

テーマ3：CDの実践

レッドハット株式会社
テクニカルセールス本部

1 CIとCDの違い

2 Gitによるソースコード管理

3 CI/CDとGitOps

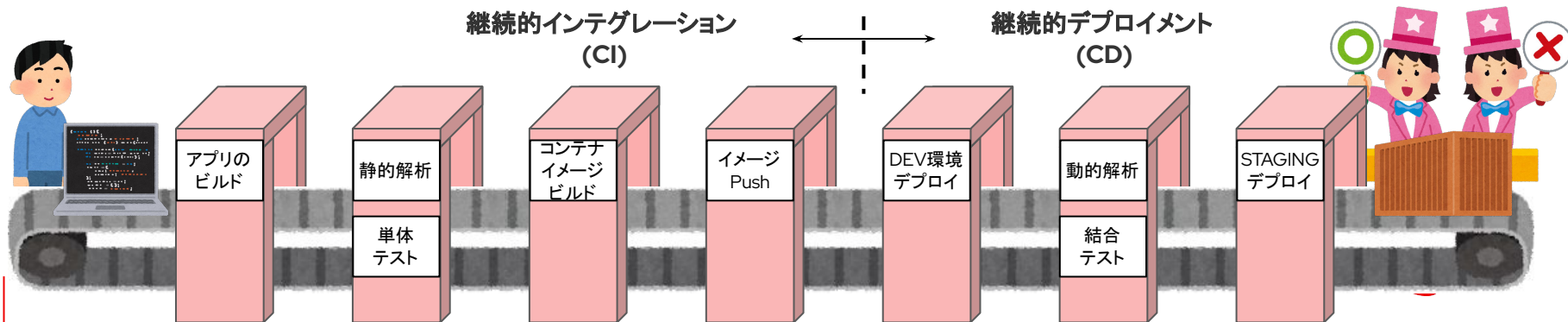
4 CDハンズオン

CIとCDの違い

開発～デプロイまでの作業を自動化するCIとCD

▶ 継続的インテグレーション (CI) と 継続的デプロイメント (CD)

- アプリケーションのビルドからコンテナのビルド、デプロイまでを自動的に行うようにする。
 - 途中で行うコードの解析やテストなども自動化する。
 - CI/CD用のソフトウェアを利用して、“パイプライン”を構成する
- 人の作業を「アプリケーションのソースコードの投入」と「デプロイの承認」など最小限にして極力自動化することで、アプリケーションデリバリーまでの作業品質を均一化する。

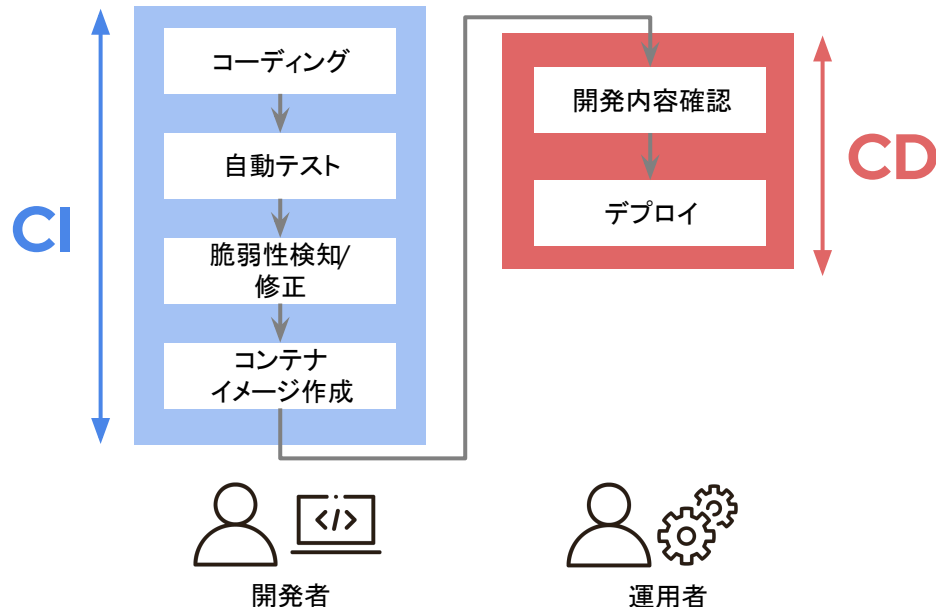


CIとCDの分離がなぜ重要なのか？

チーム間の責任分界点を明確化する

CIとCDを別々のツールを使って実施することで、役割分担が明確化し、各々のタスクに集中することが可能となります

- ・デプロイ可能な成果物を作成するまで
-> **開発チームの役割**
- ・成果物をデプロイして安定運用するまで
-> **運用チームの役割**



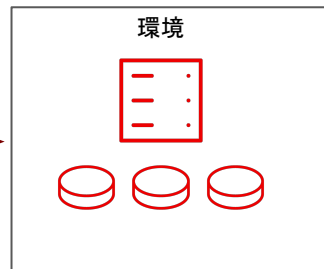
Infrastructure as Code(IaC)

宣言的な定義を思想としており
実行手順ではなく、**状態をコードで
記述**



構築

環境はコードに書かれた状態になる



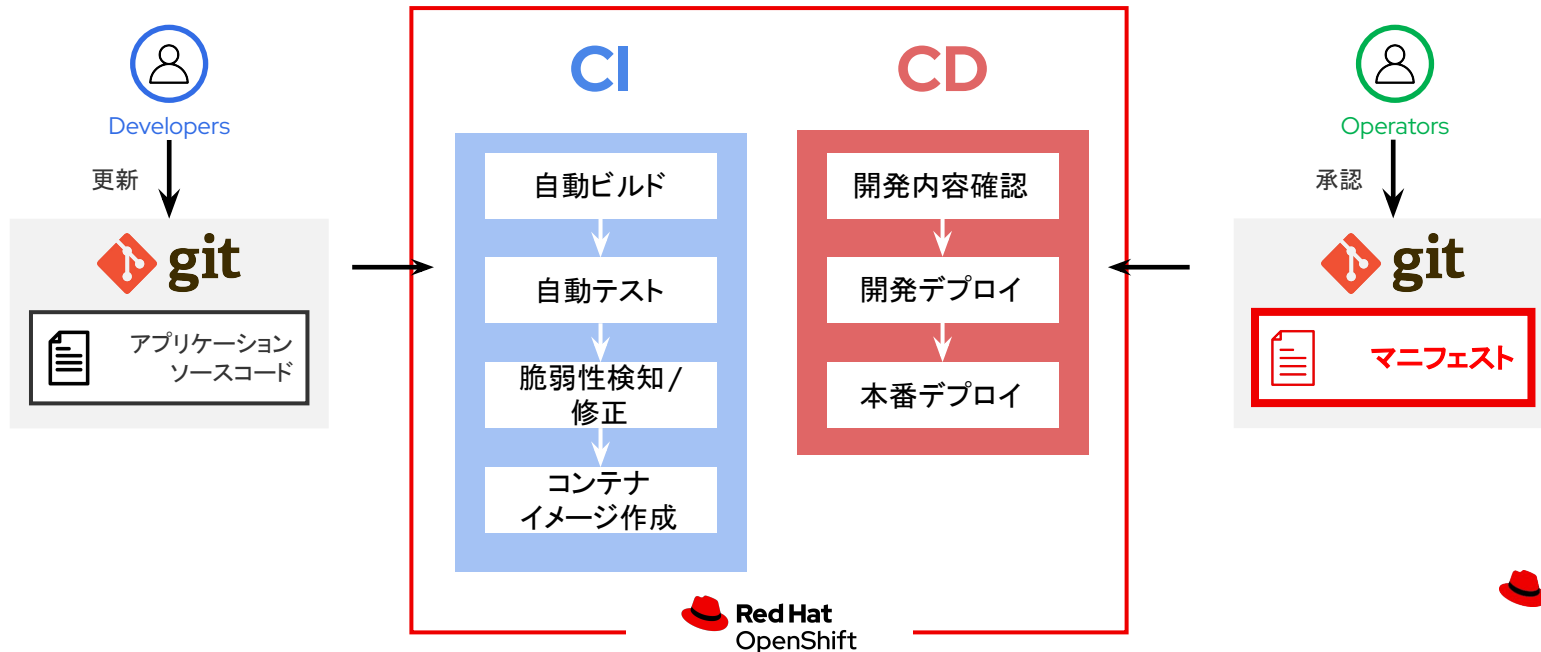
インフラ(環境)をコードで表現することが
Infrastructure as Code(IaC)の基本的な考え



コードの品質管理が重要になるため
ソフトウェア開発のナレッジ(バージョン管理/テスト)を適用する

OpenShiftのCDにおけるIaC

コードで書かれる環境 = Kubernetes マニフェスト



Kubernetes マニフェスト

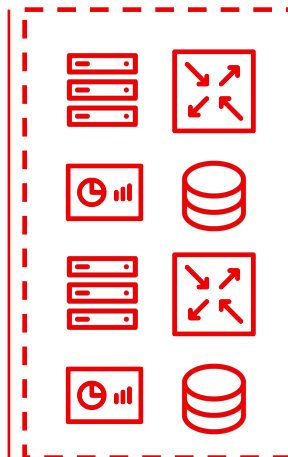
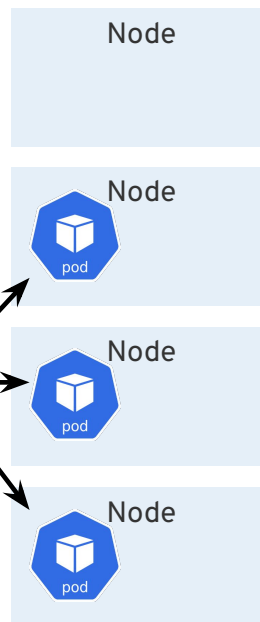
- 宣言的にKubernetesリソースのデプロイや詳細を指定



