PostgreSQL | 17.x | - |  
| Prisma | 6.x | - |

この情報があれば、AIは次のように動作します。

- 「Next.js 16ではApp Routerでこう書く」と正しく実装
- 「React 19.2のServer Componentsを活用した設計」を提案
- 「TypeScript 5.9の新機能を使用可能」と判断

バージョンを書くことは、AIへの「追加の仕様」なのです。

### 2.5.1 なぜADRが必要か

さらに重要なのは、なぜその技術・バージョンを選んだのかを記録することです。

これをADR (Architecture Decision Record) と呼びます。

ADRがあると、AIは次のように動作します。

- AIが技術選定の背景を理解し、一貫した設計提案ができる
- 後から参加したメンバーが「なぜこの技術?」を理解できる
- 将来の変更時に「当時の判断根拠」を参照できる

ADRの詳細な書き方は、第4章のARCHITECTURE.mdセクションで解説します。

#### 2.5.1.1 ADRが解決する具体的な問題

ADRがない状態で起きた実際の問題を紹介します。

あるプロジェクトで、APIの認証方式がJWTで実装されていました。新機能を追加する際、AIに「管理者向けAPIを実装して」と依頼したところ、セッションベースの認証で実装されました。

AIは「管理者向け」という言葉から、「より厳格なセキュリティが必要」と推測し、セッション認証の方が適切と判断したのです。しかし、このプロジェクトでは「マイクロサービス間の呼び出しを考慮してJWTを採用」という設計判断があり、管理者向けAPIもJWTで統一すべきでした。

問題は、この設計判断がどこにも書かれていなかったことです。元々の開発者は「当然JWTで統一だろう」と思っていましたが、AIにとっては「当然」ではありませんでした。

ADRに「認証はJWTを採用、理由はマイクロサービス間の認証を統一するため。セッション認証は使用しない」と書かれていれば、AIはこの判断を尊重し、管理者向けAPIもJWTで実装したはずです。

技術選定の背景を記録するだけで、このような「暗黙の了解」による不整合を防げます。

## 2.6 解法：仮定を排除して、仕様を"書いてから"渡す

### 2.6.1 「仮定させない」という発想

AIに「勝手に仮定するな」と言っても無駄です。情報がなければ、仮定するしかありません。

解決策は、仮定する余地をなくすことです。

つまり、仕様を書いてから渡す。

### 2.6.2 良い仕様の例

先ほどのユーザー登録APIを、仕様として書き直してみましょう。

[## 2.7 ユーザー登録API](#)

[### 2.7.1 エンドポイント](#)

```
POST /api/v1/users
```

### ### 2.7.2 リクエスト

```
```json
{
  "email": "user@example.com",
  "password": "SecureP@ss123",
  "username": "johndoe"
}
```
```

```

### ### 2.7.3 バリデーションルール

- email: 有効なメールアドレス形式、255文字以内
- password: 12文字以上、英大文字・小文字・数字・記号を各1つ以上含む
- username: 3-30文字、英数字とアンダースコアのみ

### ### 2.7.4 重複チェック

- email: 重複時は409 Conflictを返す
- username: 重複時は409 Conflictを返す

### ### 2.7.5 パスワードハッシュ

- アルゴリズム: Argon2id
- メモリコスト: 65536
- 時間コスト: 3
- 並列度: 4

### ### 2.7.6 レスポンス(成功時)

- ステータス: 201 Created
- ボディ:

```
```json
{
  "id": "uuid",
  "email": "user@example.com",
  "username": "johndoe",
  "createdAt": "2026-01-01T00:00:00Z"
}
```
```

```

### ### 2.7.7 レスポンス(エラー時)

- 400 Bad Request: バリデーションエラー
- 409 Conflict: メールまたはユーザー名の重複
- 500 Internal Server Error: サーバーエラー

### ### 2.7.8 監査ログ

- 登録成功時: audit.log に USER\_CREATED イベントを記録
- 登録失敗時: audit.log に USER\_REGISTRATION\_FAILED イベントを記録

### ### 2.7.9 既存コードとの関係

- バリデーション: srcValidators/user.ts の validateUserInput を使用
- エラーレスポンス: srcErrors/http.ts の形式に従う
- 監査ログ: srcServices/audit.ts の AuditService を使用

この仕様をAIに渡せば、推測の余地がほとんどなくなります。

## 2.7.10 仕様に含めるべき情報

最低限、以下の情報を含めましょう。

カテゴリ	含めるべき情報
What	何を作るのか、何を作らないのか
How	どう実装するのか（アルゴリズム、使用ライブラリ）
Where	既存コードのどこに配置するのか
Constraint	制約条件（パフォーマンス、セキュリティ）
Format	入出力の形式（スキーマ、エラーレスポンス）
Test	正しさの判定基準（テストケース、受け入れ条件）

### 2.7.11 Issue = 最小の仕様書

「仕様を書く」と聞くと重厚長大なドキュメントを想像するかもしれません、その必要はありません。

Issue（課題チケット）が最小の仕様書になります。

#### ## 2.8 Issue #42: ユーザー登録API の実装

##### ### 2.8.1 背景

新規ユーザー登録機能を実装する。セキュリティポリシーに準拠した  
パスワード要件を満たすこと。

##### ### 2.8.2 受け入れ基準

- [ ] POST /api/v1/users でユーザーを作成できる
- [ ] パスワードは12文字以上、英大文字・小文字・数字・記号必須
- [ ] メール・ユーザー名の重複時は409を返す
- [ ] 成功時は201と作成されたユーザー情報を返す
- [ ] 監査ログにイベントが記録される

##### ### 2.8.3 技術的制約

- パスワードハッシュはArgon2id（既存の hashService を使用）
- エラーレスポンスは src/errors/http.ts の形式に従う

##### ### 2.8.4 スコープ外

- メール確認フローは別Issue (#43) で対応
- ソーシャルログインは別Issue (#44) で対応

この程度の情報があれば、AIは「仮定」をほとんど置かずに実装できます。

## 2.9 章末チェックリスト

- 最近AIに出した指示を振り返り、「推測が必要だった箇所」を特定する
- 次にAIに指示を出すとき、What/How/Where/Constraint/Format/Testを意識する
- Issueテンプレートを作成し、「受け入れ基準」「技術的制約」「スコープ外」の項目を入れる
- プロジェクトで使用している主要技術のバージョンを確認する
- AIが未学習の可能性がある新しい技術を特定する
- 技術バージョンをMASTER.mdまたはARCHITECTURE.mdに明記する

## 2.10 次章への橋渡し

この章では、LLMの得意・不得意を理解し、「仮定を排除する」ための仕様の書き方を学びました。

次章からは、この「仕様」をプロジェクト全体で体系化する方法——7文書構成——に入っていきます。単発の仕様ではなく、プロジェクト全体を通じてAIが参照できる「生きた仕様」の作り方を解説します。

## 第2部 結論：AIに任せる開発は「仕様」が9割



本書の核心です。AIが迷わず動くために必要な「7文書構成」を提示します。MASTER.mdを中心に、PROJECT / ARCHITECTURE / DOMAIN / PATTERNS / TESTING / DEPLOYMENT の役割と連携を解説。仕様を「生きた成果物」として運用する方法を学びます。

## 第3章 仕様を"生きた成果物"にする

### 3.1 この章で学ぶこと

- 静的ドキュメントから「生きた仕様」への転換
- 変更を「差分」ではなく「影響度」で扱う発想
- 仕様→設計→テスト→運用の連鎖を維持する方法

### 3.2 仕様=合意の置き場所

#### 3.2.1 チャットログは仕様ではない

「あのとき Slack で決めたよね」「PRのコメントに書いてあったはず」「ミーティングで合意したじゃん」

これらは仕様ではありません。

なぜか？

1. 検索できない：埋もれる、見つからない、探す時間がかかる
2. 変更履歴が追えない：いつ、誰が、何を変えたかわからない
3. AIが参照できない：コンテキストとして渡せない

仕様とは、参考可能な場所に置かれた、合意の記録 です。

### 3.2.2 仕様の条件

「これが仕様である」と言えるためには、以下の条件を満たす必要があります。

条件	説明	反例
参照可能	いつでも誰でも見られる	Slackの過去ログ、会議の口頭合意
バージョン管理	変更履歴が追える	上書きされるWikiページ
単一の真実	重複がない、矛盾がない	複数のドキュメントに散在
機械可読	AIやツールが読み取れる	スキャンされたPDF、画像

#### 3.2.2.1 他のフォーマットを選ばなかった理由

仕様管理には様々なツールやフォーマットがあります。なぜMarkdownを選んだのか、他の選択肢との比較で説明します。

**Notion / Confluence**：リッチなUIで編集しやすい一方、バージョン管理が弱点です。「いつ、誰が、何を変えたか」の追跡が難しく、複数人が同時編集すると競合が発生します。また、AIがコンテキストとして参照するためにはAPI連携が必要になり、セットアップの手間が増えます。

**Google Docs**：リアルタイムコラボレーションには優れていますが、構造化された情報の管理には向きません。見出しやリストのスタイルがドキュメントごとにバラバラになりがちで、機械的な処理が困難です。

**JIRA / Linearなどのチケット管理ツール**：タスク管理には優れていますが、仕様の「全体像」を俯瞰するには向きません。情報がチケット単位で分断され、「このプロジェクトの認証方式は何か」といった横断的な質問に答えにくくなります。

**専用の仕様管理ツール（Swagger、OpenAPIなど）**：APIドキュメントには最適ですが、ビジネスルールやアーキテクチャ決定を記述するには不十分です。

Markdownファイルをリポジトリに置く方式は、これらの課題をすべて解決します。Gitによる完璧なバージョン管理、PRによるレビューの一回り、コードと同じリポジトリにあることでの参照しやすさ、そしてAIが直接読み取れるテキスト形式。シンプルですが、最も実用的な選択です。

この条件を満たす最もシンプルな形式が、**GitリポジトリにあるMarkdownファイル**です。

### 3.2.3 なぜMarkdownか

- バージョン管理：Gitで変更履歴が完璧に追える
- 差分表示：PRで変更点がレビューできる
- 構造化：見出し、リスト、テーブルで情報を整理
- AI親和性：LLMが最も扱いやすいテキスト形式
- ツール連携：静的サイト生成、Lintチェックが可能

## 3.3 変更は「差分」ではなく「影響度」で扱う

### 3.3.1 差分思考の限界

従来の仕様変更管理は「差分」に注目します。

変更前：パスワードは8文字以上

変更後：パスワードは12文字以上

差分：文字数が8→12に変更

しかし、この「差分」だけ見ても、影響の大きさはわかりません。

### 3.3.2 影響度思考への転換

「何が変わったか」ではなく、「何に影響するか」を考えます。

#### ## 3.4 変更: パスワード最小文字数 8 → 12

##### ### 3.4.1 影響度:HIGH

###### ### 3.4.2 影響範囲

- [ ] フロントエンド : バリデーションメッセージ更新
- [ ] バックエンド : バリデーションロジック更新
- [ ] テスト : パスワードバリデーションのテストケース追加
- [ ] ドキュメント : API仕様書のパラメータ説明更新
- [ ] 既存ユーザー : パスワード変更時の新ルール適用 (ログイン時は旧パスワード許容?)

###### ### 3.4.3 確認事項

- [ ] 移行期間は設けるか?
- [ ] 既存ユーザーへの通知は必要か?
- [ ] 監査ログへの記録方法は?

これが 影響度評価 (changeImpact) の発想です。

#### 3.4.4 影響度の3段階

変更を以下の3段階で分類します。

影響度	定義	例	対応
LOW	文言・スタイルの修正	エラーメッセージの文言変更、コメント修正	単独で対応可能
MEDIUM	既存概念の拡張	フィールドの追加、オプション機能の追加	関連文書の確認が必要
HIGH	概念の再定義・削除	データモデルの変更、APIの破壊的変更	全関連文書のレビューが必要

HIGH変更のときは、必ず立ち止まって影響範囲を洗い出します。

##### 3.4.4.1 HIGH変更の判断に迷うケース

影響度の分類は単純ではありません。「これはMEDIUMか、HIGHか?」と迷うケースがあります。いくつかの典型例と判断基準を示します。

###### ケース1：APIのレスポンス形式変更

レスポンスに新しいフィールドを追加するのはMEDIUMですが、既存フィールドの型を変更する (例: `string` から `number` へ) のはHIGHです。後方互換性が失われ、クライアント側のコード修正が必須になるためです。

###### ケース2：バリデーションルールの変更

緩める方向 (例: 8文字以上→6文字以上) はMEDIUMで済むことが多いですが、厳しくする方向 (例: 8文字以上→12文字以上) は既存データへの影響を考慮する必要があります、HIGHになることがあります。

###### ケース3：オプション機能のデフォルト値変更

デフォルト値が `false` から `true` に変わる場合、「オプションだから影響は少ない」と思いがちですが、既存ユーザーの挙動が変わるためにHIGHです。

判断に迷ったときは、「既存のユーザーやシステムに影響があるか」を基準にします。新規のみに影響するならMEDIUM、既存にも影響するならHIGHです。そして、迷ったらHIGHとして扱う方が安全です。

## 3.5 "仕様→設計→テスト→運用"の連鎖を切らない

### 3.5.1 連鎖が切れるとき

多くのプロジェクトで、こんな状況が起きています。

仕様：「ユーザーは商品を購入できる」

↓（連鎖）

設計：CartServiceで購入処理

↓（連鎖が切れる）

テスト：???（何をテストすべきかわからない）

↓（連鎖が切れる）

運用：???（何を監視すべきかわからない）

仕様が曖昧なまま設計に入り、設計が曖昧なままテストを書き、テストが曖昧なまま本番に出す。

この連鎖の切断が、品質問題の根本原因です。

### 3.5.2 連鎖を維持する方法

連鎖を維持するには、各フェーズの成果物が次のフェーズの入力になるように設計します。

仕様（PROJECT.md / DOMAIN.md）

↓ 何を作るか、ビジネスルールは何か

設計（ARCHITECTURE.md）

↓ どう作るか、コンポーネント構成は

テスト（TESTING.md）

↓ 何が正しいか、どう検証するか

運用（DEPLOYMENT.md）

↓ どう監視するか、障害時にどう対応するか

### 3.5.3 具体例：購入機能の連鎖

仕様（DOMAIN.md）

#### ## 3.6 購入ルール

- ユーザーは在庫がある商品のみ購入できる
- 購入数量は1~10個まで
- 購入時に在庫を減算する（楽観的ロック使用）
- 在庫不足時は購入を拒否する

設計（ARCHITECTURE.md）

#### ## 3.7 購入処理フロー

1. CartServiceが購入リクエストを受け取る
2. InventoryServiceで在庫確認（楽観的ロック取得）
3. OrderServiceで注文作成
4. InventoryServiceで在庫減算（ロック解放）
5. PaymentServiceで決済処理
6. NotificationServiceで確認メール送信

#### ### 3.7.1 障害パターン

- 在庫確認後～減算前に他ユーザーが購入：409 Conflict返却
- 決済失敗：在庫をロールバック

テスト（TESTING.md）

## ## 3.8 購入機能のテストケース

### ### 3.8.1 正常系

- [ ] 在庫がある商品を1個購入できる
- [ ] 在庫がある商品を10個購入できる

### ### 3.8.2 異常系

- [ ] 在庫0の商品は購入できない (409)
- [ ] 11個以上の購入はバリデーションエラー (400)
- [ ] 購入途中で他ユーザーに在庫を取られた場合 (409)

### ### 3.8.3 負荷テスト

- [ ] 同時100ユーザーの購入で在庫の整合性が保たれる

運用 (DEPLOYMENT.md)

## ## 3.9 購入機能の監視

### ### 3.9.1 メトリクス

- purchase\_success\_rate: 購入成功率 (目標: 99.9%)
- inventory\_conflict\_rate: 在庫競合率 (警告: 5%超過)

### ### 3.9.2 アラート

- 購入成功率が95%を下回ったら即座に通知
- 在庫競合率が10%を超えたら在庫データ確認

### ### 3.9.3 障害対応Runbook

- [購入失敗急増時の対応手順](./runbooks/purchase-failure.md)

この連鎖が維持されているから、AIは一貫した実装ができます。

## 3.10 章末チェックリスト

- 仕様がチャットログやWikiに散在していないか確認する
- 最近の仕様変更を「影響度」で分類してみる
- 仕様→設計→テスト→運用の連鎖が途切れている箇所を特定する
- 途切れている箇所を補完するドキュメントを作成する計画を立てる

## 3.11 次章への橋渡し

この章では、仕様を「生きた成果物」として維持する考え方を学びました。

次章では、この考え方を具体化した7文書構成を紹介します。どの情報をどの文書に置くか、7つの文書がどのように連携するか、詳細に解説します。

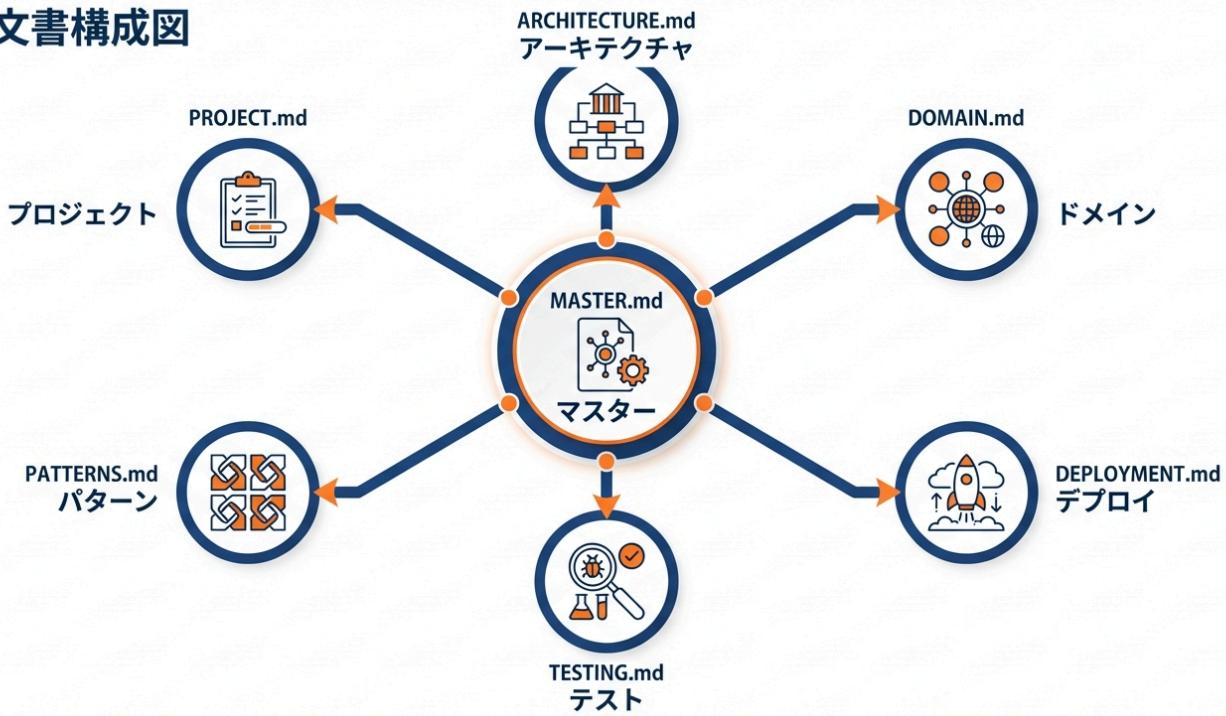
# 第4章 7文書が「AI任せ」を成立させる最小構成

## 4.1 この章で学ぶこと

- 7文書の全体像と各文書の役割
- 何が欠けるとAIが迷うのか
- 各文書の具体的な内容と書き方

## 4.2 7文書の全体像

### 7文書構成図



#### 4.2.1 なぜ7つなのか

従来のソフトウェア開発ドキュメントは、60以上のテンプレートがあることも珍しくありません。

- 要件定義書
- 基本設計書
- 詳細設計書
- テスト計画書
- テスト仕様書
- 運用手順書
- ...

しかし、AIにとって60ファイルは多すぎます。

コンテキストウィンドウの制限もありますが、それ以上に**情報が散在すると矛盾が生まれやすい**という問題があります。

7文書は、**AIが効率的に参照できる最小構成**として設計されています。

#### 4.2.1.1 7文書に絞った経緯

この7文書構成は、複数のプロジェクトでの試行錯誤から生まれました。

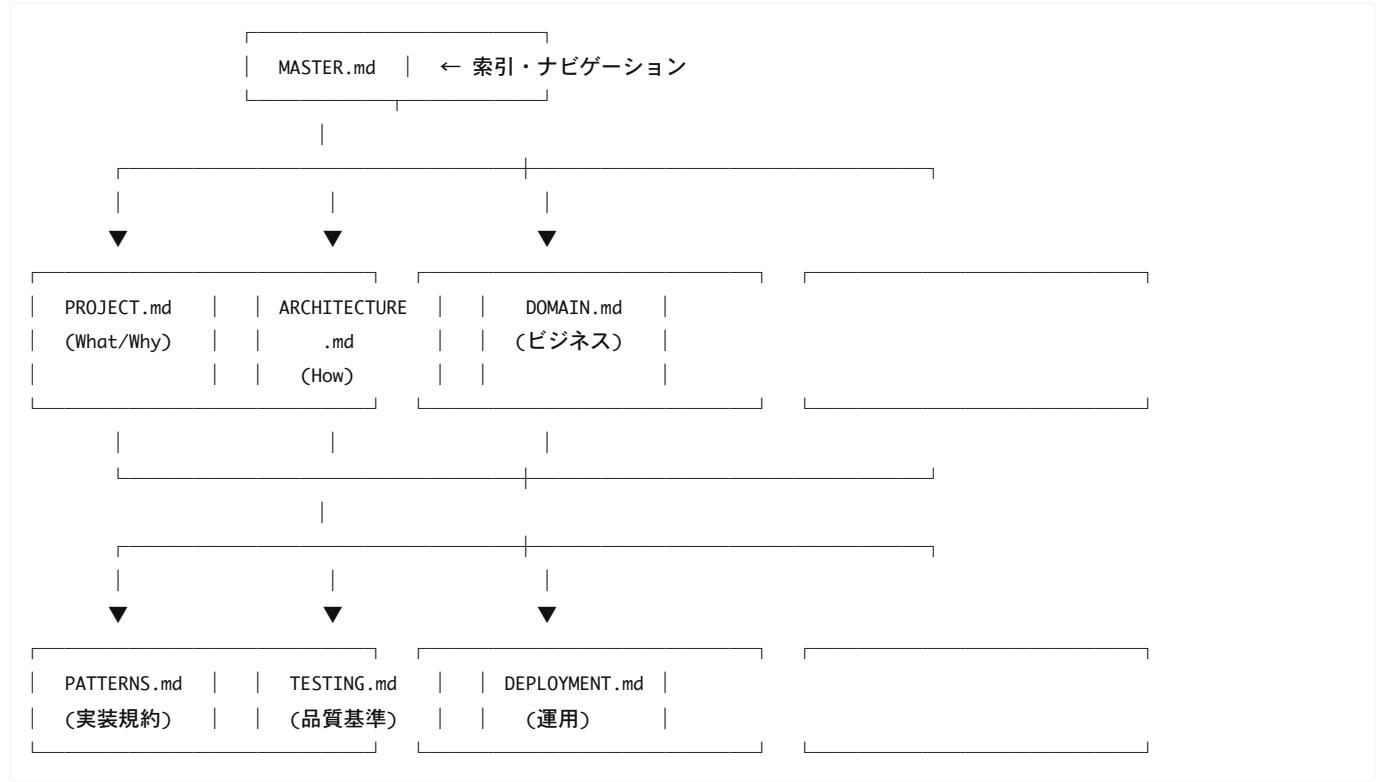
最初は「必要な文書をすべて用意する」アプローチで、12文書構成を試しました。要件定義、機能仕様、非機能要件、API仕様、DB設計、コンポーネント設計、テスト計画、テスト仕様、デプロイ手順、監視設計、障害対応、用語集——すべて別ファイルで管理しました。

結果、2つの問題が発生しました。1つ目は、**文書間の重複と矛盾**です。「ユーザー登録の仕様」が要件定義にも機能仕様にもAPI仕様にも書かれ、それぞれ微妙に異なる記述になりました。2つ目は、**AIへの指示の複雑化**です。「この機能を実装して」と言うたびに、「要件定義のこの部分と、機能仕様のこの部分と、API仕様のこの部分を参照して」と指定する必要がありました。

逆に、「最小限の文書だけでやってみよう」と3文書（README、ARCHITECTURE、PATTERNS）だけで始めたプロジェクトもありました。こちらは**ビジネスルールの置き場所がない**という問題が発生しました。「この割引ルールはどこに書く？」「この状態遷移はどこで管理する？」という疑問が頻発し、結局READMEが肥大化しました。

7文書という数は、「情報の重複を避けつつ、必要な情報が欠けない」バランスポイントとして落ち着いた結果です。それぞれの文書に明確な役割があり、「この情報はどこに書くか」が迷わない構成になっています。

#### 4.2.2 7文書の関係図



#### 4.2.3 何が欠けるとAIが迷うか

欠けている文書	AIが迷うこと	結果
MASTER.md	どこに何があるかわからない	無関係なコードを参照する
PROJECT.md	なぜこの機能が必要かわからない	要件と違う実装をする
ARCHITECTURE.md	どう実装すべきかわからない	既存と整合しない設計をする
DOMAIN.md	ビジネスルールがわからない	ルール違反の実装をする
PATTERNS.md	どう書くべきかわからない	一貫性のないコードを書く
TESTING.md	何をテストすべきかわからない	テストが漏れる/過剰になる
DEPLOYMENT.md	どうリリースすべきかわからない	運用できない実装をする

##### 4.2.3.1 実際に欠けていた事例

あるECサイトのプロジェクトで、DOMAIN.mdが存在しない状態でAIに「商品購入機能を実装して」と依頼したケースを紹介します。

AIは購入機能を実装しましたが、以下のビジネスルールが欠けていました。

- 1回の購入は10商品まで（制限なしで実装された）
- 在庫不足時は購入ボタンを非活性にする（エラーメッセージ表示のみだった）
- 購入完了後は在庫を即座に減らす（非同期で遅延減算されていた）
- 送料は5,000円以上で無料（送料計算ロジックが存在しなかった）

これらのルールはチーム内では「当たり前」でしたが、どこにも書かれていませんでした。AIはそれを知る術がなく、一般的なECサイトの実装パターンで推測しました。

レビューで指摘されて修正しましたが、もしDOMAIN.mdに「購入ルール」として明記されていれば、最初から正しい実装が得られたはずです。このプロジェクトでは、この経験をきっかけにDOMAIN.mdを整備し、同様の問題が再発しなくなりました。

