Claude Codeで開発工数を削減する実践方法



Start Challenge
Consulting co., ltd.
挑む力が、世界を拓らす

Start Challenge Consulting co., ltd.

設計・実装・テスト・レビューを効率化するAI駆動開発ガイド：／：発行：2026年8月

Confidential ／ 社外秘

はじめに

エンジニア不足と開発スピードへの要求が同時に強まるなか、生成AIを「コード補完」にとどめず、調査・設計・テスト・レビューを含む開発プロセス全体に組み込むことが工数削減の前提となる。

本資料の背景

開発現場で増加する業務負荷

機能追加・保守・障害対応が同時並行で進み、一人あたりが担う作業範囲と切り替えコストが増加している。

エンジニア不足と開発スピードへの要求

人員を増やす前提が成り立たない一方で、リリース頻度と対応速度への要求は高まり続けている。

生成AIを「コード補完」だけで終わらせない

補完による部分的な時間短縮ではなく、既存コードの調査・実装計画・テスト・レビューまでを対象にすることで、工程全体の工数に効果が及ぶ。

本資料の目的

Claude Codeを用いて開発工数を削減するための、環境準備・8つの実践方法・指示設計・ガバナンス・効果測定を体系的に整理する。

対象読者

開発責任者・CTO

投資判断とガバナンス方針の決定

プロジェクトマネージャー

工数計画・進捗管理への反映

エンジニアリングマネージャー

開発プロセスとレビュー体制の設計

導入を検討している開発者

日々の実務への具体的な適用方法

本資料は特定の製品導入を推奨するものではなく、開発プロセスへの組み込み方を整理した実務ガイドである。

注記　事実：本資料の対象読者・目的の定義／解釈：開発現場の負荷構造に関する記述は一般的な開発工程の整理に基づく。

開発工数が増大する主な原因

工数増大の要因は個々の実装作業ではなく、既存コードの読解・仕様との突合・検証の後回しといった、工程をまたいで繰り返される非生産的な作業に集中している。

調査	設計	実装	テスト	レビュー	保守
既存コードを理解するまでに時間がかかる	仕様書と実装内容が一致していない	定型的なコードを繰り返し作成している	テストコードの作成が後回しになる	コードレビューが特定メンバーに集中する	エラー調査や影響範囲の確認に時間を要する
工数への影響 **読解時間の増加**	工数への影響 **手戻りの発生**	工数への影響 **反復作業の増加**	工数への影響 **品質リスクの蓄積**	工数への影響 **着手待ちの発生**	工数への影響 **原因特定の長期化**

工程を横断して繰り返される構造要因

属人化した知識

仕様と設計の経緯が個人に蓄積し、他者が理解するまでに時間を要する。

手作業の反復

類似コードの作成と転記が繰り返され、付加価値の低い作業に時間が割かれる。

検証の後回し

テストとレビューが後工程に集中し、不具合の発見が遅れて手戻りが増える。

注記　事実：工程別の要因は本構成に基づく整理／解釈：「工数への影響」および3つの構造要因は要因からの導出であり、実測に基づく数値ではない。

Claude Codeとは

Claude Codeはリポジトリ全体を横断して把握し、調査から実装・テストまでを一連で実行するエージェント型ツールであり、単一ファイル内の補完を行う従来ツールとは適用範囲が異なる。

定義と主な特徴

Anthropicが提供する**エージェント型のコーディングツール**。ターミナルから自然言語で指示を与え、リポジトリ内のコードを横断的に読み取りながら、開発作業を一連の流れとして実行する。

1. ターミナルから自然言語で開発作業を指示
2. リポジトリ内の複数ファイルを横断的に把握
3. コードの作成・修正・テスト・調査を一連で実行
4. Gitや各種開発ツールとの連携

一般的なコード補完ツールとの違い

観点	一般的なコード補完ツール	Claude Code
作業単位	行・関数単位の補完	タスク単位の実行
参照範囲	編集中のファイルが中心	リポジトリ全体を横断
実行内容	コードの提案	調査・実装・テスト・修正
対象工程	実装工程が中心	調査から保守まで
人の関与	提案の採否を都度判断	計画と結果を確認・承認

両者は代替関係ではなく、**補完ツールは実装時の入力効率**、**Claude Codeは工程横断の作業効率**を対象とする。工数削減の余地が大きいのは、実装以外の工程を含めた後者である。

