GitHub Actionsによる CI/CD構築入門

ビルド・テスト・デプロイを自動化する実践ガイド

Start Challenge Consulting co., ltd.

2026年8月発行



Confidential ／ 無断転載・複製を禁じます

はじめに

リリース頻度の要求が高まる一方で、テストとデプロイが手作業に依存している限り、品質と速度は同時に成立しない。本資料はGitHub Actionsを用いて自動化の基盤を段階的に構築する手順を示す。

現状｜開発現場の前提

・機能追加と修正のサイクルが短期化し、リリース回数そのものが増加している

・変更のたびにビルド・テスト・デプロイという同一の作業が繰り返される

・作業量は増えるが、担当者の人数は増えない

課題｜手作業運用の限界

・手順が担当者ごとに異なり、実施漏れと操作ミスが避けられない

・不具合の発見が遅れ、修正コストが増大する

・リリース作業が特定の担当者に属人化する

目的｜本資料の狙い

・CI/CDの考え方と、GitHub Actionsの構成要素を整理する

・ビルド・テスト・デプロイの自動化を実装手順として提示する

・権限とシークレットの安全な扱いを明確にする

到達点｜読了後の状態

・自社リポジトリで最小構成のワークフローを作成できる

・CIからCDへ拡張する段階的な進め方を判断できる

・導入時の失敗パターンを事前に回避できる

想定読者

①CI/CDを初めて学ぶエンジニア ／ ②GitHubを利用している開発チーム ／ ③リリース作業を効率化したい担当者・管理者
いずれも「これから導入する」段階を前提とし、既存の自動化基盤の移行は本資料の対象外とする。

本資料の構成 Chapter 1 前提の整理（P2–P4） → Chapter 2 GitHub Actionsの基礎（P5–P8） → Chapter 3 CI/CDの実装（P9–P13） → Chapter 4 設計と運用（P14–P16）
各ページは「1ページ1メッセージ」で構成し、詳細な設定値は各ページの注記に出典を明記する。

事実：GitHub ActionsはGitHubが提供するビルド・テスト・デプロイの自動化機能である／解釈：手作業運用の限界は一般的な開発現場の傾向として整理／提案：CIから着手しCDへ段階的に拡張する。出典：GitHub Docs「About GitHub Actions」（docs.github.com）

CI/CDとは何か

CI/CDは「作業を速くする自動化」ではなく、コード変更のたびに同一の手順でビルド・検証・配布を反復し、品質を一定に保つ仕組みである。人が判断すべき箇所と機械に任せる箇所を分離することが導入の要点となる。



CI｜継続的インテグレーション

コードの変更をリポジトリへ頻繁に統合し、そのたびに自動でビルドとテストを実行する。不具合を変更直後に検出することが目的であり、CI/CD導入の最初の対象となる。

CD｜継続的デリバリー

テストを通過した成果物を、いつでもリリースできる状態まで自動で準備する。本番環境への反映には人による承認を挟むため、統制を保ったまま速度を上げられる。

CD｜継続的デプロイ

承認を挟まず、テスト通過をもって本番環境まで自動反映する。テストの網羅性と復旧手順が前提となるため、導入は最終段階に位置付ける。

CI/CDによって解決できる課題

①検証の抜け漏れ：テストの実行が人の記憶に依存せず、すべての変更に対して同一の検証が適用される。
②属人化：手順がリポジトリ内のファイルとして記述されるため、担当者が交代しても同じ結果が再現される。
③原因追跡の困難：どの変更で、どのテストが、いつ失敗したかが実行履歴として残り、障害発生時の切り分けが可能になる。

事実：GitHub ActionsではCIをpush／pull_requestイベント、CDをenvironmentを参照するジョブとして実装する／解釈：CI→継続的デリバリー→継続的デプロイの順に段階的へ拡張する構成。出典：GitHub Docs「About GitHub Actions」「Managing environments for deployment」(docs.github.com)

従来の開発・リリース作業の課題

手作業によるテストとデプロイは、担当者ごとの手順差と実施漏れを生み、不具合の発見を遅らせる。その帰結はリリース速度の低下にとどまらず、障害時の原因追跡が困難になるという経営リスクに至る。

①発生事象	②直接原因	③構造的原因	④業務への影響	⑤経営への影響
・担当者ごとにビルド・デプロイ手順が異なる	・手順が個人の記憶と口頭伝達に依存している	・リリース作業が特定の担当者に属人化している	・不具合の発見が遅れ、手戻り工数が増大する	・市場投入までの期間が長期化する
・テストの実施漏れが発生する	・実行環境が担当者の端末ごとに異なる	・検証を通っていない変更を統合できる状態にある	・リリース前の確認作業に時間が集中する	・障害発生時に原因を追跡できず、復旧が長引く
・デプロイ時の操作ミス（対象環境・実行順の誤り）が起きる	・作業記録が残らず、実施の有無を後から確認できない	・手順が仕組みとして強制されていない	・担当者の不在がリリースの停止要因になる	・品質のばらつきが顧客の信頼を損なう