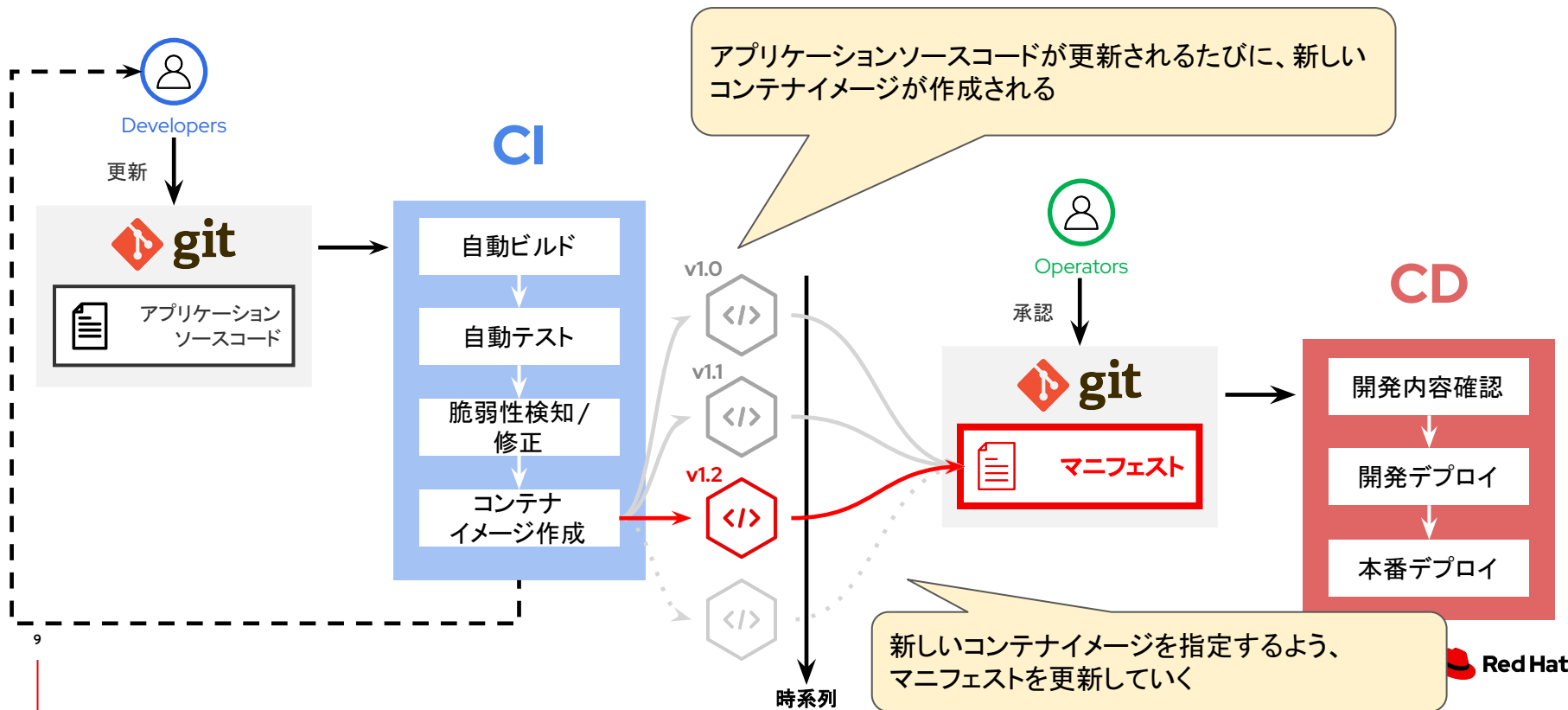
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:xxxxxxxxxxx
          ports:
            - containerPort: 80
```



Deploy



CI/CDでは継続的かつ自動でビルド/デプロイする



マニフェストにおけるコンテナイメージの指定

- 宣言的にKubernetesリソースのデプロイや詳細を指定



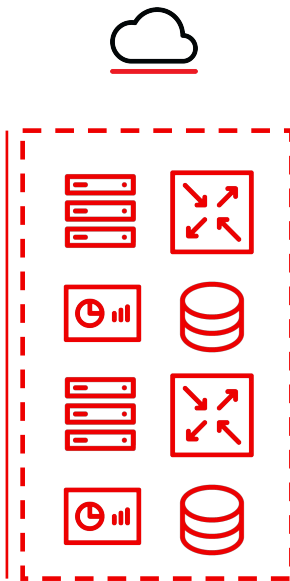
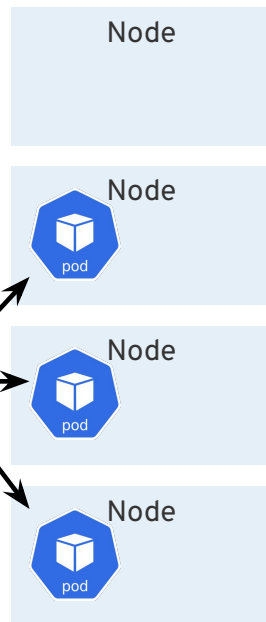
git


```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:xxxxxxxxxxx
          ports:
            - containerPort: 80
```

この部分を書き換えてコンテナイメージを指定する



Deploy





Git によるソースコード管理

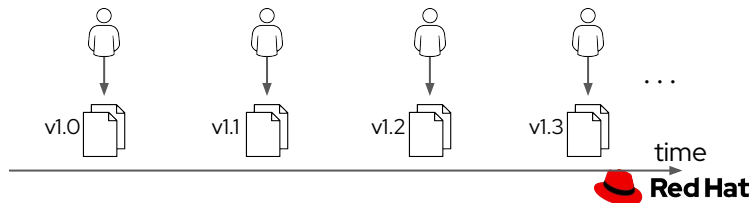
Gitとは

- バージョン管理システム (VCS: Version Control System) の1つ
- 分散したリポジトリ(保管場所)間で複製/同期をしながら管理
 - 複数人での共同管理に有用
- 今日における VCS のデファクトスタンダード
 - GitHub, GitLab, Bitbucket など様々な Git ベースのサービスがある



※バージョン管理システム (VCS: Version Control System)

- 一つのファイルやファイルの集合に対し変更を記録するシステム
- バージョン管理システムを利用することで、以下のようなことが実現できる
 - 誰が/いつ/どこをどう/なぜ変更したかの履歴を記録する
 - 過去の変更内容を比較する
 - ファイル/ディレクトリを以前の状態まで戻す



使用例: Windows PC & GitLab



開発端末(ローカル)

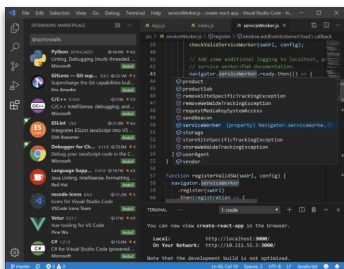


サーバー(リモート)

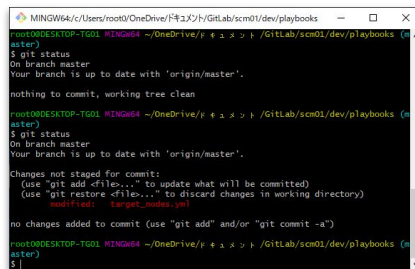
コードの開発

アップロード(Push)

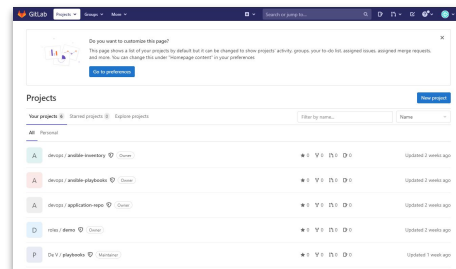
テスト・共有・管理



Visual Studio Code^[1]



Git Bash^[2]



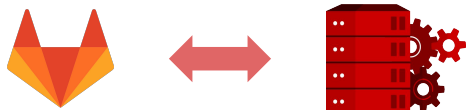
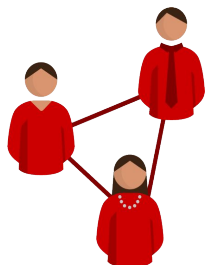
GitLab^[3]

[1] <https://code.visualstudio.com/>

[2] <https://gitforwindows.org/>

[3] <https://www.gitlab.io/>

なぜGitを使うのか



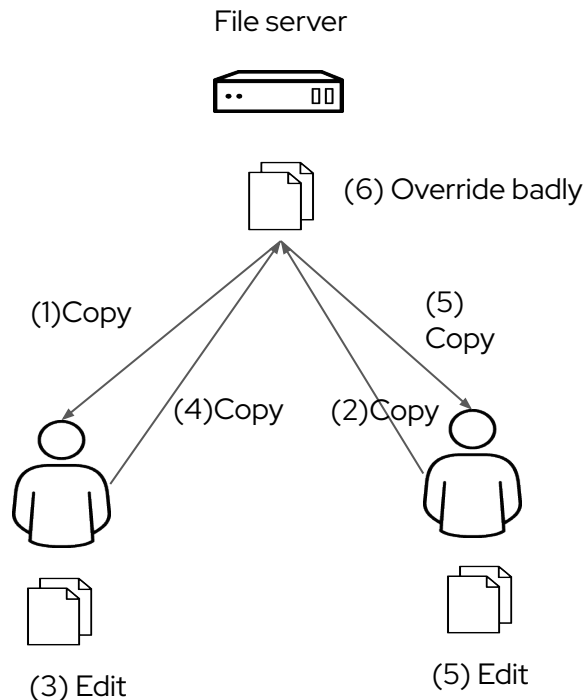
仕組みを使って「人がカバーする」余地を減らす

- 複数人でのコラボレーション
 - リポジトリ/ブランチで目的に応じた並行開発が可能に
 - 複数ユーザの変更による不整合 (コンフリクト) を検知
- 履歴はすべて Git の仕組みで記録/参照が可能

Git との連携を前提としたサービス

- 最近のツール/ソフトウェアは Git との連携を前提としているものが数多くある
- サービスの入力情報を Git で管理:
 - 役割分担を明確にする
 - Git のイベントをトリガーとして自動でアクション

Gitのない世界では...



コードの共有は「人が同期をとる」

- 1つのファイルを複数人で編集する場合に、オリジナルを上書きするタイミング/方法をケアする必要がある
- 人の力で保つため事故が発生する

Gitのない世界では...