## 4.3 MASTER.md：プロジェクト索引

### 4.3.1 役割

MASTER.mdはAIが最初に読む文書です。

プロジェクトの地図として機能し、「何がどこにあるか」を示します。

### 4.3.2 含めるべき内容

#### # MASTER.md

##### ## 4.4 プロジェクト概要

- プロジェクト名：[名前]
- 目的：[1~2文で]
- 主要技術：[言語、フレームワーク、DB]

##### ## 4.5 文書索引

文書   説明   更新日
----- ----- -----
[PROJECT.md](./PROJECT.md)   ビジョン・要件   2026-01-01
[ARCHITECTURE.md](./ARCHITECTURE.md)   システム設計   2026-01-01
[DOMAIN.md](./DOMAIN.md)   ビジネスロジック   2026-01-01
[PATTERNS.md](./PATTERNS.md)   実装パターン   2026-01-01
[TESTING.md](./TESTING.md)   テスト戦略   2026-01-01
[DEPLOYMENT.md](./DEPLOYMENT.md)   運用手順   2026-01-01

##### ## 4.6 ディレクトリ構造

```
```text
src/
├── api/          # APIエンドポイント
├── services/     # ビジネスロジック
├── repositories/ # データアクセス
├── domain/       # ドメインモデル
└── utils/        # ユーティリティ
````
```

##### ## 4.7 重要な制約

- [最重要ルール1]
- [最重要ルール2]

### 4.7.1 AIへの効果

MASTER.mdがあると、AIは次のように動作します。

- ・ 「認証の実装はsrc/services/auth/にある」と理解できる
- ・ 「DB設計はARCHITECTURE.mdを見ればいい」と判断できる
- ・ 「コーディング規約はPATTERNS.mdに従う」と認識できる

## 4.8 PROJECT.md：ビジョンと要件

### 4.8.1 役割

**What** (何を作るか) と **Why** (なぜ作るか) を定義します。

#### 4.8.2 含めるべき内容

##### # PROJECT.md

###### ## 4.9 ビジョン

[このプロジェクトが実現したい世界を1~2文で]

###### ## 4.10 ターゲットユーザー

- ペルソナ1 : [具体的な人物像]
- ペルソナ2 : [具体的な人物像]

###### ## 4.11 主要機能

###### ### 4.11.1 MVP(必須)

- [ ] 機能A : [説明]
- [ ] 機能B : [説明]

###### ### 4.11.2 Phase 2

- [ ] 機能C : [説明]

###### ## 4.12 非機能要件

- パフォーマンス : [具体的な数値目標]
- セキュリティ : [必須要件]
- 可用性 : [目標SLA]

###### ## 4.13 スコープ外

- [明示的に作らないもの]

#### 4.13.1 AIへの効果

PROJECT.mdがあると、AIは次のように動作します。

- 「この機能はMVPに含まれるのか」を判断できる
- 「パフォーマンス目標を満たす実装」を選択できる
- 「スコープ外の機能を勝手に追加しない」

#### 4.14 ARCHITECTURE.md : システム設計

##### 4.14.1 役割

**How** (どう作るか) を定義します。技術的な制約と設計判断を記録します。

##### 4.14.2 技術スタックのバージョン明記が重要な理由

第2章で解説した通り、AIには学習データのカットオフがあります。

バージョンを明記しないと、AIは「学習時点での一般的だったバージョン」を想定し、古い書き方をしてしまいます。

##### # X 悪い例:バージョンなし

###### ## 4.15 技術スタック

|                       |  |
|-----------------------|--|
| ----- -----           |  |
| Frontend   Next.js    |  |
| Backend   Node.js     |  |
| Database   PostgreSQL |  |

- AIは「どのバージョンのNext.js?」と推測するしかない
- Pages Routerの古い書き方で実装される可能性

#### # ✓ 良い例:バージョンあり

#### ## 4.16 技術スタック

|   |  |  |  |
|---|--|--|--|
| レイヤー   技術   バージョン   AIへの注意点                                 |  |  |  |
| ----- ----- ----- -----                                     |  |  |  |
| Frontend   Next.js   16.1.x   App Router使用 (Pages Router禁止) |  |  |  |
| Runtime   Node.js   22.x LTS   ES2024構文使用可                  |  |  |  |
| Database   PostgreSQL   17.x   JSON、全文検索対応                  |  |  |  |
| ORM   Prisma   6.x   型安全、マイグレーション                           |  |  |  |

→ AIはバージョン固有の書き方を選択できる

→ 非推奨APIを避けた実装が得られる

「AIへの注意点」列を追加することで、AIが特に気をつけるべきポイントを明示できます。

#### 4.16.1 含めるべき内容

#### # ARCHITECTURE.md

#### ## 4.17 システム構成図

[Mermaidやテキストでの図]

#### ## 4.18 技術スタック

|  |  |  |  |  |
|--|--|--|--|--|
| レイヤー   技術   バージョン   選定理由   ADR                           |  |  |  |  |
| ----- ----- ----- ----- -----                            |  |  |  |  |
| Frontend   Next.js   16.1.x   App Router、RSC対応   ADR-003 |  |  |  |  |
| State   Zustand   5.x   軽量、TypeScript親和性   -             |  |  |  |  |
| Backend   Hono   4.x   軽量、型安全   ADR-004                  |  |  |  |  |
| Database   PostgreSQL   17.x   リレーショナル、ACID   ADR-002    |  |  |  |  |
| ORM   Prisma   6.x   型生成、マイグレーション   -                    |  |  |  |  |
| Cache   Redis   7.x   セッション、キャッシング   -                   |  |  |  |  |

#### ## 4.19 コンポーネント設計

##### ### 4.19.1 APIレイヤー

- ルーティング: [方針]
- 認証: [方式]
- エラーハンドリング: [方式]

##### ### 4.19.2 サービスレイヤー

- 依存性注入: [方式]
- トランザクション: [方針]

##### ### 4.19.3 データアクセスレイヤー

- ORM: [使用ライブラリ]
- マイグレーション: [方式]

#### ## 4.20 設計判断記録(ADR)

[後述のADRテンプレートを使用して記録]

#### 4.20.1 ADR (Architecture Decision Record) の重要性

技術選定には「なぜその技術・バージョンを選んだのか」の記録が必要です。

これをADR（Architecture Decision Record）と呼びます。

ADRがあると、AIは次のように動作します。

- AIが技術選定の背景を理解し、一貫した設計提案ができる
- 後から参加したメンバーが「なぜこの技術？」を理解できる
- 将来の変更時に「当時の判断根拠」を参照できる
- 代替案として却下された技術を再提案しない

#### 4.20.2 ADRテンプレート（AI向け最適化版）

従来のADRテンプレートに「バージョン選定理由」と「AIへの指示」セクションを追加したものです。

##### ## 4.21 ADR-00X: [技術名]の選定

###### ### 4.21.1 ステータス

承認済み / 検討中 / 廃止

###### ### 4.21.2 コンテキスト

[なぜこの決定が必要になったのか]

###### ### 4.21.3 決定

[何を選んだのか、バージョンを含めて明記]

###### ### 4.21.4 バージョン選定理由

- \*\*なぜこのバージョンか\*\*: [具体的な理由]
- \*\*AIカットオフ対策\*\*: [このバージョンはAIの知識範囲内か、注意点は何か]
- \*\*非推奨API回避\*\*: [避けるべき古いAPIパターン]
- \*\*LTS/サポート期間\*\*: [サポート終了予定]

###### ### 4.21.5 代替技術との比較

| 技術    | 採用/却下 | 理由   |
|-------|-------|------|
| [代替1] | 却下    | [理由] |
| [代替2] | 却下    | [理由] |

###### ### 4.21.6 影響

- \*\*ポジティブ\*\*: [良い影響]
- \*\*ネガティブ\*\*: [考慮すべき制約]

###### ### 4.21.7 AIへの指示

[この技術を使う際に、AIが守るべきルール]

###### ### 4.21.8 関連

- 関連ADR: ADR-00Y
- 公式ドキュメント: [URL]

#### 4.21.9 ADR記述例：Next.js 16の選定

具体的な記述例を見てみましょう。

##### ## 4.22 ADR-003: Next.js 16.1.xの採用

###### ### 4.22.1 ステータス

承認済み

### ### 4.22.2 コンテキスト

SPAからSSR対応のフルスタックフレームワークへ移行が必要。  
SEO対応、初期表示速度の改善、サーバーコンポーネントの活用が目的。

### ### 4.22.3 決定

Next.js 16.1.x (App Router) を採用する。

### ### 4.22.4 バージョン選定理由

- \*\*なぜ16.1.xか\*\*: Turbopackファイルシステムキャッシュが安定版、開発起動時間が10~14倍高速化
- \*\*AIカットオフ対策\*\*: 2025年10月リリースのため、一部AIは未学習の可能性。  
`app/`ディレクトリ構造とRoute Handlers形式を仕様に明記すること。
- \*\*非推奨API回避\*\*:
  - ✗ `pages/api/\*.ts` (Pages Router形式) は使用禁止
  - ✗ `getServerSideProps` は使用禁止
  - ✓ `app/api/\*/route.ts` 形式を使用
- \*\*サポート期間\*\*: Next.js 16はメジャーバージョン、長期サポート対象

### ### 4.22.5 代替技術との比較

| 技術 | 採用/却下 | 理由 |

| -----                                | ----- | ----- |
|--------------------------------------|-------|-------|
| Next.js 15   却下   Turbopackキャッシュが不安定 |       |       |
| Remix   却下   エコシステム、Vercelとの親和性      |       |       |
| SvelteKit   却下   チームの学習コスト           |       |       |

### ### 4.22.6 影響

- \*\*ポジティブ\*\*:
  - Server Componentsによるバンドルサイズ削減
  - Turbopackキャッシュによる開発体験の大幅向上
- \*\*ネガティブ\*\*:
  - 一部のnpmパッケージがRSC非対応
  - AIが古いPages Router形式を提案する可能性（仕様明記で回避）

### ### 4.22.7 AIへの指示

Next.jsのコード生成時は以下を遵守：

- ルーティング: `app/`ディレクトリを使用
- APIルート: `route.ts`のHTTPメソッドエクスポート形式
- レンダリング: デフォルトはServer Component、`"use client"`は明示的に指定時のみ
- データ取得: `fetch`のキャッシングオプションを明示的に指定

### ### 4.22.8 関連

- 公式ドキュメント: <https://nextjs.org/docs>
- 移行ガイド: <https://nextjs.org/docs/app/guides/upgrading>

## 4.22.9 ADRの管理方法

ADRは以下のいずれかで管理します。

### 1. ARCHITECTURE.md内にセクションとして記載（推奨）

- 小～中規模プロジェクト向け
- 1ファイルで完結し、AIが参照しやすい

### 2. 別ファイル (DECISIONS.md) に分離

- 大規模プロジェクト向け
- ADRが10件以上になった場合

### 3. adr/ディレクトリに個別ファイル

- 非常に大規模なプロジェクト向け
- ただしAIが参照しにくくなる点に注意

#### 4.22.10 AIへの効果

ARCHITECTURE.mdがあると、AIは次のように動作します。

- 「Next.js 15のApp Routerで実装」をする
- 「PostgreSQL 16の特性を活かしたクエリ」を書く
- 「ADRで決定済みの方式」に従う
- 「却下された代替技術」を再提案しない
- 「非推奨API」を使用しない

## 4.23 DOMAIN.md : ビジネスロジック

### 4.23.1 役割

ビジネスルールの唯一の置き場所です。仕様の「正しさ」の定義がここにあります。

### 4.23.2 含めるべき内容

#### # DOMAIN.md

##### ## 4.24 ドメインモデル

###### ### 4.24.1 User

- id: UUID
- email: string (一意、必須)
- status: "active" | "suspended" | "deleted"

###### ### 4.24.2 Order

- id: UUID
- userId: UUID (必須)
- items: OrderItem[] (1件以上必須)
- status: "pending" | "confirmed" | "shipped" | "completed"

##### ## 4.25 ビジネスルール

###### ### 4.25.1 購入ルール

- ユーザーはactive状態でのみ購入可能
- 1回の注文は10商品まで
- 在庫がない商品は購入不可

###### ### 4.25.2 価格計算ルール

- 税率: 10% (税込表示)
- 割引適用順序: クーポン → ポイント
- 送料: 5,000円以上で無料、未満は500円

##### ## 4.26 状態遷移

###### ### 4.26.1 Order状態遷移

```
```text
pending → confirmed → shipped → completed
      ↓           ↓
     cancelled   cancelled
```

```

## ## 4.27 用語集

|                                  |
|----------------------------------|
| 用語   定義                          |
| ----- -----                      |
| アクティブユーザー   status="active"のユーザー |
| 有効在庫   予約済みを除いた在庫数               |

### 4.27.1 AIへの効果

DOMAIN.mdがあると、AIは次のように動作します。

- ・「active状態のチェックを入れる」ことを忘れない
- ・「税込計算の順序を正しく実装」する
- ・「用語を統一して使用」する

## 4.28 PATTERNS.md：実装パターン

### 4.28.1 役割

「どう書くべきか」のナレッジ蓄積先 です。レビューで指摘されたパターンをここに集約します。

### 4.28.2 含めるべき内容

#### # PATTERNS.md

## ## 4.29 コーディング規約

### ### 4.29.1 命名規則

- 変数 : camelCase
- 定数 : UPPER\_SNAKE\_CASE
- クラス : PascalCase
- ファイル : kebab-case.ts

### ### 4.29.2 エラーハンドリング

```
```typescript
// ✅ Good
const result = await userService.findById(id);
if (!result.ok) {
  return err(new UserNotFoundError(id));
}
return ok(result.value);

// ❌ Bad
try {
  const user = await userService.findById(id);
} catch (e) {
  throw new Error('User not found');
}
````
```

## ## 4.30 頻出パターン

### ### 4.30.1 リポジトリパターン

```
```typescript
interface UserRepository {
  findById(id: string): Promise<Result<User, NotFoundError>>;
  save(user: User): Promise<Result<User, SaveError>>;
}
````
```

### ### 4.30.2 サービスパターン

```
```typescript
class UserService {
    constructor(private readonly repo: UserRepository) {}

    async getUser(id: string): Promise<Result<UserDTO, GetUserError>> {
        // ビジネスロジック
    }
}
```
```

```

### ## 4.31 アンチパターン

#### ### 4.31.1 避けるべき実装

- any型の使用（代わりにunknownを使用）
- マジックナンバー（代わりに定数化）
- 複数責務の関数（単一責任に分割）

## 4.31.2 AIへの効果

PATTERNS.mdがあると、AIは次のように動作します。

- 「Result型を使ったエラーハンドリング」を実装する
- 「リポジトリパターン」に従った設計をする
- 「アンチパターンを避けた」コードを書く

## 4.32 TESTING.md：テスト戦略

### 4.32.1 役割

「何が正しいか」の検証方法を定義します。テストの書き方と品質基準をここに集約します。

### 4.32.2 含めるべき内容

#### # TESTING.md

### ## 4.33 テストピラミッド

- Unit: 70% (ドメインロジック中心)
- Integration: 20% (API/DB連携)
- E2E: 10% (クリティカルパス)

### ## 4.34 テストの書き方

#### ### 4.34.1 ユニットテスト

```
```typescript
describe('UserService', () => {
    describe('getUser', () => {
        it('存在するユーザーを取得できる', async () => {
            // Arrange
            const repo = createMockRepo({ findById: ok(mockUser) });
            const service = new UserService(repo);

            // Act
            const result = await service.getUser('user-1');

            // Assert
            expect(result.ok).toBe(true);
        });
    });
}
```
```

```

```
    expect(result.value.id).toBe('user-1');
  });
});
});
```

```

### ### 4.34.2 モックの方針

- 外部API：必ずモック
- DB：Integrationテストでは実DB使用
- 時間：固定値を注入

### ## 4.35 カバレッジ目標

| 対象   目標          |
|------------------|
| ----- -----      |
| ドメインロジック   90%以上 |
| サービス層   80%以上    |
| API層   70%以上     |
| ユーティリティ   80%以上  |

## 4.35.1 AIへの効果

TESTING.mdがあると、AIは次のように動作します。

- 「Arrange-Act-Assertパターン」でテストを書く
- 「適切なモック戦略」を選択する
- 「カバレッジ目標を意識した」テストを追加する

## 4.36 DEPLOYMENT.md：運用手順

### 4.36.1 役割

「どうリリースするか」「どう運用するか」を定義します。

### 4.36.2 含めるべき内容

#### # DEPLOYMENT.md

### ## 4.37 環境

| 環境   URL   用途                         |
|---------------------------------------|
| ----- ----- -----                     |
| development   localhost:3000   ローカル開発 |
| staging   staging.example.com   検証環境  |
| production   example.com   本番環境       |

### ## 4.38 デプロイフロー

1. PR作成 → CIでテスト実行
2. レビュー承認 → mainにマージ
3. 自動デプロイ → staging環境
4. 手動承認 → production環境

### ## 4.39 監視項目

| メトリクス   警告閾値   重大閾値       |
|---------------------------|
| ----- ----- -----         |
| レスポンスタイム   500ms   1000ms |
| エラーレート   1%   5%          |
| CPU使用率   70%   90%        |

## ## 4.40 障害対応

### ### 4.40.1 Runbook

- [APIレスポンス遅延](./runbooks/api-slow.md)
- [DB接続エラー](./runbooks/db-connection.md)
- [認証失敗急増](./runbooks/auth-failure.md)

## 4.40.2 AIへの効果

DEPLOYMENT.mdがあると、AIは次のように動作します。

- ・ 「環境変数の扱い方」を正しく実装する
- ・ 「監視しやすい設計」を意識する
- ・ 「運用しやすいログ出力」を入れる

## 4.41 章末チェックリスト

- ・  7文書のうち、自分のプロジェクトに存在する文書を確認する
- ・  最も欠けている（または曖昧な）文書を特定する
- ・  まずMASTER.mdを作成（または整備）する
- ・  次に欠けている文書を1つ選び、最小限の内容で作成する
- ・  技術スタックにバージョンを明記する
- ・  主要な技術選定についてADRを作成する
- ・  ADRに「バージョン選定理由」「AIカットオフ対策」「非推奨API」を含める

## 4.42 次章への橋渡し

この章では、7文書それぞれの役割と内容を学びました。

次章では、この7文書をどのように運用するか—Frontmatter、バージョン管理、検証チェックなど、日々の運用ルールを解説します。

# 第5章 「7文書」を回すための最低限のルール

## 5.1 この章で学ぶこと

- ・ Frontmatterでメタデータを統一する方法
- ・ 変更時のバージョン更新・Changelog運用
- ・ コミット前の検証チェックの仕組み

## 5.2 Frontmatterでメタデータを揃える

### 5.2.1 なぜメタデータが必要か

7文書が増えてくると、こんな問題が起きます。

- ・ 「この文書、いつ更新されたっけ？」
- ・ 「誰がこれを書いたの？」
- ・ 「このドラフト、もう確定してる？」

これらの疑問に答えるために、各文書の先頭にメタデータを記述します。

### 5.2.2 Frontmatter形式

YAML形式のFrontmatterを使います。

```
---  
title: ARCHITECTURE.md  
version: 1.2.0  
status: approved  
owner: "@tech-lead"  
created: 2026-01-01  
updated: 2024-03-15  
reviewers:  
- "@senior-dev"  
- "@security-team"  
---  
  
# ARCHITECTURE.md
```

(本文)

### 5.2.3 必須フィールド

| フィールド   | 説明           | 例                         |
|---------|--------------|---------------------------|
| title   | 文書タイトル       | ARCHITECTURE.md           |
| version | セマンティックバージョン | 1.2.0                     |
| status  | 文書の状態        | draft / review / approved |
| owner   | 責任者          | @username                 |
| created | 作成日          | 2026-01-01                |
| updated | 最終更新日        | 2026-01-05                |

### 5.2.4 オプションフィールド

| フィールド         | 説明       | 用途                  |
|---------------|----------|---------------------|
| reviewers     | レビュー一覧   | 承認フロー管理             |
| tags          | タグ       | 検索・分類               |
| related       | 関連文書     | 相互参照                |
| changelImpact | 最新変更の影響度 | LOW / MEDIUM / HIGH |

### 5.2.5 statusの運用

#### 5.2.5.1 なぜstatusが必要なのか

statusフィールドは「この文書を信じてよいか」をAIに伝えるための重要なメタデータです。

あるプロジェクトでは、statusフィールドを設けていなかったために、次のような問題が発生しました。チームが議論中だったAPI設計の「たたき台」をAIが正式仕様として解釈し、そのまま実装を進めてしまったのです。結果、レビュー時に「これ、まだ決まってないよね？」という指摘があり、3日分の実装がやり直しになりました。

statusを明示することで、AIは「draft文書は参考情報として扱い、approved文書は厳格に遵守する」という判断ができます。これは人間がドキュメントを読むときと同じです。「レビュー中」と書かれた資料を見れば、内容が変わる可能性を意識して読みますよね。AIにも同じ文脈を与えるのがstatusの役割です。

文書のライフサイクルを3段階で管理します。

`draft → review → approved`

↑ ----- |

(修正が必要な場合)

| status   | 意味      | AIへの扱い         |
|----------|---------|----------------|
| draft    | 作成中・未確定 | 参考情報として扱う      |
| review   | レビュー中   | ほぼ確定だが変更の可能性あり |
| approved | 承認済み    | 正式な仕様として遵守     |

#### 5.2.5.2 status運用でよくある失敗

status運用には、2つのよくある失敗パターンがあります。

1つ目は「reviewに留めすぎる」パターンです。「まだ変わるかもしれないから」と慎重になりすぎて、いつまでもapprovedにならない文書が増えていきます。こうなるとAIは常に「変更の可能性あり」と判断し、実装時に必要以上の確認を求めてきます。対策としては、「1週間レビューコメントがなければapproved」のような時限ルールを設けることが有効です。

2つ目は「approvedを早く出しすぎる」パターンです。急いでいるからとレビューをスキップしてapprovedにすると、後から「やっぱり違った」という修正が頻発します。AIは正式仕様として実装を進めてしまうため、手戻りのコストが大きくなります。対策としては、最低1人のレビューによる確認をapprovedの条件にしましょう。

運用のコツは、「draftは1週間以内にreviewへ、reviewは1週間以内にapprovedへ」という目安を持つことです。停滞している文書があれば、週次の振り返りで棚卸しします。

### 5.3 変更時のワークフロー：影響度評価→バージョン更新→Changelog

#### 5.3.1 変更の3ステップ

文書を変更するときは、以下の3ステップを踏みます。

1. 影響度評価 (changeImpact)

↓

2. バージョン更新

↓

3. Changelog記録

#### 5.3.2 ステップ1：影響度評価

変更内容を以下の基準で評価します。

| 影響度    | 基準         | バージョン更新           |
|--------|------------|-------------------|
| LOW    | 誤字修正、文言調整  | パッチ (0.0.x)       |
| MEDIUM | 項目追加、既存拡張  | マイナー (0.x.0)      |
| HIGH   | 構造変更、概念再定義 | メジャーバージョン (x.0.0) |

#### 5.3.3 ステップ2：バージョン更新

Frontmatterのversionを更新し、updatedを現在日付に変更します。

```
---  
# Before  
version: 1.2.0
```

updated: 2024-03-01

```
# After (MEDIUM変更の場合)
version: 1.3.0
updated: 2024-03-15
changeImpact: MEDIUM
---
```

### 5.3.3.1 なぜドキュメントにセマンティックバージョニングを適用するのか

ソフトウェアのバージョン管理でおなじみのセマンティックバージョニング（SemVer）を、なぜドキュメントにも適用するのでしょうか。

最大の理由は、**変更の影響度をバージョン番号だけで伝えられること**です。「1.2.0 → 1.2.1」を見れば誤字修正程度だとわかり、「1.2.0 → 2.0.0」を見れば大きな構造変更があったとわかります。AIがドキュメントを参照する際、このバージョン情報から「前回参照時から大きく変わっている可能性がある」と判断できます。

また、複数の文書間で影響を追跡しやすくなります。ARCHITECTURE.mdが2.0.0になったとき、関連するDOMAIN.mdやPATTERNS.mdも見直しが必要かもしれません。バージョン番号の「ジャンプ幅」が、その判断材料になります。

注意点として、ドキュメントのバージョンは**ソフトウェアのバージョンとは独立して管理します**。アプリがv3.0.0でも、ARCHITECTURE.mdはv1.5.0ということがあります。文書の内容変更に応じてバージョンを上げるのがポイントです。

### 5.3.4 ステップ3：Changelog記録

各文書の末尾、またはプロジェクトルートのCHANGELOG.mdに記録します。

#### ## 5.4 Changelog

##### ### 5.4.1 [1.3.0] - 2024-03-15

###### #### 5.4.1.1 追加

- ユーザー認証フローにMFA対応を追加

###### #### 5.4.1.2 変更

- セッションタイムアウトを30分→60分に変更

##### ### 5.4.2 [1.2.0] - 2024-03-01

###### #### 5.4.2.1 追加

- リフレッシュトークンの仕様を追加

### 5.4.3 HIGH変更のときの追加手順

影響度がHIGHの場合、追加で以下を実施します。

#### 1. 関連文書の洗い出し

##### ## HIGH変更チェックリスト

- [ ] PROJECT.md : 要件への影響確認
- [ ] DOMAIN.md : ビジネスルールへの影響確認
- [ ] TESTING.md : テストケースの更新
- [ ] DEPLOYMENT.md : 運用手順への影響確認

#### 2. レビュワーへの通知

- PRに「HIGH変更」ラベルを付与
- 関連チームメンバーをレビュワーに追加

#### 3. ADR (Architecture Decision Record) の作成

- なぜこの変更が必要か
  - 代替案とその却下理由
  - 移行計画
- 

## 5.5 コミット前に検証チェック

### 5.5.1 なぜ検証が必要か

文書が増えると、以下の問題が発生しやすくなります。

- リンク切れ
- 用語の不統一
- 構造の不整合（見出しレベルがおかしい）
- Frontmatterの記述漏れ

これらをコミット前に自動チェックします。

### 5.5.2 検証チェックの種類

| チェック種類 | 内容                    | ツール例                |
|--------|-----------------------|---------------------|
| 構造検証   | Frontmatter必須項目、見出し構造 | カスタムスクリプト           |
| リンク検証  | 内部リンクの存在確認            | markdown-link-check |
| 用語検証   | 用語集との整合性              | textlint            |
| 整合性検証  | 文書間の参照整合性             | カスタムスクリプト           |

### 5.5.3 pre-commitフックの設定

```
#!/bin/bash
# .husky/pre-commit

echo "▣ 文書検証チェックを実行中..."

