## Before: 「個人の職人技」の限界

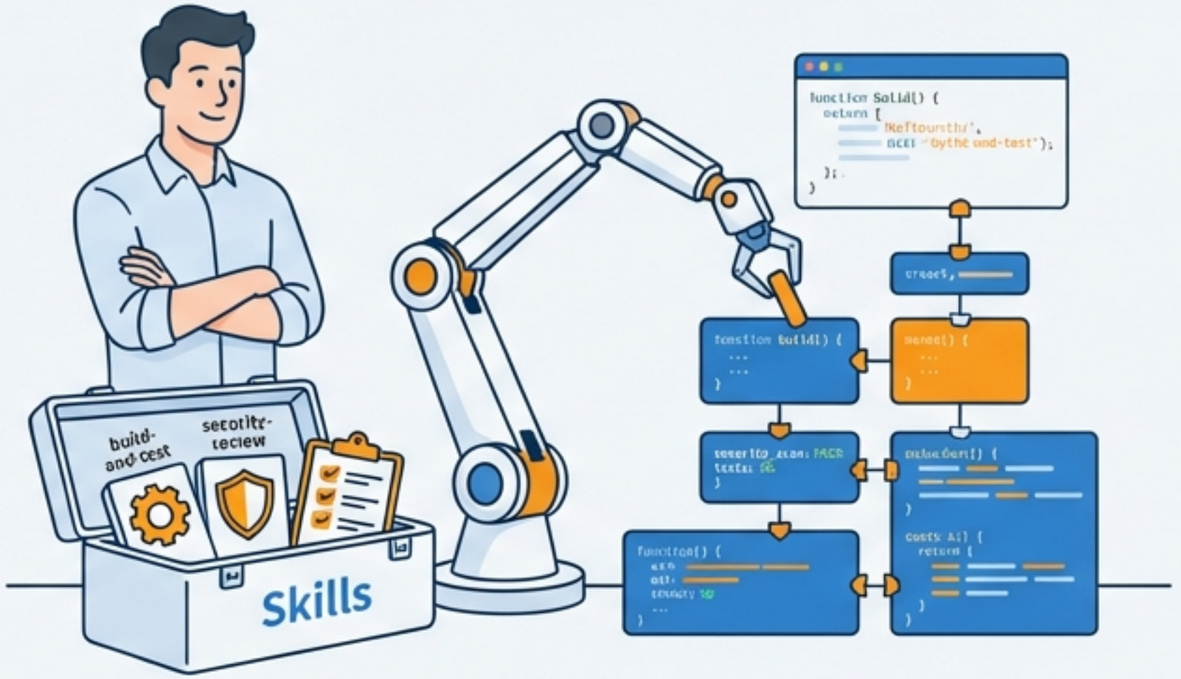

# なぜAI活用はスケールしないのか？

**現在のAI開発は、LLMの「ステートレス」な制約により、同じ指示の繰り返しや暗黙知の属人化がスケールを阻害している。**

- LLMはセッション間で記憶を保持しないため、プロジェクト固有のルールを毎回説明する必要がある。
- 手動での指示や限定的なコンテキストファイルだけでは、この「繰り返し」の問題を解決できない。
- 結果、AIへの指示が「個人の職人技」に依存し、チームとしての生産性が頭打ちになる。