```
$ ls -l hello.yml*  
  
hello.yml  
hello.yml.20200301  
hello.yml.20200309_suzuki  
hello.yml.2000315  
hello.yml.bk  
hello.yml.org
```

「変更履歴」情報がまとまらない

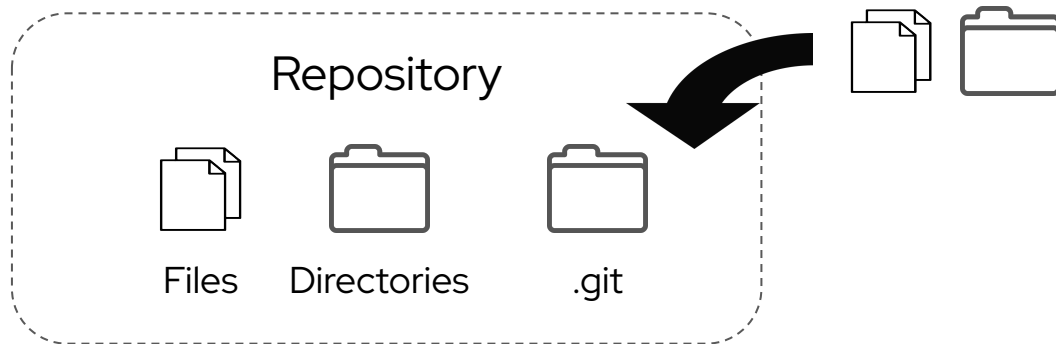
- 接尾字の追加を繰り返して大量のバックアップファイルができる
 - 変更情報 (日付, 作成者名...) をファイル名で管理
 - 名称はしばしば人に依存し、決まりは簡単にやぶられる
- 誰が/いつ/何のために作った/変更したか、まとまった情報がない
 - テキストファイルを Excel 等で管理することに

高コストな「変更追跡」の例

1. 運用手順書通りに実施したら事故が発生
2. 原因は判明したが、該当箇所は誰が/いつ/なぜ変更したのか不明
3. 過去のバックアップファイルを1つ1つ開いて確認
4. 変更履歴が適切に更新されていなかった場合は追跡不可

Gitリポジトリ

ファイルやディレクトリの変更を記録する場所



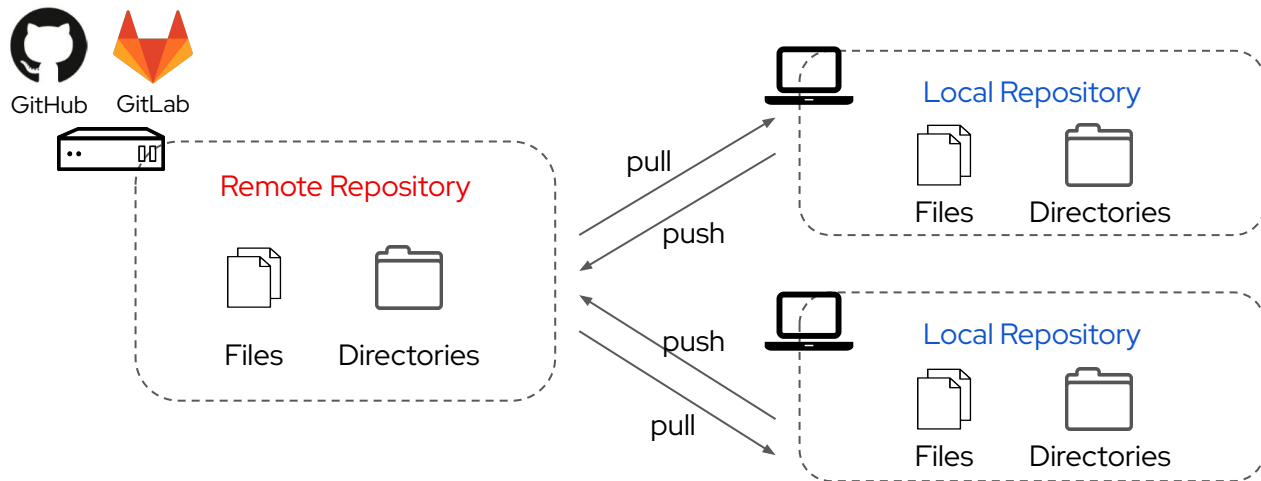
```
$ ls -ap ./your_git_repo
./ ../ .git/ .gitignore handson/
README.md

$ ls -p ./your_git_repo/.git/
branches/ COMMIT_EDITMSG config
description HEAD hooks/ index info/
logs/ objects/ packed-refs refs/
```

- 変更履歴を管理したいファイルやディレクトリ群をリポジトリ配下に配置
- gitリポジトリ直下に**.gitディレクトリ**が存在し、ファイルやディレクトリの変更を記録

リモートリポジトリとローカルリポジトリ

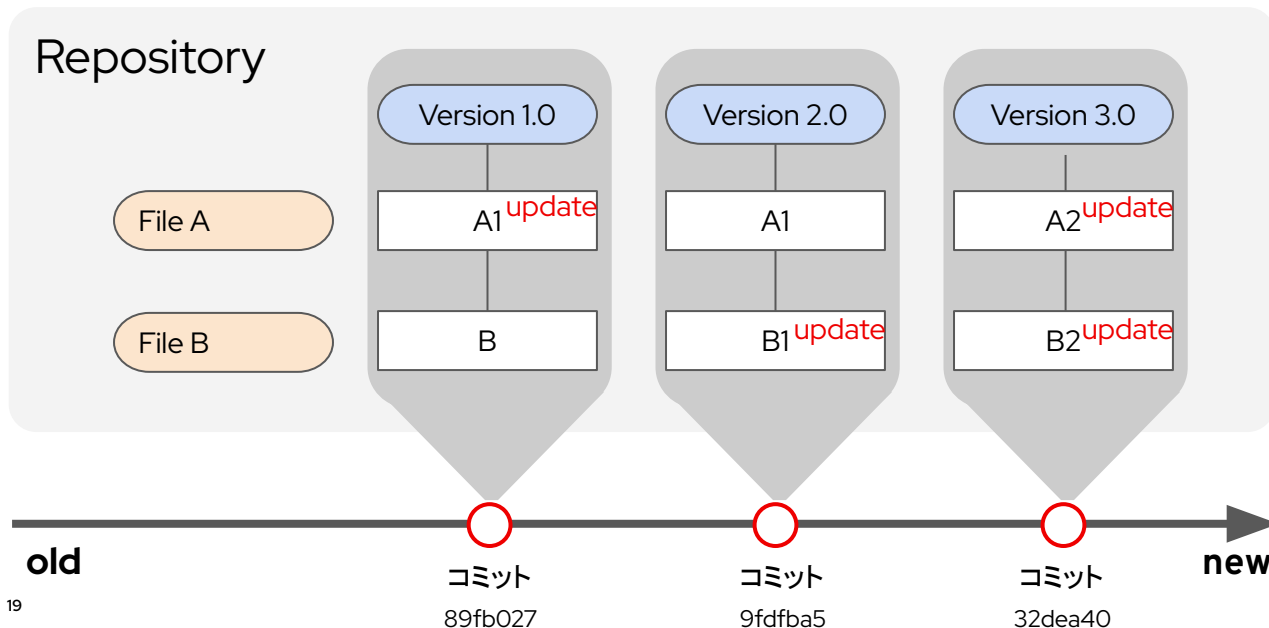
リモートリポジトリ、ローカルリポジトリの 2種類



- リモートリポジトリ: 複数人で共有するためのリポジトリ。サーバ上に作られる。(GitHub, GitLabがポピュラー)
- ローカルリポジトリ: 各個人がファイルを編集するためのリポジトリ。各端末上に作られる。
- 開発者はリモートリポジトリからローカルリポジトリにファイルを反映 (pull)し、編集後にリモートリポジトリへ変更を反映する (push)

コミット

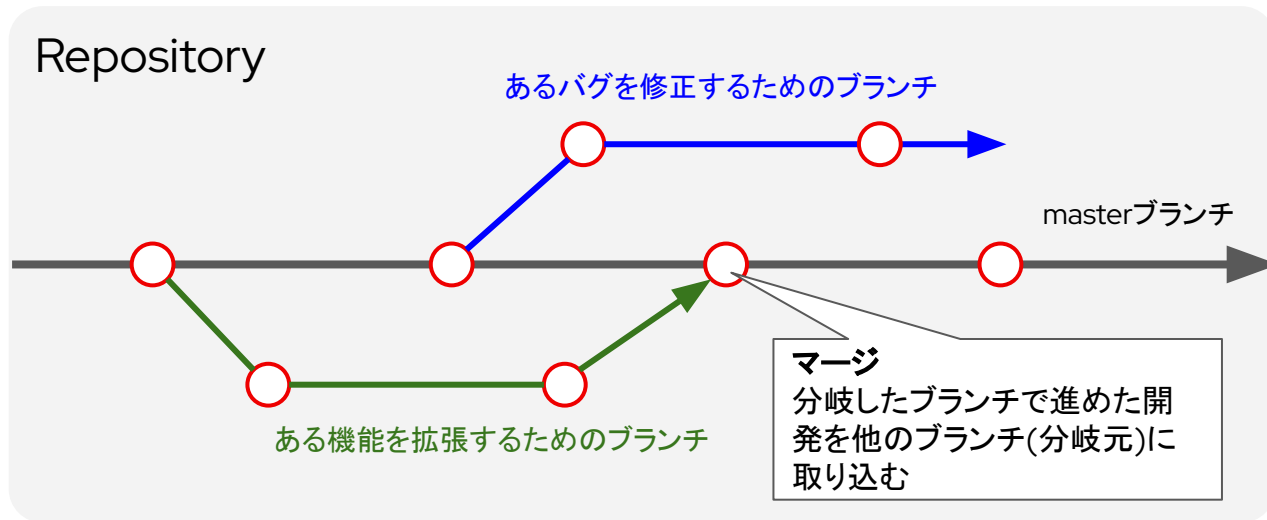
ファイルの変更を記録するための操作または変更記録そのもの
時系列に並んだコミットがリポジトリの歴史



- コミットすると、前回コミット時の状態から現在までの変更記録を作成する。
- この変更記録はコミットまたはリビジョンと呼ばれる。
- コミットからハッシュ値を生成してIDとして利用する。
- コミットはメタ情報を含む。
 - コミットオーサ
 - コミット日時
 - コミットメッセージ

ブランチ

コミットの流れを分岐させて記録するための機能
異なる目的の変更を並行で行うことが可能となる

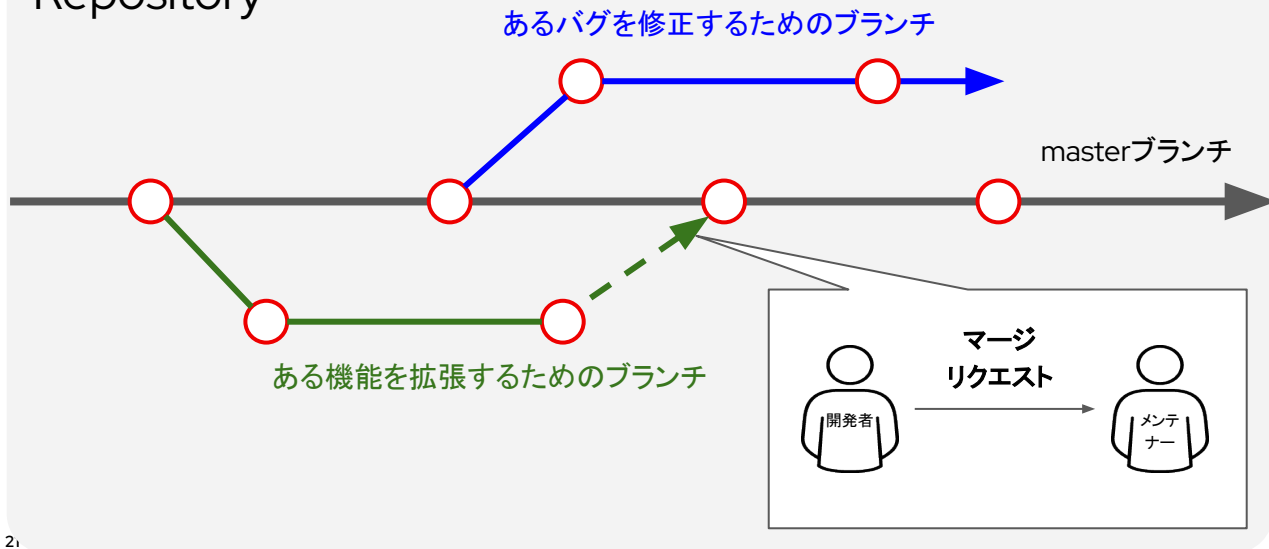


- リポジトリ作成時点では、masterブランチのみ存在する。
- 各ユーザでブランチを作成して切り替えて使う。
- あるブランチのコミットは、他のブランチに影響しない。
- 分岐したブランチは、マージすることで元のブランチに反映できる。

マージリクエスト

ブランチ間で変更内容を反映させるためのリクエスト
チーム内で変更内容の検証などを行い、コードの品質を維持できる

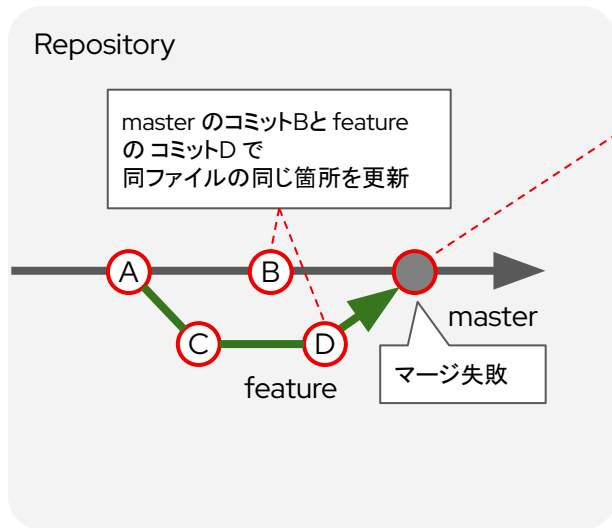
Repository



- マージはメンテナーなど限られた役割だけが実行できるようにするのが一般的である。
- 一般ユーザは自身のブランチに加えた変更をマージしたい場合は、マージリクエストを作成する。
- メンテナーはマージする対象の変更内容をレビューし、リクエストを承認 / 拒否する。
- Gitそのものの機能ではなく、VCSが提供する機能である。
 - GitHubでは『プルリクエスト』と呼ばれる

[参考]コンフリクトとその対処

同じファイルの同じ箇所を2つのブランチで変更したことで発生する不整合