# Frontmatter検証
node scripts/validate-frontmatter.js docs/*.md
if [ $? -ne 0 ]; then
  echo "✖ Frontmatter検証に失敗しました"
  exit 1
fi

# リンク検証
npx markdown-link-check docs/*.md
if [ $? -ne 0 ]; then
  echo "✖ リンク検証に失敗しました"
  exit 1
fi

# 用語検証
npx textlint docs/*.md
if [ $? -ne 0 ]; then
  echo "✖ 用語検証に失敗しました"
  exit 1
fi
```

```
echo "✅ 全ての検証チェックに合格しました"
```

#### 5.5.4 Frontmatter検証スクリプト例

```
// scripts/validate-frontmatter.js
const fs = require('fs');
const matter = require('gray-matter');

const requiredFields = ['title', 'version', 'status', 'owner', 'created', 'updated'];
const validStatuses = ['draft', 'review', 'approved'];

function validateFrontmatter(filePath) {
  const content = fs.readFileSync(filePath, 'utf-8');
  const { data } = matter(content);

  const errors = [];

  // 必須フィールドチェック
  for (const field of requiredFields) {
    if (!data[field]) {
      errors.push(`Missing required field: ${field}`);
    }
  }

  // statusの値チェック
  if (data.status && !validStatuses.includes(data.status)) {
    errors.push(`Invalid status: ${data.status}`);
  }

  // versionの形式チェック
  if (data.version && !/\d+\.\d+\.\d+/.test(data.version)) {
    errors.push(`Invalid version format: ${data.version}`);
  }

  return errors;
}
```

#### 5.5.5 Claude CodeによるAIレビュー

pre-commitでAIレビューを追加することもできます。

```
# .husky/pre-commit (追加部分)

# AIレビュー (変更されたファイルのみ)
CHANGED_DOCS=$(git diff --cached --name-only --diff-filter=ACMR | grep '\.md$')

if [ -n "$CHANGED_DOCS" ]; then
  echo "🤖 AIレビューを実行中..."
  claude-code review $CHANGED_DOCS --rules .review-rules.md
fi
```

.review-rules.md にレビュー観点を記述します。

## # 文書レビュールール

### ## 5.6 必須チェック項目

- [ ] Frontmatterが正しく記述されているか
- [ ] 用語が用語集と一致しているか
- [ ] 内部リンクが有効か
- [ ] コードブロックに言語指定があるか

### ## 5.7 推奨チェック項目

- [ ] 文が長すぎないか（100文字目安）
- [ ] 受動態より能動態を使っているか
- [ ] 具体例が含まれているか

## 5.8 章末チェックリスト

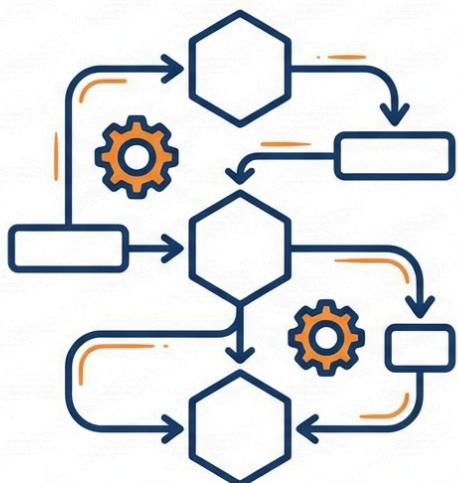
- 7文書すべてにFrontmatterを追加する
- status (draft/review/approved) の運用ルールをチームで合意する
- 変更時の3ステップ（影響度→バージョン→Changelog）を手順化する
- pre-commitフックで最低限の検証（Frontmatter、リンク）を設定する

## 5.9 次章への橋渡し

この章では、7文書を運用するための最低限のルールを学びました。

次章からは第3部「実践ワークフロー」に入ります。7文書をどのように導入し、日々の開発でどのように活用するか、具体的な手順を解説します。

## 第3部 実践：AI仕様駆動開発のワークフロー



## 第3部：実践ワークフロー

Practical Workflow

理論から実践へ。新規プロジェクトでの始め方、既存プロジェクトへの導入、日々の開発フロー、文書追加の判断基準、変更時の影響度評価まで。明日から使える具体的なワークフローを解説します。

## 第6章 導入手順：既存プロジェクト／新規プロジェクト

### 6.1 この章で学ぶこと

- 新規プロジェクトでの7文書構成の始め方
- 既存プロジェクトへの段階的な導入方法
- 「最初から完璧」を捨てる実践的アプローチ

### 6.2 新規プロジェクト：docs構造を作り、最初の7文書を生成する

新規プロジェクトでは、コードを行も書く前に、まず7文書の構造を作ります。

なぜ最初にドキュメントなのか。第1部で見たように、AIは「何を作るか」が曖昧なまま指示を受けると、推測で補完して意図しない実装をします。7文書を先に用意することで、AIに「この仕様に従って実装して」と明確な指示を出せるようになります。

また、プロジェクト開始時が最も構造を整えやすいタイミングです。既存コードがない状態なら、理想的なディレクトリ構成を自由に決められます。後からドキュメントを追加するより、最初から「ドキュメントありき」で始める方が定着しやすいのです。

最初の7文書は完璧である必要はありません。「TODO」や「未定」と書いておき、プロジェクトが進むにつれて埋めていきます。大切なのは構造が最初から存在することです。

#### 6.2.1 ステップ1：ディレクトリ構造の作成

まず、番号付きディレクトリ構造を作成します。番号付けにより、ファイルが増えても整理しやすくなります。

```
# ディレクトリ構造を作成
mkdir -p docs/{00-planning,01-context,02-design,03-implementation,04-quality,05-operations,06-reference,07-project-management,08-knowledge}

# 7文書を作成
touch docs/MASTER.md
touch docs/01-context/PROJECT.md
touch docs/02-design/ARCHITECTURE.md
touch docs/02-design/DOMAIN.md
touch docs/03-implementation/PATTERNS.md
touch docs/04-quality/TESTING.md
touch docs/05-operations/DEPLOYMENT.md
```

作成される構造：

```
docs/
├── MASTER.md                      # 索引・ナビゲーション
├── 00-planning/                   # 企画段階の文書
│   └── PROJECT.md                # ビジョン・要件
├── 01-context/                    # ビジネスロジック
│   └── ARCHITECTURE.md          # システム設計
└── 02-design/                     # 実装パターン
    ├── DOMAIN.md                 # ビジネスロジック
    └── PATTERNS.md              # テスト戦略
├── 03-implementation/            # 実装パターン
    └── TESTING.md               # テスト戦略
└── 04-quality/                  # 実装パターン
    └── DEPLOYMENT.md           # 部署戦略
```

```

├── 05-operations/
│   └── DEPLOYMENT.md          # 運用手順
├── 06-reference/             # API仕様・外部参照
├── 07-project-management/    # 進捗・チケット管理
└── 08-knowledge/             # ナレッジ・ADR

```

### 6.2.2 ステップ1.5：AIエージェント設定ファイルの作成（任意）

多くのAIコーディングツールは、リポジトリルートの設定ファイルを自動で読み込みます。

| ツール                            | 設定ファイル    |
|--------------------------------|-----------|
| Claude Code                    | CLAUDE.md |
| GitHub Copilot, Cursor, Codex等 | AGENTS.md |

7文書構成と連携させる最小限の設定例：

#### # CLAUDE.md(または AGENTS.md)

このリポジトリでは docs/ 配下の7文書に仕様が記載されています。

実装前に必ず docs/MASTER.md を読んでください。

**Note:** 詳細は「付録：AIエージェント設定ファイル一覧」を参照してください。

### 6.2.3 ステップ2：MASTER.mdを最初に書く

MASTER.mdは他の文書へのナビゲーションです。最初に骨格を作ります。

```

---
title: MASTER.md
version: 0.1.0
status: draft
owner: "@your-name"
created: 2026-01-01
updated: 2026-01-01
---
```

#### # プロジェクト名

#### ## 6.3 概要

[1~2文でプロジェクトの目的を記述]

#### ## 6.4 技術スタック

- 言語：[未定]
- フレームワーク：[未定]
- データベース：[未定]

#### ## 6.5 文書索引

|                                                                 |  |  |
|-----------------------------------------------------------------|--|--|
| 文書   状態   説明                                                    |  |  |
| ----- ----- -----                                               |  |  |
| [PROJECT.md](./01-context/PROJECT.md)   draft   ビジョン・要件         |  |  |
| [ARCHITECTURE.md](./02-design/ARCHITECTURE.md)   draft   システム設計 |  |  |
| [DOMAIN.md](./02-design/DOMAIN.md)   draft   ビジネスロジック           |  |  |
| [PATTERNS.md](./03-implementation/PATTERNS.md)   draft   実装パターン |  |  |
| [TESTING.md](./04-quality/TESTING.md)   draft   テスト戦略           |  |  |

| [DEPLOYMENT.md](./05-operations/DEPLOYMENT.md) | draft | 運用手順 |

## ## 6.6 ディレクトリ構造

```text

(プロジェクト構造が決まり次第記述)

### ### 6.6.1 ステップ3：PROJECT.mdでビジョンを固める

技術的な詳細より先に、「何を作るのか」「誰のためか」を明確にします。

```markdown

---

```
title: PROJECT.md
version: 0.1.0
status: draft
owner: "@your-name"
created: 2026-01-01
updated: 2026-01-01
```

---

# PROJECT.md

## ## 6.7 ビジョン

[このプロジェクトが実現したい世界を1~2文で]

## ## 6.8 解決する課題

[ユーザーが抱える具体的な問題]

## ## 6.9 ターゲットユーザー

- ペルソナ1：[具体的な人物像]
- ペルソナ2：[具体的な人物像]

## ## 6.10 主要機能 (MVP)

1. [機能A]：[説明]
2. [機能B]：[説明]
3. [機能C]：[説明]

## ## 6.11 スコープ外 (MVPでは作らないもの)

- [機能X]：[理由]
- [機能Y]：[理由]

### 6.11.1 ステップ4：残りの文書は「最小限」で開始

最初から完璧な文書を作る必要はありません。各文書は見出しだけでも構いません。

---

```
title: ARCHITECTURE.md
version: 0.1.0
status: draft
owner: "@your-name"
created: 2026-01-01
updated: 2026-01-01
```

---

# ARCHITECTURE.md

## ## 6.12 システム構成

(設計が進んだら記述)

## ## 6.13 技術スタック

(決定次第記述)

## ## 6.14 コンポーネント設計

(実装開始時に記述)

大事なのは「場所を確保すること」です。

## 6.15 既存プロジェクト：既存設計を"吸い上げ"て、欠けた文書を補完する

### 6.15.1 ステップ1：現状把握

既存プロジェクトには、散在した情報がすでにあります。まずそれを洗い出します。

#### ## 6.16 既存情報の棚卸し

##### ### 6.16.1 README.md

- [ ] プロジェクト概要 → MASTER.mdに転記
- [ ] セットアップ手順 → DEPLOYMENT.mdに転記

##### ### 6.16.2 既存ドキュメント

- [ ] API仕様書 → ARCHITECTURE.mdに転記
- [ ] ER図 → DOMAIN.mdに転記
- [ ] テスト手順 → TESTING.mdに転記

##### ### 6.16.3 コード内コメント

- [ ] 重要なビジネスロジック → DOMAIN.mdに転記
- [ ] 設計意図のコメント → ARCHITECTURE.mdに転記

##### ### 6.16.4 チャットログ・議事録

- [ ] 技術選定の経緯 → ARCHITECTURE.md (ADR) に転記
- [ ] 要件の合意事項 → PROJECT.mdに転記

### 6.16.5 ステップ2：優先順位をつける

すべてを一度にやろうとしないでください。以下の優先順位で進めます。

| 優先度 | 文書              | 理由             |
|-----|-----------------|----------------|
| 1   | MASTER.md       | 他の文書への入り口      |
| 2   | PATTERNS.md     | 日々の開発で最も参照される  |
| 3   | ARCHITECTURE.md | 新規実装時に必須       |
| 4   | DOMAIN.md       | ビジネスロジックの誤解を防ぐ |
| 5   | TESTING.md      | 品質基準を統一        |
| 6   | PROJECT.md      | 長期的な方向性        |
| 7   | DEPLOYMENT.md   | 運用が安定してから      |

#### 6.16.5.1 なぜこの順番なのか

この優先順位には理由があります。

MASTER.mdを最初に作るのは、これがAIにとって「地図」になるからです。MASTER.mdがあれば、AIは「他にどんな文書があるか」「自分が知らない情報はどこを見ればよいか」を把握できます。たとえ他の文書が空でも、「ARCHITECTURE.mdという文書が存在する」という情報自体に価値があります。

次にPATTERNS.mdを優先するのは、日々の開発で最も頻繁に参照されるからです。「変数名の命名規則は?」「エラーハンドリングはどう書く?」といった質問は、毎日のように発生します。PATTERNS.mdがあれば、AIは一貫したコードを生成できます。

ARCHITECTURE.mdを3番目にしたのは、新機能の実装時に必須だからです。「このAPIはどのレイヤーに置く?」「DBへのアクセスはどこで行う?」といった判断に必要です。

PROJECT.mdを後回しにしているのは意外に思われるかもしれません。しかし、既存プロジェクトでは「何を作るか」はすでにコードとして存在しています。まずは「どう作るか」の知識を整理し、「何のために作ったか」は余裕ができるから遡って記録すれば十分です。

#### 6.16.6 ステップ3：AIに「吸い上げ」を手伝わせる

既存コードから情報を抽出するのは、AIが得意な作業です。

##### ## 6.17 プロンプト例:コードから設計意図を抽出

以下のコードを分析して、ARCHITECTURE.mdに記載すべき設計方針を抽出してください。

[コードを貼り付け]

以下の観点で整理してください：

- レイヤー構成
- 依存関係の方向
- エラーハンドリングのパターン
- 使用しているデザインパターン

##### ## 6.18 プロンプト例:テストからテスト戦略を抽出

以下のテストコードを分析して、TESTING.mdに記載すべきテスト方針を抽出してください。

[テストコードを貼り付け]

以下の観点で整理してください：

- テストの種類 (Unit/Integration/E2E)
- モック戦略
- テストデータの管理方法
- アサーションのパターン

#### 6.18.0.1 抽出がうまくいかないケース

AIによる抽出がうまくいかないケースもあります。代表的なパターンと対策を紹介します。

##### コードにコメントがほとんどない場合

コメントがないコードからは、「なぜそうしたか」の情報が抽出できません。この場合、gitログを活用します。

##### ## 6.19 プロンプト例:gitログから設計意図を推測

以下のコミット履歴を分析して、設計の変遷と意図を推測してください。

[git log --oneline --since="6 months ago" の出力を貼り付け]

### 設計意図が口頭伝承になっている場合

長く運用されているプロジェクトでは、重要な決定が「暗黙知」になっていることがあります。この場合、関係者へのヒアリングが必要です。ただし、いきなり「設計思想を教えてください」と聞いても答えにくいものです。

効果的なのは、AIが生成した「たたき台」を見せて「ここ違いますか？」と確認する方法です。人は白紙から説明するより、間違いを指摘するほうが簡単です。

#### ## 6.20 ヒアリング用たたき台の作成

以下のコードを分析して、「おそらくこういう設計意図だろう」という仮説を5つ挙げてください。

仮説は「～のために、～という設計にしている」という形式で書いてください。

[コードを貼り付け]

このたたき台を関係者に見せ、「合っている/違う」を確認することで、暗黙知を効率的に文書化できます。

### 6.20.1 ステップ4：段階的に育てる

最初は「現状を記録した」だけの文書でOKです。

その後、以下のタイミングで文書を育てていきます：

- 新機能追加時：PROJECT.mdに機能を追記、ARCHITECTURE.mdに設計を追記
- バグ修正時：DOMAIN.mdにルールを明確化、PATTERNS.mdにアンチパターンを追記
- レビュー指摘時：PATTERNS.mdにパターンを蓄積
- 障害発生時：DEPLOYMENT.mdにRunbookを追加

## 6.21 「最初から完璧」を捨てる

### 6.21.1 パレートの法則を適用する

文書の80%の価値は、20%の労力で生み出せます。

| 完成度  | 得られる価値         | 必要な労力 |
|------|----------------|-------|
| 20%  | 骨格だけでもAIは参照できる | 1時間   |
| 50%  | 日常の開発で十分使える    | 3時間   |
| 80%  | ほぼ完璧、例外対応も記載   | 1日    |
| 100% | 完璧（ただし陳腐化が早い）  | 1週間   |

狙うべきは50%の完成度です。

### 6.21.2 ドラフト→レビュー→更新の反復

完璧を目指すより、反復で育てるほうが効率的です。

ドラフト（30分）

↓

実際に使ってみる

↓

「これが足りない」と気づく

↓

更新（15分）  
↓  
また使ってみる  
↓  
...

この反復を3回繰り返せば、使い物になる文書が完成します。

### 6.21.3 完璧主義を手放すコツ

「最初から完璧を捨てる」と言っても、エンジニアにとっては心理的に難しいものです。コードには厳密さを求めるのに、ドキュメントだけ「雑でいい」と言われても違和感があります。

ここで大切なのは、**ドキュメントの目的を再認識すること**です。ドキュメントの目的は「完璧な仕様書を作ること」ではなく、「AIと人間が同じ理解を持つこと」です。AIが正しく動作するために必要な最低限の情報があれば、それで十分なのです。

また、「不完全なドキュメント」と「ドキュメントがない」では、天と地ほどの差があります。50%の完成度でも、AIの出力精度は劇的に改善します。0%から50%への改善効果は、50%から100%への改善効果よりもはるかに大きいのです。

もう一つのコツは、「書く」ではなく「記録する」と考えることです。完璧な文章を書こうとするから手が止まります。「今わかっていることを箇条書きで記録しておく」と考えれば、ハードルは下がります。箇条書きでも、AIは十分に理解できます。

### 6.21.4 「最小限で始める」テンプレート

各文書の最小限バージョンを示します。

#### # ARCHITECTURE.md(最小限版)

##### ## 6.22 技術スタック

- Backend: Node.js + Express
- Database: PostgreSQL
- Cache: Redis

##### ## 6.23 レイヤー構成

API → Service → Repository → Database

##### ## 6.24 最重要ルール

- Service層にビジネスロジックを集約
- Repositoryは純粋なデータアクセスのみ
- 直接SQLは書かない（ORMを使用）

これだけあれば、AIは「Serviceにロジックを書く」「Repositoryは薄く保つ」という判断ができます。

## 6.25 章末チェックリスト

### 6.25.1 新規プロジェクトの場合

- 番号付きディレクトリ構造（00～08）と7つの文書を作成した
- MASTER.mdに概要と文書索引を書いた
- PROJECT.mdにビジョンとMVP機能を書いた
- 他の文書は見出しだけでも作成した

### 6.25.2 既存プロジェクトの場合

- 既存情報の棚卸しを行った
- 優先順位に基づいて、まずMASTER.mdを作成した
- 次にPATTERNS.md（またはARCHITECTURE.md）を作成した

- 残りは「今後育てる」と割り切った
- 

## 6.26 次章への橋渡し

この章では、7文書構成の導入手順を学びました。

次章では、日々の開発フロー——Issueを作り、AIにタスクを渡し、PRを出すまでの流れを解説します。冒頭で紹介した「3つの戦略」を、具体的なワークフローに落とし込みます。

# 第7章 日々の開発フロー：AIに"タスク"を渡す前にやること

## 7.1 この章で学ぶこと

- Issueベースのスコープ絞り込み
  - Issue = AIへのコンテキスト選択 という考え方
  - AIに渡す前の3つの確認ポイント
  - 70%完成度でのPR運用の具体的な流れ
- 

## 7.2 なぜIssueベースの開発が重要なのか

AI仕様駆動開発では、Issueを起点にすべてが始まります。これは単なるタスク管理ではなく、AIとの協働を最適化するための設計です。

### 7.2.1 スコープの明確化

Issueに記載することで、作業範囲が自然と小さくなります。「ログイン機能を作る」という曖昧な指示ではなく、「パスワードリセット機能のメール送信部分を実装する」という具体的なスコープに落とし込めます。AIは限定されたスコープの方が精度高く作業できます。

### 7.2.2 コンテキストの集約

Issueには、参照すべきドキュメント、関連するコード、受け入れ基準がすべて記載されています。AIはIssueを読むだけで「何を作るか」「どの仕様に従うか」「どうなれば完成か」を把握できます。

### 7.2.3 マージ可否の判断基準

Issueに書かれた受け入れ基準は、そのままマージの条件になります。AIはこの基準を見て「この実装で受け入れ基準を満たしているか」を自己判断できます。レビュー時も、レビューは受け入れ基準に照らして確認するだけで済みます。

### 7.2.4 履歴の追跡性

AIは、Issue・ブランチ・PRの連携を追跡できます。「以前のIssue #42で何を決めたか」「関連するPR #58でどんな実装をしたか」を参照することで、プロジェクトの文脈を理解した上で作業できます。過去の意思決定を踏まえた一貫性のある実装が可能になります。

---

## 7.3 Issueベースの開発フロー全体像

# AI Development Flow

## 開発フロー



### 7.3.1 フローの概要

1. Issueを作成（スコープを絞る）  
↓
2. 3つの確認（仕様・設計・テスト）  
↓
3. AIにタスクを渡す  
↓
4. 70%完成度でPRを作成  
↓
5. レビュー→修正→マージ  
↓
6. 指摘をナレッジ化

このフローを回すことでのAIの出力品質は継続的に向上します。

## 7.4 ステップ1：Issueを作成する

### 7.4.1 なぜIssueから始めるのか

「ログイン機能を作って」とAIに直接指示するのではなく、まずIssueを作る。

これには3つの理由があります。

1. スコープが明確になる：何を作る/作らないが記録される
2. 追跡可能になる：後から「なぜこう実装したか」がわかる
3. AIへの入力になる：Issueそのものがコンテキストになる

### 7.4.2 良いIssueの構造

## ## 7.5 Issue #42: [機能名の動詞形]

### ### 7.5.1 背景

[なぜこの機能が必要か、1~2文で]

### ### 7.5.2 受け入れ基準

- [ ] [具体的な動作1]
- [ ] [具体的な動作2]
- [ ] [具体的な動作3]

### ### 7.5.3 技術的制約

- [使用すべきライブラリ/パターン]
- [既存コードとの接続点]
- [パフォーマンス要件]

### ### 7.5.4 スコープ外

- [明示的に今回やらないこと]

### ### 7.5.5 関連文書

- [ARCHITECTURE.md#認証](../docs/ARCHITECTURE.md#認証)
- [DOMAIN.md#ユーザー](../docs/DOMAIN.md#ユーザー)

## 7.5.6 具体例：ログイン機能

## ## 7.6 Issue #42: ログインAPIエンドポイントの実装

### ### 7.6.1 背景

ユーザーがメールアドレスとパスワードでログインできるようにする。  
セキュリティ要件としてブルートフォース対策を含む。

### ### 7.6.2 受け入れ基準

- [ ] POST /api/v1/auth/login でログインできる
- [ ] 成功時はJWTトークン（アクセス/リフレッシュ）を返す
- [ ] パスワード誤り5回でアカウントを15分ロック
- [ ] ログイン試行を監査ログに記録

### ### 7.6.3 技術的制約

- JWT署名はRS256（既存の鍵ペアを使用）
- リフレッシュトークンはRedisに保存
- パスワード検証はArgon2id
- レート制限はRedisのスライディングウィンドウ

### ### 7.6.4 スコープ外

- パスワードリセット (Issue #43)
- OAuth/ソーシャルログイン (Issue #44)
- MFA (Issue #45)

### ### 7.6.5 関連文書

- [ARCHITECTURE.md#認証フロー](../docs/ARCHITECTURE.md#認証フロー)
- [DOMAIN.md#認証ルール](../docs/DOMAIN.md#認証ルール)

## 7.6.6 悪いIssueの例と改善

良いIssueの例を見ましたが、「悪いIssue」を知ることも重要です。実際にAIの出力が乱れた例を紹介します。

### 7.6.6.1 受け入れ基準が曖昧なIssue

#### ## 7.7 Issue #99: ユーザー管理機能の改善

##### ### 7.7.1 背景

ユーザー管理画面が使いにくいので改善したい。

##### ### 7.7.2 受け入れ基準

- [ ] UIを改善する
- [ ] パフォーマンスを向上させる
- [ ] エラーハンドリングを追加する

このIssueでは、AIは「UIを改善する」の意味を推測するしかありません。結果、デザイナーの意図とは違うレイアウト変更をしたり、不要な機能を追加したりします。「テーブルのソート機能を追加する」「ローディング表示を0.5秒以内にする」のように、**検証可能な基準**を書く必要があります。

### 7.7.2.1 技術的制約が欠落したIssue

#### ## 7.8 Issue #100: メール送信機能を追加

##### ### 7.8.1 受け入れ基準

- [ ] ユーザー登録時にメールを送信する

技術的制約がないため、AIは自由にメール送信ライブラリを選びます。プロジェクトで使っているSendGridではなく、別のサービスを使った実装が出てくるかもしれません。また、既存のメール送信基盤があることを知らずに、一から実装することもあります。

### 7.8.1.1 スコープ外が不明確なIssue

#### ## 7.9 Issue #101: 検索機能を実装

##### ### 7.9.1 受け入れ基準

- [ ] 商品を検索できる

スコープ外が書かれていないと、AIは「全文検索も必要?」「ファセット検索は?」「オートコンプリートは?」と拡大解釈しがちです。本来はシンプルなキーワード検索だけでよかったですのに、Elasticsearchを使った本格的な検索基盤を提案されることがあります。

## 7.10 Issue = AIへのコンテキスト選択

### 7.10.1 全7文書を毎回読む必要はない

ここで重要な概念を紹介します。

Issue作成とは、「AIに何を読ませるか」を選択する行為です。

プロジェクトには7文書（MASTER、PROJECT、ARCHITECTURE、DOMAIN、PATTERNS、TESTING、DEPLOYMENT）があり、さらに開発が進むとADR、ナレッジ、詳細仕様など、文書は増えていきます。

しかし、すべてのタスクで全文書を読む必要はありません。

- ログイン機能の実装 → ARCHITECTURE（認証フロー）、DOMAIN（認証ルール）を読めばよい
- CSSの修正 → PATTERNSのスタイリング規約だけで十分
- デプロイ設定の変更 → DEPLOYMENTのみ

Issueの「関連文書」セクションは、AIに「このタスクではこれだけ読めばよい」と指示する場所なのです。

## 7.10.2 タスク種別ごとの推奨参照文書

以下の表を参考に、タスク種別に応じた文書を選択してください。

| タスク種別    | 必須参照                         | 推奨参照              | 通常不要       |
|----------|------------------------------|-------------------|------------|
| 新機能追加    | MASTER, ARCHITECTURE, DOMAIN | PATTERNS, TESTING | DEPLOYMENT |
| バグ修正     | 関連Issue, PATTERNS            | TESTING           | DOMAIN全体   |
| リファクタリング | ARCHITECTURE, PATTERNS       | TESTING           | DOMAIN     |
| インフラ変更   | MASTER, DEPLOYMENT           | ARCHITECTURE      | DOMAIN     |
| ドキュメント更新 | MASTER                       | 対象文書              | その他        |

ポイント：

- 「必須参照」はIssueに必ずリンクを記載
- 「通常不要」は明示的に「スコープ外」に記載することで、AIの無駄な参照を防ぐ

## 7.10.3 関連Issueの参照

過去の類似Issueも重要なコンテキストです。

### ### 7.10.4 関連Issue

- #123 ログイン機能の基本実装（参考：認証フローの詳細）
- #145 認証エラーハンドリング（参考：エラーレスポンス形式）

関連Issueを参照することで：

- 過去の設計判断を引き継げる
- 同じ失敗を繰り返さない
- 一貫した実装が得られる

## 7.10.5 スコープ外の明示

「読まなくてよい文書」を明示することも重要です。

### ### 7.10.6 スコープ外(今回は参考不要)

- DEPLOYMENT.md（インフラ変更なし）
- GLOSSARY.md（新用語の追加なし）
- PROJECT.md（ビジネス要件に変更なし）

これにより、AIは必要なコンテキストだけに集中できます。

## 7.10.7 「読まなくてよい」の判断基準

「スコープ外」に何を書くべきか、迷うことがあります。以下の判断基準を参考にしてください。

「このタスクの実装中に、その文書を参照する場面があるか？」を問いかけます。

- DEPLOYMENT.md：「このコードはどうデプロイされるか」を意識する必要があるか？→ 通常のバックエンド実装なら不要
- PROJECT.md：「なぜこの機能が必要か」を確認する必要があるか？→ 要件が明確なら不要
- DOMAIN.md：「このビジネスルールに従うべきか」を確認する必要があるか？→ 技術的なリファクタリングなら不要

逆に、「必要かもしれない」と思ったらスコープ外に入れないことが重要です。AIが必要な情報を見つけられないほうが、余計な情報を読むよりも問題が大きいからです。

注意点として、過剰にコンテキストを絞りすぎると、AIが既存のパターンを無視した実装をすることがあります。迷ったら「PATTERNS.mdは常に参照」としておくと、一貫性のある実装が得られます。

## 7.11 ステップ2：3つの確認ポイント

Issueを作成したら、AIに渡す前に3つの確認を行います。

### 7.11.1 確認1：仕様の粒度

受け入れ基準が具体的に書けているか？

**✗** 暗昧な受け入れ基準

- [ ] ログインできる
- [ ] エラーハンドリングする

**✓** 具体的な受け入れ基準

- [ ] POST /api/v1/auth/login でログインできる
- [ ] メール//パスワードが正しければ200とトークンを返す
- [ ] メールが存在しなければ401とエラーメッセージを返す
- [ ] パスワードが間違っていれば401とエラーメッセージを返す

確認の問い合わせ：「これを読んだ人が、テストケースを書けるか？」

### 7.11.2 確認2：設計の粒度

アーキテクチャ制約が明示されているか？

**✗** 制約が不明確

- 認証を実装する

**✓** 制約が明確

- JWT署名はRS256
- 既存のauthServiceを拡張
- エラーレスポンスはsrc/errors/auth.tsの形式

確認の問い合わせ：「どこに・どう実装すべきか、迷う余地がないか？」

### 7.11.3 確認3：テストの粒度

テストが仕様を「代替」していないか？

これは重要なポイントです。

**✗** テストが仕様の代替になっている

「テストコード見ればわかるでしょ」

→ テストを読まないと要件がわからない

**✓** 仕様が先、テストは検証

1. 仕様 (Issue) : 「パスワード誤り5回でロック」
2. テスト : その仕様をコードで検証

確認の問い合わせ：「仕様を読まずにテストだけ見て、正しさを判断できるか？」

→ できてしまうなら、仕様が足りない。

#### 7.11.3.1 「仕様が先」を実践する難しさ

「仕様が先、テストは検証」という考え方には、TDD（テスト駆動開発）に慣れた人には違和感があるかもしれません。TDDでは「テストを先に書く」ことで要件を明確にするアプローチを取るからです。

しかし、AI仕様駆動開発で「仕様が先」と言っているのは、**人間が書くべき仕様の話**です。テストコードではなく、自然言語で書かれた仕様（Issue、受け入れ基準、DOMAIN.mdのルールなど）が先に存在すべきという意味です。

実際のワークフローでは、こう考えてください。

1. **人間が仕様を書く**：「パスワード誤り5回でロック」（Issueの受け入れ基準）
2. **AIがテストを生成**：仕様に基づいたテストコード
3. **AIが実装を生成**：テストを通す実装コード

この順序であれば、テストは「仕様の検証」として機能します。一方、テストコードだけを見て「これが仕様だ」と言わっても、AIは文脈がわかりません。「なぜ5回なのか」「15分ロックの根拠は？」といった背景情報はテストコードには書かれていないからです。

TDDの「テストファースト」とAI仕様駆動開発の「仕様ファースト」は矛盾しません。人間が仕様を書き、AIがテストと実装を生成する。この分担がポイントです。

## 7.12 ステップ3：AIにタスクを渡す

### 7.12.1 プロンプトの構造

#### ## 7.13 タスク

Issue #42 を実装してください。

#### ## 7.14 コンテキスト

- Issue: [IssueのURL or 内容をコピー]

- 関連文書:

- docs/ARCHITECTURE.md
- docs/DOMAIN.md
- docs/PATTERNS.md

#### ## 7.15 制約

- 既存の認証サービス（src/services/auth/）を拡張する形で実装
- 新規ファイルは src/services/auth/login.ts に作成
- テストは src/services/auth/\_\_tests\_\_/login.test.ts に作成

#### ## 7.16 出力

1. 実装コード
2. テストコード
3. 影響を受ける既存ファイルの変更点

### 7.16.1 AIコーディングツールを使う場合

Claude Code、GitHub Copilot、Cursorなど、いずれのツールでもIssueの内容をそのまま渡せます。

