Engineer

# CI

変更を頻繁に統合し、そのたびに自動でビルドとテストを行う実践

## どのようなものか

CIはcontinuous integration、継続的インテグレーションの略です。チーム全員が一週間に何度も、コードの変更を一つの共通の開発ラインへ取り込み、そのたびに機械がビルドとテストを実行する進め方です。これにより、変更した本人がまだ作業を覚えているうちに、誤りが表面化します。略称と同じくらい、継続的インテグレーションという正式名称もよく使われます。

この言葉は二つの異なるものに使われるため、区別すると多くの混乱を避けられます。一つ目は、人に関わる実践です。多くのチームが最終的に引用する出典となるMartin Fowlerの記事は、習慣の集まりとして説明しています。メインラインを一つに保つ、ビルドを自動化する、ビルドに自己テストを組み込む、少なくとも毎日メインラインへプッシュする、毎回のプッシュをビルドする、壊れたビルドはすぐ直す、ビルドを速く保つ、未完成の作業はブランチに残すのではなく機能を隠す、全員がビルドの状態を見られるようにする、といった習慣です。記事は名称の由来をExtreme Programmingにたどり、それ以前にGrady Boochもこの表現を使っていたと記しています。

二つ目は、機械です。「CIが赤い」と言うときは、ビルドを実行し、失敗を報告したサービスを指します。GitHub Actions、GitLab CI/CDなどは、同じ構造を持っています。作業を定義するファイルをリポジトリへ置き、GitHubでは `.github/workflows/*.yml` 、GitLabではルートの `.gitlab-ci.yml` を使います。プッシュ、プルリクエストやマージリクエスト、スケジュール、人がボタンを押す操作などのイベントが、実行のきっかけになります。専用に用意したマシンであるランナーがコミットをチェックアウトし、そのファイルのコマンドを実行して、成功・失敗の結果を、そのコミットに関連付けて報告します。

サーバーだけでは、継続的インテグレーションにはなりません。作業を一か月間ブランチに置き、サーバーで各ブランチをビルドするチームは、役立つビルドの自動化を実現していますが、統合は最後にまとめて行っています。実践の中心

は頻繁なマージであり、機械はそれを続けられる程度に安全にする役割を担います。

継続的デリバリーと継続的デプロイは、その後に続く段階です。デリバリーは、成功したビルドを常にリリース可能な状態に保ち、デプロイはそれぞれを自動的にリリースします。どちらも先にCIが整っていることを必要とするため、三つはCI/CDとしてまとめて書かれることがよくあります。

概念を判断に生かす

「CIとは？」を実際の判断に生かしましょう。その概念を、開発または購入するシステムに結び付けて考えるお手伝いをします。 変更を検証し、リリースし、元に戻せる仕組みを整え、金曜の深夜ではなく火曜の午後にお客様へ届けられるようにします。

**サービスを見る** お問い合わせ →

用語の背後にある前提を検証する

解説とリンク先の実績を参考に、プロジェクトについての質問を整理してください。ツールを選ぶ前に、前提を一緒に検討できます。 GitHub ActionsでGIS SaaSのデプロイを自動化 — デプロイ、データ、報告を手作業から解放

**GitHub ActionsでGIS SaaSのデプロイを自動化** お問い合わせ →

## なぜ重要なのか

二つの作業系統をマージする負担は、別々に進めた期間に比例する以上に増えます。昨日、同じモジュールを変更した二人なら、両方が意図を覚えているため、数分で競合を解消できます。六週間前に分かれた二人は、差分から意図を再構築することになります。小さな変更を頻繁にマージすれば、競合も一つ一つ小さく保てます。機械が関わる前に、すでに効果の大部分が得られます。

プッシュのたびにビルドすると、もう一つ加わります。同じ結果を確認できる、余計なもののないマシンです。「自分のノートパソコンでは動く」というのは、独自にインストールしたライブラリーや残存ファイルを持つ、一台のノートパソコンについての主張です。毎回、新しいチェックアウトから始めるランナーは、同じ結果を再現するか、そのパソコン固有の条件を示します。

価値を生むのは速さです。Fowlerの記事は、主要なビルドの目標を約十分間としています。その理由は実務でも成り立ちます。プッシュから十分後に報告された失敗は、変更がまだ画面にあるうちに、原因を作った人へ戻ります。翌朝の失敗は、たまたま見た人のもとへ届き、まず一時間かけて過去の経緯を掘り起こすことになります。

可視性も重要です。メインラインの状態を全員に示せば、壊れたビルドを土台に、そうと知らずに新しい作業を始める人はいなくなります。修復は誰か他の人の問題ではなく、チームが最初に取り組む仕事になります。

CIはホステッドサービスである必要もありません。pre-commitは、コミットやプッシュの前にGitフックとしてプロジェクトのチェックを実行し、文書では同じ一式をサーバー上で `pre-commit run --all-files` により実行する方法も示しています。Taskは、一つの `Taskfile.yml` で各チェックに名前を付け、人とランナーが同じコマンドを呼べるようにします。このサイトを支えるチェックも、そのモデルに従っています。ページを執筆するローカル環境で動き、ホステッド型のビルドサービスを挟まず、毎回のプッシュ前に全チェックを実行します。