# 解決の要約：Agent Skillsがもたらす変化

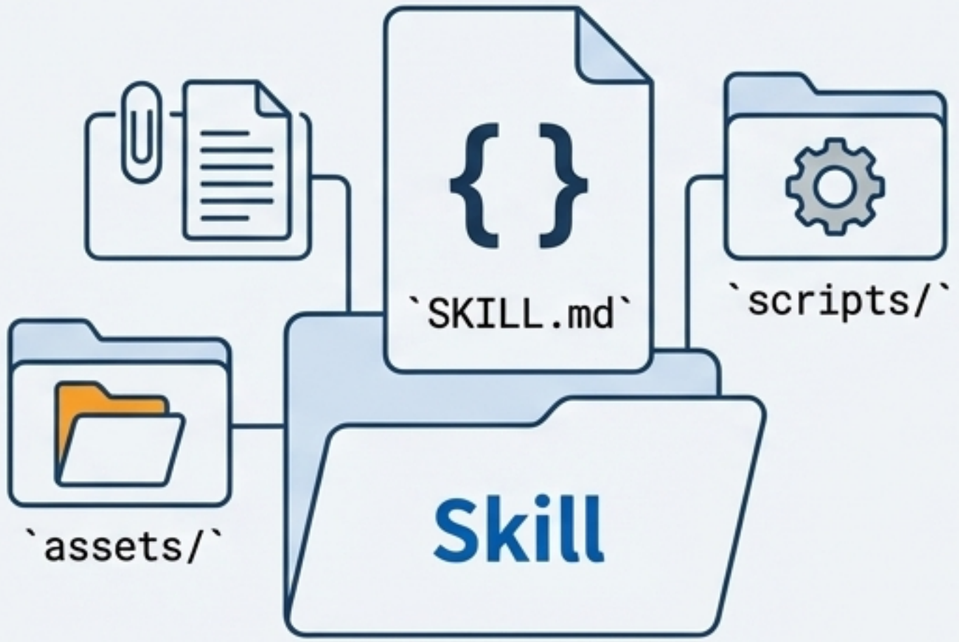
「個人の職人技」から
「チームの資産」へ




**Agent Skillsは、AIに「専門知識」をオンデマンドで注入し、
開発ワークフローを組織の「再利用可能な資産」へと変える。**

- 繰り返し必要な手続き的知識を「Skill」としてパッケージ化し、AIが必要な時に自律的に読み込む。
- 個人の暗黙知が、チームで共有・改善可能な形式知（バージョン管理されたドキュメント　）へと変換される。
- AIは単なるアシスタントから、プロジェクトの文脈を理解し一貫した振る舞いをする「チームの一員」へと進化する。

# Agent Skillsとは何か：1スライドで即理解

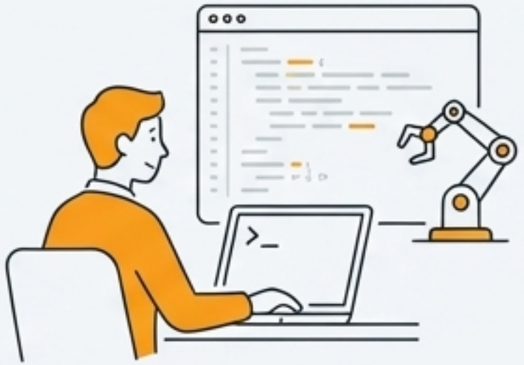



> 「新入社員向けの
> オンボーディングガイド」

**Agent Skillsとは、AIエージェント向けの「オンボーディングガイド」であり、
指示・スクリプト・リソースをまとめた再利用可能な知識パッケージである。**

- 実体は、`SKILL.md`という指示ファイルを中心としたシンプルなフォルダ構造。
- AIはタスクに応じて、利用可能な全Skillsの中から最適なものを自律的に発見し、その知識を活用する。
- AIの「使い方（How-to)」ではなく、「できること自体（Capability)」を拡張し、「どう振る舞うべきか（Behavior)」を定義する。