注記　事実：Claude Codeの提供元・基本機能に関する記述／解釈：比較表の観点整理および両者の位置付けは本資料による整理。最新の機能仕様は提供元の公式ドキュメントを参照のこと。

Claude Codeで削減できる開発工数

効果は実装工程だけでなく調査・設計・テスト・レビュー・保守の全工程に及び、なかでも既存コードの読解と定型作業の比重が大きい工程で削減余地が大きい。

開発工程	活用例	期待できる効果
調査	既存コードの構造分析	**読解時間の短縮**
設計	実装方針・変更範囲の整理	**設計漏れの防止**
実装	機能追加・定型コード生成	**コーディング時間の短縮**
テスト	テストケース・テストコード生成	**テスト作成の効率化**
レビュー	差分・問題点の確認	**レビュー負荷の軽減**
保守	エラー原因の特定	**調査・修正時間の短縮**

記載の効果は、**Git・テスト・Lint・型チェックによる自動検証環境が整備されていること**を前提とした想定である（仮説）。導入効果は前提条件の整備状況とタスクの性質により変動するため、P17の測定方法に沿って自組織で検証することを推奨する。

注記　事実：工程・活用例の整理／仮説：「期待できる効果」は導入前提が満たされた場合の想定であり、実測値ではない（仮）／提案：導入前後の実測による検証。

導入前に準備すべき開発環境

AIが生成した変更を機械的に検証できる環境（Git・テスト・Lint・型チェック）を先に整えることが、Claude Code導入の成否を分ける。

対象リポジトリと利用範囲

どのリポジトリ・どの作業でClaude Codeを利用するかを事前に定義し、対象外の範囲も明示する。

Gitによるバージョン管理

変更差分を常に追跡でき、問題があれば即座に巻き戻せる状態を維持する。

テスト・Lint・型チェック

生成された変更を人手を介さず検証できる仕組みを、実行コマンドとともに整備する。

開発環境の再現手順

誰でも同じ環境を構築・実行できる手順を用意し、環境差異による検証不能を防ぐ。

機密情報の取扱いルール

入力してはならない情報と認証情報の保管方法を明文化し、周知する。

AIと人の役割分担

AIに任せる作業と、人が判断・承認する作業の線引きをあらかじめ定める。

POINT

Claude Codeの導入前に、生成された変更を自動で検証できる環境を整えておく。検証の自動化なくして、工数削減は成立しない。

注記　事実：準備項目の整理／提案：導入判断の前提として、6項目の整備状況をチェックリスト化し、未整備項目は導入対象から除外する。

既存コードの調査を効率化する

読解に費やしていた時間を、構成・処理フロー・依存関係の説明として出力させることで短縮し、同時に属人化した知識をドキュメントとして残す。

主な活用場面

01 ディレクトリ構成と主要コンポーネントの把握

02 処理フローやデータの流れを説明させる

03 特定機能に関係するファイルの抽出

04 使用されているライブラリや依存関係の整理

05 変更時に影響を受ける箇所の洗い出し

06 属人化したコードのドキュメント化

指示例

> 「この機能の処理開始からデータ保存までの流れを、関連ファイルとともに説明してください」

対象機能と出力してほしい粒度（関連ファイル名を含める等）を明示することで、調査結果をそのまま設計・実装の入力として使える。

得られる成果物

- 機能単位の処理フロー説明
- 関連ファイル・依存関係の一覧
- 変更時の影響範囲メモ（設計の入力）

注記　事実：活用場面・指示例は本構成に基づく／提案：調査結果は必ず人が実コードと突合し、誤った説明をそのまま設計に持ち込まないこと。

実装前の計画を作成させる

いきなりコードを書かせず、タスク分解・変更対象・検証方法を計画として先に出力させ、人が確認してから実装に進むことで手戻りを抑制する。

計画に含めるべき内容

01 要件をタスク単位に分解

02 変更対象となるファイルを特定

03 既存設計との整合性を確認

04 実装手順と検証方法を整理

05 想定されるリスクや例外処理を抽出

計画レビューで人が確認する観点

変更対象ファイルに**過不足がないか**

既存の設計方針・命名規約と**矛盾しないか**

検証方法が**自動テストで確認可能か**

影響範囲に**未検討の領域がないか**

計画段階での指摘は、実装後の修正に比べて修正コストが小さい。

推奨フロー



注記　事実：推奨フローの各ステップ／提案：「人による確認」を省略可能な例外条件は事前に定義し、原則として承認ゲートを飛ばさない運用とする。