## 実際の活用例

あるチームでは、手作業のリリースをやめたことで、約4,000時間を取り戻しました。ソフトウェアの公開は、人によって少しずつ異なる手作業に頼っていましたが、ビルド、テスト、デプロイの自動化で置き換えました。変更から本番までの経路が、その場しのぎではなく標準的なものになり、リリースは速さと確実さの両方が向上しました。

長期運用のデータベースを持つあるプロジェクトでは、スキーマを二通りで記述しています。一つは初期構築用、もう一つは036から1102までの番号が付いた1,022本のマイグレーションです。変更のたびに、CIジョブが両方の方法でデータベースを構築します。一方は新規用のスキーマから、もう一方はすべてのマイグレーションを順に再実行して作り、構造と初期投入データを比較します。二つに違いがあれば、ジョブは失敗し、その差を出力します。これにより、新しい環境と長く使った環境が同じデータベースになることを、推測ではなくチェックで確認できます。

CIは、一つのアプリケーションの二つの部分の整合性も保てます。別のプロジェクトでは、データベースを起点とする生成処理の連鎖から、APIの記述とフロントエンドのクライアントを作っています。クライアントだけでも、型付きTypeScriptが44,076行あります。CIチェックは、フロントエンドが送る内容をAPIが期待する内容と照合し、食い違えばビルドを失敗させます。これこそCIが目指す結果です。片側だけで変わったフィールドが、利用者のブラウザーでのエラーになる前に、変更から数分後のビルドで失敗として見つかります。

# 問題が起きるところ

CIの仕組みの多くは、突然壊れるのではなく、徐々に劣化します。以下はどれも小さく始まり、最後には、チームが回避しながら作業することに慣れたビルドになります。

**作業そのものより、ビルドに時間がかかります。** プッシュの結果が返るまで四十分かかるようになると、人は待つのをやめ、複数の変更をまとめてプッシュし、その頻度も下げます。実践の目指す方向とは逆です。ハードウェアを追加する前に、どこで時間を使っているかを測定します。依存関係をキャッシュし、速いチェックから実行し、遅いテスト群は第二段階かスケジュール実行へ移し、主要なビルドを十分間程度に保ちます。

**メインラインが一日中失敗したままです。** 誰も直さない壊れたビルドが通常の状態になり、二つ目の失敗が最初の陰に隠れて入ってきます。防ぐルールは明快です。壊した人が今すぐ直すか、変更を元に戻し、修正には自分の時間を使います。ビルドが成功しているメインラインこそが、他の全員の開発の土台です。

**サーバーが各ブランチをビルドしていても、ブランチが何週間も残っています。** すべてのブランチは成功を示し、マージした日に問題が一度に現れます。これは統合を伴わない、ビルドの自動化です。少なくとも毎日メインラインへマージし、未完成の機能は分離したままにするのではなく、無効にしておきます。

**サーバーとノートパソコンで、チェックの一覧が違います。** 手作業で管理する二つのコマンド一覧はずれていき、変更がローカルでは成功してサーバーで失敗したり、その逆になったりして、どちらも信頼されなくなります。チェックの入口を一つにし、両方から呼びます。pre-commitなら、サーバーでも各開発者とまったく同じフックを実行できます。

```
# .gitlab-ci.yml
check:
    image: python:3.13
    script:
        - pip install pre-commit
        - pre-commit run --all-files --show-diff-on-failure
```

**不安定なテストのせいで、全員が再実行ボタンを押すようになります。** 変更とは関係のない理由で二十回に一回失敗するテストは、チームに、結果を読むより再実行する習慣を付けます。その後は、本当の失敗も同じように扱われます。そ

のテストを隔離し、修正するか削除します。頻繁に押される再実行ボタンは、それ自体を問題の兆候として扱います。

## 参考資料

- 継続的インテグレーション：Martin Fowlerの詳しい記事です。各実践、十分間のビルド、名称の由来を説明します。
- GitHub Actionsを理解する：GitHub自身の定義に沿って、ワークフロー、イベント、ジョブ、ランナーを説明します。
- GitLab CI/CD：GitLabの用語に沿って、`.gitlab-ci.yml`、パイプライン、ステージ、ジョブ、ランナーを説明します。
- pre-commit：設定ファイルで一度だけ定義したチェックを、Gitフックとして実行し、CIサーバーの手順としても実行する方法を説明します。
- Task：各チェックに一つの名前を付け、人とランナーが同じコマンドを入力できるようにする `Taskfile.yml` を説明します。

用語集

実際の制約に合わせて選ぶ

デプロイの流れ、運用上のリスク、緊急対応の担当者をお知らせください。検査、復旧手段、明確な責任の所在を伴う変更の範囲を定められます。

**料金を見る** お問い合わせ →

## 用語を実務に結び付ける

「CIとは？」がお客様の組織にどう当てはまるかをご相談ください。現在のシステムの具体例が一つあれば、話を始められます。

**サービスを見る**

welcome@engineer.company +45 50 10 61 84 デンマーク、コペンハーゲン

このページを持ち歩く

連絡先カード

心を込めて丁寧に制作： D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

著作権 © 2025 – 現在 · Engineer ApS · すべての権利を留保します。

印刷元： https://engineer.company/ja/glossary/ci/