観点	手作業による運用（As-Is）	CI/CD導入後（To-Be）
手順の再現性	担当者と端末に依存し、結果が一致しない	ワークフローファイルとして定義され、常に同一手順で実行される
不具合の検出時点	リリース直前、または本番稼働後	変更のプッシュ時・プルリクエスト時
記録と追跡	作業記録が残らず、事後の検証が困難	実行ログ・成果物が履歴として保存され、変更単位で追跡できる

事実：ワークフローは.github/workflows配下のYAMLとして定義され、実行ログが履歴として残る／解釈：①〜⑤の因果構造は一般的な手作業運用から導いた整理であり、個別組織の実態は別途確認が必要／提案：③構造的原因（属人化・検証の未強制）から着手する。出典：GitHub Docs「Workflow syntax for GitHub Actions」（docs.github.com）

GitHub Actionsの基礎知識

GitHub減ctionsはリポジトリと同一環境で動作する自動化基盤であり、CI/CD専用サーバーの構築・運用を必要としない。ソースコードと自動化手順を同じリポジトリで管理できることが、外部CIツールとの最大の差である。

GitHub Actionsとは

ビルド・テスト・デプロイのパイプラインを自動化できる、GitHubに組み込まれた継続的インテグレーション／継続的デリバリー（CI/CD）のプラットフォーム。リポジトリ内のイベント（プッシュ、プルリクエスト、Issue作成など）をきっかけに、GitHubが提供する仮想マシン（ランナー）上で任意の処理を実行する。

主な利用用途

ビルド	テスト	コードチェック	デプロイ	定期処理
依存パッケージを解決し、アプリケーションを実行可能な成果物へ変換する	単体テストを自動実行し、失敗時はプルリクエストのマージを止める	静的解析・書式統一・依存関係の脆弱性検査を変更単位で適用する	検証環境・本番環境へ成果物を反映する。承認を組み込むことも可能	cron書式で夜間バッチや定期検査を実行する（最短5分間隔）