定型的な実装を自動化する

繰り返し発生する定型実装は削減効果が最も明確な領域であり、既存実装を参照させて生成することでコードの統一性を維持したまま時間を短縮できる。

自動化しやすい定型実装

01	CRUD処理の作成
02	APIエンドポイントの追加
03	データモデル・型定義の作成
04	入力チェック処理の実装
05	ログ・例外処理の追加
06	既存コードのパターンに合わせた機能実装

注意点

ゼロから自由に生成させるのではなく、既存実装を参照させて統一性を保つ。

参照元を指定しない生成は、同一リポジトリ内に異なる設計パターンを混在させ、結果として保守工数を増加させる。

統一性を保つための指示要素

- 参照する既存ファイルを明示する
- 命名規約・ディレクトリ配置を指定する
- エラー処理・ログ出力の既定方式を伝える
- 生成後にLint・型チェックの実行を指示する

注記　事実：定型実装の分類／解釈：削減効果が明確とする判断は、作業の反復性と検証容易性に基づく整理であり、実測値ではない。

複数ファイルの修正を効率化する

リポジトリ全体を横断できる特性により、人手では参照漏れが生じやすい複数ファイルにまたがる変更を、影響範囲の確認まで含めてまとめて実施できる。

横断的な修正が発生する場面

01	API変更に伴うフロント・バックエンドの同時修正
02	データベース項目追加に伴う関連コードの更新
03	関数名や型定義の一括変更
04	共通処理のコンポーネント化
05	古いライブラリやAPIの移行
06	修正後の参照漏れ・影響範囲の確認

効果

人手では漏れやすい横断的な変更を、参照箇所の特定から修正・確認までまとめて実施できる。

実施時の進め方

1. 変更対象と参照箇所を先に一覧化させる
2. 一覧を人が確認し、対象外を明示的に除外する
3. 修正を実行し、テスト・型チェックで検証する
4. Git差分で変更範囲が想定内かを確認する

大規模な一括変更は、コミット単位を分割して差分を追跡可能に保つ。

注記　事実：横断修正が発生する場面の整理／解釈：効果の記述はリポジトリ横断参照という機能特性からの導出であり、削減率を示すものではない。

テストコードを自動生成する

後回しになりがちなテスト作成を、観点抽出から異常系・境界値の生成まで委ねることで効率化できる一方、実装を追認するだけのテストになっていないかの確認が不可欠となる。

テスト工程での活用範囲

01	既存コードからテスト観点を抽出
02	正常系・異常系・境界値のテスト作成
03	単体テストコードの生成
04	モック・テストデータの作成
05	不足しているテストケースの洗い出し
06	テスト失敗時の原因調査と修正案の提示

重要ポイント

生成されたテストが「現在の実装を追認するだけ」になっていないか確認する。

実装の挙動をそのまま期待値にしたテストは、バグが存在してもそれを正解として固定してしまう。

追認テストを避ける確認手順

- 期待値の根拠を**仕様側**から示させる
- 異常系・境界値が含まれているか点検する
- 意図的に実装を壊し、テストが失敗するか確かめる
- 重要な業務ロジックは人がテスト観点を定義する

注記　事実：活用範囲の整理／提案：確認手順の4項目をテストレビュー時のチェック観点として運用ルールに組み込む。

エラー調査とデバッグを短縮する

原因が特定できないまま調査範囲が広がる状況に対し、原因候補を可能性の高い順に整理させることで、調査の優先順位付けと修正までの時間を短縮する。

デバッグの標準手順 － 各ステップをClaude Codeに実行させる

STEP 01
ログ解析
エラーログやスタックトレースを解析し、発生箇所を特定する。

STEP 02
再現条件の整理
どの入力・状態で発生するかを整理し、再現手順を明確化する。

STEP 03
原因候補の優先順位付け
可能性の高い順に候補を挙げ、調査順序を提示させる。

STEP 04
関連箇所の確認
関連コードと設定ファイルを横断的に確認する。

STEP 05
修正案の作成
影響範囲を踏まえた修正案を作成させる。

STEP 06
テスト実行と確認
修正後にテストを実行し、結果を人が確認する。

指示例

> 「エラーの原因候補を挙げ、可能性の高い順に調査手順を提示してください」

修正コードを直接求めるのではなく、まず**原因候補と調査手順**を求めることで、誤った前提のまま修正が進むことを防ぐ。

入力時に添えるべき情報

- エラーメッセージとスタックトレースの全文
- 発生した操作手順と入力値
- 実行環境とバージョン情報
- 直近の変更内容（Git差分）
- すでに切り分け済みの事項