```
$ git merge feature
Auto-merging hello.yml
CONFLICT (add/add): Merge conflict in
hello.yml
Automatic merge failed; fix conflicts and
then commit the result
```

```
$ vi playbook.yml
<<<<<<< HEAD
- name: Ensure httpd is present
=====
- name: Ensure httpd is existing
>>>>>> feature

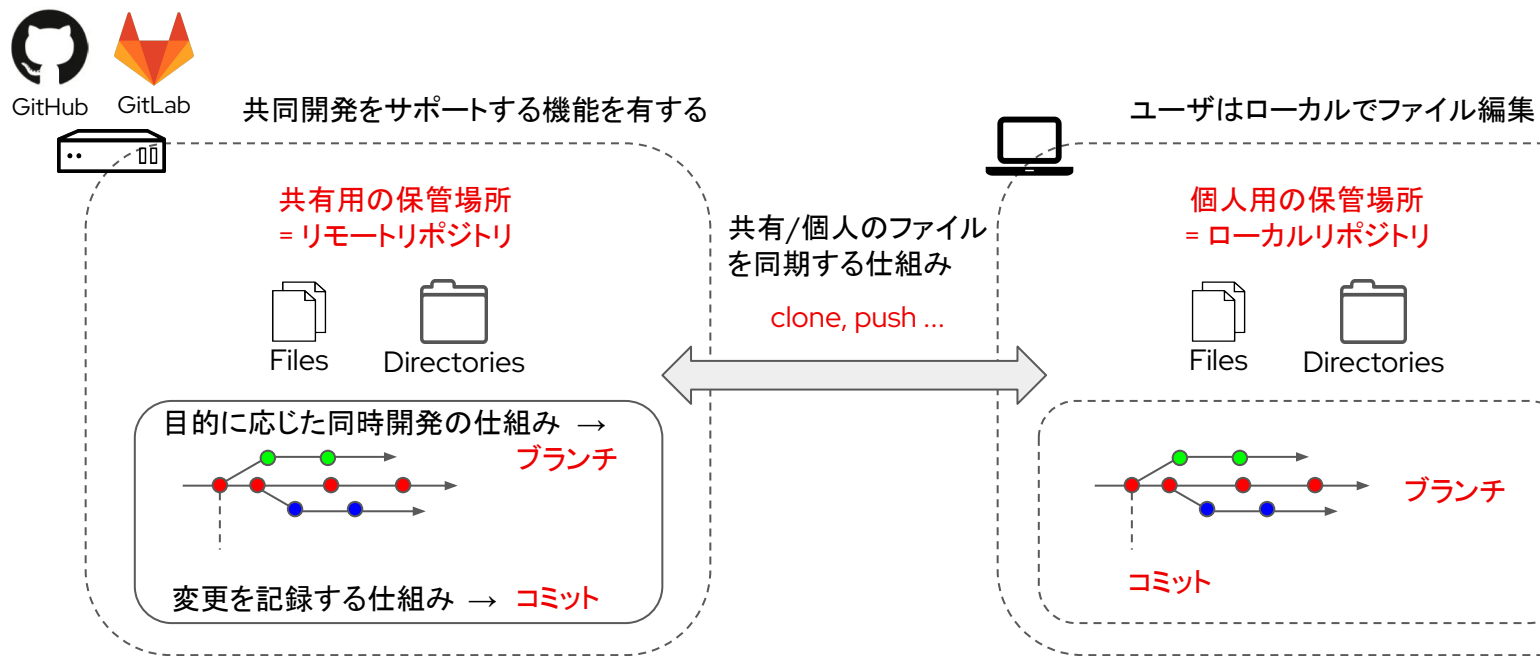
# 例えば上記を以下のように修正

- name: Ensure httpd is installed

$ git commit -m"modify: conflict between
master and feature"
```

- コンフリクトが発生するとマージ処理は中断され、対象のファイルにコンフリクトマーカーが付与される。
- 対象ファイルを開き、コンフリクトマーカーの情報を元に内容を修正する。
- 対象ファイルを修正した後は、再度 git commit コマンドを実行しコミットし直す。

Git アーキテクチャ全体像



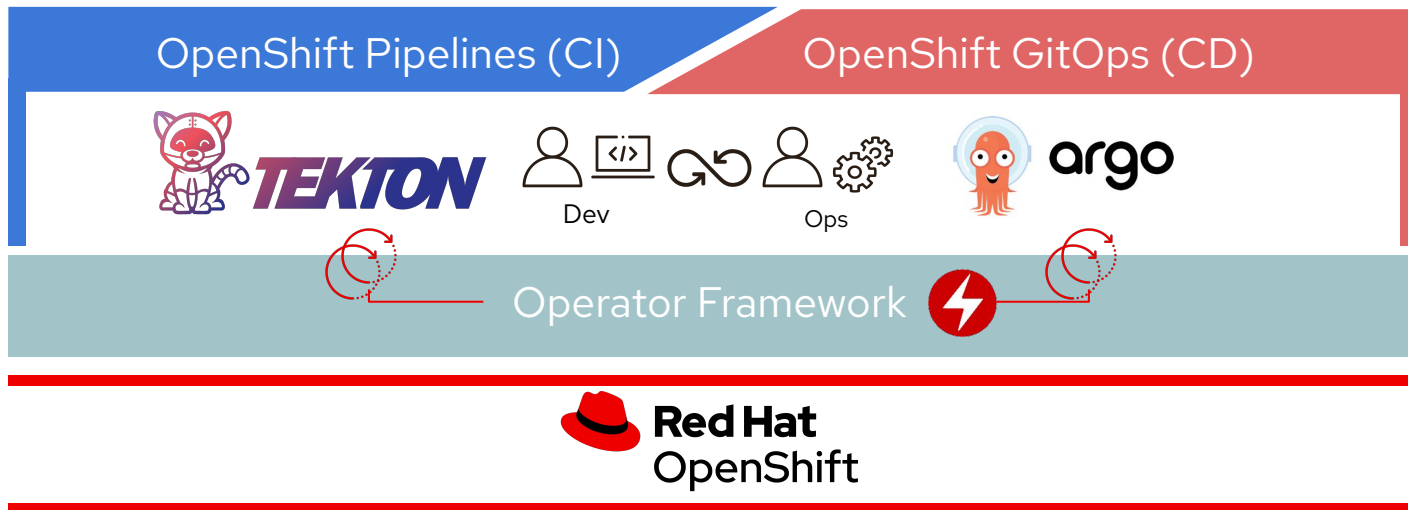


CICD と GitOps

OpenShift Pipelines / OpenShift GitOps

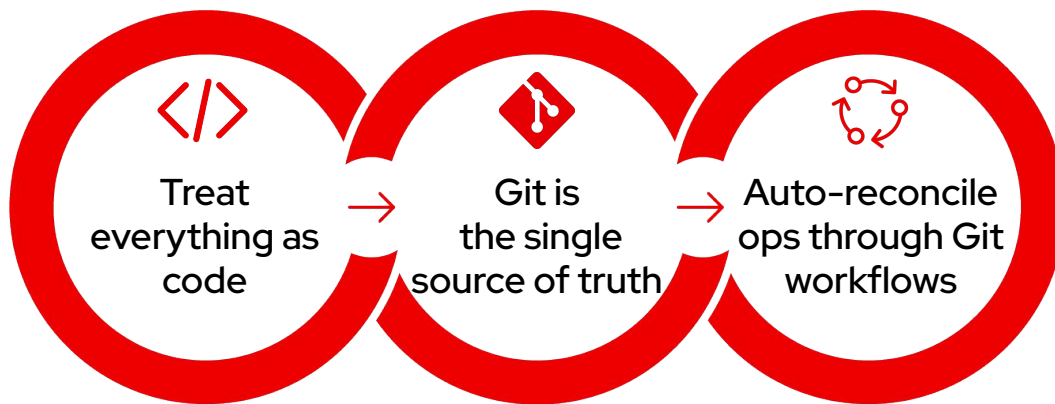
OpenShiftにおけるCI/CD

OpenShiftでは、CIをOSSのTektonをベースとするOpenShift Pipelinesで、CDをArgo CDをベースとしたOpenShift GitOpsにて実施します。それぞれOperatorを使いインストール/管理されます。