# 導入価値：エンジニアとマネージャ、それぞれのメリット



## For Engineers

### 反復作業の自動化とベストプラクティスの実践

- 面倒な手順（テスト、デプロイ、レビュー）を一度定義すれば、あとはAIに任せられる。
- チームのベストプラクティスを簡単に利用・実践できる。

## For Tech Lead / Manager

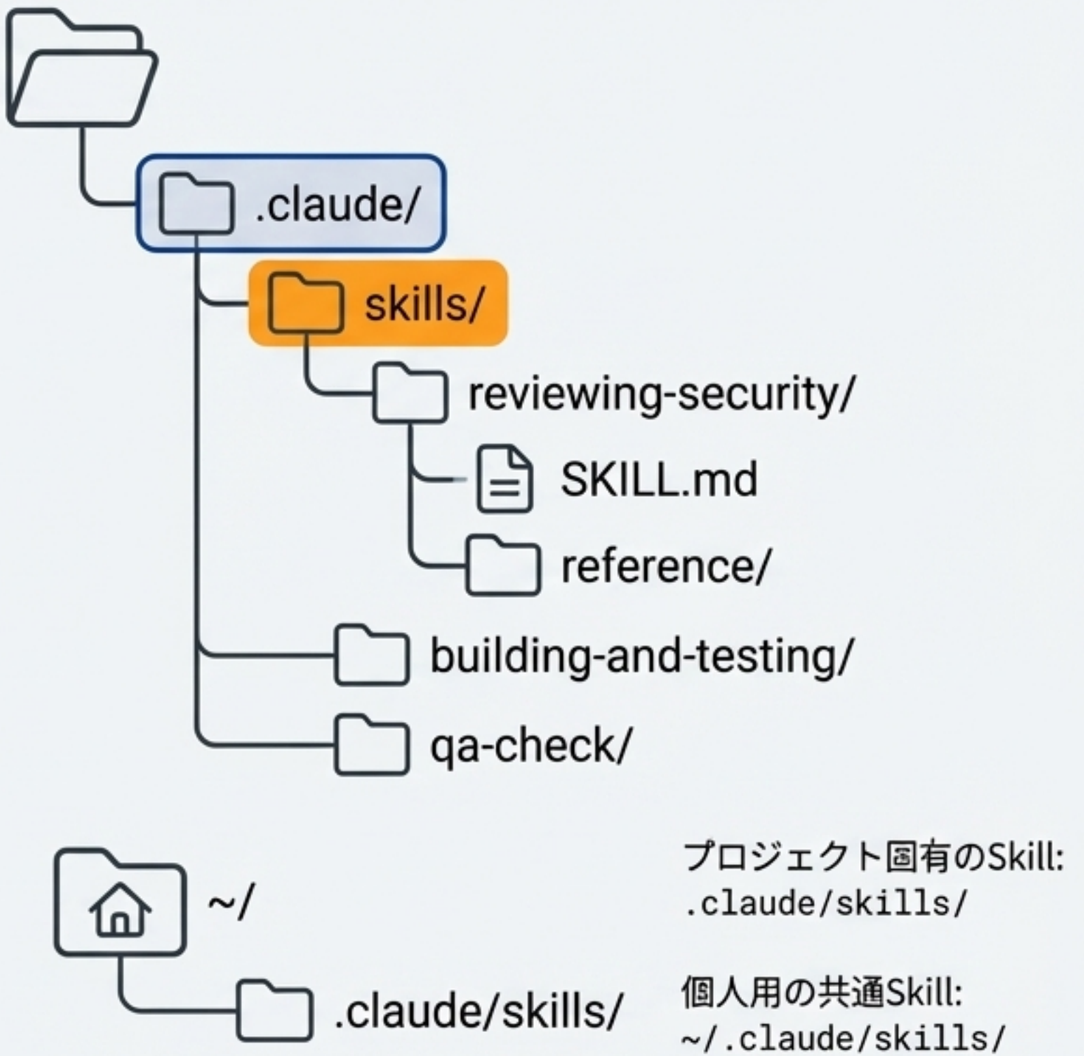
### チームのナレッジ資産化と品質の安定化

- 開発ワークフローをコードのようにバージョン管理し、チーム全体の生産性とアウトプット品質を底上げできる。
- 属人化しがちなノウハウを組織の資産として蓄積できる。