ログに含まれる個人情報・認証情報は必ず除去する。

注記　事実：デバッグ手順・指示例の整理／提案：原因候補の妥当性は人が判断し、確証が得られるまで修正を確定しない。

リファクタリングを効率化する

重複や複雑な分岐の抽出は機械的な処理に適しており、修正前後のテストによって振る舞いが変わらないことを担保できる場合にのみ、安全に効率化できる。

リファクタリングの対象

01 重複コードの抽出と共通化

02 長大な関数やクラスの分割

03 命名やコードスタイルの統一

04 複雑な条件分岐の整理

05 未使用コード・不要な依存関係の検出

06 振る舞いを変えずに構造を改善

前提条件

修正前後のテストによる安全性確認ができない範囲は、リファクタリングの対象としない。

テストで守られていないコードの構造変更は、外形上は改善に見えても、振る舞いの変化を検知できない。

安全に進める順序

1. 対象範囲のテストを先に整備する
2. 変更前のテストが通ることを確認する
3. 構造改善を小さな単位で実施する
4. 同じテストが通ることを再確認する

注記 事実:リファクタリング対象の整理／提案:機能追加とリファクタリングは同一コミットに混在させず、差分の意図を明確に保つ。

コードレビューを支援させる

差分の要約と機械的に検出可能な指摘を一次確認として先行させることで、レビュアーは設計妥当性と業務要件への適合という判断領域に時間を集中できる。

AIに任せるレビュー観点

01	Git差分の要約
02	仕様との不整合の確認
03	バグ・例外処理漏れの検出
04	セキュリティ上の懸念点の洗い出し
05	パフォーマンスへの影響確認
06	テスト不足の指摘・重点確認箇所の提示

位置付け

AIレビューを最終承認とはせず、人によるレビュー前の「一次確認」に位置付ける。

役割分担

一次確認（AI）
差分要約、記述漏れ・例外処理漏れ、規約逸脱、テスト不足の機械的検出

最終承認（人）
設計判断の妥当性、業務要件への適合、リリース可否の意思決定

AIの指摘がないことは、問題がないことの証明にはならない。

注記　事実：レビュー観点・位置付けの整理／提案：レビュー記録には、AIの一次確認結果と人の承認を区別して残す。

効果を高める指示の出し方

工数削減の成否は指示の質に依存する。目的・範囲・参照実装・完了条件を定型化し、大きな作業を調査・計画・実装・検証に分割することが再現性を生む。

指示に含めるべき7つの要素

01	目的と背景を伝える
02	対象範囲を明確にする
03	参照すべき既存コードを指定する
04	制約条件と禁止事項を記載する
05	完了条件を定義する
06	テスト・Lint・型チェックの実行を指示する
07	大きな作業は調査・計画・実装・検証に分ける

指示テンプレート

目的	何を達成したいか・背景
対象機能	対象となる機能・画面・API
変更範囲	変更してよいファイル・領域
技術的な制約	利用可否のライブラリ・禁止事項
参考にする既存実装	パターンを踏襲すべきファイル
完了条件	満たすべき状態の定義
実行する検証項目	テスト・Lint・型チェック

POINT

有効だった指示はテンプレートとして共有資産化し、担当者による品質差を縮小する。

注記　事実：指示要素・テンプレート項目の整理／提案：テンプレートはリポジトリ内に配置し、プロジェクト固有の制約を追記して運用する。

導入時の注意点とガバナンス

情報管理・権限・承認・記録の4領域にルールを設けることが導入の前提であり、ガバナンスの整備なくして工数削減の効果を組織的に定着させることはできない。

領域	遵守事項	目的・想定リスク
情報管理	機密情報や個人情報を入力しない	情報漏えい・目的外利用の防止
情報管理	APIキーやパスワードをリポジトリに保存しない	認証情報の流出と不正利用の防止
品質・法務	生成コードのライセンス・依存関係を確認する	ライセンス違反・脆弱な依存の混入防止
承認	重要な設計判断は人が承認する	誤った前提に基づく実装の拡大防止
権限	本番環境への直接変更を禁止する	サービス停止・データ破壊の回避
権限	権限と実行可能なコマンドを制限する	想定外の操作範囲の抑制
記録	ログと変更履歴を残す	追跡可能性と説明責任の確保
品質・法務	セキュリティテストを省略しない	脆弱性の見落とし防止と品質担保