GitOpsとは？

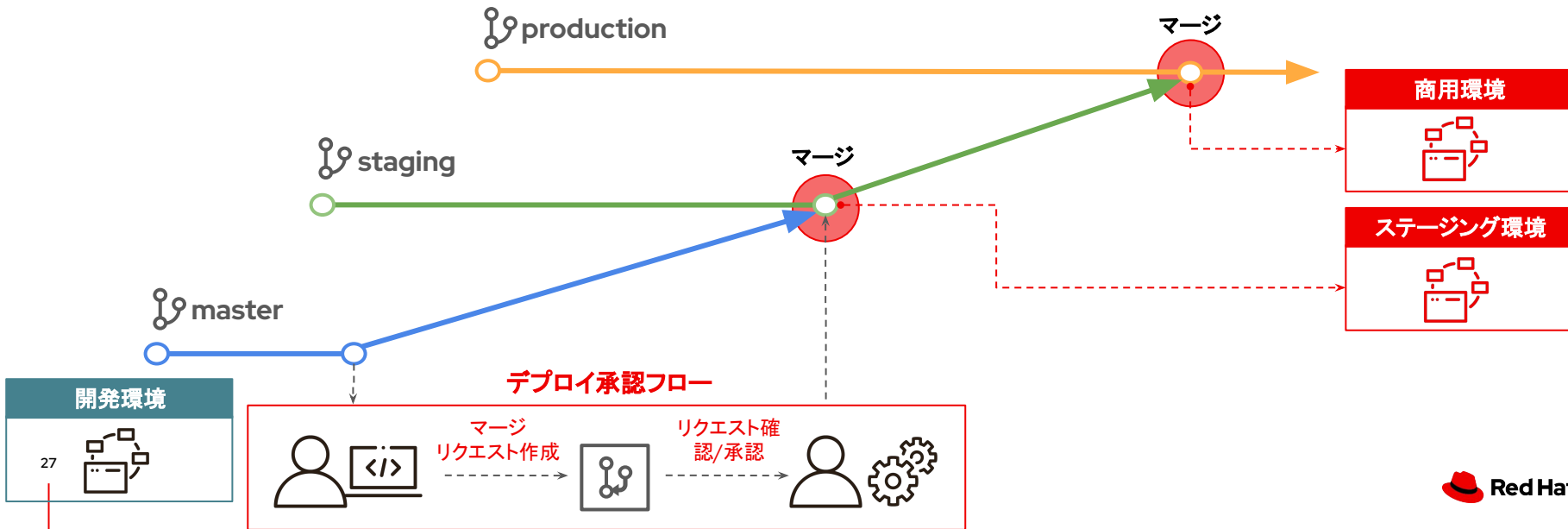
- Git によるソースコード管理を前提にインフラとアプリを構成管理するプラクティスを示す
- Kubernetes では Git リポジトリに格納した Deployment や Service などのマニフェストを **Single source of truth** として扱い、クラスタに登録された設定と Git 上のマニフェストを一致させる
 - 最新コミットのマニフェストを自動でクラスタにデプロイ
 - 手動でリソースを変更しても Git リポジトリの状態と一致しなければアラートを上げたり、リポジトリ側を正としてリソースを自動修正
 - 変更は全てGitで管理されるため、ロールバックが容易



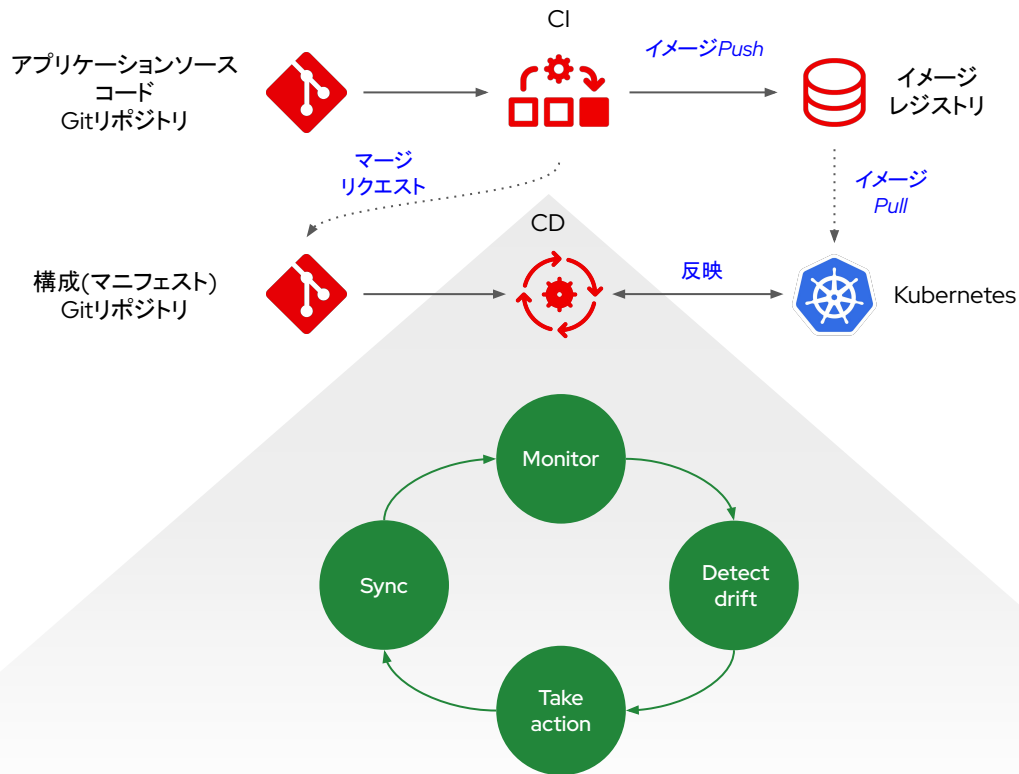
GitOpsのイメージ

Gitのブランチとマージリクエストを使ったデプロイ

CDツールを使ってステージングや商用環境にデプロイする場合、Gitのマージリクエストの仕組みを使って承認フローを構築することができます。承認者はリクエストの中身を確認し、ブランチをマージしていくことで対応する環境へのデプロイが行われます。



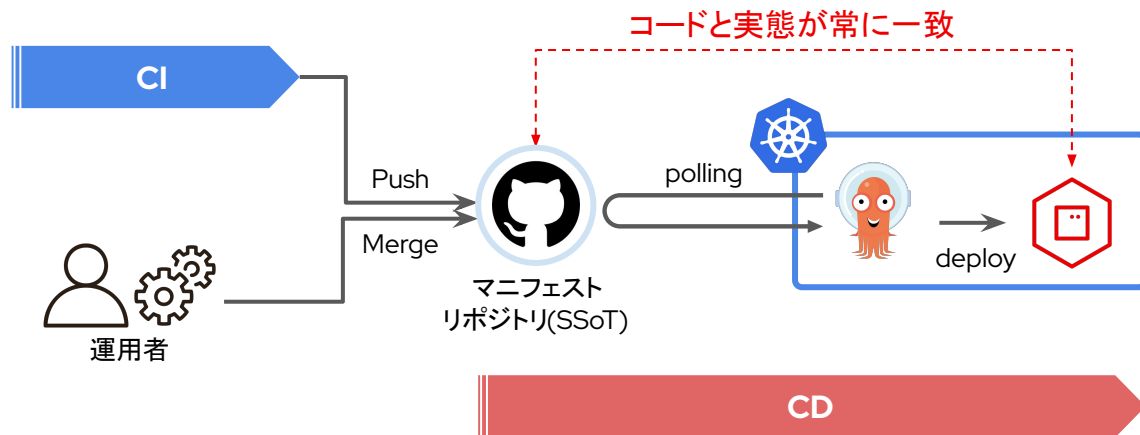
GitOpsのアプリケーションデリバリーモデル



コードによるインフラの管理

GitOpsによる運用を実現

Gitリポジトリに置かれたコードをSingle Source of Truthとみなし、コードとインフラを常に同じ状態に保つ運用のベストプラクティスをGitOpsと呼びます。Argo CDはGitリポジトリの状態を定期的を確認し、変更を自動的にデプロイすることでGitOpsを実現します。