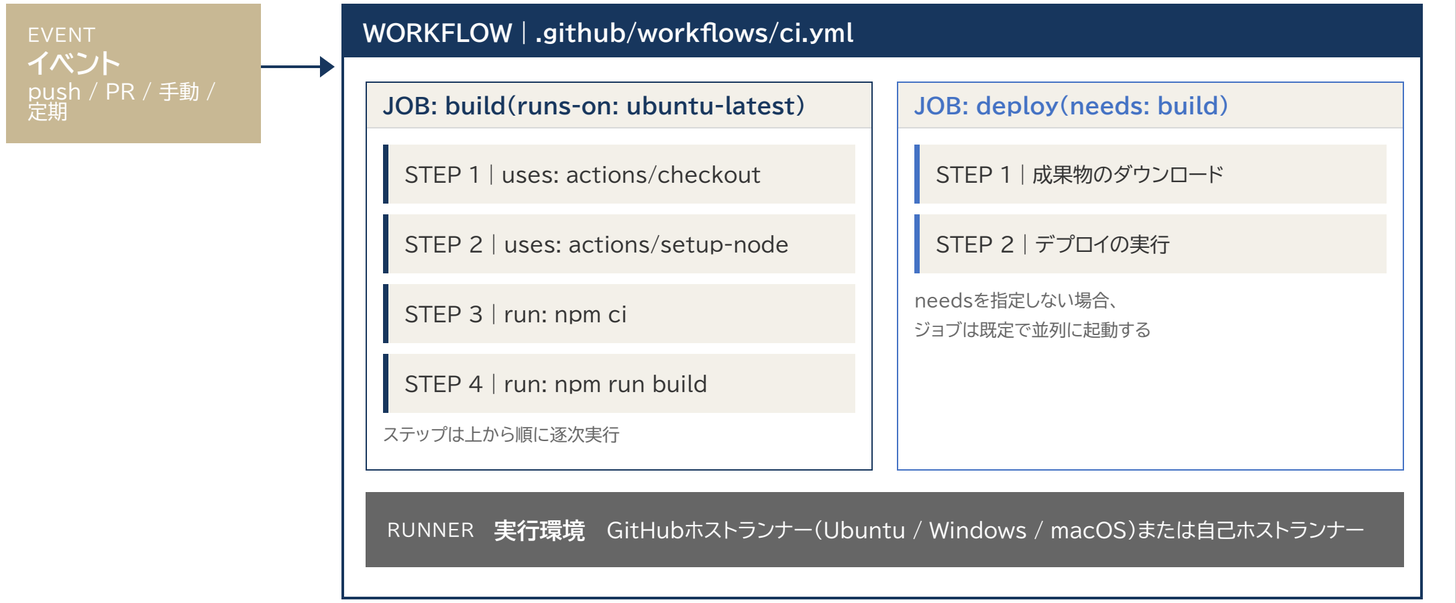
Jenkinsなど外部CI/CDツールとの主な違い

観点	Jenkins（セルフホスト型の代表例）	GitHub Actions
実行環境	主に自社で用意したサーバー上で運用する	GitHubホストランナーと自己ホストランナーを併用できる
定義の記述形式	宣言的パイプライン／スクリプト型パイプラインの2系統	YAML形式に統一（.github/workflows配下で管理）
構成要素の対応	stages減 減teps減 減gent減 減pstreamProjects減 減ron	jobs減 減teps減 減uns-on減 減eeds減 減n.schedule減 対応

事実：スケジュール実行の最短間隔は5分。Jenkinsのstages／steps／agent／upstreamProjects／cronはGitHub減ctionsのjobs／steps／runs-on／needs／on.scheduleに対応する（公式移行ガイド記載）／解釈：「専用サーバー不要」は運用形態上の特性。出典：GitHub減ocs「Migrating減rom減enkins減o減itHub減ctions」「Workflow減yntax減or減itHub減ctions」（docs.github.com）

GitHub Actionsの基本構成

GitHub Actionsは「Workflow ⊃ Job ⊃ Step」という3階層で構成される。ジョブは既定で並列に起動し、ステップは同一ランナー上で逐次実行される。この並列と逐次の区別が、CI/CD設計の起点となる。



要素	定義
Workflow	1つ以上のジョブを実行する自動化された手順。YAMLファイル1本が1ワークフローに相当する
Event	ワークフローを起動するきっかけ。リポジトリ内の操作、手動実行、スケジュール等
Job	同一ランナー上で実行される一連のステップ。既定で他のジョブと並列実行される
Step	ジョブ内の個別処理。コマンド実行またはActionの呼び出しで、逐次実行される
Action	ステップから呼び出す再利用可能な処理単位。公式・第三者・自作のものが使える
Runner	ステップが動作する仮想マシン環境。GitHubホスト型と自己ホスト型を選択できる

設計上の留意点

・並列と逐次の使い分け：独立した検証（テストとLint）は別ジョブに分けて並列化し、実行時間を短縮する。順序が必要な処理は**needs**で明示的に依存関係を宣言する。

・ジョブは別々のランナーで動作するため、ファイルはジョブ間で共有されない。成果物の受け渡しにはArtifactを用いる（P11参照）。

・GitHubホストランナーのジョブ最大実行時間は**6時間**、ワークフロー実行全体の最大継続時間は**35日**。長時間処理は分割を前提に設計する。

事実：ジョブは既定で並列実行され、needsで依存関係を指定する。ステップは同一ランナー上で逐次実行。GitHubホストランナーのジョブ上限は6時間、ワークフロー実行全体は35日／解釈：並列化による実行時間短縮は設計方針。出典：GitHub Docs「Workflow syntax for GitHub Actions」「Usage limits, billing, and administration」(docs.github.com)

ワークフローファイルの作成方法

ワークフローは.github/workflows/ 直下のYAMLファイルとして定義する。サブディレクトリは対象外であり、記述はインデントで階層が決まるため、構文ミスの多くはインデントとタブ文字に起因する。

基本的な記述項目

キー	区分	役割
name	任意	Actionsタブに表示する名称。省略時はファイルパスが表示される
on	必須	起動トリガー。指定がなければワークフローは実行されない
jobs	必須	実行するジョブの集合。任意のジョブIDを配下に定義する
runs-on	必須	ジョブを実行するランナー環境（例：ubuntu-latest）
steps	必須	ジョブ内で実行する処理の並び。usesでAction、runでコマンドを実行
permissions	任意	GITHUB_TOKENの権限範囲。最小権限の設定に用いる（P13参照）

記述時の注意点

- 保存場所は.github/workflows/ 直下。サブディレクトリはサポートされない
- 拡張子は .yml または .yaml。1リポジトリに複数ファイルを配置できる
- インデントは半角スペースのみ。タブ文字は構文エラーとなる
- 1ファイルの上限は500KB。超過した場合は実行が開始されない
- エディタのYAML検証機能を用い、コミット前に構文を確認する

最小構成のサンプルワークフロー

mainへのプッシュとmain宛のプルリクエストで起動し、取得 → Node.js準備 → 依存関係 → テストを実行する

.github/workflows/ci.yml

```yaml
name: node-ci

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

permissions:
  contents: read

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v6
        with:
          node-version: '20'
          cache: 'npm'
      - run: npm ci
      - run: npm test
```

事実：保存場所は.github/workflows直下（サブディレクトリ非対応）、拡張子は.yml／.yaml、1ファイル上限500KB、nameは任意で省略時はファイルパスを表示／提案：コミット前にYAML検証を実施する。出典：GitHub Docs「Workflow syntax for GitHub Actions」「Usage limits, billing, and administration」（docs.github.com）