```
# Claude Codeの例
claude "Issue #42 を実装して。docs/配下の仕様に従って。"

# GitHub Copilot（チャット）の例
# @workspace Issue #42 を実装して。docs/配下の仕様に従って。
```

7文書がリポジトリにあれば、AIツールは自動的に参照します。

## 7.17 ステップ4：70%完成度でPRを作成

### 7.17.1 完璧を求めない

AIが生成したコードを見て、「ここが微妙だな」と思うことがあるでしょう。

そこで止まらないでください。

70%の完成度でPRを出す理由：

1. レビューで具体的な指摘を得られる
2. 指摘内容がAIへの次の入力になる
3. 完璧を目指す時間より、反復の方が速い

### 7.17.2 PRの書き方

#### ## 7.18 PR #42: ログインAPIエンドポイントの実装

##### ### 7.18.1 概要

Issue #42 の実装です。

##### ### 7.18.2 変更内容

- `src/services/auth/login.ts`: ログインサービスを追加
- `src/api/routes/auth.ts`: ログインエンドポイントを追加
- `src/services/auth/\_\_tests\_\_/login.test.ts`: テストを追加

##### ### 7.18.3 確認ポイント

- [ ] JWT署名がRS256になっているか
- [ ] ブルートフォース対策が正しく動作するか
- [ ] エラーレスポンスの形式が統一されているか

##### ### 7.18.4 今後の対応(レビュー後)

- エッジケースの追加テスト
- パフォーマンス最適化 (必要に応じて)

##### ### 7.18.5 関連

- Closes #42

## 7.19 ステップ5：レビュー→修正→マージ

### 7.19.1 レビュー指摘をAIに渡す

レビューコメントは、AIへの次の入力として最適です。

#### ## 7.20 レビュー指摘

「validateCredentials関数、バリデーションとDB問い合わせが混ざってる。分離して。」

#### ## 7.21 AIへの指示

上記のレビュー指摘に対応してください。

validateCredentials関数を以下のように分割してください：

1. validateLoginInput: 入力値のバリデーション
2. findUserByEmail: ユーザー検索
3. verifyPassword: パスワード検証

それぞれ単一責任になるよう実装してください。

レビュー指摘には具体的な問題と期待される解決方向が含まれているため、AIは的確に修正できます。

### 7.21.1 反復の効果

1回目のPR → レビュー指摘5件

↓ AIで修正

2回目 → レビュー指摘2件

↓ AIで修正

3回目 → LGTM

反復するたびに、スコープがさらに狭まるため、AIの出力精度は上がります。

## 7.22 ステップ6：指摘をナレッジ化

### 7.22.1 繰り返される指摘を記録する

レビューで同じ指摘が何度も出る場合、それはPATTERNS.mdに蓄積すべき知識です。

#### ## 7.23 よくある指摘と対応

##### ### 7.23.1 関数の責務分離

✗ 1つの関数で複数のことをする

✓ 単一責任の関数に分割する

例：

- validateAndSave → validate + save
- fetchAndTransform → fetch + transform

##### ### 7.23.2 エラーレスポンスの統一

✗ throw new Error('message')

✓ return err(new SpecificError(details))

##### ### 7.23.3 テストの粒度

✗ 1つのテストで複数のケースをテスト

✓ 1テスト1ケース (Arrange-Act-Assert)

### 7.23.4 自動チェックへの昇格

何度も指摘されるパターンは、自動チェックに組み込むことを検討します。

- ESLintカスタムルール
- pre-commitフック
- サブエージェントによるレビュー

これにより、「指摘される前に防ぐ」サイクルが回り始めます。

## 7.24 章末チェックリスト

- 次のタスクをIssueとして作成する
- Issueに「受け入れ基準」「技術的制約」「スコープ外」を含める
- タスク種別に応じた必須参照文書を選択する
- 関連Issueをリンクする
- 読む必要のない文書を「スコープ外」に明記する
- AIに渡す前に3つの確認（仕様・設計・テスト）を行う
- 70%完成度でPRを出す心構えを持つ
- レビュー指摘はAIへの入力として活用する

- 繰り返される指摘はPATTERNS.mdに記録する
- 

## 7.25 次章への橋渡し

この章では、日々の開発フローを学びました。

次章では、文書追加の意思決定——「この情報はどの文書に置くべきか」という判断基準を解説します。

# 第8章 文書追加の意思決定 (Decision Matrix)

## 8.1 この章で学ぶこと

- 新しい情報をどの文書に置くべきかの判断基準
  - 具体的なケーススタディ (DB設計、認証、権限など)
  - MASTER.mdの索引更新を習慣化する方法
- 

## 8.2 「これはどこに書く？」の判断基準

### 8.2.1 問題：情報が散在する

7文書構成を導入しても、こんな迷いが生じます。

- 「認証の仕様って、ARCHITECTURE ? DOMAIN？」
- 「このAPIの詳細はどこに書く？」
- 「監査ログの要件は？」

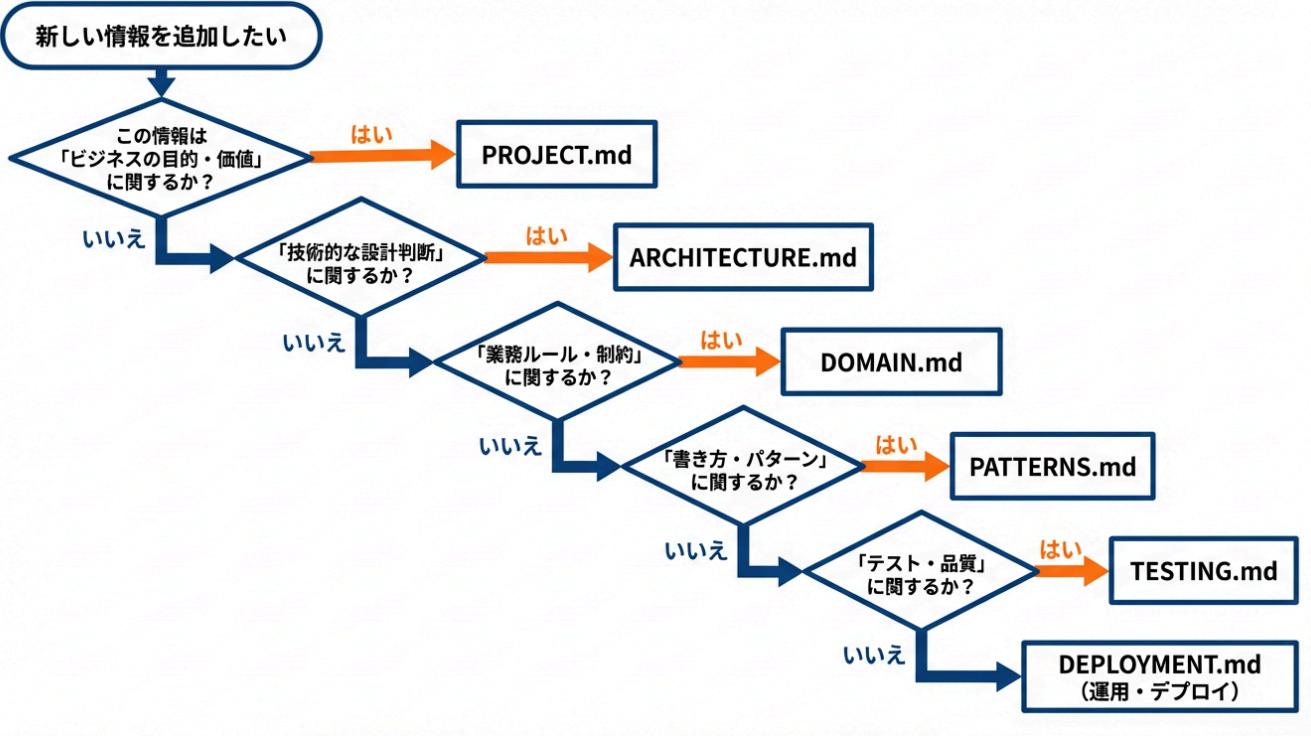
判断基準がないと、人によって置き場所が変わり、結局情報が散在します。

### 8.2.2 Decision Matrix

以下のマトリクスで判断します。

| 情報の種類   | 置き場所            | 判断基準          |
|---------|-----------------|---------------|
| なぜ作るか   | PROJECT.md      | ビジネス目標、ユーザー価値 |
| 何を作るか   | PROJECT.md      | 機能一覧、要件定義     |
| どう作るか   | ARCHITECTURE.md | 技術選定、システム構成   |
| ルールは何か  | DOMAIN.md       | ビジネスロジック、制約   |
| どう書くか   | PATTERNS.md     | コーディング規約、パターン |
| 何が正しいか  | TESTING.md      | テスト戦略、品質基準    |
| どう運用するか | DEPLOYMENT.md   | デプロイ、監視、障害対応  |

### 8.2.3 判断フローチャート



#### 8.2.4 フローチャートで決まらないとき

このフローチャートは便利ですが、「どちらにも当てはまる」「どれにも当てはまらない」と感じるケースがあります。そのときの対処法を知っておきましょう。

##### 8.2.4.1 どちらにも当てはまる場合

「キャッシング戦略」を例に考えます。「Redisを使う」は技術判断（ARCHITECTURE）ですが、「キャッシングの有効期限を30分にする」はビジネス要件にも関係します。このような場合は、**情報の性質**で分割します。技術的な「どう実現するか」はARCHITECTURE、ビジネス的な「なぜその値か」はDOMAINに書きます。分割することで、技術変更とビジネス変更を独立して管理できます。

##### 8.2.4.2 どれにも当てはまらない場合

「外部APIのエンドポイント一覧」のような情報は、フローチャートを辿っても行き場がありません。このような運用に近い参照情報は、DEPLOYMENT.mdに「外部連携」セクションを設けるか、専用の参考ドキュメント（REFERENCES.md）を追加することを検討します。ただし、文書を増やすとAIの参照コストが上がるため、まずは既存の7文書に収める努力をしましょう。

#### 8.2.5 判断に迷いややすい典型例

実務で迷いややすい例をいくつか紹介します。

| 情報         | 迷うポイント                         | 推奨配置       | 理由                                    |
|------------|--------------------------------|------------|---------------------------------------|
| バリデーションルール | DOMAIN ? PATTERNS ?            | DOMAIN     | 「メールは一意」は業務ルール。「入力値チェックの書き方」はPATTERNS |
| エラーコード一覧   | DOMAIN ? PATTERNS ?            | DOMAIN     | エラーの種類と意味は業務に根ざす。コードの定数化方法はPATTERNS   |
| 環境変数一覧     | ARCHITECTURE ?<br>DEPLOYMENT ? | DEPLOYMENT | 本番・ステージング等の環境差異は運用の関心事                |
| ページネーション   | ARCHITECTURE ?<br>PATTERNS ?   | PATTERNS   | 実装のHow-Toに近い。ただし、上限値の根拠はDOMAIN        |

迷ったときの最終判断基準は、「この情報が変わったとき、誰が気にするか」です。ビジネス側が気にするならPROJECT/DOMAIN、開発者が気にするならARCHITECTURE/PATTERNS、運用担当が気にするならDEPLOYMENTです。

## 8.3 ケーススタディ

### 8.3.1 ケース1：DB設計

「ユーザー一テーブルのスキーマをどこに書く？」

判断：情報の性質によって分割します。

#### ## 8.4 ARCHITECTURE.md に書くこと

- 使用するDB (PostgreSQL)
- テーブル間のリレーション概要
- インデックス戦略
- マイグレーション方針

#### ## 8.5 DOMAIN.md に書くこと

- User エンティティの属性と制約
- 状態遷移 (active → suspended → deleted)
- ビジネスルール (メールは一意、など)

理由：

- 「PostgreSQLを使う」は技術的設計判断 → ARCHITECTURE
- 「メールは一意」はビジネスルール → DOMAIN

### 8.5.1 ケース2：認証

「JWT認証の仕様をどこに書く？」

判断：

#### ## 8.6 ARCHITECTURE.md に書くこと

- 認証方式 (JWT)
- トークン署名アルゴリズム (RS256)
- トークン保存場所 (Cookie/LocalStorage)
- リフレッシュトークンの保存先 (Redis)

#### ## 8.7 DOMAIN.md に書くこと

- ログイン可能な条件 (status=activeのみ)
- パスワードポリシー (12文字以上、等)
- ロックアウトルール (5回失敗で15分ロック)

#### ## 8.8 PATTERNS.md に書くこと

- 認証ミドルウェアの実装パターン
- トークン検証のコードパターン

#### ## 8.9 TESTING.md に書くこと

- 認証関連のテストケース一覧
- モック戦略 (トークン生成のモック方法)

### 8.9.1 ケース3：権限管理

「ロールベースアクセス制御をどこに書く？」

判断：

## **## 8.10 PROJECT.md に書くこと**

- なぜ権限管理が必要か（ビジネス要件）
- 想定されるロール一覧（admin, member, viewer）

## **## 8.11 DOMAIN.md に書くこと**

- 各ロールの権限定義
- リソースとアクションのマトリクス
- 権限チェックのビジネスルール

## **## 8.12 ARCHITECTURE.md に書くこと**

- 権限チェックの実装場所（ミドルウェア？ サービス？）
- 権限データの保存方法

## **## 8.13 PATTERNS.md に書くこと**

- 権限チェックのコードパターン
- デコレータ/ガード関数の使い方

### **8.13.1 ケース4：監査ログ**

「監査ログの要件をどこに書く？」

判断：

## **## 8.14 PROJECT.md に書くこと**

- なぜ監査ログが必要か（コンプライアンス要件）
- 保存期間の要件

## **## 8.15 DOMAIN.md に書くこと**

- 記録すべきイベントの種類
- 各イベントの記録項目
- 個人情報のマスキングルール

## **## 8.16 ARCHITECTURE.md に書くこと**

- ログの保存先（CloudWatch, Elasticsearch等）
- ログの転送方式
- ログの構造化フォーマット

## **## 8.17 DEPLOYMENT.md に書くこと**

- ログの検索・閲覧方法
- アラート設定
- ログ保管・削除の運用手順

### **8.17.1 ケース5：SLO (Service Level Objective)**

「レスポンスタイム目標をどこに書く？」

判断：

## **## 8.18 PROJECT.md に書くこと**

- ユーザー体験としての要件（「3秒以内に応答」）
- ビジネスインパクト（遅延による離脱率）

## **## 8.19 TESTING.md に書くこと**

- パフォーマンステストの閾値
- 負荷テストのシナリオ

## ## 8.20 DEPLOYMENT.md に書くこと

- 具体的なSLO数値 (p99 < 500ms)
- 監視設定とアラート閾値
- SLO違反時の対応手順

### 8.20.1 情報が複数文書にまたがるときの管理

ケーススタディを見てお気づきかもしれません、「認証」「権限」「監査ログ」といった横断的な機能は、複数の文書に情報が分散します。これをどう管理するかが実務上の課題です。

#### 8.20.1.1 代表文書を決める

まず、その機能の「代表文書」を決めます。認証であればARCHITECTURE.md、権限であればDOMAIN.mdが代表になることが多いでしょう。代表文書に「関連する記述は〇〇もあります」という相互参照を入れておきます。

## ## 8.21 認証(Authentication)

認証方式はJWTを採用します。

### \*\*関連情報\*\* :

- ログイン可能条件・パスワードポリシー → DOMAIN.md#認証ルール
- 認証ミドルウェアの実装パターン → PATTERNS.md#認証
- テストのモック戦略 → TESTING.md#認証テスト

### 8.21.0.1 MASTER.mdにクロスリファレンスを集約する

複数の文書にまたがる情報は、MASTER.mdの索引セクションにクロスリファレンスとしてまとめておくと、AIが全体像を把握しやすくなります。

## ## 8.22 横断的機能の参照先

|      |                           |              |        |          |         |            |  |
|------|---------------------------|--------------|--------|----------|---------|------------|--|
| 機能   | PROJECT                   | ARCHITECTURE | DOMAIN | PATTERNS | TESTING | DEPLOYMENT |  |
| ---  | ---                       | ---          | ---    | ---      | ---     | ---        |  |
| 認証   | -   ○   ○   ○   ○   -     |              |        |          |         |            |  |
| 権限   | ○   ○   ○   ○   ○   -   - |              |        |          |         |            |  |
| 監査ログ | ○   ○   ○   ○   -   -   ○ |              |        |          |         |            |  |

◎: 代表文書 ○: 関連情報あり

このマトリクスがあれば、「認証について知りたい」というときに、AIはARCHITECTURE.mdを中心に、関連するDOMAIN.md、PATTERNS.md、TESTING.mdも参照すべきだと判断できます。

## 8.23 MASTER.mdの索引更新を必須タスクにする

### 8.23.1 なぜ重要なか

7文書に情報を追加しても、MASTER.mdの索引が古いままだと、AIは新しい情報を見つけられません。

### 8.23.2 ルール：文書変更時のチェックリスト

PRを出す前に、以下を確認します。

## ## 8.24 文書変更時チェックリスト

- [ ] 追加した情報は適切な文書に配置されているか
- [ ] MASTER.md の文書索引が最新か

- [ ] MASTER.md のディレクトリ構造が最新か
- [ ] 関連する他の文書に矛盾がないか

### 8.24.1 自動化の例

CIで「MASTER.mdが更新されているか」をチェックします。

```
# .github/workflows/docs-check.yml
name: Docs Check

on:
  pull_request:
    paths:
      - 'docs/**'

jobs:
  check-master-updated:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0
      - name: Check MASTER.md updated
        run: |
          # docs/配下が変更されている場合、MASTER.mdも変更されているか確認
          DOCS_CHANGED=$(git diff --name-only origin/${{ github.base_ref }} | grep '^docs/' | wc -l)
          MASTER_CHANGED=$(git diff --name-only origin/${{ github.base_ref }} | grep '^docs/MASTER.md' | wc -l)

          if [ "$DOCS_CHANGED" -gt 0 ] && [ "$MASTER_CHANGED" -eq 0 ]; then
            echo "::error::docs/ was changed but MASTER.md was not updated"
            echo "Please update MASTER.md to reflect the changes"
            exit 1
          fi
```

## 8.25 AIに判断を任せる

### 8.25.1 プロンプト例

新しい情報をどこに置くべきか、AIに判断を委ねることもできます。

#### ## 8.26 質問

以下の情報を7文書構成のどこに配置すべきか判断してください。

#### ### 8.26.1 追加したい情報

[情報の内容]

#### ### 8.26.2 判断基準

- PROJECT.md: ビジネス目標、ユーザー価値、要件
- ARCHITECTURE.md: 技術選定、システム構成、設計判断
- DOMAIN.md: ビジネスロジック、ルール、エンティティ
- PATTERNS.md: コーディング規約、実装パターン
- TESTING.md: テスト戦略、品質基準
- DEPLOYMENT.md: デプロイ、監視、運用

#### ### 8.26.3 回答形式

1. 配置先の文書
2. 配置先のセクション（見出し）
3. 判断理由
4. 他の文書に関連情報を追加すべきか

#### 8.26.4 AIの判断例

##### ## 8.27 回答

###### ### 8.27.1 追加情報:「APIレート制限を1分あたり100リクエストに設定」

1. \*\*配置先の文書\*\*: ARCHITECTURE.md
2. \*\*配置先のセクション\*\*: ## API設計 > ### レート制限
3. \*\*判断理由\*\*: レート制限の数値は技術的な設計判断であり、ビジネスルール (DOMAIN) ではないため
4. \*\*関連情報\*\*:
  - TESTING.md: レート制限のテストケースを追加
  - DEPLOYMENT.md: レート制限のモニタリング設定を追加
  - DOMAIN.md: レート制限に達した場合のユーザー向けメッセージを追加（該当する場合）

#### 8.28 章末チェックリスト

- Decision Matrixを印刷またはブックマークする
- 最近追加した情報が適切な文書に配置されているか確認する
- MASTER.mdの索引が最新か確認する
- 「文書変更時チェックリスト」をPRテンプレートに追加する

#### 8.29 次章への橋渡し

この章では、情報をどの文書に配置すべきかの判断基準を学びました。

次章では、**変更に強い運用**—影響度評価 (changeImpact) を使って手戻りを最小化する方法を解説します。

## 第9章 変更に強い運用：影響度評価で手戻りを消す

### 9.1 この章で学ぶこと

- 変更の種類と影響度の分類
- HIGH変更時の必須チェックリスト
- 仕様変更→AI再実装を安全に繰り返す設計

### 9.2 なぜ影響度評価が必要か

#### 9.2.1 変更は避けられない

ソフトウェア開発において、**仕様変更**は必ず起きます。

- ビジネス要件の変更
- ユーザーフィードバックへの対応
- 技術的負債の解消
- セキュリティ対応

問題は変更そのものではなく、**変更の影響**が見えないこと です。

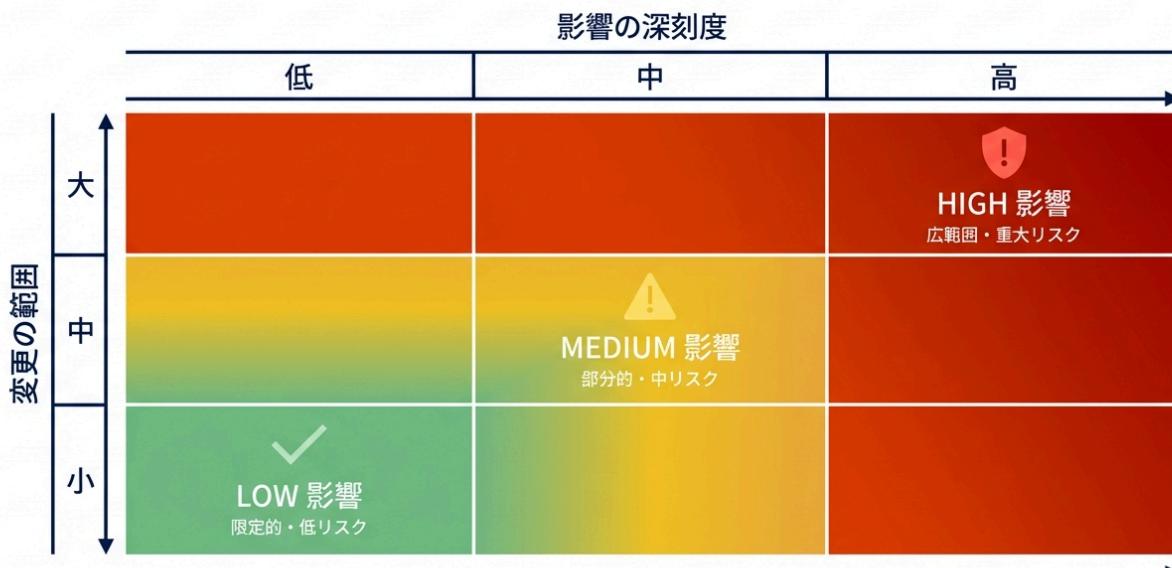
### 9.2.2 見えない影響が手戻りを生む