作業の属人化を防止

デプロイはArgo CDが自動で行うため、デプロイ作業が特定の個人のスキルに依存することがなくなります。



デプロイ状況の見える化

リポジトリのコードを見れば、クラスターに今何がデプロイされているか確認することができます。

OpenShift GitOps



宣言的なデプロイメントの実現

- Gitに記録されるマニフェスト情報を正とするデプロイメント
- マニフェストからの逸脱 (ドリフト) の検知と自動修正



マルチクラスタへの対応

- 複数のOpenShift/Kubernetesクラスタにおけるデプロイメントの管理・制御
- 複雑なデプロイ戦略への柔軟な対応



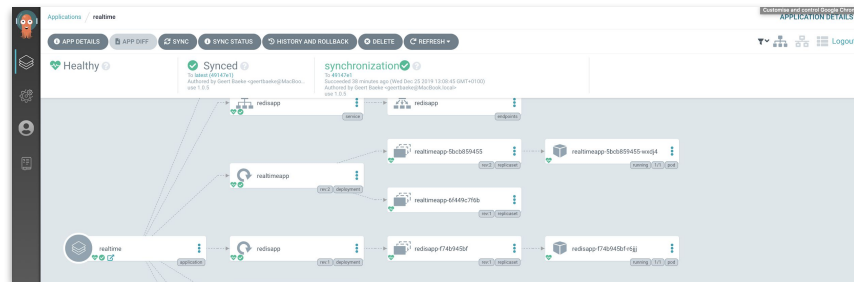
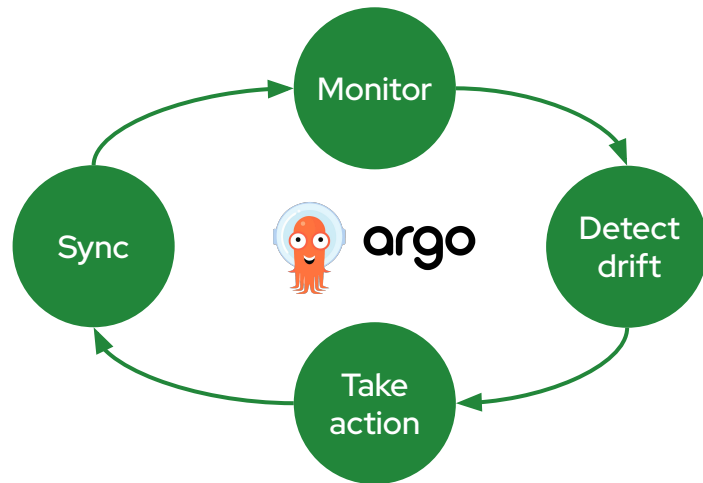
デプロイメントと環境の可視化

- デプロイメントとKubernetesリソースの可視化
- デプロイメント履歴の表示とロールバック



自動的なインストールと構成

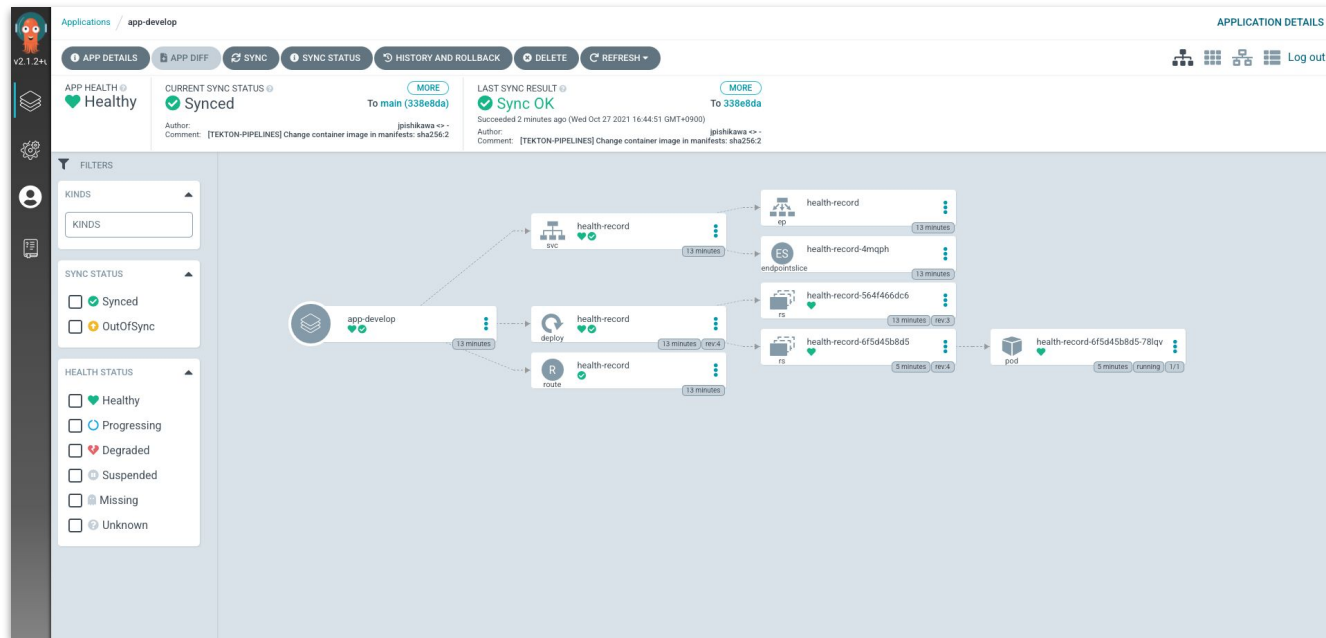
- OperatorHubを使った自動的なインストール・構成・アップグレード



Argo CDコンソール

デプロイされたアプリケーションの状況を可視化

Argo CDコンソールでは現在デプロイされているアプリケーションの状態をトポロジービューとして確認できます。また特定の過去バージョンへのロールバックなどについてもコンソールから実施できます。



CD ハンズオン

ハンズオンの流れ

1 CDのための事前準備

- ・GitリポジトリとOpenShift namespaceの準備
- ・CIパイプラインの作成と実行