# 全体アーキテクチャ概要：ファイルシステムベースのシンプルさ

**Agent Skillsは、プロジェクトルートの`.claude/skills/`ディレクトリに配置するだけで、AIが自動認識するファイルシステムベースのシンプルなアーキテクチャである。**

- Skillの最小構成はSKILL.mdファイル1つ。Skillの最小構成はSKILL.mdファイル1つ。nameとdescriptionを持つYAML frontmatterが必須。
- プロジェクト固有のSkillは`.claude/skills/`、個人用の共通Skillは`~/.claude/skills/`に配置してスコープを管理する。
- 補助的なスクリプトはscripts/、ドキュメントやテンプレートはassets/やreference/に配置する。



# 機能の使い分け：Skills / Commands / Hooks / Subagents

各機能は明確に役割が分かれており、Skillsは「専門知識の注入」、
Subagentsは「コンテキストの分離」が重要な判断基準となる。

| 機能名 | 役割 | コンテキスト | ユースケース例 |
| --- | --- | --- | --- |
| **Skills** | **専門知識の注入**<br>AIの振る舞いを定義 | **親と共有** | 継続的な対話作業<br>(TDD、リファクタリング) |
| **Subagents** | **コンテキストの分離**<br>専門タスクを委任 | **親から独立** | 試行錯誤が多い調査<br>(エラー調査、リサーチ) |
| **Commands** | **手動ショートカット**<br>ユーザーが明示的に実行 | ユーザーが起動 | 定型的な一連のコマンド実行<br>(/test-and-commit) |
| **Hooks** | **決定的な処理**<br>特定イベントで必ず実行 | イベントに紐づく | 品質ゲート<br>(コミット前の自動フォーマット) |