```
「パスワードを12文字以上に変更」  
↓  
影響を見落とす  
↓  
- フロントのバリデーションが古いまま  
- テストケースが更新されていない  
- APIドキュメントが古い  
- 既存ユーザーへの案内が必要だった  
↓  
本番でバグ発覚、手戻り
```

影響度評価は、この「見えない影響」を事前に可視化する仕組みです。

## 9.3 変更の3分類

### 変更影響度評価



すべての変更を以下の3つに分類します。

### 9.3.1 LOW：文言修正・スタイル調整

特徴：

- 機能的な影響なし
- 他の文書への波及なし
- レビュー1人で十分

例：

- エラーメッセージの文言修正
- コメントの追加・修正
- ドキュメントの誤字修正

- コードフォーマットの調整

対応：

#### ## 9.4 変更内容

エラーメッセージを「無効なパスワード」→「パスワードが正しくありません」に変更

#### ## 9.5 影響度: LOW

#### ## 9.6 対応: 単独で変更可能

### 9.6.1 MEDIUM：既存概念の拡張

特徴：

- 既存の機能に影響する可能性あり
- 関連文書の確認が必要
- レビュー複数人推奨

例：

- 新しいフィールドの追加
- オプション機能の追加
- APIにパラメータ追加
- 新しいエラーケースの追加

対応：

#### ## 9.7 変更内容

ユーザー モデルにprofileImageUrl フィールドを追加

#### ## 9.8 影響度: MEDIUM

#### ## 9.9 確認が必要な文書

- [ ] DOMAIN.md: Userエンティティの定義更新
- [ ] ARCHITECTURE.md: 画像ストレージの設計確認
- [ ] TESTING.md: プロフィール画像関連のテストケース

#### ## 9.10 後方互換性

- 既存APIは影響なし（オプショナルフィールド）
- 既存データはnullで初期化

### 9.10.0.1 LOW vs MEDIUMの境界

「これはLOW？それともMEDIUM？」と迷うケースがあります。判断のポイントは、他の箇所に波及するかどうかです。

たとえば、エラーメッセージの文言変更は通常LOWです。しかし、そのエラーメッセージをフロントエンドがパースして分岐処理していたらどうでしょう。「無効なパスワード」を「パスワードが正しくありません」に変えると、フロントエンドのコードが壊れます。これはMEDIUMです。

同様に、コメントの追加は通常LOWですが、そのコメントがAPIドキュメントに自動生成される仕組みになっていれば、MEDIUMになる可能性があります。

判断に迷ったときは、MEDIUMとして扱うのが安全です。LOWと判断して漏れがあると手戻りが発生しますが、MEDIUMとして扱って実はLOWだったとしても、追加の確認をしただけで済みます。

### 9.10.1 HIGH：概念の再定義・削除

特徴：

- 根本的な設計変更
- 複数の文書への波及
- 全関連チームのレビュー必須

例：

- データモデルの構造変更
- APIの破壊的変更
- 認証方式の変更
- ビジネスルールの根本的変更
- 機能の削除

対応：

### ## 9.11 変更内容

認証方式をセッションベースからJWTに変更

### ## 9.12 影響度: HIGH

### ## 9.13 必須チェックリスト

- [ ] PROJECT.md: セキュリティ要件との整合性
- [ ] ARCHITECTURE.md: 認証フロー全体の再設計
- [ ] DOMAIN.md: セッション関連ルールの削除/JWT関連ルールの追加
- [ ] PATTERNS.md: 認証関連のコードパターン更新
- [ ] TESTING.md: 認証関連のテストケース全更新
- [ ] DEPLOYMENT.md: セッションストア削除、JWT鍵管理追加

### ## 9.14 移行計画

1. Phase 1: JWT認証を並行稼働（2週間）
2. Phase 2: 新規ログインはJWTのみ（1週間）
3. Phase 3: セッション認証を完全廃止

### ## 9.15 ADR(Architecture Decision Record)

- ADR-005: 認証方式の変更.md

#### 9.15.0.1 MEDIUM vs HIGHの境界

MEDIUMとHIGHの境界も迷いやすいポイントです。判断基準は、既存の設計を維持できるかどうかです。

MEDIUMは「拡張」です。既存の設計の枠内で、新しい要素を追加します。たとえば、ユーザー モデルに新しいフィールドを追加するのはMEDIUMです。既存のコードは影響を受けず、新しいフィールドを使うコードだけが変更されます。

一方、HIGHは「再設計」です。既存の設計を根本から変える必要があります。たとえば、ユーザーIDをUUIDから連番に変更するのはHIGHです。すべての参照箇所、外部キー、API、フロントエンドの表示ロジックまで影響が波及します。

もう一つの判断ポイントは、**ロールバックの難易度**です。MEDIUMの変更は比較的簡単にロールバックできます。追加したフィールドを無視すれば、旧バージョンに戻れます。しかし、HIGHの変更はロールバックが困難です。データ移行が絡む場合、元に戻すにも移行作業が必要になります。

#### 9.15.0.2 影響度を過小評価したときの教訓

あるプロジェクトで、「ユーザー名のバリデーションルールを変更」という作業が発生しました。担当者は「バリデーションの文字数を変えるだけ」と判断し、LOWとして対応しました。

実際には、以下の影響がありました。

- フロントエンドのフォームバリデーション（別リポジトリ）
- モバイルアプリのバリデーション（別チーム管轄）
- 既存ユーザーのユーザー名が新ルールに違反するケース

- CSVインポート機能のバリデーション
- 管理画面の表示幅

本番デプロイ後、モバイルアプリで登録できないという問い合わせが発生。調査と修正に3日を要しました。最初からMEDIUM以上として扱い、関連チームに確認していれば、1日で完了していたはずです。

この教訓から、「単純に見える変更ほど慎重に」というルールがチームに定着しました。

## 9.16 HIGH変更の必須チェックリスト

HIGH変更を行う際は、以下のチェックリストを必ず実施します。

### 9.16.1 1. 影響範囲の洗い出し

#### ## 9.17 影響を受ける文書

- [ ] PROJECT.md: [該当セクション]
- [ ] ARCHITECTURE.md: [該当セクション]
- [ ] DOMAIN.md: [該当セクション]
- [ ] PATTERNS.md: [該当セクション]
- [ ] TESTING.md: [該当セクション]
- [ ] DEPLOYMENT.md: [該当セクション]

#### ## 9.18 影響を受けるコード

- [ ] [ディレクトリ/ファイル1]
- [ ] [ディレクトリ/ファイル2]

#### ## 9.19 影響を受ける外部システム

- [ ] [外部API/サービス]

### 9.19.1 2. 後方互換性の確認

#### ## 9.20 後方互換性

##### ### 9.20.1 維持できる点

- [ ] 互換性が保たれる機能]

##### ### 9.20.2 破壊的変更

- [ ] 互換性が失われる機能]
- 対応策: [移行方法]

### 9.20.3 3. 移行計画

#### ## 9.21 移行計画

##### ### 9.21.1 Phase 1: 準備(～開始前)

- [ ] 新旧両対応の実装
- [ ] 移行用テストの追加

##### ### 9.21.2 Phase 2: 並行稼働

- [ ] 新方式を有効化
- [ ] モニタリング強化

##### ### 9.21.3 Phase 3: 旧方式の廃止

- [ ] 旧コードの削除
- [ ] ドキュメントの整理

#### 9.21.4 4. ロールバック計画

##### ## 9.22 ロールバック計画

###### ### 9.22.1 トリガー条件

- エラーレートが5%を超えた場合
- レスポンスタイムがp99で1秒を超えた場合

###### ### 9.22.2 ロールバック手順

1. [具体的な手順1]
2. [具体的な手順2]
3. [具体的な手順3]

#### 9.22.3 5. ADR (Architecture Decision Record)

HIGH変更は、必ず意思決定の記録を残します。

##### # ADR-005: 認証方式の変更

##### ## 9.23 ステータス

承認済み

##### ## 9.24 コンテキスト

現在のセッションベース認証では、水平スケールが困難。  
マイクロサービス化に向けて、ステートレスな認証が必要。

##### ## 9.25 決定

JWTベースの認証に移行する。

##### ## 9.26 理由

- ステートレスでスケーラブル
- マイクロサービス間での認証が容易
- 業界標準で知見が豊富

##### ## 9.27 却下した代替案

###### ### 9.27.1 セッション認証の維持

- 理由: スケール時にセッション共有が複雑化

###### ### 9.27.2 OAuth2 + OpenID Connect

- 理由: 現時点ではオーバースペック (将来的には検討)

##### ## 9.28 結果

- JWT実装により認証サーバーが不要になった
- レスポンスタイムが平均20%改善
- セッションストア (Redis) のコストが削減

#### 9.29 仕様変更→AI再実装を安全に繰り返す設計

##### 9.29.1 安全な反復のための3原則

###### 9.29.1.1 原則1：変更は必ず文書から始める

✖ 危険なフロー

コード変更 → 後からドキュメント更新

#### 安全なフロー

ドキュメント変更 → 影響度評価 → コード変更

ドキュメントを先に変更することで、影響範囲を事前に把握できます。

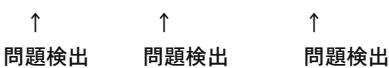
#### 9.29.1.2 原則2：影響度に応じたレビュ一体制

| 影響度    | レビュ一体制          |
|--------|-----------------|
| LOW    | 1人承認でOK         |
| MEDIUM | 2人以上、関連チーム1人以上  |
| HIGH   | 全関連チーム、テックリード必須 |

#### 9.29.1.3 原則3：段階的なデプロイ

HIGH変更は一気にデプロイしません。

開発環境 → ステージング → カナリア (10%) → 本番 (100%)



各段階で問題が検出されたら、即座にロールバックできる状態を維持します。

#### 9.29.2 AIへの指示テンプレート

仕様変更後にAIに再実装を依頼する際のテンプレート：

##### ## 9.30 仕様変更対応の依頼

###### ### 9.30.1 変更内容

[DOMAIN.mdの該当部分を抜粋]

###### ### 9.30.2 影響度

[LOW / MEDIUM / HIGH]

###### ### 9.30.3 変更対象ファイル

- src/services/auth/login.ts
- src/services/auth/\_\_tests\_\_/login.test.ts

###### ### 9.30.4 制約

- 後方互換性を維持すること
- 既存のテストが全てパスすること
- 新しいテストケースを追加すること

###### ### 9.30.5 確認ポイント

- [ ] 変更前の動作を壊していないか
- [ ] 新しい仕様に従っているか
- [ ] テストカバレッジが維持されているか

#### 9.31 章末チェックリスト

- 最近の変更を3分類（LOW/MEDIUM/HIGH）で振り返る
- HIGH変更のチェックリストをPRテンプレートに追加する
- ADRを書く習慣をつける（まずは1件から）

- 「コード先、ドキュメント後」の習慣を「ドキュメント先」に変える

## 9.32 次章への橋渡し

この章では、影響度評価を使った変更管理を学びました。

次章からは第4部「現場FAQ」に入ります。「結局エンジニアが全部書くのでは？」「品質は担保できるのか？」といった現場でよく出る疑問に答えます。

## 第4部 現場で揉めるポイントへの回答



「結局エンジニアが書くのでは？」「品質は大丈夫？」「責任は誰が取る？」——導入時に必ず出る疑問に答えます。懐疑派を納得させる論理と、実装を自動化するツール活用法を解説。

## 第10章 「それ、結局エンジニアが全部書くのでは？」への答え

### 10.1 この章で学ぶこと

- 人間とAIの役割分担の再設計
- 「書く」から「編集する」へのシフト
- 80%できたら修正指示にする戦略

### 10.2 よくある誤解

#### 10.2.1 「仕様書を書く時間でコードが書ける」

AI仕様駆動開発を説明すると、こんな反応が返ってきます。

- 「仕様書を書く時間があったら、コードを書いた方が早い」

- ・ 「結局、人間が全部考えるなら、AIいらなくない？」
- ・ 「二度手間じゃん」

この反応は理解できます。しかし、前提が間違っています。

### 10.2.2 本当の比較対象

#### ✗ 誤った比較

仕様を書く時間 vs コードを書く時間

#### ✓ 正しい比較

仕様を書く時間 + AI実装 vs コードを書く時間 + デバッグ + リファクタ + 手戻り

実際の開発では、最初のコード作成は全体の一部に過ぎません。

- ・ バグ修正
- ・ レビュー対応
- ・ リファクタリング
- ・ 仕様変更への対応
- ・ ドキュメント整備

これらを含めた総工数で比較すると、仕様駆動のアプローチが優位になります。

### 10.2.3 従来アプローチの隠れたコスト

「コードを書く時間」だけを見ていると、隠れたコストが見えません。複数の開発現場での観察から、エンジニアの作業時間は概ね以下のようない傾向があります（プロジェクトの特性により大きく異なります）。

- ・ コードを書く時間：約20%
- ・ コードを読む時間：約40%
- ・ デバッグ・調査の時間：約25%
- ・ コミュニケーションの時間：約15%

つまり、コードを書いている時間は全体の2割程度です。残りの8割は「すでにあるコードの理解」と「問題の特定」に費やされています。

AI仕様駆動開発では、この構成が変わります。仕様が明確であれば、AIが生成したコードの意図は理解しやすくなります。また、仕様に沿ってテストが自動生成されるため、デバッグ時間も短縮されます。「仕様を書く時間」は増えますが、「読む時間」「デバッグ時間」が大幅に減るのであります。

## 10.3 役割分担の再設計

### 10.3.1 従来の役割分担

人間：

- 要件定義
- 設計
- コーディング
- テスト作成
- デバッグ
- ドキュメント作成

AI：

- コード補完（数行レベル）

### 10.3.2 新しい役割分担

#### 人間：

- 仕様の定義 (What/Why)
- 設計判断 (How)
- レビューと検証
- ナレッジの蓄積

#### AI：

- コード生成
- テスト生成
- ドキュメント生成
- リファクタリング実行
- エラーの修正

### 10.3.3 人間がやるべきこと

| 領域   | 人間の責務        | AIが代替できない理由         |
|------|--------------|---------------------|
| 要件定義 | ビジネス価値の判断    | ユーザーのニーズは人間しか理解できない |
| 設計判断 | トレードオフの決定    | 「何を優先するか」は文脈依存      |
| レビュー | 品質の最終判断      | 「正しい」の定義は人間が持つ      |
| 合意形成 | ステークホルダーとの調整 | 人間関係の構築は人間の仕事       |

### 10.3.4 AIがやるべきこと

| 領域     | AIの責務     | 人間より優れている理由   |
|--------|-----------|---------------|
| コード生成  | パターンの組み立て | 既知パターンの再現は高速  |
| テスト作成  | 網羅的なケース生成 | 漏れなくケースを列挙    |
| リファクタ  | 一貫した変換    | 機械的な作業は得意     |
| ドキュメント | 構造化された記述  | 決まった形式への変換は確実 |

### 10.3.5 境界が曖昧な作業をどう扱うか

上の表では明確に分かれていますが、実際には「人間とAI、どちらがやるべきか」が曖昧な作業があります。

**コードレビュー**：人間とAIの両方が担当します。AIは「PATTERNS.mdに沿っているか」「型エラーがないか」といった機械的なチェックを担当し、人間は「この設計で本当にいいか」「ビジネス要件を満たしているか」という判断を担当します。まずAIにレビューさせ、機械的な指摘を漬してから人間がレビューすると効率的です。

**テスト設計**：テスト戦略（何をテストするか）は人間が決め、テストケースの実装はAIが担当します。「境界値をテストすべき」と人間が決めれば、具体的な境界値のテストコードはAIが生成できます。

**バグの調査**：原因の仮説立ては人間が行い、仮説の検証（ログの解析、コードの追跡）はAIが得意です。「このあたりが怪しい」と人間が当たりをつけ、「この関数の呼び出し元をすべて調べて」とAIに指示するのが効率的です。

**アーキテクチャの検討**：複数の選択肢を洗い出すのはAIが得意ですが、最終的な判断は人間が行います。「認証方式の選択肢を3つ挙げて、それぞれのメリット・デメリットを整理して」と指示し、その情報をもとに人間が決定します。

## 10.4 「書く」から「編集する」へ

### 10.4.1 パラダイムシフト

従来：「人間が書く」

人間 → [コード] → 完成

新しいアプローチ：「AIが書き、人間が編集する」

人間 → [仕様] → AI → [80%のコード] → 人間が編集 → 完成

#### 10.4.2 編集の方が効率的な理由

##### 1. ゼロからより、修正の方が認知負荷が低い

- 白紙から書くのは難しい
- 既存のものを直すのは比較的簡単

##### 2. 問題点が見えやすい

- 動くコードを見れば、改善点がわかる
- 抽象的な議論より、具体的なコードで議論

##### 3. コミュニケーションコストが下がる

- 「こう直して」の方が「こう書いて」より伝わりやすい

#### 10.4.3 編集指示の例

##### ## 10.5 AIが生成したコード

```
```typescript
async function validateUser(email: string, password: string) {
  const user = await db.user.findByEmail(email);
  if (!user) throw new Error('User not found');
  if (!await bcrypt.compare(password, user.password)) {
    throw new Error('Invalid password');
  }
  return user;
}
```

## 10.6 編集指示

1. エラーをthrowではなくResult型で返して
2. 関数を「ユーザー検索」と「パスワード検証」に分割して
3. 型定義を追加して

この指示で、AIは的確に修正できます。

##### ### 10.6.1 効果的な編集指示のコツ

編集指示を出すときの実践的なコツを紹介します。

###### #### 10.6.1.1 「なぜ」を添える

「Result型で返して」だけでなく、「エラーをthrowすると呼び出し元でtry-catchが必要になり、エラーハンドリングが煩雑になるため、Result型で返して」と理由を添えると、AIはより適切なコードを生成します。理由がわかれれば、指示していない箇所でも同じ方針を適用してくれます。

###### #### 10.6.1.2 具体例を示す

「関数を分割して」より、「findUserByEmailとverifyPasswordに分割して」のように、具体的な関数名まで指定する方が確実です。AIは指

示を解釈する余地が減り、期待通りの結果が得られやすくなります。

#### #### 10.6.1.3 一度に多くを求める

修正指示は3つ程度に抑えます。多すぎると、AIが一部の指示を見落としたり、相互に矛盾する変更を加えたりすることがあります。大きな修正が必要な場合は、複数回に分けて指示を出します。

#### #### 10.6.1.4 修正後の確認ポイントを伝える

「分割した後、両方の関数にユニットテストがあることを確認して」のように、修正後に何をチェックすべきかを伝えると、AIは自己検証してから出力します。

---

### ## 10.7 80%できたら修正指示にする

#### ### 10.7.1 80%ルール

AIが生成したコードが\*\*80%程度の完成度\*\*に達したら、そこから先は\*\*修正指示\*\*で進めます。

0% → [AIに生成させる] → 80% 80% → [修正指示] → 90% 90% → [微調整] → 100%

#### ### 10.7.2 なぜ80%か

- \*\*80%までは高速\*\* : AIはパターン適用が得意
- \*\*80%以降は難しい\*\* : 細かい調整は人間の判断が必要
- \*\*100%を目指すと時間がかかる\*\* : 完璧を求めるとき効率が落ちる