2 OpenShift GitOpsとGitを使ったアプリのデプロイ

- ・Argo CDによるGitOpsの体験(Auto-sync)
- ・Gitブランチを使ったステージング/本番へのデプロイ

3 アプリの更新をステージング/本番へ適用

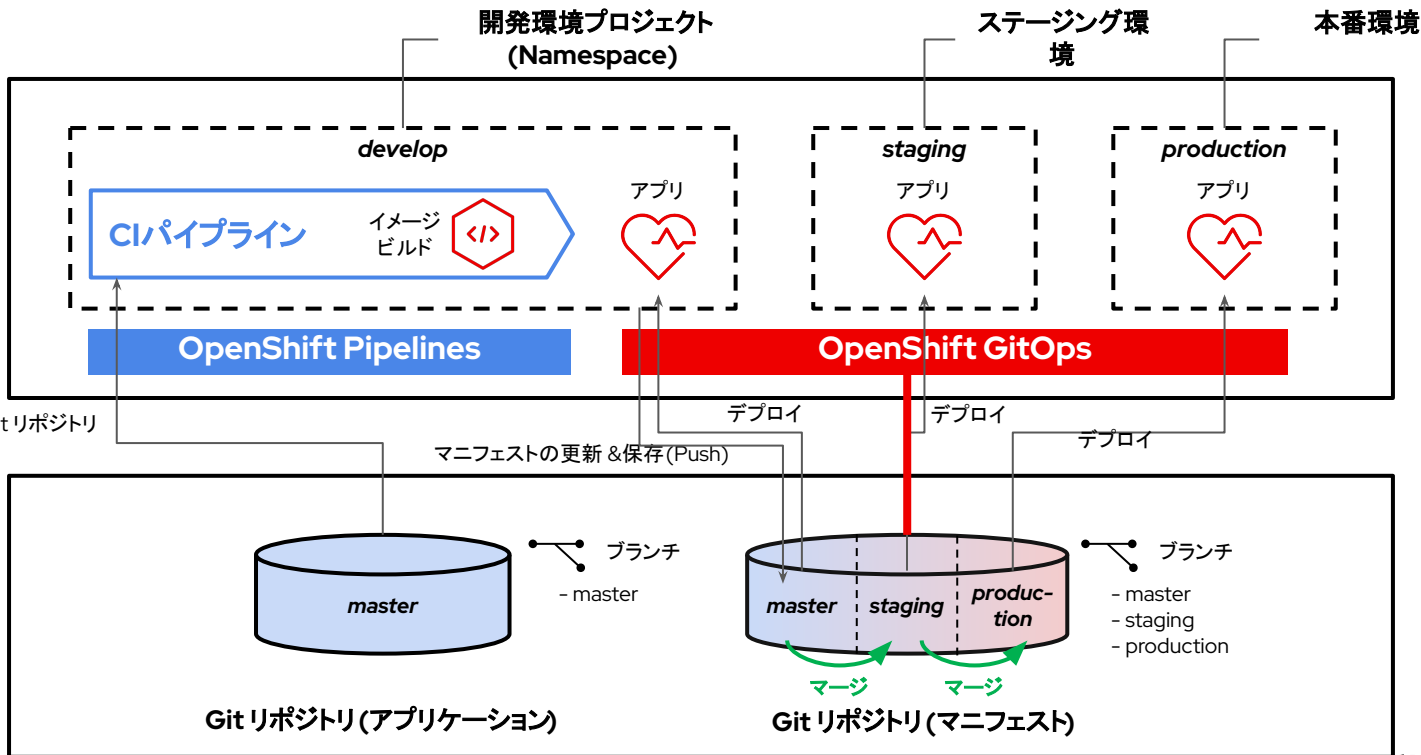
- ・GitLabのマージリクエストを使った更新適用
- ・アプリケーションのロールバック

ハンズオンシナリオ

開発環境で CI パイプラインを用いてコンテナイメージをビルドし、次に同じくパイプラインを用いてステージングや本番環境にビルドしたイメージをデプロイします。

ハンズオンのリンクはこちらになります。 <https://gitlab.com/openshift-starter-kit/cd-practice>

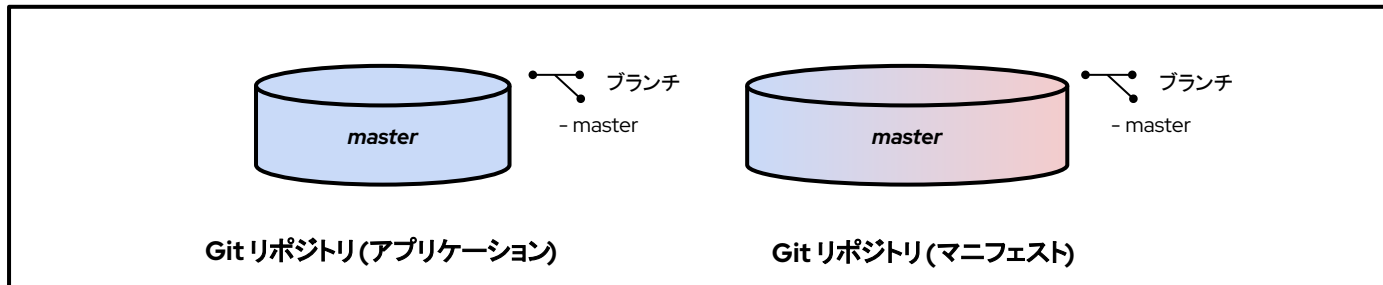
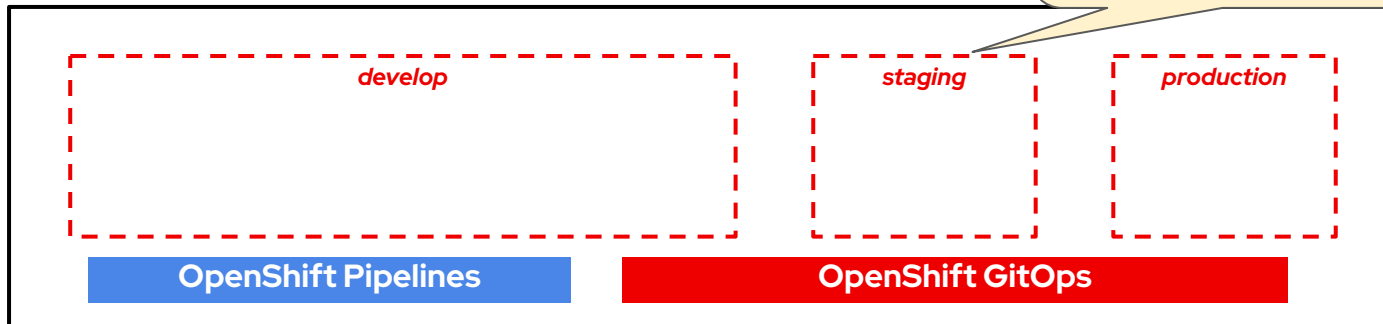
パイプラインでデプロイする環境の全体像



ステップ1: CDのための事前準備

- ▶ OpenShift クラスタからプロジェクト(namespace)を作成する

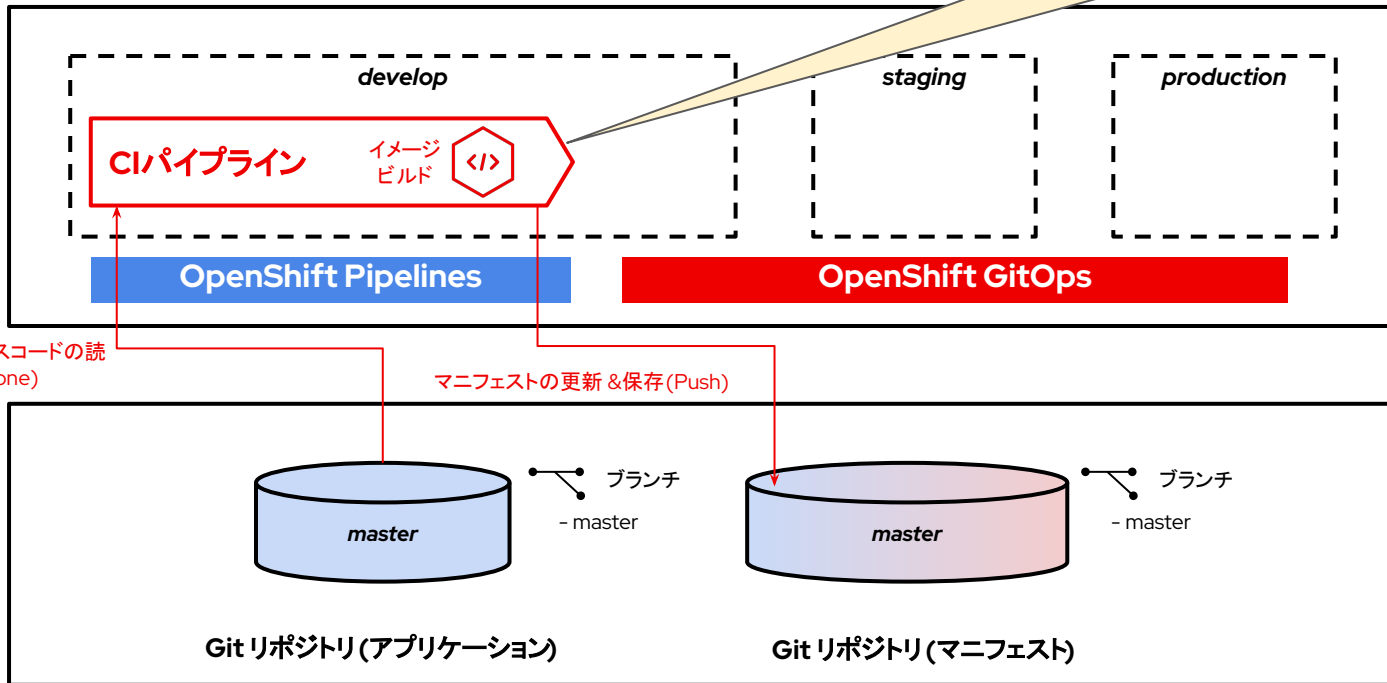
- develop: 開発環境
- staging: ステージング環境
- production: 本番環境



ステップ1: CDのための事前準備

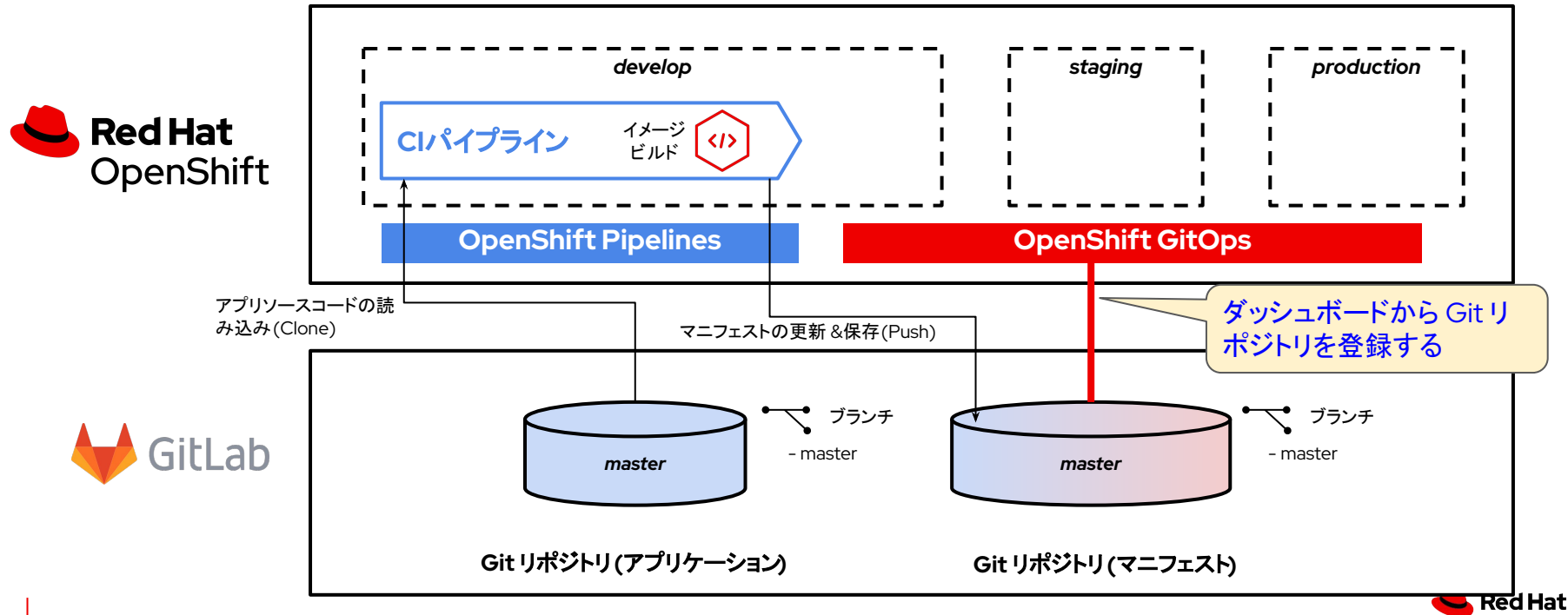
- ▶ 開発環境でCIパイプラインを作成・実行する

- Git リポジトリのクローン
- コンテナイメージのビルド
- マニフェストの更新



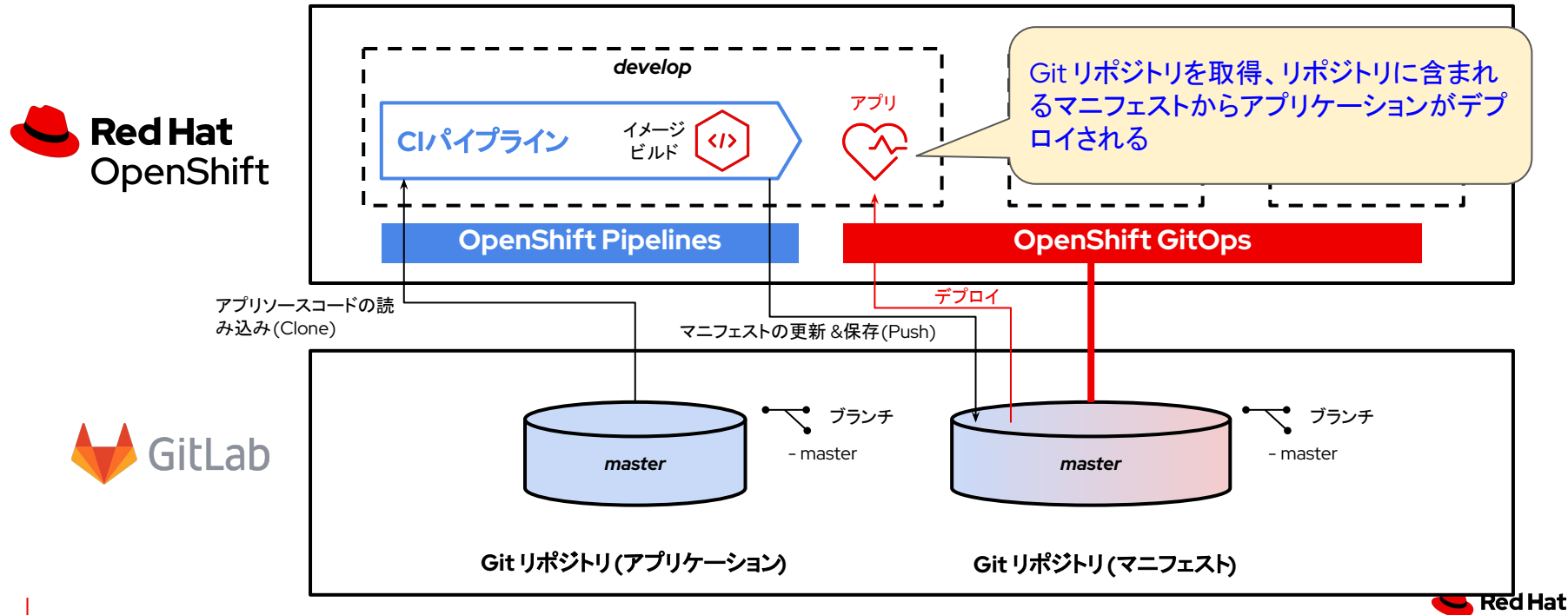