ワークフローを開始するトリガー

トリガーは「検証はプルリクエスト時、配布はマージ後」という役割分担で選ぶ。ブランチとパスによる絞り込みを併用することで、無関係な変更による不要な実行を抑え、実行時間と利用コストの両方を削減できる。

トリガー	起動条件	主な用途	留意点
push	対象ブランチ・タグへのプッシュ時に自動実行	マージ後のビルド、成果物生成、デプロイの起点	全ブランチを対象にすると実行回数が急増する
pull_request	既定では opened／synchronize／reopened の3種で自動実行	マージ前のテスト・静的解析。必須チェックとの組み合わせ	コミット追加ごとに再実行される（synchronize）
workflow_dispatch	Actionsタブの「Run workflow」から手動実行	臨時のデプロイ、再実行、パラメータ指定の運用作業	ファイルが既定ブランチ上にある場合のみ受信される
schedule	POSIX cron書式（分 時 日 月 曜日）で定期実行。既定はUTC	夜間の全体テスト、依存関係の脆弱性検査、定期バッチ	最短間隔は5分。既定ブランチの最新コミットで実行される

ブランチ・パスによる実行条件の指定

```yaml
on:
  push:
    branches:
      - main
      - 'release/**'      # パターン指定が可能
    paths-ignore:
      - 'docs/**'         # 文書のみの変更では実行しない
      - '**.md'
  pull_request:
    branches: [main]
    paths:
      - 'src/**'          # 対象コードの変更時のみ検証
  workflow_dispatch:      # 手動実行も併用
```

用途に応じた使い分けと、不要な実行を減らす設計

使い分けの原則

- マージ前の品質確認 → pull_request
- マージ後の配布・デプロイ → push（mainなどに限定）
- 臨時作業・再実行 → workflow_dispatch／定期検査 → schedule

不要な実行を減らす方法

- branches／pathsで対象を限定し、無関係な変更では起動しない
- concurrencyのcancel-in-progressで、同一ブランチの古い実行を取り消す
- schedule実行の頻度は目的に対して過剰にしない

事実：pull_requestの既定アクティビティはopened／synchronize／reopened。workflow_dispatchは既定ブランチ上のファイルのみ受信。scheduleは最短5分間隔・既定タイムゾーンはUTCで既定ブランチの最新コミットに対して実行。branchesとbranches-ignore、pathsとpaths-ignoreは同一イベント内で併用不可／提案：pathsによる限定でコストを抑える。出典：GitHub Docs「Workflow syntax for GitHub Actions」「Events that trigger workflows」（docs.github.com）

CI環境の構築① ビルドの自動化

ビルドの自動化は「チェックアウト → ランタイム準備 → 依存関係のインストール → ビルド」の4段階に定型化できる。依存関係のキャッシュを併用することで、変更のない部分の再取得を省き、実行時間を短縮する。

1. **ソースコードのチェックアウト**
   actions/checkout@v4 でリポジトリをランナー上に取得する
2. **言語・ランタイムのセットアップ**
   actions/setup-node@v6 等でバージョンを固定し、環境差をなくす
3. **依存パッケージのインストール**
   npm ci のようにロックファイルに従い、再現性のある取得を行う
4. **アプリケーションのビルド**
   npm run build を実行し、配布可能な成果物を生成する
5. **ビルド結果の確認**
   実行ログで失敗箇所を特定し、成果物はArtifactとして保存する（P11）