#### ### 10.7.3 80%の判断基準

| 観点 | 80%到達の目安 |

|-----|-----|

| 機能 | 主要機能が動く |

| 構造 | 適切なファイル・関数分割 |

| テスト | 主要ケースがパス |

| エラー処理 | 基本的なエラーをハンドリング |

| 観点 | 残り20%（人間が仕上げる） |

|-----|-----|

| エッジケース | 細かい境界条件 |

| パフォーマンス | 最適化 |

| UX | 細かいメッセージやタイミング |

| 運用考慮 | ログ、モニタリング |

---

### ## 10.8 具体的なワークフロー

#### ### 10.8.1 ステップ1：初回生成 (0%→60%)

```markdown

#### ## 10.9 AIへの指示

Issue #42 を実装してください。

docs/配下の仕様に従ってください。

### 10.9.1 ステップ2：初回レビュー（60%→80%）

#### ## 10.10 AIが生成したコードを確認

- 基本的な機能は動く
- ファイル構成は適切
- エラーハンドリングが不十分
- 一部の関数が複雑すぎる

### 10.10.1 ステップ3：修正指示（80%→90%）

#### ## 10.11 修正指示

1. validateAndSave関数を、validate関数とsave関数に分割して
2. エラーをResult型で返すように変更して
3. 入力バリデーションのエラーメッセージをi18n対応して

### 10.11.1 ステップ4：微調整（90%→100%）

#### ## 10.12 最終調整

1. ログレベルをdebug→infoに変更
2. タイムアウト値を定数化
3. コメントを追加

## 10.13 よくある反論への回答

### 10.13.1 「仕様を書くスキルが必要じゃないか」

回答：そのスキルこそがエンジニアの本質的な価値です。

コードを書く能力は自動化されますが、「何を作るべきか」を判断する能力は自動化できません。仕様を書くスキルを磨くことは、エンジニアとしてのキャリアを強化します。

### 10.13.2 「AIが間違えたら結局直すんだろう」

回答：その通りです。だからこそ「編集」という視点が重要です。

AIが間違えることを前提に、修正しやすい形で出力させる。完璧を期待せず、80%で修正に切り替える。これが現実的なアプローチです。

### 10.13.3 「ジュニアが育たないのでは」

回答：むしろ育ちやすくなります。

従来：コードを書いて覚える（時間がかかる） 新しいアプローチ：コードを読んで、直して覚える（早い）

AIが生成したコードをレビューし、修正指示を出す過程で、コードの品質について深く学べます。

## 10.14 章末チェックリスト

- 自分の業務を「人間がやるべき」「AIがやるべき」に分類する
- 次のタスクで「生成→編集」のフローを試す
- 80%到達の判断基準を明文化する
- 修正指示のテンプレートを作る

## 10.15 次章への橋渡し

この章では、人間とAIの役割分担を整理しました。

次章では、**品質・セキュリティ・責任の所在**——「AIが書いたコードの責任は誰が取るのか」という問い合わせます。

# 第11章 品質・セキュリティ・責任の所在

## 11.1 この章で学ぶこと

- 「AIが書いたコード」の責任の考え方
- テスト戦略を先に固める意味
- 監査可能性を確保する仕組み

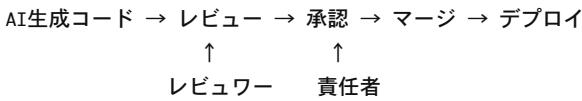
## 11.2 「AIが書いたコード」の責任は誰にあるか

### 11.2.1 よくある懸念

- 「AIが書いたコードにバグがあったら、誰の責任？」
- 「セキュリティホールがあったら？」
- 「本番障害が起きたら？」

これらの懸念は当然です。しかし、**責任の所在は明確**です。

### 11.2.2 責任は「承認した人」にある



AIはツールです。ツールが output したものを承認した人間に責任があります。

これは、IDEの自動補完やコードフォーマッタと同じ考え方です。

- Prettierが変なフォーマットをしても、Prettierの責任ではない
- 設定した人、承認した人の責任

### 11.2.3 責任を取れる状態を作る

責任を取るためにには、以下が必要です。

#### 1. レビュー可能であること

- コードが理解できる粒度であること
- 変更理由が追跡できること

#### 2. テストがあること

- 動作が検証されていること
- 回帰テストが実行されること

#### 3. ロールバック可能であること

- 問題が起きたら戻せること
- 影響範囲が把握できること

AI仕様駆動開発は、これらすべてを仕組み化します。

#### 11.2.3.1 責任を取れない状態の具体例

「責任を取れない状態」とは具体的にどういう状況でしょうか。実際にあった例を紹介します。

あるスタートアップで、AIに「課金機能を実装して」と依頼し、生成されたコードをほぼそのままマージしました。レビューは「動いていいからOK」という軽いものでした。

1ヶ月後、顧客から「二重課金された」という報告が入りました。調査を始めましたが、以下の問題に直面しました。

まず、**コードが理解できない**。AIが生成した課金ロジックは複雑で、なぜこの順序で処理しているのか、担当者にも説明できませんでした。レビュー時に深く理解していなかったためです。

次に、**テストがない**。正常系の簡単なテストしかなく、ネットワークエラー時のリトライ動作がテストされていませんでした。二重課金の原因是、リトライ処理のバグでした。

最後に、**変更理由が追跡できない**。なぜこのリトライ回数なのか、なぜこのタイムアウト値なのか、どこにも記録がありませんでした。

結局、原因特定に3日、修正と検証に2日、顧客対応に1週間かかりました。責任を取れる状態——レビュー可能、テストあり、ロールバック可能——が整っていれば、問題は発生しなかったか、発生しても数時間で解決できたはずです。

## 11.3 テスト戦略を先に固める意味

### 11.3.1 テストは「仕様の検証」

テストを先に考えることの本質的な意味は、「正しさの定義」を先に決めることです。

従来：

コード書く → テスト書く → 「これでいいか」を確認

AI仕様駆動：

仕様を書く → テスト戦略を定義 → AIがコード+テスト生成 → 検証

テスト戦略が明確なら、AIに「これをテストしろ」と指示できます。

### 11.3.2 テスト戦略の定義項目

TESTING.mdには以下を含めます。

#### ## 11.4 テスト戦略

##### ### 11.4.1 テストの種類と比率

- Unit: 70% (ビジネスロジック中心)
- Integration: 20% (API/DB連携)
- E2E: 10% (クリティカルパス)

##### ### 11.4.2 何をテストするか

- ビジネスロジックのすべてのパス
- 境界値 (0, 1, max, max+1)
- エラーケース
- セキュリティ関連 (認証、認可、入力検証)

##### ### 11.4.3 何をテストしないか

- 外部ライブラリの内部動作
- フレームワークの標準動作
- UIの見た目 (E2Eで最低限)

##### ### 11.4.4 モック戦略

- 外部API：必ずモック
- DB：Unitはモック、Integrationは実DB

- 時間：固定値を注入
- ランダム：シード固定

### 11.4.5 AIへの効果

テスト戦略が定義されていると、AIは次のように動作します。

- ・「この関数にはユニットテストが必要」と判断できる
- ・「境界値テストを追加すべき」と認識できる
- ・「モックの使い方」を正しく適用できる

#### 11.4.5.1 テスト戦略がないときのAIの挙動

テスト戦略が定義されていない場合、AIはどのようなテストを生成するでしょうか。実際に観察された傾向を紹介します。

まず、**カバレッジが偏る**傾向があります。正常系のテストは充実しますが、エラーケースや境界値のテストが不足しがちです。AIは「典型的なテスト」を学習しているため、例外的なケースへの意識が低くなります。

次に、**モックの使い方が不適切**になります。「外部APIはモック」「DBは実DBを使用」といった方針がないと、AIは一貫性のないモック戦略でテストを書きます。あるテストではDBをモックし、別のテストでは実DBを使う——といった混乱が生じます。

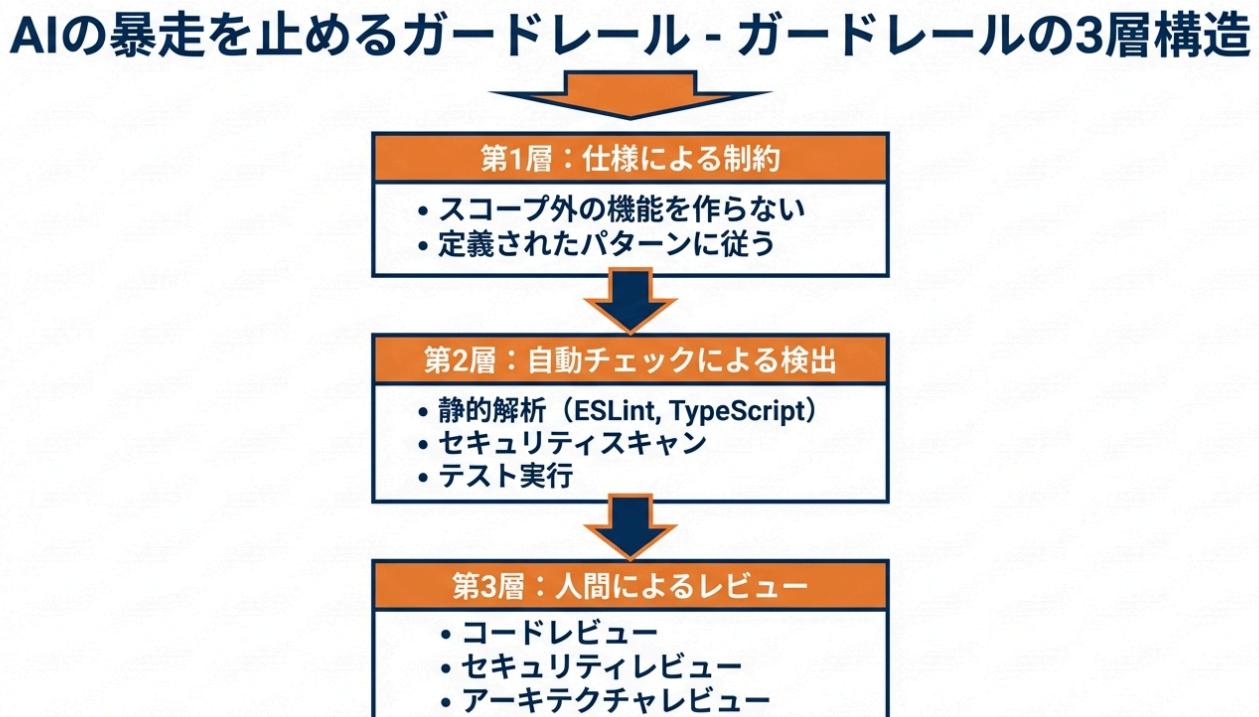
また、**テストの粒度が安定しない**傾向もあります。1つの関数に対して10個のテストを書くこともあれば、複雑な関数でも1つのテストで済ませることもあります。「ビジネスロジックは手厚く、ユーティリティは最低限」といった基準がないためです。

さらに、**不必要的テストが増える**こともあります。フレームワークの標準動作をテストしたり、ライブラリの内部実装をテストしたりすることがあります。「何をテストしないか」が明示されていないためです。

TESTING.mdでテスト戦略を明示することで、これらの問題を防ぎ、一貫した品質のテストが生成されるようになります。

## 11.5 AIの暴走を止めるガードレール

### 11.5.1 ガードレールの3層構造



AIの品質を担保するために、3層のガードレールを設けます。

- 第1層：仕様による制約 – スコープ外の機能を作らない、定義されたパターンに従う
- 第2層：自動チェックによる検出 – 静的解析（ESLint, TypeScript）、セキュリティスキャン、テスト実行
- 第3層：人間によるレビュー – コードレビュー、セキュリティレビュー、アーキテクチャレビュー

各層の詳細を見ていきましょう。

### 11.5.2 第1層：仕様による制約

AIに「やってはいけないこと」を明示します。

#### ## 11.6 セキュリティ制約(ARCHITECTURE.mdより)

##### ### 11.6.1 禁止事項

- ユーザー入力を直接SQLに埋め込まない
- 認証なしでAPIを公開しない
- 秘密情報をログに出力しない
- 動的コード実行や危険なDOM操作を行わない

##### ### 11.6.2 必須事項

- すべての入力をバリデーションする
- すべてのAPIに認証ミドルウェアを適用する
- 機密データは暗号化して保存する

### 11.6.3 第2層：自動チェック

CIパイプラインで自動検出します。

```
# .github/workflows/security.yml
name: Security Check

on: [push, pull_request]

jobs:
  security:
    runs-on: ubuntu-latest
    steps:
      # 依存関係の脆弱性チェック
      - name: Dependency scan
        run: npm audit --audit-level=high

      # コードのセキュリティスキャン
      - name: CodeQL Analysis
        uses: github/codeql-action/analyze@v2

      # 機密情報の漏洩チェック
      - name: Secret scan
        uses: trufflesecurity/trufflehog@main
        with:
          path: ./
```

### 11.6.4 第3層：人間によるレビュー

最終的には人間が確認します。

#### ## 11.7 セキュリティレビューチェックリスト

### ### 11.7.1 認証・認可

- [ ] すべてのエンドポイントに認証が設定されているか
- [ ] 権限チェックが適切に行われているか
- [ ] セッション管理は安全か

### ### 11.7.2 入力検証

- [ ] すべてのユーザー入力がバリデーションされているか
- [ ] SQLインジェクション対策がされているか
- [ ] XSS対策がされているか

### ### 11.7.3 データ保護

- [ ] 機密データは暗号化されているか
- [ ] ログに機密情報が出力されていないか
- [ ] エラーメッセージに機密情報が含まれていないか

## 11.8 監査可能性：いつ誰が何を変えたかを追える

### 11.8.1 なぜ監査可能性が重要か

問題が起きたとき、「なぜこうなったか」を追跡できる必要があります。

- セキュリティインシデント時の調査
- コンプライアンス監査への対応
- バグの原因特定

### 11.8.2 追跡可能性の確保

#### 11.8.2.1 1. コミット履歴

すべての変更はGitで追跡されます。

```
# いつ、誰が、何を変えたか
git log --oneline --author="name" --since="2026-01-01"

# 特定のファイルの変更履歴
git log --follow -p -- path/to/file
```

#### 11.8.2.2 2. PRとIssueの紐付け

すべてのPRはIssueに紐付けます。

### ## 11.9 PR #42: ログイン機能の実装

Closes #42

### ### 11.9.1 変更内容

- ログインAPI追加
- セッション管理追加

これにより、「なぜこの変更が必要だったか」がIssueから追跡できます。

### 11.9.1.1 3. ADR (Architecture Decision Record)

重要な決定は文書化します。

### # ADR-006: パスワードハッシュアルゴリズムの選定

## ## 11.10 日付

2024-01-15

## ## 11.11 決定者

@security-team, @tech-lead

## ## 11.12 決定

Argon2idを使用する

## ## 11.13 理由

- OWASP推奨
- bcryptより高いセキュリティ
- ライブリが安定している

## ## 11.14 代替案

- bcrypt: 却下 (Argon2idの方がセキュア)
- scrypt: 却下 (ライブラリが不安定)

### 11.14.0.1 4. 監査ログ

重要な操作は監査ログに記録します。

```
// 監査ログの記録
await auditService.log({
  action: 'USER_LOGIN',
  actor: userId,
  target: { type: 'user', id: userId },
  details: { ip: request.ip, userAgent: request.headers['user-agent'] },
  timestamp: new Date(),
});
```

## 11.15 AIが書いたコードと人間が書いたコード

### 11.15.1 区別する必要があるか？

必要ありません。

レビューを通過し、テストをパスし、承認されたコードは、誰が書いたかに関係なく「チームのコード」です。

### 11.15.2 重要なのはプロセス

AIが書いたか人間が書いたかより、適切なプロセスを経たかが重要です。

- 仕様に基づいているか
- レビューを受けたか
- テストをパスしたか
- セキュリティチェックをパスしたか
- 承認されたか

これらすべてをクリアしたコードは、品質が担保されています。

## 11.16 章末チェックリスト

- テスト戦略 (TESTING.md) を整備する
- セキュリティ制約をARCHITECTURE.mdに明記する

- CIにセキュリティスキャンを追加する
  - セキュリティレビューチェックリストを作成する
  - PRテンプレートにIssue紐付けを必須にする
- 

## 11.17 次章への橋渡し

この章では、品質・セキュリティ・責任の所在について整理しました。

次章では、ツール実装—Claude CodeとSkillsを使って仕様駆動を自動化する具体的な方法を解説します。

# 第12章 ツール実装：Claude Code / GitHub Copilotで"仕様駆動"を自動化する

## 12.1 この章で学ぶこと

- Claude CodeとGitHub Copilotの両方で仕様駆動開発を自動化する方法
  - 使用するツールに関わらず、同等のレビュー機能を実現できること
  - 各ツール固有のエージェント/スキルの設計方法
- 

## 12.2 読者へのメッセージ

本章では、Claude CodeとGitHub Copilotの両方を取り上げます。

どちらのツールでも、仕様駆動開発を支援する同等のエージェント機能を利用できます。普段お使いのツールに合わせて、該当するセクションを参照してください。

重要な違い:

| 観点     | Claude Code               | GitHub Copilot            |
|--------|---------------------------|---------------------------|
| プラグイン  | pr-review-toolkitなど公式提供済み | なし（自作が必要）                 |
| 準備の手間  | すぐに使える                    | 本章のテンプレートを導入              |
| カスタマイズ | Skills/Agentsファイル         | .github/agents/*.agent.md |

本章では、GitHub Copilotユーザー向けに**6つのエージェントテンプレート**を提供します。これらはClaude Codeのpr-review-toolkitと同等の機能を実現するためのものです。

---

## 12.3 Skillsの考え方

### 12.3.1 Skillsとは

Claude CodeのSkillsは、特定のワークフローをモジュール化したものです。

繰り返し行う作業を「スキル」として定義しておくことで：

- 毎回同じ指示を書かなくて済む
- チーム全体で統一された方法で作業できる
- プロジェクト横断で再利用できる

### 12.3.2 スキルの基本構造

## # スキル名

### ## 12.4 説明

このスキルが何をするか

### ## 12.5 トリガー

どんなときにこのスキルを使うか

### ## 12.6 実行手順

1. ステップ1
2. ステップ2
3. ステップ3

### ## 12.7 入力

- 必要な情報1
- 必要な情報2

### ## 12.8 出力

- 生成されるもの

## 12.8.1 スキルを作るべきタイミング

「どの作業をスキル化すべきか」という判断は難しいものです。以下の基準を参考にしてください。

スキル化すべき作業の条件:

1. 繰り返し発生する : 週に3回以上同じ作業をしている
2. 手順が定型化できる : 毎回ほぼ同じステップを踏む
3. ミスが起きやすい : 手順を忘れたり、漏れが発生したりする

逆に、スキル化しないほうがよいケースもあります。

スキル化を避けるべきケース:

- 頻度が低い作業（月に1回程度）: スキルを作るコストに見合わない
- 毎回判断が異なる作業: 定型化が難しく、スキルの柔軟性が足りなくなる
- 一度きりの作業: 使い回せないスキルは無駄

まずは「この作業、もう3回目だな」と感じたらスキル化を検討しましょう。最初から完璧なスキルを作る必要はありません。簡単なスキルから始めて、使いながら改善していく方が効率的です。

## 12.9 仕様駆動を支援するスキル例

### 12.9.1 スキル1：プロジェクト初期化

## # spec-init

### ## 12.10 説明

7文書構成を新規プロジェクトに導入する

### ## 12.11 トリガー

「7文書を初期化して」 「spec-initを実行して」

### ## 12.12 実行手順

1. docs/ディレクトリを作成
2. 7つのテンプレートファイルを生成

3. MASTER.mdにプロジェクト情報を記入
4. .claude/settings.jsonにドキュメントパスを設定

#### ## 12.13 入力

- プロジェクト名
- 技術スタック（言語、フレームワーク、DB）
- 簡単な説明

#### ## 12.14 出力

- docs/MASTER.md
- docs/PROJECT.md
- docs/ARCHITECTURE.md
- docs/DOMAIN.md
- docs/PATTERNS.md
- docs/TESTING.md
- docs/DEPLOYMENT.md

### 12.14.1 スキル2：Issue作成支援

#### # create-issue

#### ## 12.15 説明

仕様駆動に適したIssueを作成する

#### ## 12.16 トリガー

「Issueを作って」 「タスクをIssue化して」

#### ## 12.17 実行手順

1. ユーザーから機能の概要をヒアリング
2. 関連する仕様文書（DOMAIN.md, ARCHITECTURE.md）を参照
3. 受け入れ基準を具体化
4. 技術的制約を洗い出し
5. スコープ外を明確化
6. Issue形式で出力

#### ## 12.18 入力

- 機能の概要
- 関連する既存機能（あれば）

#### ## 12.19 出力

- Issue本文（Markdown形式）
- 関連文書へのリンク

### 12.19.1 スキル3：文書検証

#### # validate-docs

#### ## 12.20 説明

文書の整合性と完全性を検証する

#### ## 12.21 トリガー

「ドキュメントを検証して」 「validate-docs」

#### ## 12.22 実行手順

1. 全文書のFrontmatterを検証
2. 内部リンクの有効性を確認
3. 用語の統一性をチェック
4. 文書間の参照整合性を確認
5. MASTER.mdの索引が最新か確認
6. 問題点をレポート

#### ## 12.23 入力

- 検証対象のディレクトリ（デフォルト: docs/）

#### ## 12.24 出力

- 検証結果レポート
- 修正が必要な箇所のリスト

### 12.24.1 スキル4：影響度評価

#### # assess-impact

#### ## 12.25 説明

文書変更の影響度を評価する

#### ## 12.26 トリガー

「この変更の影響度は？」 「assess-impact」

#### ## 12.27 実行手順

1. 変更内容を分析
2. 影響を受ける文書を特定
3. 影響を受けるコードを特定
4. 影響度（LOW/MEDIUM/HIGH）を判定
5. 対応チェックリストを生成

#### ## 12.28 入力

- 変更内容の説明
- または変更対象の文書パス

#### ## 12.29 出力

- 影響度（LOW/MEDIUM/HIGH）
- 影響を受ける文書リスト
- 対応チェックリスト
- HIGH の場合はADRテンプレート

### 12.29.1 スキル5：コミット前チェック

#### # pre-commit-check

#### ## 12.30 説明

コミット前に仕様準拠を確認する

#### ## 12.31 トリガー

コミット時に自動実行（huskyフック）

#### ## 12.32 実行手順

1. 変更されたファイルを取得
2. 関連する仕様文書を特定

3. 仕様との整合性を確認
4. PATTERNS.mdのルールに違反していないか確認
5. MASTER.mdの更新が必要か判断
6. 問題があれば警告

#### ## 12.33 入力

- ステージングされたファイル

#### ## 12.34 出力

- チェック結果 (OK/警告/エラー)
- 警告・エラーの詳細

### 12.34.1 スキル同士を組み合わせる

個々のスキルは単独でも便利ですが、組み合わせることでより強力なワークフローを構築できます。

#### 12.34.1.1 典型的なワークフロー例

プロジェクト開始から日々の開発まで、スキルを連携させた流れを示します。

プロジェクト開始時:

```
spec-init → validate-docs → create-issue  
(初期化 → 検証 → 最初のIssue作成)
```

機能開発時:

```
create-issue → [AIで実装] → pre-commit-check → assess-impact  
(Issue作成 → 実装 → コミット前チェック → 影響度評価)
```

リリース前:

```
validate-docs → multi-review → assess-impact  
(文書検証 → 多視点レビュー → 影響度の最終確認)
```

### 12.34.1.2 スキル連携のポイント

スキルを連携させるときは、前のスキルの出力が次のスキルの入力になるように設計します。

例えば、`create-issue` が出力する「関連文書へのリンク」は、AIが実装するときのコンテキストになります。また、`assess-impact` が出力する「影響を受ける文書リスト」は、`validate-docs` でチェックすべき対象になります。

このように、スキル間でデータが自然に流れる設計になると、手動での情報の受け渡しが減り、ワークフロー全体が効率化します。

## 12.35 サブエージェントによる多視点レビュー

### 12.35.1 なぜ多視点が必要か

1人のレビュー（人間でもAIでも）には盲点があります。

多視点レビューは、異なる観点を持つ複数のエージェントが同時にレビューすることで、盲点をカバーします。

### 12.35.2 レビュワーエージェントの例

#### 12.35.2.1 セキュリティレビュワー

```
# security-reviewer
```

#### ## 12.36 役割

セキュリティ観点でコードをレビューする

### **## 12.37 確認観点**

- 認証・認可の実装
- 入力検証の網羅性
- 機密データの取り扱い
- 依存ライブラリの脆弱性
- ログ出力に機密情報が含まれていないか

### **## 12.38 出力形式**

### **## 12.39 セキュリティレビュー結果**

#### **### 12.39.1 重大な問題**

- [ファイル:行] 問題の説明

#### **### 12.39.2 警告**

- [ファイル:行] 懸念点の説明

#### **### 12.39.3 推奨事項**

- 改善提案

### **12.39.3.1 パフォーマンスレビュー**

#### **# performance-reviewer**

### **## 12.40 役割**

パフォーマンス観点でコードをレビューする

### **## 12.41 確認観点**

- N+1クエリの有無
- 不要なループ処理
- メモリ効率
- キャッシュ戦略
- 非同期処理の適切性

### **## 12.42 出力形式**

### **## 12.43 パフォーマンスレビュー結果**

#### **### 12.43.1 問題**

- [ファイル:行] 問題の説明と推定影響

#### **### 12.43.2 最適化提案**

- 改善案と期待される効果

### **12.43.2.1 アーキテクチャレビュー**

#### **# architecture-reviewer**

### **## 12.44 役割**

既存アーキテクチャとの整合性をレビューする

### **## 12.45 確認観点**

- レイヤー構成の遵守
- 依存関係の方向
- 責務の分離

- ARCHITECTURE.mdとの整合性
- PATTERNS.mdのパターン適用

#### ## 12.46 出力形式

#### ## 12.47 アーキテクチャレビュー結果

##### ### 12.47.1 違反

- [ファイル] 違反内容と参照すべき文書

##### ### 12.47.2 改善提案

- よりよい設計案

### 12.47.3 複数エージェントの統合実行

#### # multi-review

##### ## 12.48 説明

複数のレビュワーエージェントを並行実行する

##### ## 12.49 トリガー

「PRをレビューして」 「multi-review」

##### ## 12.50 実行手順

1. 変更ファイルを取得
2. 以下のエージェントを並行実行
  - security-reviewer
  - performance-reviewer
  - architecture-reviewer
3. 結果を統合してレポート

##### ## 12.51 出力

##### ## 12.52 統合レビュー結果

##### ### 12.52.1 セキュリティ

[security-reviewerの結果]

##### ### 12.52.2 パフォーマンス

[performance-reviewerの結果]

##### ### 12.52.3 アーキテクチャ

[architecture-reviewerの結果]

##### ### 12.52.4 総合判定

[Approve / Request Changes / Comment]

### 12.52.5 pr-review-toolkit (公式プラグイン)

Claude Codeには、PRレビュー専用の公式プラグイン「pr-review-toolkit」が提供されています。6つの専門エージェントを組み合わせた包括的なレビューシステムです。

#### 12.52.5.1 使い方

```
# すべてのレビューを実行  
/pr-review-toolkit:review-pr
```

```

# 特定のアспектのみ
/pr-review-toolkit:review-pr tests errors

# すべてを並列実行
/pr-review-toolkit:review-pr all parallel

```

#### 12.52.5.2 6つの専門エージェント

| エージェント                | 役割        | 主な検出対象                |
|-----------------------|-----------|-----------------------|
| code-reviewer         | コード品質     | CLAUDE.md違反、バグ、スタイル問題 |
| silent-failure-hunter | エラーハンドリング | 空のcatchブロック、沈黙する失敗    |
| code-simplifier       | 簡潔化       | ネスト過多、冗長コード           |
| comment-analyzer      | コメント品質    | 不正確なコメント、コメント腐れ       |
| pr-test-analyzer      | テスト分析     | テスト不足、エッジケース漏れ        |
| type-design-analyzer  | 型設計       | カプセル化不足、不变性の問題        |

#### 12.52.5.3 code-reviewerの信頼度スコア

code-reviewerは検出した問題に0-100の信頼度スコアを付与し、**80以上のみを報告します。**

| スコア    | 意味                | 報告   |
|--------|-------------------|------|
| 0-25   | 誤検出または既存の問題       | しない  |
| 26-50  | マイナー（ガイドラインに明記なし） | しない  |
| 51-79  | 有効だが低影響           | しない  |
| 80-90  | 重要な問題             | する   |
| 91-100 | クリティカル            | 必ずする |

この仕組みにより、ノイズの少ない高品質なレビュー結果が得られます。

## 12.53 GitHub Copilot Agents

**Note:** GitHub Copilotには、Claude Codeのpr-review-toolkitのような公式プラグインが提供されていません。同等の機能を実現するには、本セクションで紹介するエージェントテンプレートを自分でリポジトリに追加する必要があります。

#### 12.53.1 GitHub Copilot Agentsとは

GitHub Copilotのカスタムエージェントは、**特定のタスクに特化したAIの専門家**を定義できる機能です。VS Code 1.107以降で利用可能です。

| 項目     | 内容                        |
|--------|---------------------------|
| 保存場所   | .github/agents/ フォルダ      |
| ファイル形式 | *.agent.md (Markdown)     |
| 対応環境   | VS Code 1.107+、GitHub.com |

#### 12.53.2 4種類のエージェントタイプ

GitHub Copilotには4種類のエージェントがあります。

| 種類       | 実行場所       | 特徴                | 用途         |
|----------|------------|-------------------|------------|
| ローカル     | VS Code    | 対話的・リアルタイム        | 小～中規模タスク   |
| バックグラウンド | VS Code    | Git worktreeで並行作業 | 大規模リファクタ   |
| クラウド     | GitHub.com | 自律的にPR作成          | Issue解決    |
| サブ       | 親エージェント内   | 専門タスクを委譲          | 専門知識が必要な部分 |

### 12.53.3 カスタムエージェントの作成方法

.github/agents/ ディレクトリに \*.agent.md ファイルを作成します。

```
---
description: エージェントの目的や役割の説明
tools:
- "/*"
---

# エージェント名

## 12.54 役割
エージェントの役割を説明

## 12.55 確認観点
- チェックポイント1
- チェックポイント2

## 12.56 出力形式
期待する出力フォーマット
```

### 12.56.1 VS Code設定

カスタムエージェントを有効にするには、VS Codeの settings.json に以下を追加します。

```
{
  "github.copilot.chat.cli.customAgents.enabled": true
}
```

### 12.56.2 仕様駆動開発向けエージェント（6種）

以下は、Claude Codeのpr-review-toolkitと同等の機能を実現するためのエージェントテンプレートです。

#### 12.56.2.1 1. code-reviewer.agent.md

```
---
description: プロジェクトガイドラインへの準拠をチェックし、バグ、スタイル違反、コード品質問題を検出するコードレビューエージェント
tools:
- "/*"
---

# Code Reviewer
```

プロジェクトガイドラインへの準拠をチェックし、高信頼度の問題のみを報告するコードレビューエージェントです。

#### ## 12.57 役割

- CLAUDE.md, README.md、その他のプロジェクトガイドラインとの照合
- バグ検出
- スタイル違反の特定
- コード品質問題の発見

#### ## 12.58 分析プロセス

1. プロジェクトのガイドラインファイル(CLAUDE.md等)を読み込む
2. 変更されたファイルを特定する(git diff)
3. 各変更をガイドラインと照合する
4. 問題に信頼度スコアを付与する

#### ## 12.59 信頼度スコア

各問題には0-100の信頼度スコアを付与してください:

| スコア    | 意味                   | 報告    |
|--------|----------------------|-------|
| 10-25  | 誤検出または既存の問題          | 報告しない |
| 26-50  | マイナーな指摘(ガイドラインに明記なし) | 報告しない |
| 51-75  | 有効だが低影響              | 報告しない |
| 76-90  | 重要な問題                | 報告する  |
| 91-100 | クリティカルなバグまたは明示的な違反   | 必ず報告  |

\*\*報告閾値: 信頼度80以上ののみ報告\*\*

#### ## 12.60 出力形式

#### ## 12.61 Code Review Results

##### ### 12.61.1 Critical Issues (信頼度 91-100)

- [ファイル名:行番号] 問題の説明
  - 信頼度: XX
  - 理由: なぜこれが問題か
  - 修正提案: どう修正すべきか

##### ### 12.61.2 Important Issues (信頼度 76-90)

- [ファイル名:行番号] 問題の説明
  - 信頼度: XX
  - 理由: なぜこれが問題か
  - 修正提案: どう修正すべきか

##### ### 12.61.3 Summary

- 検出された問題数: X
- Critical: X
- Important: X

#### ## 12.62 注意事項

- 信頼度80未満の問題は報告しない
- 既存のコード(変更されていない部分)の問題は報告しない

- 推測や曖昧な指摘は避ける
- 具体的な修正提案を含める

#### 12.62.0.1 2. error-handler-hunter.agent.md