ステップ2: OpenShift GitOpsとGitを使ったアプリのデプロイ

- ▶ マニフェストを格納するGitリポジトリを登録する



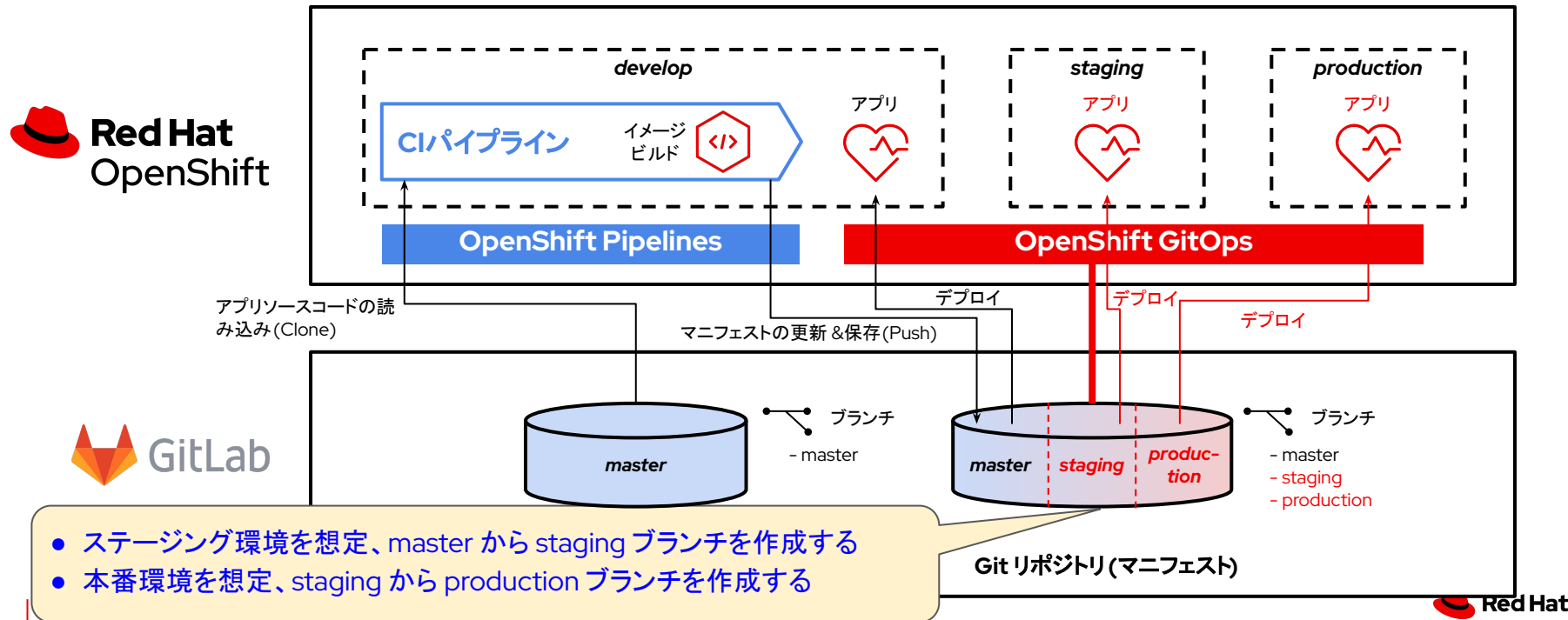
ステップ2: OpenShift GitOpsとGitを使ったアプリのデプロイ

- ▶ 開発環境にアプリケーションをデプロイする



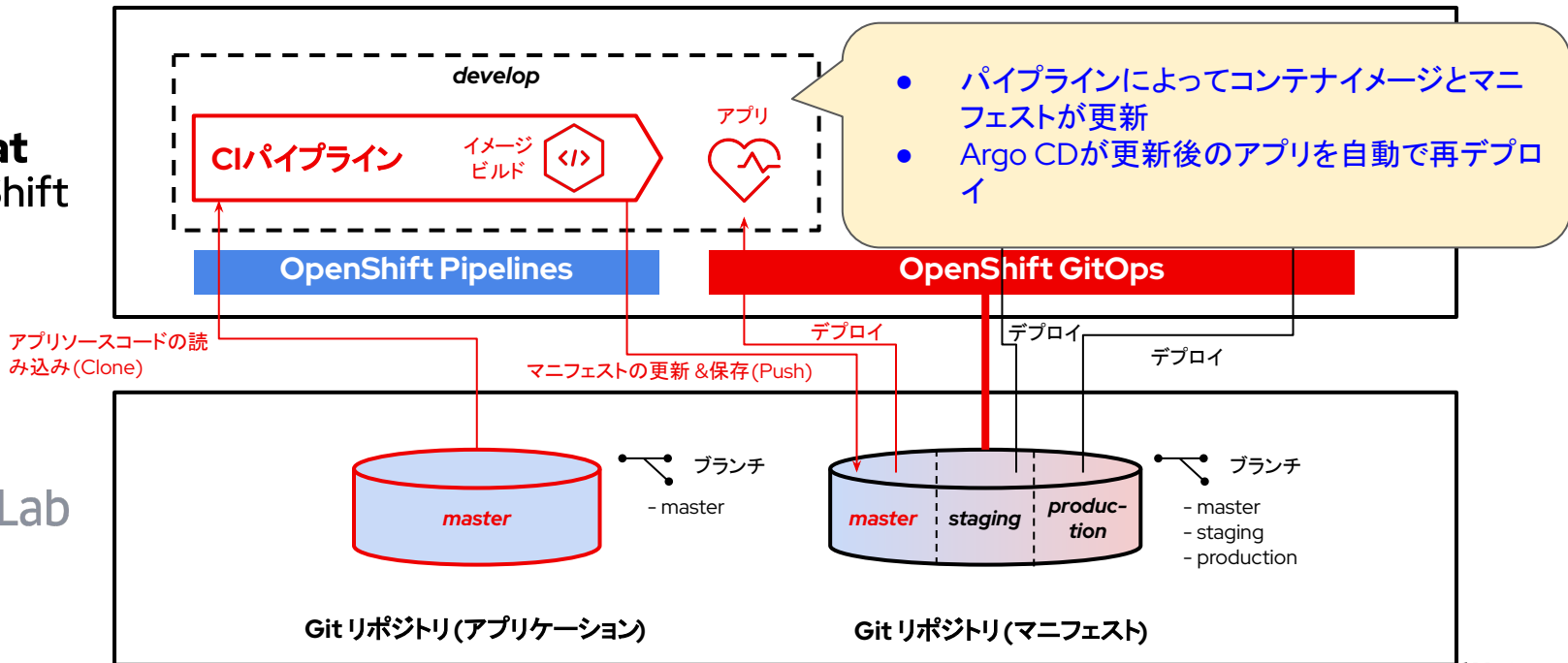
ステップ2: OpenShift GitOpsとGitを使ったアプリのデプロイ

- ▶ ステージングと本番環境向けにGit リポジトリのブランチを作成する



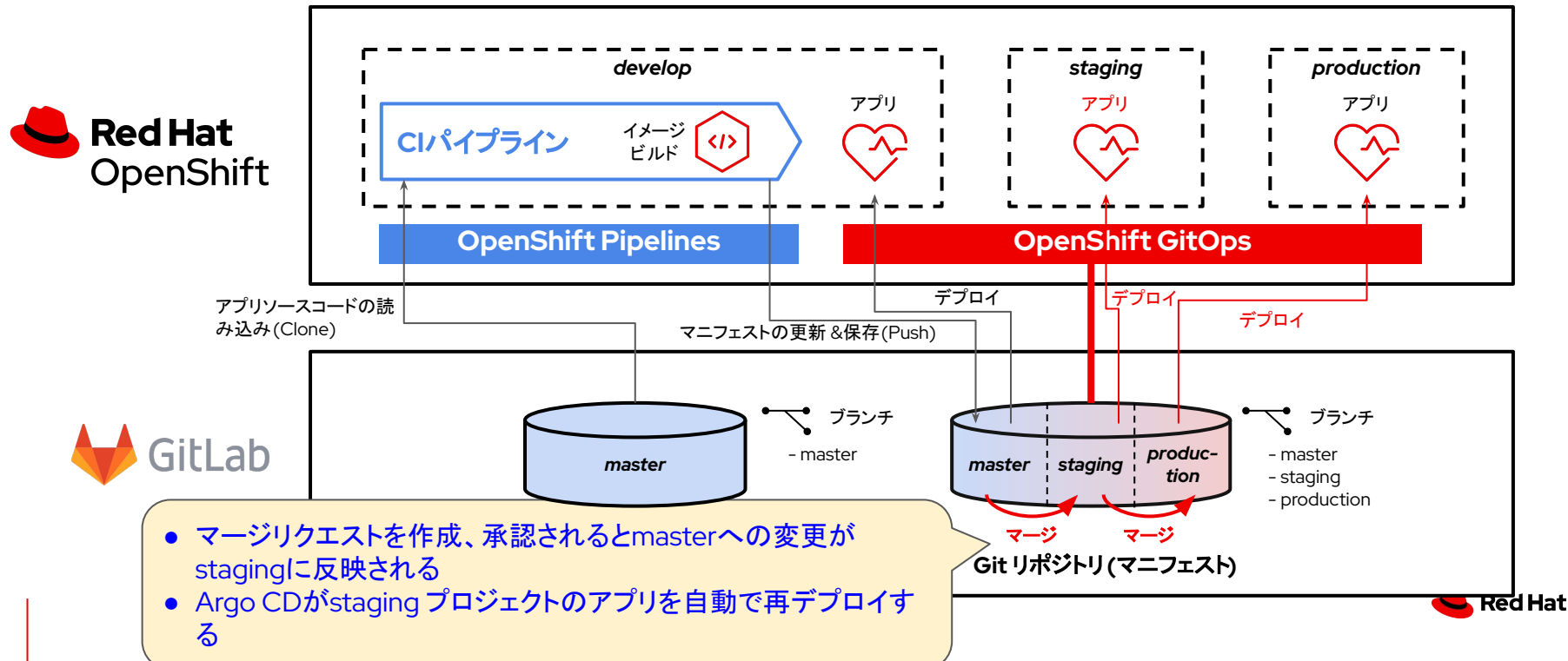
ステップ3: アプリの更新をステージング/本番へ適用

- アプリケーションのソースコードを更新、パイプラインを再実行する



ステップ3: アプリの更新をステージング/本番へ適用

- ▶ ステージングと本番環境にアプリケーションをデプロイする

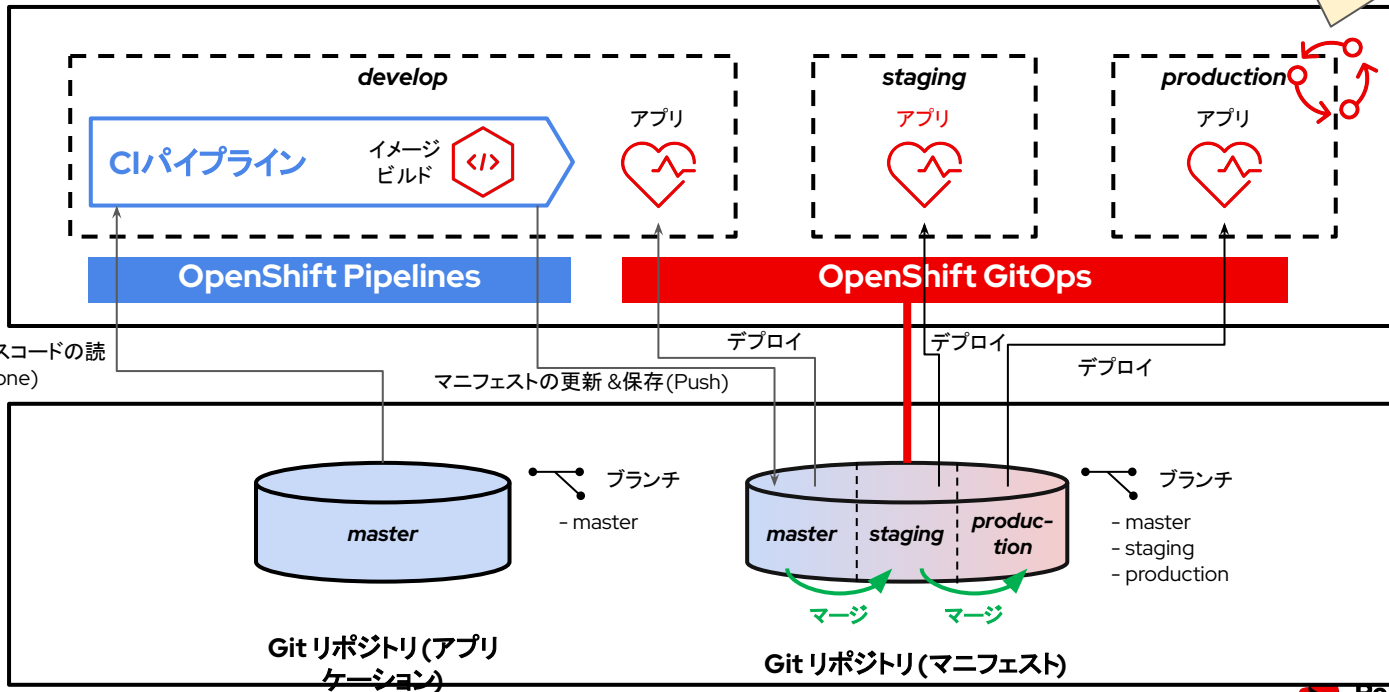


- マージリクエストを作成、承認されるとmasterへの変更がstagingに反映される
- Argo CDがstagingプロジェクトのアプリを自動で再デプロイする

ステップ3: アプリの更新をステージング/本番へ適用

- ▶ 本番環境でアプリケーションをロールバックする

更新前のバージョンにロールバックさせる



Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.



linkedin.com/company/red-hat



youtube.com/user/RedHatVideos



facebook.com/redhatinc



twitter.com/RedHat