# Progressive Disclosure：賢い知識アクセスの設計思想

Agent Skillsは、3段階の「段階的開示」アーキテクチャにより、コンテキスト消費を最小限に抑えつつ大規模な知識の活用を可能にする。

**Level 1: メタデータ**
起動時に全Skillの`name`と`description`のみをロード。AIは「何ができるか」の全体像を把握。

約100トークン/Skill

**Level 2: 指示**
タスクに関連するSkillがトリガーされた時、`SKILL.md`本文をロード。
関係ないSkillは読み込まない

**Level 3: リソース**
`SKILL.md`内の指示に基づき、必要なスクリプトや参照ファイルを追加でロード。

Just-In-Time戦略

**推論空間の段階的絞り込み**

# BDDとの対応関係：人とAIの共通契約

Skillsは、振る舞い駆動開発（BDD）と同様の構造を持ち、AIエージェントの「振る舞い」を自然言語で定義・保証する「人とAIの共通契約」として機能する。


プログラミング不要で、誰もがAIの振る舞いを定義・レビューできる。

UD Shin Go NT

# 実践例①：小さなSkillの積み重ね

**定型的なビルドやテストのような単純な連続コマンドも、Skill化することで「テストして」という自然言語指示だけで毎回正確に実行させられる。**

```
# building-and-testing/SKILL.md
---
name: building-and-testing
description: "Rustプロジェクトのビルドとテスト実行。フォーマット、lint、ユニットテスト、ビルド確認を一括実行。Use when: ビルド，テスト，チェックを依頼された時。"
---
# ビルドとテスト

## 実行手順
1. フォーマットチェック: `cargo fmt --check`
2. Lint実行: `cargo clippy -- -D warnings`
3. ユニットテスト: `cargo test --workspace`
4. ビルド確認: `cargo build --workspace`

## 一括実行
```bash
cargo fmt --check && cargo clippy -- -D warnings && cargo test --workspace && cargo build --workspace
```

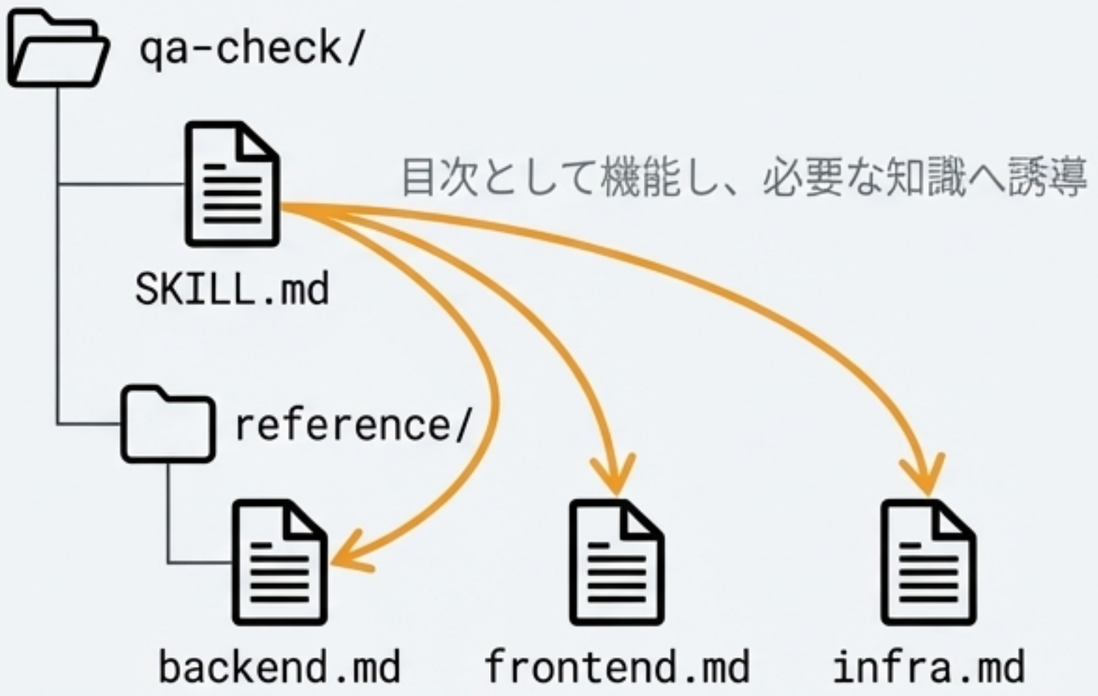
AIが起動タイミングを判断するためのトリガーワードを記述。

単純なコマンドの連続も、一つのSkillとしてパッケージ化。

- **このような小さなSkillの積み重ねが、日々の開発における「小さな手間」を確実に削減する。**
```

# 実践例②：品質・レビュー系Skill

**段階的開示（Progressive Disclosure）を活用すれば、レビュー対象に応じて必要なチェックリストだけを読み込ませる、効率的で高精度なレビューSkillを構築できる。**




### reference/backend.md の抜粋

```
# Rust バックエンド QAチェック項目

## エラーハンドリング
- [ ] `unwrap()`を本番コードで使用していないか
- [ ] `Result`型を適切に伝播しているか

## セキュリティ
- [ ] SQLインジェクション対策（パラメータ使用）
- [ ] 機密情報のログ出力がないか

## パフォーマンス
- [ ] N+1クエリが発生していないか
- [ ] 不要な`clone()`がないか
```