```
---
description: エラーハンドリングの品質を検査し、沈黙する失敗を検出するエージェント
tools:
- "/*"
---
```

##### # Error Handler Hunter

沈黙する失敗を許さない、エラーハンドリングの厳格な検査官です。

##### ## 12.63 役割

- try-catchブロックの検査
- 沈黙する失敗の検出
- 空のcatchブロックの禁止
- フォールバックロジックの正当性確認

##### ## 12.64 コア原則(譲歩不可)

1. 沈黙する失敗は受け入れられない
2. ユーザーは実行可能なフィードバックに値する
3. フォールバックは明示的で正当化される必要がある
4. キャッチブロックは特定的でなければならない

##### ## 12.65 重大度レベル

| レベル      | 説明                                     | 例 |
|----------|----------------------------------------|---|
| CRITICAL | サイレント失敗、ブロードcatch / 空のcatchブロック        |   |
| HIGH     | 不十分なエラーメッセージ / console.log("error") のみ |   |
| MEDIUM   | コンテキスト不足 / エラーの原因が不明確                  |   |

##### ## 12.66 出力形式

##### ## 12.67 Error Handling Analysis Results

###### ### 12.67.1 CRITICAL Issues

- [ファイル名:行番号] 問題の説明
- コード: 問題のあるコード
- 問題: 何が問題か
- 修正提案: 推奨される修正

###### ### 12.67.2 Summary

- CRITICAL: X
- HIGH: X
- MEDIUM: X

#### 12.67.2.1 3. test-analyzer.agent.md

```
---
```

description: テストカバレッジの品質を分析し、クリティカルなギャップを特定するエージェント  
tools:  
  - "/\*"

```
---
```

#### # Test Analyzer

行カバレッジではなく、動作カバレッジの観点からテスト品質を分析するエージェントです。

#### ## 12.68 役割

- 動作カバレッジの分析
- クリティカルなテストギャップの特定
- エッジケースとエラー条件のカバレッジ確認

#### ## 12.69 識別対象のギャップ

1. テストされていないエラーハンドリングパス
2. 境界条件のエッジケース(空入力, null, 最大値/最小値)
3. クリティカルなビジネスロジック分岐
4. ネガティブテストケース
5. 非同期/並行処理

#### ## 12.70 優先度スケール

|                                        |
|----------------------------------------|
| /優先度/意味/                               |
| ----- -----                            |
| /9-10/クリティカル(データ損失、セキュリティ、システム障害の可能性)/ |
| /7-8/重要(ユーザー向けエラーの可能性)/                |
| /5-6/エッジケース(混乱や軽微な問題)/                 |
| /3-4/Nice-to-have/                     |
| /1-2/オプショナル/                           |

#### ## 12.71 出力形式

#### ## 12.72 Test Coverage Analysis Results

##### ### 12.72.1 Critical Gaps (優先度 9-10)

- [機能名] ファイル: path/to/file.ts
- テストされていない動作: 説明
- リスク: 影響
- 優先度: X
- 推奨テストケース: 具体的なテスト案

#### 12.72.1.1 4. code-simplifier.agent.md

```
---
```

description: コードの簡潔性と可読性を向上させるエージェント。機能を変更せずに、不要な複雑性を排除します  
tools:  
  - "/\*"

```
---
```

#### # Code Simplifier

機能を保持したまま、コードの簡潔性と可読性を向上させるエージェントです。

## ## 12.73 役割

- 不要な複雑性の排除
- 可読性の向上
- 冗長なコードの削減

## ## 12.74 簡潔化のルール

### ### 12.74.1 推奨する変更

1. ネストした三項演算子 → if/else文へ
2. 深いネスト → 早期リターンパターンへ
3. 巧妙なコード → 分かりやすいコードへ

### ### 12.74.2 禁止事項

- 機能の変更
- 新機能の追加
- テストの削除

## ## 12.75 出力形式

## ## 12.76 Code Simplification Results

### ### 12.76.1 Simplification Opportunities

- [ファイル名:行番号]
- 現在のコード: ...
- 提案: ...
- 理由:なぜこの変更が可読性を向上させるか

## ## 12.77 注意事項

- 機能を絶対に変更しない
- 最近変更されたコードのみに焦点を当てる
- 簡潔性より明確性を優先する

## 12.77.0.1 5. comment-analyzer.agent.md

```
---  
description: コードコメントの正確性、完全性、長期的な保守性を分析するエージェント  
tools:  
- "/*"  
---
```

## # Comment Analyzer

コードコメントの品質を分析し、技術的負債を防ぐエージェントです。

## ## 12.78 役割

- コメントと実コードの照合
- コメント腐れ(技術的負債)の検出
- 誤解を招く・時代遅れなコメントの特定

## ## 12.79 検証プロセス

1. 事実精度の確認(関数署名、説明された動作)
2. 完全性の評価(仮定、副作用、エラー状態)
3. 長期的価値の評価('なぜ'を説明しているか)
4. 誤解要素の特定(曖昧性、古い参照)

## ## 12.80 出力形式

### ## 12.81 Comment Analysis Results

#### ### 12.81.1 Critical Issues(事実として不正確)

- [ファイル名:行番号]
  - コメント: "..."
  - 問題: コメントが実際のコードと矛盾
  - 推奨修正: ...

#### ### 12.81.2 Improvement Opportunities(改善可能)

- [ファイル名:行番号]
  - 問題: 情報が不完全
  - 推奨追加内容: ...

### 12.81.2.1 6. type-design-analyzer.agent.md

```
---  
description: 型設計の品質と不变性を分析するエージェント  
tools:  
  - "/*"  
---
```

#### # Type Design Analyzer

型設計品質と不变性の表現を分析し、堅牢な型システムの構築を支援するエージェントです。

## ## 12.82 役割

- 型カプセル化の評価
- 不变性表現の分析
- アンチパターンの検出

## ## 12.83 評価軸(各1-10スコア)

| 評価軸                   | 評価内容        |
|-----------------------|-------------|
| Encapsulation         | 内部実装の隠蔽度    |
| Invariant Expression  | 不变性の型による表現度 |
| Invariant Usefulness  | 実バグ防止への有効性  |
| Invariant Enforcement | 構築時・変異時の検証度 |

## ## 12.84 アンチパターン

- 貧血メインモデル(データのみで振る舞いがない)
- 変更可能な内部の公開
- ドキュメント依存の不变性

- 構築境界での検証不足

## 12.85 出力形式

## 12.86 Type Design Analysis Results

#### 12.86.1 [型名]

- ファイル: path/to/file.ts

- スコア: Encapsulation X/10, Invariant Expression X/10, ...

- 総合スコア: X/10

- 検出されたアンチパターン: ...

- 改善提案: ...

## 12.86.2 エージェントの呼び出し方

VS CodeのCopilot Chatで @ に続けてエージェント名を入力します。

```
@code-reviewer このPRをレビューして
```

```
@test-analyzer テストカバレッジを分析して
```

```
@error-handler-hunter エラーハンドリングを検査して
```

## 12.86.3 推奨ワークフロー

コミット前:

```
@code-reviewer  
@error-handler-hunter
```

PR作成前:

```
@code-reviewer  
@test-analyzer  
@error-handler-hunter  
@comment-analyzer
```

新しい型を追加した場合:

```
@type-design-analyzer
```

## 12.86.4 Premium Requestsとコスト

GitHub CopilotのカスタムエージェントはPremium Requestsを消費します。

| モデル                   | 消費量   |
|-----------------------|-------|
| GPT-4o / GPT-4.1      | 無料    |
| Claude Haiku 4.5      | 0.33倍 |
| Claude Sonnet 4 / 4.5 | 1倍    |
| Claude Opus 4.5       | 3倍    |
| Claude Opus 4.1       | 10倍   |

コストを抑えたい場合は、GPT-4oを使用することで無料でエージェントを実行できます。

## 12.87 Claude Code vs GitHub Copilot 比較

### 12.87.1 基本比較

| 観点      | Claude Code Skills      | GitHub Copilot Agents     |
|---------|-------------------------|---------------------------|
| 実行環境    | Claude Code CLI         | VS Code / GitHub.com      |
| 設定ファイル  | plugin.json + commands/ | .github/agents/*.agent.md |
| Git連携   | Bash経由                  | ネイティブ (worktree, PR作成)    |
| 並列実行    | 順次                      | Background/Sub-Agents     |
| コスト     | サブスクリプション               | Premium Requests          |
| 公式プラグイン | pr-review-toolkit等あり    | なし (自作が必要)                |

### 12.87.2 レビューエージェント対応表

| 目的        | Claude Code (pr-review-toolkit) | GitHub Copilot Agent          |
|-----------|---------------------------------|-------------------------------|
| コードレビュー   | code-reviewer                   | code-reviewer.agent.md        |
| サイレント失敗検出 | silent-failure-hunter           | error-handler-hunter.agent.md |
| コード簡素化    | code-simplifier                 | code-simplifier.agent.md      |
| コメント分析    | comment-analyzer                | comment-analyzer.agent.md     |
| テスト分析     | pr-test-analyzer                | test-analyzer.agent.md        |
| 型設計評価     | type-design-analyzer            | type-design-analyzer.agent.md |
| 包括的レビュー   | /review-pr                      | (複数エージェント組み合わせ)               |

どちらのツールを使っても、同等のレビュー機能を実現できます。

## 12.88 スキルの設計ポイント

### 12.88.1 1. 起動条件 (description) の重要性

スキルは「いつ起動するか」が重要です。

#### ## 12.89 良いdescription

「文書を初期化したいとき」「新規プロジェクトでspec-initを実行」

#### ## 12.90 悪いdescription

「プロジェクト設定」(曖昧すぎる)

明確な起動条件により、ユーザーが意図したタイミングでスキルが発動します。

### 12.90.1 2. 入力の明確化

スキルに必要な入力を明確にします。

## ## 12.91 入力

- プロジェクト名（必須）：リポジトリ名と同じ
- 技術スタック（必須）：言語、フレームワーク、DBを列挙
- 説明（任意）：1~2文で概要

### 12.91.1 3. 幕等性

スキルは何度実行しても同じ結果になるように設計します。

## ## 12.92 幕等性の確保

- ファイルが存在する場合は上書きしない（または確認する）
- 部分的に実行された場合も再実行で完了する

### 12.92.1 4. エラーハンドリング

失敗時の動作を定義します。

## ## 12.93 エラー時の動作

- 途中で失敗した場合、完了したステップを報告
- ロールバックが必要な場合は手順を提示
- 再実行方法を案内

### 12.93.1 うまくいかなかったスキルの事例

スキル設計で実際に起きた失敗例を紹介します。同じ轍を踏まないための参考にしてください。

#### 12.93.1.1 descriptionが曖昧で誤発動

あるチームが「code-review」というスキルを作りました。descriptionを「コードをレビューする」と書いたところ、「このコードをレビューして」以外にも、「このコードの意味を教えて」「このコードのバグを探して」など、意図しないタイミングで発動するようになりました。

解決策として、descriptionを「PRのコード変更をPATTERNS.mdに基づいてレビューする」と具体化しました。「PRの」「PATTERNS.mdに基づいて」という限定条件を入れることで、意図したタイミングでのみ発動するようになりました。

#### 12.93.1.2 入力が複雑すぎて使われない

別のチームが「full-spec-check」というスキルを作りました。入力として「対象ディレクトリ」「チェックレベル（厳密/標準/緩め）」「除外パターン」「レポート形式」「通知先」の5つを要求していました。

結果、誰も使いませんでした。毎回5つの入力を指定するのが面倒で、「手動でチェックした方が早い」となったのです。

解決策として、入力を「対象ディレクトリ」だけにし、他はデフォルト値を持つオプションにしました。「90%のケースはデフォルトで十分」という設計に変えたところ、利用率が上がりました。

#### 12.93.1.3 粒度が大きすぎて柔軟性がない

「deploy-all」というスキルは、テスト実行→ビルド→デプロイ→通知をすべて一気に実行するものでした。しかし、「テストだけ実行したい」「ビルドまで止めたい」というケースに対応できず、結局使われなくなりました。

解決策として、「run-tests」「build」「deploy」「notify」に分割し、必要な組み合わせで使えるようにしました。さらに、よく使う組み合わせを「deploy-standard」としてまとめることで、利便性と柔軟性を両立しました。

## 12.94 章末チェックリスト

### 12.94.1 共通

- プロジェクトで繰り返している作業を洗い出す
- 最もよく使う作業をスキル/エージェント化する
- description (起動条件) を明確に書く
- 複数視点レビューの導入を検討する
- スキル/エージェントをチームで共有する方法を決める

#### 12.94.2 Claude Code ユーザー

- pr-review-toolkitプラグインを導入する
- `/pr-review-toolkit:review-pr` をPR前に実行する習慣をつける
- 必要に応じてカスタムスキルを作成する

#### 12.94.3 GitHub Copilot ユーザー

- `.github/agents/` ディレクトリを作成する
- 本章の6つのエージェントテンプレートを導入する
- VS Code設定でカスタムエージェントを有効化する
- `@code-reviewer` などのエージェントをPR前に実行する習慣をつける

### 12.95 次章への橋渡し

この章では、Claude CodeとGitHub Copilotの両方で仕様駆動開発を自動化する方法を学びました。どちらのツールでも、同等のレビュー機能を実現できることがわかりました。

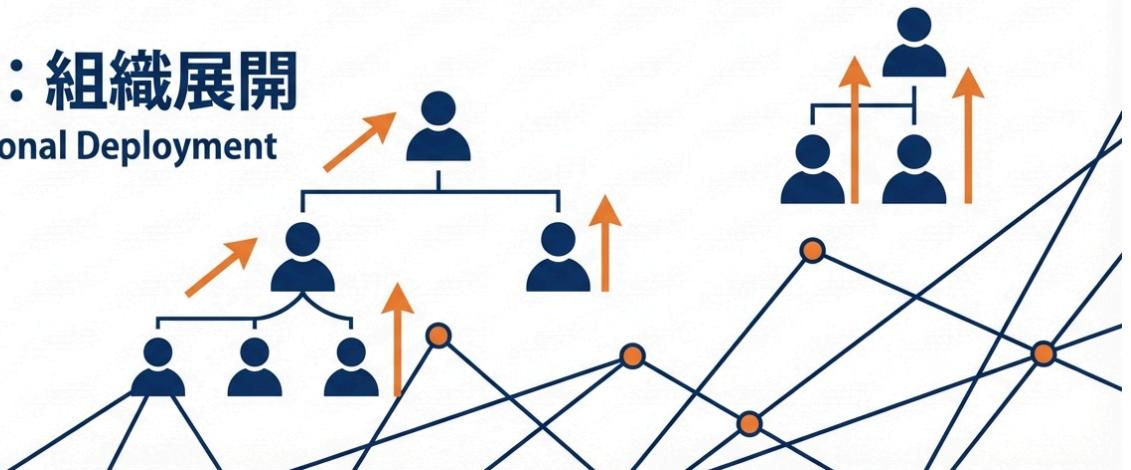
次章からは第5部「組織に展開する」に入ります。個人の取り組みをチーム・組織の標準にする方法を解説します。

## 第5部 組織に展開する

# 組織展開

## 第5部：組織展開

Organizational Deployment



個人からチームへ、チームから組織へ。レビューの重心を「コード」から「仕様」に移し、ナレッジを蓄積する仕組みを構築します。AI仕様駆動開発を組織の標準にするためのロードマップを示します。

# 第13章 チーム標準化：レビューの中心を「コード」から「仕様」

へ

## 13.1 この章で学ぶこと

- 仕様レビュー→タスク→実装レビューの順序
- 「MASTERが更新されていないPRは受け付けない」ルール
- チーム全体での運用定着

## 13.2 なぜレビューが重要なのか

コードレビューは、ソフトウェア開発において品質を担保する最後の砦です。

レビューには複数の目的があります。

- バグの早期発見：本番環境に出る前に問題を発見できる
- 知識の共有：チームメンバー間でコードの理解が広がる
- 設計の改善：第三者の視点で設計上の問題に気づける
- 標準化の維持：コーディング規約やパターンの一貫性を保つ

しかし、レビューには課題もあります。レビューの負担が大きく、レビューが形骸化しがちです。「LGTG (Looks Good To Me=問題なさそう)」と一言だけで承認されるレビューを見たことがある人も多いでしょう。

AI仕様駆動開発では、この課題に対してレビューの対象を変えることで対処します。コードだけでなく「仕様」をレビューすることで、より本質的な問題を早期に発見できるようになります。

## 13.3 レビューの順序を変える

### 13.3.1 従来のレビューフロー

実装 → コードレビュー → 修正 → マージ

↑

「これ、仕様と違うよね？」

「そもそもこの仕様でいいの？」

コードを書いてからレビューすると、根本的な指摘が後から出てきます。

その結果：

- 大幅な手戻り
- モチベーションの低下
- 時間の浪費

### 13.3.2 新しいレビューフロー

仕様レビュー → タスク化 → 実装 → 実装レビュー → マージ

↑

↑

「この仕様でいい？」

「仕様通りに書けてる？」

仕様を先にレビューすることで：

- 根本的な問題を早期発見
- 実装レビューは「仕様通りか」に集中
- 手戻りが最小化

## 13.4 仕様レビューの実践

### 13.4.1 仕様レビューの対象

| 対象    | レビュー観点                 |
|-------|------------------------|
| Issue | 受け入れ基準は具体的か、スコープは適切か   |
| 文書変更  | 影響度は正しいか、関連文書は更新されているか |
| ADR   | 決定理由は妥当か、代替案は検討されているか  |

### 13.4.2 仕様レビューのチェックリスト

#### ## 13.5 仕様レビューチェックリスト

##### ### 13.5.1 Issue

- [ ] 受け入れ基準が具体的で検証可能か
- [ ] 技術的制約が明示されているか
- [ ] スコープ外が明確か
- [ ] 関連文書へのリンクがあるか

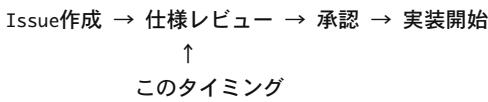
##### ### 13.5.2 文書変更

- [ ] 影響度 (LOW/MEDIUM/HIGH) が正しく評価されているか
- [ ] Frontmatter (version, updated) が更新されているか
- [ ] MASTER.md の索引が更新されているか
- [ ] 他の文書との矛盾がないか

##### ### 13.5.3 ADR

- [ ] 決定の背景が理解できるか
- [ ] 代替案が検討されているか
- [ ] 却下理由が妥当か
- [ ] 将来の参照に十分な情報があるか

## 13.5.4 仕様レビューのタイミング



実装を始める前にレビューを完了させます。

### 13.5.4.1 仕様レビューの工数感

「仕様レビューを追加すると、工数が増えるのでは？」という懸念があります。実際のところ、仕様レビューにかかる時間は以下の程度です。

- Issueのレビュー : 5~15分
- 文書変更のレビュー : 10~30分
- ADRのレビュー : 15~30分

一見すると「追加の工数」に見えますが、これは「コードレビューで発生していた仕様議論」が前倒しになっただけです。

あるチームで計測した結果、仕様レビューを導入する前はコードレビューに平均2.5時間かかっていました。導入後はコードレビューが平均1時間に短縮され、仕様レビュー（平均30分）を加えても合計1.5時間で済むようになりました。トータルでは40%の時間削減です。

さらに重要なのは、「根本的なやり直し」がなくなったことです。仕様レビュー前は月に2~3回「これ、そもそも設計からやり直し」という事態が発生していましたが、導入後はほぼゼロになりました。

## 13.6 「MASTERが更新されていないPRは受け付けない」ルール

### 13.6.1 なぜこのルールが必要か

7文書構成の中心はMASTER.mdです。

MASTER.mdが最新でないと：

- AIが古い情報を参照する
- チームメンバーが情報を見つけられない
- 文書が形骸化する

### 13.6.2 ルールの実装

#### 13.6.2.1 GitHub Actionsでの自動チェック

```
# .github/workflows/docs-check.yml
name: Documentation Check

on:
  pull_request:
    paths:
      - 'docs/**'
      - 'src/**'

jobs:
  check-master:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Check MASTER.md updated
        run: |
          # docs/配下が変更されている場合、MASTER.mdも変更されているか確認
          DOCS_CHANGED=$(git diff --name-only origin/${{ github.base_ref }} | grep '^docs/' | wc -l)
          MASTER_CHANGED=$(git diff --name-only origin/${{ github.base_ref }} | grep '^docs/MASTER.md' | wc -l)

          if [ "$DOCS_CHANGED" -gt 0 ] && [ "$MASTER_CHANGED" -eq 0 ]; then
            echo "::error::docs/ was changed but MASTER.md was not updated"
            echo "Please update MASTER.md to reflect the changes"
            exit 1
          fi

      - name: Validate Frontmatter
        run: |
          node scripts/validate-frontmatter.js docs/*.md
```

#### 13.6.2.2 PRテンプレートでの確認

```
<!-- .github/pull_request_template.md -->
```

## ## 13.7 チェックリスト

### ### 13.7.1 コード変更時

- [ ] 関連するテストを追加・更新した
- [ ] 既存のテストがすべてパスする

### ### 13.7.2 文書変更時

- [ ] MASTER.mdの索引を更新した
- [ ] Frontmatter (version, updated) を更新した
- [ ] 影響度を評価した (LOW/MEDIUM/HIGH)

### ### 13.7.3 すべてのPR

- [ ] 関連するIssueをリンクした
- [ ] レビュワーを適切に設定した

#### 13.7.3.1 チェックが形骸化するパターン

チェックリストやルールを導入しても、時間が経つと形骸化するケースがあります。典型的なパターンと対策を紹介します。

##### パターン1：全部チェックする習慣

チェックボックスを確認せずに全部チェックしてしまうケースです。「レビューがチェックするだろう」という意識が原因です。対策として、CIで自動検証できる項目は自動化し、チェックリストは「人間の判断が必要な項目」だけに絞ります。

##### パターン2：例外の常態化

「今回は急ぎだから」「この変更は軽微だから」と例外を認め続けると、ルールが有名無実化します。対策として、例外を認める条件を明文化し、例外を使った場合は記録に残します。例外が月3回を超したら、ルール自体を見直すトリガーにします。

##### パターン3：更新されない文書

MASTER.mdの更新がPRの通過条件になっていても、「更新しました」と最小限の変更だけして通過するケースがあります。対策として、文書の変更内容もレビュー対象に含め、「変更の意図が伝わる更新になっているか」を確認します。

形骸化の根本原因是、ルールの目的が共有されていないことです。「なぜこのルールがあるのか」をチームで定期的に確認する場を設けましょう。

## 13.8 チーム全体での運用定着

### 13.8.1 段階的な導入

一度にすべてを導入しようとすると失敗します。段階的に進めましょう。

#### 13.8.1.1 Phase 1 : MASTER.mdだけ始める (1週間)

- MASTER.mdを作成
- 「MASTER.mdを見れば、プロジェクトの全体像がわかる」状態を作る
- まずは1人が主導して更新

#### 13.8.1.2 Phase 2 : Issueテンプレートを導入 (2週間)

- 受け入れ基準・技術的制約・スコープ外のテンプレート
- 新規Issueはテンプレートに従う
- 仕様レビューの習慣化

#### 13.8.1.3 Phase 3 : 残りの文書を整備 (1ヶ月)

- ARCHITECTURE.md、PATTERNS.mdを優先
- 既存の暗黙知を文書化
- レビューで「PATTERNS.md参照」と言えるようにする

#### 13.8.1.4 Phase 4 : 自動チェックを導入 (1ヶ月)

- CI/CDに文書チェックを追加

- pre-commit フックの設定
- 違反時はマージブロック

### 13.8.2 抵抗への対処

#### 13.8.2.1 「ドキュメント書く時間がない」

回答：最初は時間がかかりますが、長期的には時間を節約します。

- 同じ質問に何度も答える時間が減る
- レビューでの根本的な指摘が減る
- 新メンバーのオンボーディングが速くなる

数値で示す：「過去1ヶ月で、仕様確認のやり取りに何時間かかったか」

#### 13.8.2.2 「形骸化するだけ」

回答：だから自動チェックを入れます。

- 更新されていないPRはマージできない
- 形骸化したら即座に検知される
- 「更新しないとマージできない」が習慣になる

#### 13.8.2.3 「既存プロジェクトには適用できない」

回答：一度に全部やる必要はありません。

- まずMASTER.mdだけ
- 新機能から7文書構成を適用
- 既存部分は触るときに少しずつ文書化

#### 13.8.2.4 段階的導入の成功事例

あるスタートアップ（エンジニア4名）での導入事例を紹介します。

最初の1週間は、リーダーが1人でMASTER.mdを作成しました。既存のSlackやNotionに散らばった情報を集め、「プロジェクトの全体像がわかる1ページ」を作りました。この時点では他のメンバーには「参照してね」と伝えるだけで、更新義務は課しませんでした。

2週目からIssueテンプレートを導入しました。最初は「テンプレートに沿って書くのが面倒」という声がありました。3週目には「受け入れ基準が明確だと実装が楽」という声に変わりました。仕様レビューも「Issueを書いた時点でSlackに投げる」という軽い運用から始めました。

1ヶ月目の終わりにPATTERNS.mdを整備しました。このとき、過去1ヶ月のコードレビューで繰り返し指摘されていた内容をリストアップし、「これはPATTERNS.mdに書いておこう」という形で文書化しました。「レビューで言われる前にPATTERNS.mdを読む」という習慣が生まれました。

2ヶ月目にCIチェックを導入し、3ヶ月目には「仕様レビューなしのPRは受け付けない」という運用が自然に定着しました。導入前と比べて、PRの平均レビュー往復回数は3.2回から1.8回に減少し、「仕様の認識違い」による手戻りは月平均5件からほぼゼロになりました。

## 13.9 定着の指標

### 13.9.1 定量指標

| 指標           | 目標   | 測定方法              |
|--------------|------|-------------------|
| 仕様レビュー実施率    | 100% | Issueのレビューコメント数   |
| MASTER.md更新率 | 100% | docs変更時のMASTER更新率 |
| PRの平均レビュー回数  | 2回以下 | PRの往復回数           |
| 仕様起因の手戻り     | 20%減 | 根本的変更を求めるコメント数    |

### 13.9.2 定性指標

- ・ 「あの仕様どこ？」という質問が減る
- ・ 新メンバーが自力でキャッチアップできる
- ・ レビューが「仕様通りか」に集中する

## 13.10 章末チェックリスト

- ・  仕様レビュー→実装→コードレビューの順序を決める
- ・  Issueテンプレートを作成する
- ・  PRテンプレートにチェックリストを追加する
- ・  CI/CDに文書チェックを追加する
- ・  段階的導入の計画を立てる

## 13.11 次章への橋渡し

この章では、チームでの標準化について学びました。

次章では、ロードマップとナレッジ蓄積—成長フェーズで追加すべき文書と、属人性を排除した知識の蓄積について解説します。

# 第14章 ロードマップとナレッジ蓄積

## 14.1 この章で学ぶこと

- ・ 成長フェーズで追加する文書
- ・ ドキュメント増加の管理戦略（分割・アーカイブ・月次チェック）
- ・ 属人性を排除した知識の蓄積方法
- ・ AIに「食わせられる」形でナレッジを残す

## 14.2 成長フェーズで増やす文書

### 14.2.1 7文書から始まる拡張

7文書は最小構成です。プロジェクトが成長するにつれて、追加の文書が必要になります。

#### 初期 (7文書)