Node.jsを例にした設定イメージ

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      # 1. チェックアウト
      - uses: actions/checkout@v4
      # 2. ランタイム＋依存関係キャッシュ
      - uses: actions/setup-node@v6
        with:
          node-version: '20'
          cache: 'npm'
      # 3. 依存パッケージの取得
      - run: npm ci
      # 4. ビルド
      - run: npm run build
```

キャッシュによる処理時間の短縮｜依存関係の再取得を省く

- setup-node の **cache: 'npm'** を指定するだけで、依存関係のキャッシュ復元と保存が自動化される
- より細かい制御が必要な場合は actions/cache@v4 を用い、キーとパスを明示する
- キャッシュはあくまで高速化手段であり、ロックファイルによる再現性の担保が前提となる
- キャッシュ容量はリポジトリあたり**既定10GB**。超過時は最終アクセスの古いものから自動削除される
- **7日間**アクセスのないキャッシュエントリは自動的に削除される
- キャッシュ操作にはレート制限がある（1リポジトリあたり200アップロード／分など）

事実：キャッシュ容量はリポジトリあたり既定10GB、7日間アクセスされないエントリは自動削除、キャッシュのアップロードは1リポジトリあたり200回/分が上限。setup-nodeのcacheオプションで依存関係キャッシュを自動化できる／解釈：4段階への定型化は実装手順の整理。出典：GitHub Docs「Caching dependencies to speed up workflows」「Usage limits, billing, and administration」（docs.github.com）

CI環境の構築② テストの自動化

テストの自動実行は、ブランチ保護の必須ステータスチェックと組み合わせて初めて「品質の門」として機能する。実行するだけでは失敗を止められないため、マージ条件としての設定が不可欠である。

プルリクエスト時に実行する検証

検証の種類	内容	実行例
単体テスト	機能単位の動作を検証し、変更による退行を検出する	npm test
静的解析・Lint	実行前に構文上の問題や規約違反を検出する	npm run lint
書式チェック	整形の統一を機械的に確認し、レビュー範囲を論点に集中させる	npm run format:check
複数版テスト	複数のランタイム版・OSでの動作を並列に検証する	strategy.matrix

テスト失敗時のマージ防止 | 必須ステータスチェックの設定

・対象ブランチにブランチ保護ルールを設定し、テストジョブを**必須ステータスチェック**として指定する
・指定したチェックが成功しない限り、保護ブランチへのマージがブロックされる
・ワークフローを作るだけではマージは止まらない。リポジトリ設定側の指定が必要である

テスト結果をプルリクエストで確認する

・各ジョブの成否はプルリクエスト画面のチェック一覧に表示され、ログから失敗箇所を追跡できる
・ステップから **GITHUB_STEP_SUMMARY** へ書き込むと、実行結果の要約をMarkdownで表示できる

マトリクスビルドの活用 | 複数バージョンの並列検証

```
s:
test:
  runs-on: ${{ matrix.os }}
  strategy:
    fail-fast: false     # 既定はtrue
    max-parallel: 4      # 同時実行数の上限
    matrix:
      os: [ubuntu-latest, windows-latest]
      node-version: ['18.x', '20.x', '22.x']
  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v6
      with:
        node-version: ${{ matrix.node-version }}
        cache: 'npm'
    - run: npm ci
    - run: npm test
```

マトリクス設計の要点

・変数の値の組み合わせごとにジョブが生成される。上の例では2 OS × 3バージョン = 6ジョブ
・**fail-fast** の既定値は true。1つが失敗すると進行中の他ジョブが取り消される。全組み合わせの結果を確認したい場合は false を指定する
・**max-parallel** で同時実行数を制御し、実行枠の占有を抑える
・マトリクスで生成できるジョブ数は1ワークフロー実行あたり**最大256**。組み合わせを増やすほど利用分数も比例して増加する

事実：fail-fastの既定値はtrue、マトリクスのジョブ上限は1ワークフロー実行あたり256、必須ステータスチェックにより保護ブランチへのマージがブロックされる、GITHUB_STEP_SUMMARYでMarkdown要約を表示できる／提案：ワークフロー作成とブランチ保護設定は必ず同時に行う。出典：GitHub Docs「Running variations of jobs in a workflow」「About protected branches」「Workflow commands for GitHub Actions」(docs.github.com)

成果物の保存と受け渡し

ジョブは別々のランナーで動作するためファイルは共有されない。Artifactはジョブ間の受け渡し手段であり、同時に「CIで検証した成果物をそのままCDで配布する」という同一性の担保手段でもある。

JOB: build（CI）

ビルドを実行し、生成された dist/ を成果物としてアップロードする

```
- uses: actions/upload-artifact@v4
  with:
    name: build-output
    path: dist/
```

Artifactとして保存

GitHub上のストレージ

・実行単位で成果物を保管
・実行画面から直接ダウンロード可能
・保持期間の経過後に自動削除

別ジョブで取得

JOB: deploy（CD）

検証済みの成果物を取得し、再ビルドせずそのまま配布する

```
- uses: actions/download-artifact@v4
  with:
    name: build-output
    path: dist/
```

Artifactとは｜保存する目的

・ワークフロー実行中に生成したファイルを、実行終了後も保持できる仕組み
・**①ジョブ間の受け渡し**：ビルドジョブの出力をデプロイジョブへ引き継ぐ
・**②検証と配布の同一性**：テストを通った成果物そのものを本番へ届ける
・**③記録の保全**：テストレポート・ログを保存し、事後の原因追跡に用いる

保存期間と、不要なデータを残さない運用

・既定の保持期間は**90日**。経過後は自動削除される
・ワークフロー側で **retention-days** を指定し、用途ごとに短縮できる
・保持期間の既定値はリポジトリ・Organizationの設定で変更可能
・成果物はGitHub Packagesと共通のストレージ枠で課金されるため、大容量を長期保持しない運用ルールが必要

CIからCDへ成果物を連携する考え方

デプロイジョブで再ビルドを行うと、テストした対象と配布する対象が別物になり得る。CIで生成・検証した成果物をArtifactとして受け渡し、CDでは「配布のみ」を行う構成が、検証結果と本番稼働物の同一性を担保する。

保持期間を短縮する記述例

テストレポートなど短期間のみ必要なものは、明示的に期間を絞る。