**「バックエンドのコードをレビューして」という指示で reference/backend.md のみがロードされ、コンテキストを節約しつつ専門的なレビューが実行される。**

# 実践例③：要件定義〜設計のSkillセット

複数のSkillを組み合わせることで、要件定義、システム設計、設計レビューといった複雑な開発フェーズ全体を体系化し、成果物の品質を標準化できる。

**要件定義 Skill**
抜け漏れのない
要件定義を支援

→

**システム設計 Skill**
アーキテクチャ設計
プロセスをガイド

→

**設計レビュー Skill**
NFRやセキュリティ観点
でのレビューを構造化

`srs-template.md`

`user-story-template.md`

`architecture-spec-template.md`

`adr-template.md`

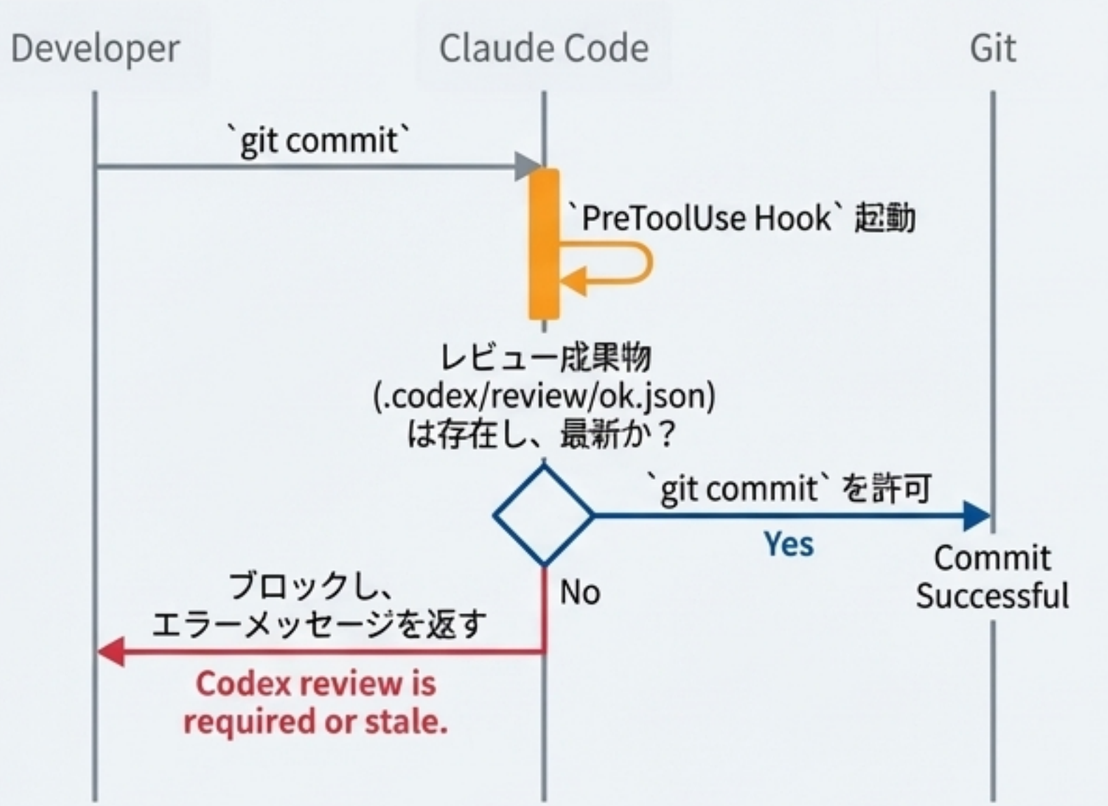
`design-review-checklist.md`

`risk-register-template.md`

# Hooks連携による「推奨」から「必須」へ

**Skillsが「推奨される手順」を定義するのに対し、Hooksを組み合わせることで、特定のワークフロー（例：コミット前のコードレビュー）を「必須」の品質ゲートにできる。**

- Skillsはモデルが自律的に判断して起動するため、実行が100%保証されるわけではない。
- `PreToolUse` Hookは、`git commit`のような特定ツール実行前に割り込み、条件を満たさない場合は実行をブロックできる。
- 「Skillでレビュー手順を定義」し、「Hookでレビュー成果物の存在をチェック」することで、品質ゲートを自動化する。