意思決定事項

導入前に「入力禁止情報の定義」「実行権限の範囲」「人の承認が必要な変更」「記録の保存方法」の4点を明文化し、開発ルールとして合意する。

注記　事実：遵守事項8項目の整理／解釈：領域分類と想定リスクは本資料による整理／提案：自組織の情報管理規程・セキュリティポリシーと整合させて運用する。

導入効果の測定方法

効果は「時間」「品質」「利用状況」の3系統で測定する。導入前の現状工数を測定してから小規模に試行し、前後比較によって拡大判断を行う。

測定する主なKPI

時間	時間	時間	時間
機能開発にかかった時間 同種タスクの所要時間で比較	**コード調査にかかった時間** 既存コード読解に要した時間	**テストコード作成時間** テスト整備に要した工数	**PR作成から承認までの時間** レビュー待ちを含むリードタイム
品質	**品質**	**利用状況**	**利用状況**
修正の手戻り件数 差し戻し・再修正の発生回数	**リリース後の不具合件数** 品質低下の有無を確認	**利用したタスクの割合** 適用範囲の広がりを把握	**利用頻度と満足度** 開発者ごとの定着度を確認

測定を成立させる前提

比較条件を揃える

同種・同規模のタスクで前後を比較し、難易度差の影響を抑える。

時間と品質を併記する

時間短縮のみで評価せず、不具合件数と併せて判断する。

目標値は実測後に設定

削減率の目標は、現状工数の測定結果を踏まえて設定する。

導入ステップ

1. 対象業務の現状工数を測定
2. 小規模チームで試験導入
3. 導入前後の数値を比較
4. 有効な指示をテンプレート化
5. 対象プロジェクトを段階的に拡大

ステップ1を省略すると比較基準が存在せず、効果の有無を判断できない。**現状測定を起点とすること**が測定設計の要件となる。

注記　事実：KPI項目・導入ステップの整理／提案：目標削減率は本資料では設定せず、ステップ1の実測値に基づき各組織で設定する（未実測のため数値記載なし）。

まとめ

定型作業と既存コード調査から限定的に着手し、実装前の計画確認・自動テスト・人によるレビューを組み合わせて効果を測定しながら、段階的に適用範囲を拡大する。

本資料の要点

01 Claude Codeは調査からテストまで幅広い工程で活用できる

02 特に定型作業と既存コード調査で工数削減効果を得やすい

03 実装前に計画を確認することで手戻りを抑えられる

04 自動テストと人によるレビューを組み合わせる

05 まずは限定的な業務から導入し、効果を測定する

06 開発ルールとガバナンスを整備して段階的に展開する

ご支援メニュー

Claude Code 導入支援
対象リポジトリの選定から検証環境の整備、初期運用の立ち上げまで

AI駆動開発の業務整理・ルール策定
役割分担・承認フロー・情報管理ルールの設計と文書化

開発チーム向けハンズオン研修
実践方法①～⑧と指示テンプレートを用いた実務演習

効果測定の設計・検証支援
現状工数の測定設計とKPIによる導入前後の比較評価

お問い合わせ先は次ページに記載しています。

次のアクション：対象業務を1つ選定し、現状工数の測定から着手する。

注記　事実：本資料の要点／提案：ご支援メニューは対象範囲と体制に応じて組み合わせて実施。／Confidential ／ 社外秘

お問い合わせ

導入検討の初期段階から、対象業務の選定・検証環境の整備・効果測定の設計までご相談いただけます。まずは現状の課題共有からお気軽にお問い合わせください。

お問い合わせ先

Start Challenge Consulting co., ltd.

挑む力が、世界を照らす

担当	**平出**
メールアドレス	**info@startchg.com**

ご連絡いただく際にお知らせください

① 対象システム・開発体制の規模　② 現在の開発工程上の課題
③ ご検討の時期とご希望のご支援内容

ご相談いただける内容

01 導入対象の選定と適用範囲の切り分け

02 検証環境（テスト・Lint・型チェック）の整備

03 情報管理ルールと承認フローの策定

04 効果測定の設計と導入前後の比較評価

お問い合わせ後の流れ

1 ご連絡内容の確認・日程調整

2 現状ヒアリング（オンライン・約1時間）

3 対象業務とご支援範囲のご提案

4 試験導入の開始

注記　事実：連絡先情報／提案：ご相談内容・お問い合わせ後の流れは想定手順であり、ご要望に応じて調整いたします。／Confidential ／ 社外秘