```
- uses: actions/upload-artifact@v4
  with: { name: test-report, path: reports/, retention-days: 7 }
```

事実：upload-artifact／download-artifactの現行メジャーはv4（v4は2023年12月14日に一般提供、v3は2024年4月16日に廃止告知）。既定の保持期間は90日でretention-daysにより変更可能。成果物はGitHub Packagesと共通のストレージ枠で課金される／提案：CDでは再ビルドせずArtifactを配布する。出典：GitHub Docs「Storing and sharing data from a workflow」／GitHub Changelog「Artifacts v4 is now Generally Available」（github.blog）

CD環境の構築とデプロイの自動化

デプロイの自動化は「CI成功を前提条件とし、本番反映のみ人が承認する」構成が基本形となる。needsで順序を、environmentで承認と権限を制御することで、速度と統制を同時に成立させられる。

JOB 1
CI(build / test)
ビルドとテストを実行し、成果物をアップロード

needs →

JOB 2
ステージングへ自動デプロイ
environment: staging(承認なし)

needs →

GATE
手動承認(必須レビュー)
承認されるまでジョブは待機状態

approve →

JOB 3
本番環境へデプロイ
environment: production

デプロイ用Jobの作成例(CI成功後に実行)

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps: [ ... ビルドとテスト ... ]

  deploy-staging:
    needs: build            # CI成功が前提条件
    if: github.ref == 'refs/heads/develop'
    runs-on: ubuntu-latest
    environment:
      name: staging         # 承認なしで自動反映
    steps: [ ... 検証環境へ配布 ... ]

  deploy-production:
    needs: build
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest
    environment:
      name: production      # 必須レビューを設定した環境
      url: https://example.com
    steps: [ ... 本番へ配布 ... ]
```

ブランチと環境の対応例

ブランチ	対応環境	デプロイ方式
develop	開発・検証環境	マージ後に承認なしで自動デプロイ
main	本番環境	必須レビューによる承認後にデプロイ

手動承認を組み込む方法(Environment側の設定)

・環境に**必須レビュー担当者**を設定する(最大6人／6チーム、承認は1人で成立)
・承認されるまでジョブは待機し、承認後に初めて環境シークレットへアクセスできる
・待機時間(wait timer)は最大43,200分＝30日まで設定可能で、待機時間は課金対象に含まれない

デプロイ失敗時の確認ポイント
①ジョブの実行ログ(どのステップで失敗したか) ②認証情報の有効性と権限範囲 ③環境のデプロイ対象ブランチ制限 ④成果物の取得結果 ⑤前段ジョブの成否(needsの連鎖)

事実:environmentには必須レビュー担当者を最大6人／6チーム設定でき承認は1人で成立、wait timerは最大43,200分(30日)で待機時間は課金対象外、needsにより前段ジョブが失敗した場合は後続ジョブは実行されない／解釈:develop＝検証環境、main＝本番環境の対応は代表的な設計例。出典:GitHub Docs「Managing environments for deployment」「Workflow syntax for GitHub Actions」(docs.github.com)

シークレットと権限の安全な管理

自動化はリポジトリに認証情報を集約させる。パスワードやAPIキーをワークフローへ直接記述せず、Secretsで管理し、権限は最小限に絞る。クラウド認証はOIDCにより長期の認証情報そのものを持たない構成が望ましい。

ワークフローに認証情報を直接記述する危険性

ワークフローファイルはリポジトリの履歴として永続的に残るため、一度コミットした認証情報は削除しても過去のコミットから取得できる。リポジトリの閲覧権限を持つ全員に露出し、フォーク・クローンを通じて外部へ拡散する。

GitHub Secrets | 3階層による管理

設定単位	適用範囲と用途	個数上限
Repository	当該リポジトリの全ワークフローで利用。共通の外部サービス連携など	100件／リポジトリ
Environment	その環境を参照するジョブのみ利用可。本番の認証情報はここに置く	100件／環境
Organization	複数リポジトリで共有。対象リポジトリを限定して付与できる	1,000件／Organization

設定の原則 本番環境の認証情報はEnvironment単位で管理し、必須レビューを設定した環境に限定する。承認前のジョブは環境シークレットへアクセスできないため、承認フローがそのまま認証情報の保護になる。1つの値の最大サイズは48KB。

GITHUB_TOKENの役割と、必要最小限の権限設定

- ワークフロー実行ごとに自動生成される一時的な認証トークン。事前登録は不要
- ランナー上でのみ有効で、ジョブ終了時または最大有効期間の経過で失効する（GitHubホストランナーのジョブ上限は6時間）
- Organization／リポジトリ設定で既定の権限を**読み取り専用**に変更できる
- ワークフロー側では **permissions** キーで必要なスコープのみを付与する

```yaml
permissions:        # 未指定のスコープは自動的に none
  contents: read
  id-token: write   # OIDC利用時のみ必要
```