# ベストプラクティス：効果的なSkillの3原則

## 簡潔に書く (Concise is key)

AIが既に知っていることは書かず、トークンとAIの思考ノイズを減らす。SKILL.mdは500行未満を推奨。

## 自由度を適切に設定 (Set appropriate degrees of freedom)

リスクの高い作業（DBマイグレーション）は手順を厳密に、創造性が求められる作業（コードレビュー）は自由度を高く設定する。

## フィードバックループを入れる (Implement feedback loops)

長いワークフローでは検証ステップ（例: validate.pyの実行）を挟み、早期にエラーを検出・修正させる。

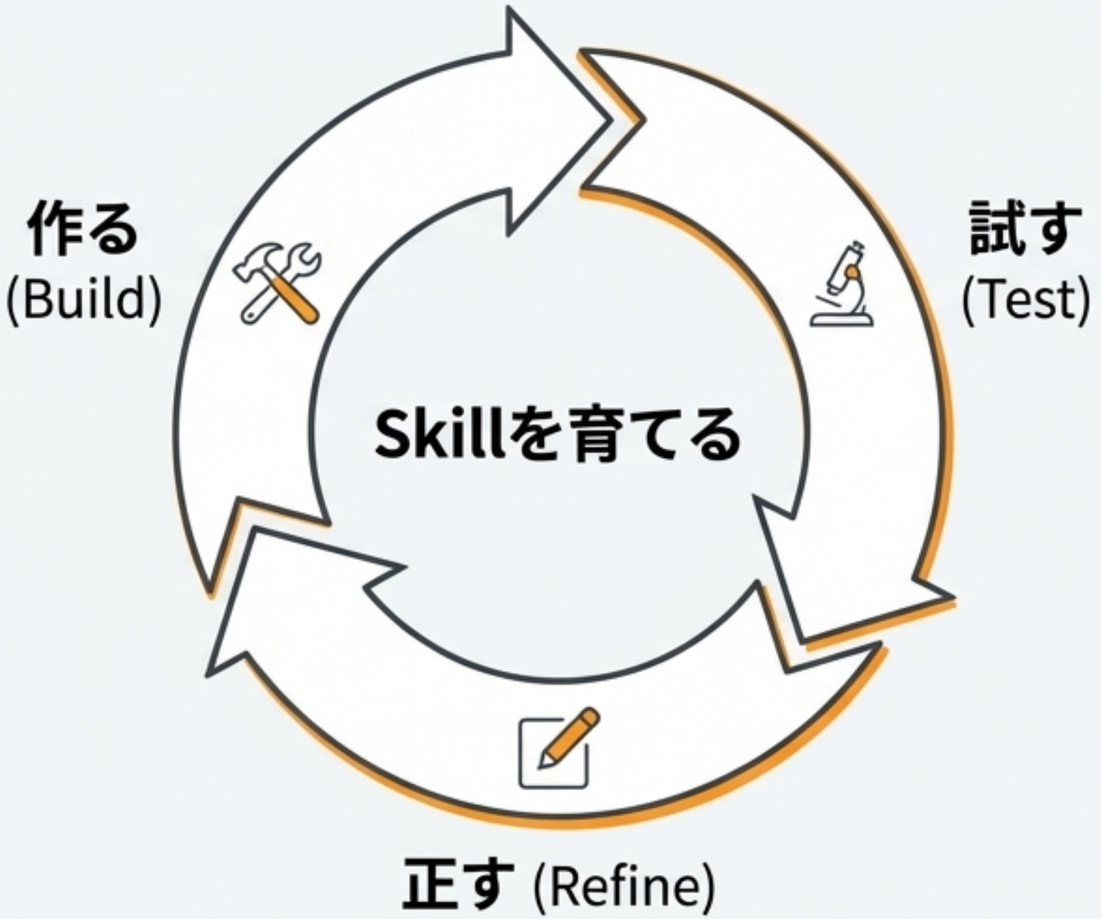
# よくある失敗と制約：現実的な課題

**Skill開発は、`description`の試行錯誤やデバッグの難しさといった固有の課題があり、プロンプトエンジニアリングに近いスキルが求められる。**


- **起動しない**
  - **原因：**`description`が曖昧でAIがタスクとの関連性を見出せない。
  - **対策：**トリガーワードの追加や「いつ使うか」を明記する。
- **デバッグが難しい**
  - **原因：**「なぜこのSkillが選ばれなかったか」の判断過程がブラックボックス。
  - **対策：**試行錯誤が基本。最小限の変更で挙動を確認する。
- **Skill同士の競合**
  - **原因：**似た`description`を持つSkillが増えると、意図しない方が起動する。
  - **対策：**Skillの責務を明確に分離し、`description`の重複を避ける。

# 成長サイクル：「作る・試す・正す」で育てる

**Skillsは一度作って完成ではなく、「作る→試す→正す」のサイクルを通じて、チームと共に継続的に育てるアジャイルな「組織知の実験装置」である。**




- 完璧な設計を目指すより、まずは最小限のSkillで始め、実際のタスクで試すことが重要（評価駆動開発）。
- 「期待通りに動かない」というフィードバックこそが、`description`や手順を改善し、LLMの振る舞いを理解する最良の機会となる。