```
|--- MASTER.md  
|--- PROJECT.md  
|--- ARCHITECTURE.md  
|--- DOMAIN.md  
|--- PATTERNS.md  
|--- TESTING.md  
|--- DEPLOYMENT.md
```

#### 成長期 (+α)

```
|--- GLOSSARY.md      ← 用語集  
|--- DECISIONS.md    ← 意思決定の履歴 (ADR集)  
|--- ROADMAP.md     ← 今後の計画  
|--- LESSONS_LEARNED.md ← 振り返りと教訓
```

### 14.2.2 GLOSSARY.md : 用語集

#### 14.2.2.1 なぜ必要か

- チーム内で用語の認識がズレる
- 新メンバーが専門用語を理解できない
- AIが用語を誤解する

システム開発において、開発メンバーが共通に使う言葉は用語集があると認識のズレを防げます。これはAIに対しても同じです。

たとえば、「リスト画面」なのか「一覧画面」なのか。プロンプトでの言い方を整理しておくと、AIへの指示が的確になります。用語が統一されていないと、AIが「そんな画面はないから新たに作る」といった誤った判断をしてしまうことがあります。

「ユーザー詳細画面」と「ユーザープロフィール画面」が同じものを指しているなら、用語集で「ユーザー詳細画面（＝ユーザープロフィール画面）」と明記しておくことで、AIは既存の画面を正しく認識できます。

#### 14.2.2.2 構造

# GLOSSARY.md

##### ## 14.3 ドメイン用語

###### ### 14.3.1 アクティブユーザー

status="active"のユーザー。ログイン可能、購入可能。

###### ### 14.3.2 サスペンデッドユーザー

status="suspended"のユーザー。ログイン不可、既存注文は継続処理。

###### ### 14.3.3 有効在庫

物理在庫 - 予約済み数。注文可能な実質在庫数。

##### ## 14.4 技術用語

###### ### 14.4.1 トークンリフレッシュ

アクセストークンの有効期限切れ時に、リフレッシュトークンを使って新しいアクセストークンを取得するプロセス。

###### ### 14.4.2 楽観的ロック

更新時にバージョン番号を比較し、競合を検出する方式。

本プロジェクトではORMのバージョンカラムで実装。

#### 14.4.2.1 用語集を更新するタイミング

用語集は作って終わりではなく、継続的に更新する必要があります。しかし、「いつ更新すべきか」が曖昧だと、いつの間にか陳腐化します。

用語集を更新すべきタイミングは以下の通りです。

- 新メンバーから質問があったとき：「この用語は何ですか？」という質問は、用語集に追加すべきサインです。
- コードレビューで用語の認識違いが発覚したとき：「アクティブユーザーってログイン中のユーザーのことでは？」といった指摘があれば、用語の定義を明確化します。
- AIが用語を誤解して実装したとき：AIが「有効在庫」を「すべての在庫」と解釈して実装した場合、用語集の説明が不足しています。
- 新しい概念がドメインに追加されたとき：新機能で新しいビジネス概念が登場したら、その時点で用語集に追加します。後回しにすると忘れます。

これらのタイミングをIssueテンプレートやPRテンプレートに組み込むと、更新漏れを防げます。「この変更で新しい用語を使っていますか？→ GLOSSARY.mdを更新してください」という項目を追加するだけで効果があります。

#### 14.4.3 DECISIONS.md：意思決定の履歴

##### 14.4.3.1 なぜ必要か

- ・「なぜこの技術を選んだか」が失われる
- ・同じ議論が繰り返される
- ・過去の決定を覆すときに根拠が必要

#### 14.4.3.2 構造

# DECISIONS.md

#### ## 14.5 意思決定一覧

| ID      | タイトル                | 日付         | 状態  |
|---------|---------------------|------------|-----|
| ADR-001 | [認証方式の選定](#adr-001) | 2024-01-15 | 承認済 |
| ADR-002 | [DB選定](#adr-002)    | 2024-01-20 | 承認済 |
| ADR-003 | [キャッシュ戦略](#adr-003) | 2024-02-01 | 承認済 |

---

#### ## 14.6 ADR-001: 認証方式の選定

##### ### 14.6.1 ステータス

承認済み (2024-01-15)

##### ### 14.6.2 コンテキスト

...

##### ### 14.6.3 決定

...

##### ### 14.6.4 理由

...

##### ### 14.6.5 代替案

...

#### 14.6.5.1 ADRを書く心理的ハードル

ADRの重要性はわかっていても、実際に書き始めると心理的なハードルを感じることがあります。

よくある抵抗感とその対処法を紹介します。

「完璧に書かなければ」という意識：ADRは完璧である必要はありません。「後から見てわかる程度」で十分です。決定の理由が1行でも書いてあれば、何も書いてないよりはるかに価値があります。

「小さな決定は書かなくていい」という判断：どの決定が「小さい」かは、後からでないとわかりません。迷ったら書く方を選びましょう。5分で書けるADRが、将来の1時間の議論を防ぐこともあります。

「書く時間がない」という現実：技術選定の議論に1時間かけて、ADRを書く10分がないということは通常ありません。議論が終わった直後、記憶が新鮮なうちに書くのが最も効率的です。後回しにすると、詳細を思い出すのに時間がかかります。

「誰も読まない」という諦め：最初はそうかもしれません。しかし、半年後に「なぜこの技術を選んだんだったっけ？」という質問が出たとき、ADRがあれば「ADR-003を読んで」で終わります。その1回のやり取りで、ADRを書いた時間は回収できます。

最初のADRは短くても構いません。「決定した技術」「なぜ選んだか（2-3行）」「何を却下したか」だけで始めてみてください。

#### 14.6.6 ROADMAP.md：今後の計画

##### 14.6.6.1 なぜ必要か

- ・全員が同じ方向を向く
- ・優先順位の判断基準になる
- ・AIに「今は何をすべきか」を伝えられる

#### 14.6.6.2 構造

# ROADMAP.md

##### ## 14.7 現在のフェーズ

MVP開発（～2024年3月）

##### ## 14.8 マイルストーン

###### ### 14.8.1 M1: MVP(2024年3月)

- [x] ユーザー認証
- [x] 商品一覧表示
- [ ] カート機能
- [ ] 決済連携

###### ### 14.8.2 M2: ベータ(2024年6月)

- [ ] レビュー機能
- [ ] お気に入り
- [ ] 検索最適化

###### ### 14.8.3 M3: 正式リリース(2024年9月)

- [ ] 管理画面
- [ ] 分析ダッシュボード
- [ ] パフォーマンス最適化

##### ## 14.9 今はやらないこと(Icebox)

- ソーシャルログイン（M2以降で検討）
- 多言語対応（正式リリース後）
- ネイティブアプリ（未定）

#### 14.9.1 LESSONS\_LEARNED.md : 振り返りと教訓

##### 14.9.1.1 なぜ必要か

- ・同じ失敗を繰り返さない
- ・成功パターンを再現する
- ・チームの成長を可視化する

##### 14.9.1.2 構造

# LESSONS\_LEARNED.md

##### ## 14.10 2024年Q1の教訓

###### ### 14.10.1 障害: 決済APIタイムアウト多発(2024-02-15)

###### #### 14.10.1.1 何が起きたか

決済APIのタイムアウトが1%→5%に急増。ユーザーから決済できない報告多数。

###### #### 14.10.1.2 原因

決済APIのレスポンスタイムが悪化したが、タイムアウト値が短すぎた(3秒)。

###### #### 14.10.1.3 対応

タイムアウトを10秒に延長、リトライ機能を追加。

#### ##### 14.10.1.4 教訓

- 外部APIのタイムアウトは余裕を持った設定に
- 外部依存の監視を強化する
- SLOを明確に定義しておく

#### ##### 14.10.1.5 反映先

- DEPLOYMENT.md: モニタリング設定に決済API監視を追加
- ARCHITECTURE.md: 外部API呼び出しポリシーに追記

## 14.11 ドキュメント増加の管理戦略

### 14.11.1 文書は必然的に増える

7文書から始まっても、開発が進むにつれて文書は増えていきます。

- ADR（技術選定の記録）
- ナレッジ（障害対応、トラブルシューティング）
- 詳細仕様（複雑な機能の設計書）
- 用語集、FAQ

これは問題ではなく、プロジェクトの成熟を示す健全な状態です。

問題は「増えること」ではなく、「管理されないまま増えること」です。

### 14.11.2 成長のフェーズ

Phase 1（初期）： 7文書  
↓ プロジェクト開始～MVP

Phase 2（成長）： 7文書 + ADR集約 + ナレッジ  
↓ MVP～運用開始

Phase 3（成熟）： 階層化（親文書 + 子文書）  
↓ 運用開始～

### 14.11.3 ファイルサイズの制限

AIツールにはコンテキストの制限があります。そのため、1ファイルあたりの行数を制限します。

| しきい値   | アクション |
|--------|-------|
| 500行超  | 分割を検討 |
| 800行超  | 分割を推奨 |
| 1200行超 | 分割を必須 |

#### 14.11.3.1 分割の方法

親文書（索引）と子文書（詳細）に分割します。

DEPLOYMENT.md (222行 - 索引)

```
|—— deployment/  
|   |—— git-workflow.md  
|   |—— ci-cd.md
```

```
|   └── infrastructure.md  
|   └── monitoring.md
```

親文書には概要と子文書へのリンクを記載します。

#### 14.11.4 分割のトリガー

| 条件        | アクション              |
|-----------|--------------------|
| 800行超過    | 子文書に分割             |
| ADR 10件超過 | DECISIONS.mdに分離    |
| ナレッジ蓄積    | 08-knowledge/配下に整理 |

#### 14.11.5 アーカイブ戦略

古くなった文書はarchive/ディレクトリに移動します。

```
docs/  
└── archive/           ← 古い文書を保管  
    ├── ADR-001-old.md  
    └── legacy-api.md  
├── MASTER.md  
└── ...
```

##### 14.11.5.1 アーカイブの基準

- 6ヶ月間参照されていない
- 技術的に陳腐化（例：廃止されたライブラリのADR）
- 別文書に統合された

##### 14.11.5.2 アーカイブの手順

1. Frontmatterに status: archived を追加
2. archive/ディレクトリに移動
3. MASTER.mdから参照を削除
4. 元の場所にリダイレクト用のメモを残す（任意）

#### 14.11.6 月次参照チェック

毎月、文書の参照状態を確認します。

##### 14.11.6.1 チェックリスト

###### ## 14.12 月次ドキュメント参照チェック(YYYY-MM)

###### ### 14.12.1 1. MASTER.mdからの参照確認

- [ ] 新規文書がMASTER.mdから参照されているか
- [ ] 参照リンクが有効か（リンク切れなし）
- [ ] 削除・統合された文書の参照が残っていないか

###### ### 14.12.2 2. ファイルサイズ確認

- [ ] 800行を超えた文書はないか
- [ ] 超えている場合、分割計画はあるか

###### ### 14.12.3 3. 鮮度確認

- [ ] 6ヶ月以上更新されていない文書はないか

- [ ] 更新されていない文書の内容は現状と合っているか

#### ### 14.12.4.4. 孤立文書の確認

- [ ] どこからも参照されていない文書はないか

### 14.12.4.1 自動化

リンク切れチェックをCIで自動化できます。

```
# .github/workflows/docs-link-check.yml
name: Docs Link Check

on:
  pull_request:
    paths:
      - 'docs/**'
      - '**/*.md'

jobs:
  link-check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Link Checker
        uses: lycheeverse/lychee-action@v2
        with:
          args: --verbose --no-progress 'docs/**/*.md'
          fail: true
```

## 14.13 "知見"をAIに食わせられる形で残す

### 14.13.1 属人化の問題

「あの人には聞かないとわからない」——これが属人化です。

属人化した知識は：

- 担当者がいないと進まない
- AIに伝えられない
- 引き継ぎが困難

### 14.13.2 構造化されたナレッジ

AIに「食わせられる」形とは、構造化されたテキストです。

#### ## 14.14 悪い例(属人化した知識)

「あれ、認証まわりは田中さんに聞いて」

#### ## 14.15 良い例(構造化された知識)

##### ### 14.15.1 認証トラブルシューティング

###### #### 14.15.1.1 症状:ログインできない

###### 1. JWTトークンの有効期限を確認

- 期限切れの場合：リフレッシュトークンで再発行
- リフレッシュトークンも期限切れ：再ログインを促す

## 2. Redisのセッション情報を確認

```
```bash
redis-cli get "session:${userId}"
```

## 3. 認証サービスのログを確認

```
kubectl logs -l app=auth-service | grep "auth error"
```

### ### 14.15.2 ナレッジ蓄積のトリガー

以下のタイミングでナレッジを蓄積します。

トリガー	蓄積先	内容
レビューで同じ指摘が3回	PATTERNS.md	コーディングパターン
障害対応後	LESSONS_LEARNED.md	教訓と対策
技術選定時	DECISIONS.md	決定理由と代替案
新メンバーからの質問	GLOSSARY.md	用語の説明
仕様の議論	DOMAIN.md	ビジネスルール

### ### 14.15.3 蓄積のワークフロー

問題発生・質問受領 | ▼ 「これ、前にも説明したな」 | ▼ 適切な文書に追記 | ▼ MASTER.mdの索引更新 | ▼ 次回は「〇〇.md参照」で済む

---

## ## 14.16 ナレッジの保守

### ### 14.16.1 陳腐化を防ぐ

ナレッジは放っておくと陳腐化します。

#### #### 14.16.1.1 定期レビュー

```
```markdown
```

### ## 14.17 ナレッジ保守スケジュール

#### ### 14.17.1 月次

- LESSONS\_LEARNED.md: 新しい教訓の追加
- ROADMAP.md: 進捗更新

#### ### 14.17.2 四半期

- 全文書のFrontmatterを確認
- リンク切れチェック
- 用語集の更新

#### ### 14.17.3 年次

- 全文書の棚卸し
- 古くなったADRの見直し
- 構造の再評価

#### 14.17.3.1 自動化

```
# .github/workflows/docs-review-reminder.yml
name: Docs Review Reminder

on:
  schedule:
    - cron: '0 9 1 * *' # 毎月1日9時

jobs:
  reminder:
    runs-on: ubuntu-latest
    steps:
      - name: Create Issue
        uses: actions/github-script@v7
        with:
          script: |
            github.rest.issues.create({
              owner: context.repo.owner,
              repo: context.repo.repo,
              title: '月次ドキュメントレビュー',
              body: '## 確認項目\n- [ ] LESSONS_LEARNED.md\n- [ ] ROADMAP.md\n- [ ] リンク切れチェック',
              labels: ['documentation', 'maintenance']
            })
}
```

## 14.18 章末チェックリスト

- 7文書以外に必要な文書を特定する
- GLOSSARY.md（用語集）を作成する
- ADRを書く習慣を始める（まずは1件）
- ナレッジ蓄積のトリガーを決める
- 定期レビューのスケジュールを設定する
- 800行を超えた文書があれば分割計画を立てる
- archive/ディレクトリを作成する
- 月次参照チェックをカレンダーに登録する

## 14.19 次章への橋渡し

この章では、成長に伴う文書拡張とナレッジ蓄積を学びました。

次章はおわりに——本書のまとめと、明日から始める最初の一歩を示します。

## おわりに

## 14.20 本書のまとめ

### 14.20.1 核心メッセージ

本書で最も伝えたかったことは、シンプルです。

AIに任せるために必要なのは、「魔法のプロンプト」ではなく「仕様というOS」

AIコーディングツールは、入力が曖昧なら曖昧な出力を返します。入力が明確なら、明確な出力を返します。

これはAIの限界ではなく、AIの仕様です。

だから、**入力を整える**——それが本書のアプローチです。

#### 14.20.2 3つの戦略

「はじめに」で提示した3つの戦略を振り返ります。

##### 戦略1：Issueでスコープを絞る

スコープを絞ることは、コンテキストエンジニアリングそのものです。AIが推測する余地をなくし、明確な境界を与えることで、プレのない実装を得られます。

##### 戦略2：PRは70%の完成度でいい

完璧を求めず、レビューを通じてスコープを収束させます。レビュー指摘は、AIへの次の入力として最適です。反復するたびに精度が上がります。

##### 戦略3：指摘をナレッジ化する

同じ指摘を繰り返さないために、ナレッジを蓄積し、自動チェックに組み込みます。これにより、プロジェクト固有の文脈をAIに学習させ続けることができます。

#### 14.20.3 7文書構成

3つの戦略を支える基盤が、7文書構成です。

文書	役割
MASTER.md	AIが迷子にならない地図
PROJECT.md	何を作るか、なぜ作るか
ARCHITECTURE.md	どう作るか
DOMAIN.md	ビジネスルール
PATTERNS.md	どう書くか
TESTING.md	何が正しいか
DEPLOYMENT.md	どう運用するか

これらが揃っていれば、AIは一貫した実装ができます。

### 14.21 次にやること

本書を読んだあなたが、明日から始める最初の一歩を示します。

#### 14.21.1 最初の1時間

##### 1. MASTER.mdを作る（30分）

プロジェクトのルートに `docs/MASTER.md` を作成し、以下だけ書きます。

# プロジェクト名

## 概要

[1~2文]

## 技術スタック

- 言語: [X]
- フレームワーク: [Y]
- DB: [Z]

#### ## ディレクトリ構造

[簡単な説明]

## 2. 次のタスクをIssue化する (30分)

今取り組もうとしているタスクを、Issueとして書き出します。

#### ## 受け入れ基準

- [ ] [具体的な動作1]
- [ ] [具体的な動作2]

#### ## スコープ外

- [今回やらないこと]

### 14.21.2 最初の1週間

#### 3. 1つの機能を「仕様→実装」で回す

Issue → AIに実装依頼 → 70%でPR → レビュー → マージ

この一連の流れを1回経験します。

#### 4. レビュー指摘をPATTERNS.mdに記録する

「次回から気をつけること」をPATTERNS.mdに書きます。

### 14.21.3 最初の1ヶ月

#### 5. 残りの文書を少しづつ整備する

- ARCHITECTURE.md (技術選定の理由)
- DOMAIN.md (主要なビジネスルール)
- TESTING.md (テスト方針)

#### 6. チームに共有する

「こういうやり方を試している」と共有し、フィードバックをもらいます。

## 14.22 最後に

「AIに任せる開発なんて無理」

この本を手に取る前、あなたもそう思っていたかもしれません。

でも、本当の問題は「AIが使えない」ことではありませんでした。

「入力が仕様になっていない」——これが根本原因でした。

仕様を整え、スコープを絞り、反復で精度を上げる。

このシンプルなアプローチで、AIは驚くほど頼れるパートナーになります。

「AIに任せる開発なんて無理」を、

「AIに任せたら、本当に楽になった」に。

その第一歩を、今日から踏み出してください。

## 14.23 著者について

[著者プロフィール]

## 14.24 謝辞

[謝辞]

## 14.25 参考文献

- [AI仕様駆動開発 公式ドキュメント](#)
- [GitHub Spec-Kit](#)
- [Microsoft Spec-Driven Development](#)

# 付録：AIエージェント設定ファイル一覧

**注意:** この情報は2026年1月6日時点の調査結果です。AIコーディングツールの仕様は頻繁に更新されるため、最新情報は各ツールの公式ドキュメントを確認してください。

## 14.26 AGENTS.mdとは

AGENTS.md は、AIコーディングエージェント向けのオープンスタンダードです。

- Linux Foundation傘下のAgentic AI Foundation が管理
- 60,000以上のリポジトリで採用（2026年1月時点）
- GitHub Copilot, Cursor, OpenAI Codex等の主要ツールが対応

AGENTS.mdは「AIエージェント向けのREADME」として設計されています。README.mdが人間向けの説明であるのに対し、AGENTS.mdはAIが効率的にコードを理解・生成するための指示を記述します。

### 14.26.1 公式リソース

- 公式サイト: <https://agents.md>
- GitHub: <https://github.com/agentsmd/agents.md>
- OpenAI Codexガイド: <https://developers.openai.com/codex/guides/agents-md/>

## 14.27 ツール別設定ファイル一覧

ツール	設定ファイル	AGENTS.md対応
Claude Code	CLAUDE.md	✗
GitHub Copilot	.github/copilot-instructions.md, AGENTS.md	✓
OpenAI Codex	AGENTS.md, SKILL.md	✓
Cursor	AGENTS.md, .cursor/rules/*.mdc	✓
Windsurf	AGENTS.md, Cascade Rules	✓
Gemini CLI	AGENTS.md	✓
Google Antigravity	~/.gemini/GEMINI.md, AGENTS.md	✓
Kiro (AWS)	.kiro/steering/*.md, AGENTS.md	✓

Cline	.clinerules	-
Factory/Droid	AGENTS.md, .factory/droids/*.md	✓

### 14.27.1 Claude Codeの注意点

Claude CodeはAGENTS.md標準に **ネイティブ対応していません。** CLAUDE.md のみを読み込みます。

ワークアラウンド: CLAUDE.md 内でAGENTS.mdを参照するよう記述すれば、間接的に読み込ませることは可能です。

#### # CLAUDE.md

このリポジトリのルールは AGENTS.md に記載されています。

実装前に必ず AGENTS.md を読んでください。

**Agent Skills:** Anthropicは独自の「Agent Skills」システムを開発しています。 SKILL.md ファイルを使用し、Claude.ai、Claude Code、Claude Agent SDKで利用可能です。AGENTS.mdがエージェントの要件・設計・セキュリティ等を定義するのに対し、Agent Skills/SKILL.mdはClaudeが実行時に動的に読み込むスキルモジュールです。両者は補完関係にあり、併用することで効果を発揮します。

複数のAIツールを併用する場合は、 CLAUDE.md と AGENTS.md の両方を用意することを推奨します。

### 14.27.2 Google Antigravityの特徴

Google Antigravity (2025年11月リリース) は、Googleが提供するエージェント型開発プラットフォームです。

- グローバル設定: ~/geminis/GEMINI.md に個人の設定を記述
- AGENTS.md対応: リポジトリルートのAGENTS.mdも読み込み可能
- アーティファクト機能: Google Docsスタイルのコメントでエージェントにフィードバック可能

### 14.27.3 Kiro (AWS) の特徴

Kiro (2025年7月リリース) は、AWSが提供する仕様駆動開発に特化したエージェント型IDEです。

- Steeringファイル: .kiro/steering/\*.md でプロジェクト固有のルールを定義
- AGENTS.md対応: AGENTS.mdも読み込み可能 (ただし inclusion modes は非対応)
- Spec駆動: requirements.md, design.md, tasks.md による仕様管理
- Foundation files: product.md, tech.md, structure.md でプロジェクトコンテキストを提供

## 14.28 7文書との連携

### 14.28.1 基本方針

AIエージェント設定ファイルには、7文書への参照を明記します。これにより、AIは自動的に仕様を参照してからコードを生成します。

### 14.28.2 Claude Code向け (CLAUDE.md)

#### # CLAUDE.md

#### ## 14.29 プロジェクト概要

このリポジトリでは docs/ 配下の7文書に仕様が記載されています。

#### ## 14.30 必読ドキュメント

実装前に必ず以下を読んでください :

- [docs/MASTER.md](docs/MASTER.md) - プロジェクト全体の索引

## 2. [docs/03-implementation/PATTERNS.md](docs/03-implementation/PATTERNS.md) - 実装パターン

### ## 14.31 コーディングルール

- マジックナンバー禁止（定数化必須）
- エラーハンドリングはResult型を使用
- テストカバレッジ80%以上を維持

### ## 14.32 Git運用

- コミットメッセージはConventional Commits形式
- PRはdevelopブランチにマージ

## 14.32.1 AGENTS.md対応ツール向け

### # AGENTS.md

### ## 14.33 Project Context

This repository uses a 7-document structure under docs/.  
Always read docs/MASTER.md before implementing any feature.

### ## 14.34 Build & Test

- Build: `npm run build`
- Test: `npm test`
- Lint: `npm run lint`

### ## 14.35 Coding Standards

- No magic numbers (use named constants)
- Error handling with Result type
- Maintain 80%+ test coverage

### ## 14.36 PR Guidelines

- Target branch: develop
- Use Conventional Commits format
- Include issue number in PR title

## 14.37 モノレポでの運用

AGENTS.mdはモノレポ構成にも対応しています。

各パッケージに個別のAGENTS.mdを配置すると、最も近いファイルが優先されます。

```
monorepo/
├── AGENTS.md          # リポジトリ全体のルール
├── packages/
|   ├── frontend/
|   |   └── AGENTS.md    # フロントエンド固有のルール
|   └── backend/
|       └── AGENTS.md    # バックエンド固有のルール
```

```
└── docs/
    └── MASTER.md
```

## 14.38 参考リンク

### 14.38.1 AGENTS.md関連

- [AGENTS.md 公式サイト](#)
- [GitHub agentsmd/agents.md](#)
- [OpenAI Codex AGENTS.mdガイド](#)
- [InfoQ: AGENTS.md Emerges as Open Standard](#)

### 14.38.2 各ツールの公式リリース・ドキュメント

ツール	公式リリース	ドキュメント
Claude Code	<a href="#">Claude 3.7 Sonnet and Claude Code</a>	<a href="#">Claude Code Docs</a>
GitHub Copilot	<a href="#">Coding Agent for GitHub Copilot</a>	<a href="#">GitHub Copilot Docs</a>
OpenAI Codex	<a href="#">Codex is now generally available</a>	<a href="#">Codex Changelog</a>
Cursor	<a href="#">Cursor 2.0</a>	<a href="#">Cursor Docs</a>
Windsurf	<a href="#">Windsurf Launch</a>	<a href="#">Windsurf Changelog</a>
Gemini CLI	<a href="#">Introducing Gemini CLI</a>	<a href="#">Gemini CLI Docs</a>
Google Antigravity	<a href="#">Build with Google Antigravity</a>	<a href="#">Antigravity Docs</a>
Kiro (AWS)	<a href="#">Kiro Launch</a>	<a href="#">Kiro Docs</a>
Cline	<a href="#">GitHub cline/cline</a>	<a href="#">Cline Docs</a>
Factory/Droid	<a href="#">Factory is GA</a>	<a href="#">Factory Docs</a>