シークレットをログへ出力しないための注意点

- 登録済みのシークレットはログ上で自動的にマスクされるが、これは最後の防御線に過ぎない
- JSONやXMLなど**構造化データを値に含めるとマスクが機能しない**場合がある。値は平文の単一文字列とする
- デバッグ目的で認証情報を出力する処理を残さない。環境変数の一括表示も避ける

クラウド認証におけるOIDCの活用

- OpenID Connectにより、実行ごとに**短命なクラウドアクセストークン**を取得する
- 長期有効なクラウド認証情報をGitHub側に保存・複製する必要がなくなる
- 利用には **id-token: write** の権限が必要。AWS・Azure・GCP・HashiCorp Vault等で利用可

事実：シークレットの上限はリポジトリ100件・環境100件・Organization1,000件、1値の最大サイズ48KB。GITHUB_TOKENは実行ごとに自動生成され、permissions未指定のスコープはnone。OIDCにはid-token: write権限が必要。第三者Actionはフル長のコミットSHAでの固定が公式推奨／提案：本番認証情報はEnvironment単位で管理。出典：GitHub Docs「Using secrets in GitHub Actions」「Automatic token authentication」「Security hardening for GitHub Actions」（docs.github.com）

実務で必要なCI/CD設計

CI/CDの設計とは、ブランチ戦略と自動化の対応付けである。「プルリクエスト時は検証のみ、マージ後に配布」という原則にブランチ・環境・承認を対応させることで、権限と責任の境界が明確になる。



プルリクエスト時に実行する処理

- 単体テスト、静的解析、書式チェック
- 必要に応じた複数バージョンでのテスト
- 原則として**デプロイは行わない**
- 実行時間を短く保ち、レビューを妨げない

マージ後に実行する処理

- ビルドと成果物（Artifact）の保存
- 検証環境への自動デプロイ
- 本番環境へのデプロイ（承認を経由）
- リリースノート等の付随処理

環境の分離と承認フローの設計

- 開発／検証／本番をEnvironmentで分離
- 環境ごとに認証情報と保護ルールを設定
- 本番のみ必須レビューを設定する
- 自己承認の禁止設定も併せて検討する

同時実行の制御　同一ブランチへ連続してプッシュされた場合、古い実行を取り消して実行枠と分数を節約する。デプロイジョブでは同一環境への同時反映を防ぐ用途にも用いる。

```
concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true
```

再利用可能なワークフロー　共通処理を1本に集約し、複数リポジトリから呼び出す。小規模チームでは「CI用1本＋CD用1本」を共通化し、各リポジトリは呼び出しのみとする構成が保守しやすい。

```
jobs:
  call-ci:
    uses: my-org/shared/.github/workflows/ci.yml@v1
```

事実：concurrencyのcancel-in-progressにより同一グループの進行中の実行を取り消せる（同時実行グループあたりのキュー上限は100実行）。再利用可能ワークフローはon: workflow_callで定義し、接続できるワークフローは最大10段まで／解釈：ブランチと処理の対応付けは代表的な設計例であり、組織のブランチ戦略に合わせた調整が必要（提案）。出典：GitHub Docs「Workflow syntax for GitHub Actions」「Reusing workflows」（docs.github.com）

導入時によくある失敗と改善策

導入の失敗は技術的な難易度ではなく、範囲設定と統制の欠落に起因する。テストが不十分なまま本番デプロイを自動化することは、不具合の本番到達を高速化するに等しい。小さく導入し、段階的に対象を広げる。

#	よくある失敗	発生する問題	改善策
1	最初からすべてを自動化しようとする	構築が長期化し、稼働前に形骸化する。設定の複雑化で保守できなくなる	ビルドとテストの2工程のみで開始し、動作を確認してから対象を広げる
2	テストが不十分なままデプロイを自動化する	検証されない変更が本番へ到達し、障害の発生頻度が上がる	自動テストの整備を前提条件とし、CDはテストの信頼性が確認できた範囲に限定する
3	本番環境へ直接デプロイできる状態にする	誤操作や意図しないブランチからの反映が発生し、統制が効かない	本番はEnvironmentに必須レビューを設定し、デプロイ対象ブランチを限定する
4	権限を過剰に付与する	ワークフローの不具合や侵害時に、影響範囲がリポジトリ全体へ及ぶ	GITHUB_TOKENの既定を読み取り専用にし、permissionsで必要分のみ付与する
5	外部Actionを無条件で利用する	タグは書き換え可能なため、参照先の内容が意図せず変わる可能性がある	利用するActionを限定し、フル長のコミットSHAで固定する（公式推奨）
6	実行時間や利用コストを管理していない	無料枠を超過し、想定外の費用が発生する。実行待ちが常態化する	トリガーを絞り、キャッシュとconcurrencyを活用し、利用分数を定期的に確認する
7	エラー通知の仕組みがない	失敗が放置され、CI/CDが「常に赤い」状態となり信頼されなくなる	失敗時の通知先と一次対応の担当・手順を、導入と同時に定める