- このサイクルを回すことで、Skillは徐々にチームのワークフローに最適化された、価値の高い資産へと成長する。

# 開発プロセスの未来像：ワークフローの資産化

Agent Skillsは、AIとの協働を「プロンプトの職人技」から
「ワークフローの資産化」へとパラダイムシフトさせる。

## Before

### 個人（暗黙知）

良いプロンプト、効率的な手順など、
個人の頭の中にあったノウハウ

**Agent Skills**

`SKILL.md`による形式知化

## After

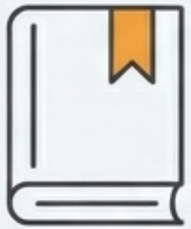
### チーム（形式知）

チーム全体でレビュー、改善、
再利用が可能な組織の資産

今後のAIエンジニアリングは、「LLMにどう推論させるか」ではなく、
「LLMの推論をどう制約し、組織の資産として 蓄積するか」が中心となる。

# まとめ：3つの要点

**Agent Skillsは、AI開発のスケール課題を解決し、ワークフローを資産化する、育てるべき仕組みである。**

1. 

## What

AI向けの「オンボーディングガイド」で、ステートレスなLLM に専門知識を注入する。

2. 

## Why

同じ指示の繰り返しを防ぎ、暗黙知をチームの形式知（資産）に変え、開発をスケールさせるため。

3. 


## How

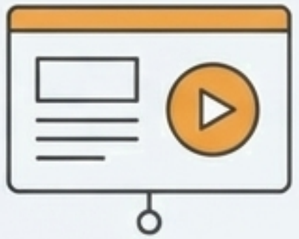
「作る・試す・正す」サイクルを回し、チームのワークフローと共に継続的に改善していく。

# 今日からできること：具体的なToDo

まずはビルトインSkillを体験し、5分で最小のカスタムSkillを作成することで、その価値を即座に実感できる。

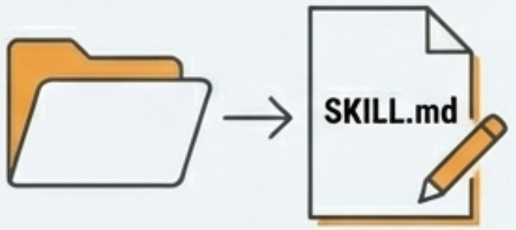
## 1. ビルトインSkillを体験する（1分）

claude.aiで「PowerPointで自己紹介スライドを作って」と依頼してみる。



## 2. 最小のカスタムSkillを作る（5分）

~/.claude/skills/に`hello-skill/SKILL.mdを作成し、「挨拶して」と依頼してみる。




## 3. コミュニティの作例を眺める（10分）

GitHubの`awesome-claude-skills`リポジトリを覗き、実践的な書き方を参考にする。