小さく導入し、段階的に改善する進め方

PHASE 1
CI（検証）から着手
1リポジトリでビルドとテストのみ自動化

→

PHASE 2
マージ条件として組み込む
ブランチ保護と必須ステータスチェックを設定

→

PHASE 3
検証環境へ自動デプロイ
影響範囲の小さい環境で運用に慣れる

→

PHASE 4
承認付きで本番へ拡張
必須レビューと通知・復旧手順を整備した上で適用

事実：第三者Actionはフル長のコミットSHAでの固定が公式に推奨されている（タグは書き換え可能）。GITHUB_TOKENの既定権限はリポジトリ／Organization設定で読み取り専用に変更できる／解釈：失敗パターンと改善策の対応は一般的な導入知見に基づく整理／提案：PHASE 1〜4の順序で段階的に適用する。出典：GitHub Docs「Security hardening for GitHub Actions」「Automatic token authentication」（docs.github.com）

まとめ・CI/CD導入チェックリスト

GitHub Actions導入の要点は「ビルドとテストの自動化から始め、統制（ブランチ保護・承認・最小権限）を整えてから本番デプロイを自動化する」ことにある。以下の5ステップと5項目のチェックリストを着手基準とする。

導入ステップ

1. **自動化する作業を整理する**
   現在の手作業を洗い出し、繰り返し発生するものから対象を選ぶ
2. **ビルドとテストから開始する**
   pull_requestトリガーで検証を自動化し、必須チェックに設定する
3. **ステージング環境へ展開する**
   Artifactで成果物を受け渡し、検証環境へ自動デプロイする
4. **承認付きで本番デプロイを導入する**
   Environmentに必須レビューを設定し、対象ブランチを限定する
5. **セキュリティと運用ルールを整備する**
   最小権限・OIDC・通知と復旧手順を定め、定期的に見直す

導入チェックリスト

- [ ] 自動テストが整備され、失敗が検知できる状態になっている
- [ ] ブランチ保護が設定され、必須ステータスチェックが指定されている
- [ ] シークレットが安全に管理され、ワークフローに直接記述されていない
- [ ] 本番デプロイに承認手順（必須レビュー）が組み込まれている
- [ ] 失敗時の通知先と復旧手順が明文化されている
- [ ] GITHUB_TOKENの権限が最小限に設定され、外部ActionがSHAで固定されている

自社でのCI/CD構築・開発支援のご案内

現状のリリース手順の可視化、ワークフローの設計・実装、ブランチ戦略と承認フローの整備、権限設計とセキュリティ点検、運用定着の支援まで一貫してご対応します。既存環境の調査から段階的な導入計画の策定まで、規模に応じて構成いたします。

Start Challenge Consulting co., ltd.
挑む力が、世界を照らす

会社名 Start Challenge Consulting co., ltd.
ご担当 平出

お問い合わせ info@startchg.com
ご相談内容 CI/CD構築・開発支援

詳しいご連絡先は次ページをご参照ください

事実：必須ステータスチェックによる保護ブランチへのマージ制御、Environmentの必須レビュー（最大6人／6チーム、承認は1人で成立）、permissionsによる最小権限設定はいずれもGitHubの標準機能である／提案：導入ステップ1〜5とチェックリスト6項目を着手基準として用いる。出典：GitHub Docs「About protected branches」「Managing environments for deployment」「Security hardening for GitHub Actions」（docs.github.com）

お問い合わせ

本資料の内容に関するご質問、および自社でのCI/CD構築・開発支援のご相談は、下記までお気軽にご連絡ください。現状の確認から段階的な導入計画の策定まで、規模に応じてご提案いたします。

Start Challenge
Consulting co., ltd.
挑む力が、世界を照らす

ご連絡先	
会社名	Start Challenge Consulting co., ltd.
ご担当	平出
メールアドレス	info@startchg.com

ご相談いただける内容

・現状のリリース手順の可視化と課題の整理　／　・ワークフローの設計・実装
・ブランチ戦略と承認フローの整備　／　・権限設計とセキュリティ点検
・段階的な導入計画の策定と運用定着の支援

まずは現状の確認から
「どこから自動化すべきか」の判断が難しい段階でも構いません。既存のリリース手順とリポジトリの状況を確認したうえで、着手すべき範囲を具体的にご提示します。

本資料は情報提供を目的としたものであり、記載内容の利用により生じた損害について当社は責任を負いません。仕様・料金等は変更される場合があります。