アプリケーション開発・運用の変革とサーバ構築の自動化

Ver1.2 2013/11/26
中井悦司（Twitter @enakai00）

自己紹介

- 中井悦司（なかいえつじ）
  - Twitter @enakai00

- 日々の仕事
  - Senior Solution Architect and Cloud Evangelist at Red Hat K.K.
    企業システムでオープンソースの活用を希望される
    お客様を全力でご支援させていただきます。

- 昔とった杵柄
  - 素粒子論の研究（超弦理論とか）
  - 予備校講師（物理担当）
  - インフラエンジニア（Unix/Linux専門）

「Linux独習書の新定番」
書きました！

読者の声より ——
「今はインターネット上に情報が溢れているけど、質の高い入門書が少なくなっているのは不幸なことだと思う。そんな中、この本はすごくいい」
「平易な言葉でありながら、決して足りなくはない。慎重に選ばれています。脳みそに染みこんで来ます」

はじめに：企業システムの自動化における問題意識

- クラウドコンピューティングの実現により、さまざまな「インフラの自動化」が可能になりました。
  - アクセス変動に敏感なWebアプリケーション（現在のパブリッククラウド利用の主流）では、「オートスケール」などは確かに便利な機能です。

- 今後、一般企業の業務アプリケーションをクラウドで利用する場合、どのような観点で、どのような処理を自動化することが必要なのでしょうか？
  - 既存の手作業をそのまま自動化するのでは、「自動化のための自動化」に陥る危険性はないでしょうか？
  - 「アプリケーション開発、システム運用プロセスの変革/改善と一体化した自動化の適用」を研究するべき時期がきているのではないでしょうか？

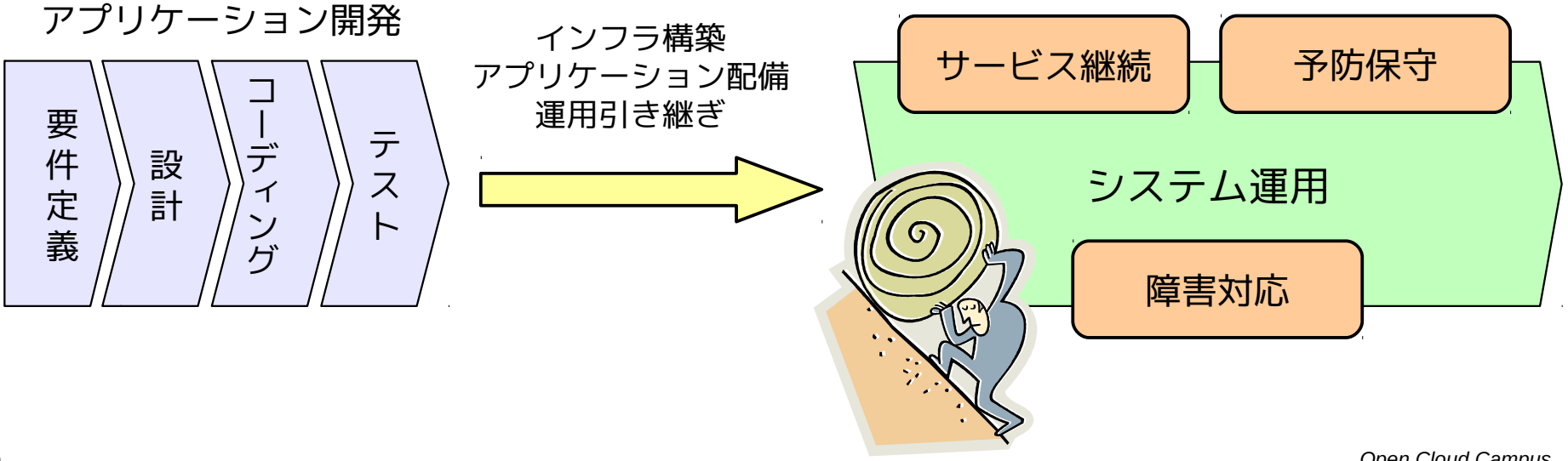
Contents

- システム運用の現状と課題
- 運用自動化への道 – ソフトウェアライフサイクルの観点から
- アプリケーションデプロイの自動化の現状
- JEOS方式の自動化ツール
- サービス継続性から見る自動化の適用と考え方

システム運用の現状と課題

システム運用業務の目的

- サービスの継続提供
  - システムの機能/性能を維持して安定したサービスを継続すること
- システムの予防保守
  - 障害が発生しないように健全なシステム状態を保持すること
- 障害対応
  - 障害発生時に迅速にサービスを復旧して、再発を防止すること

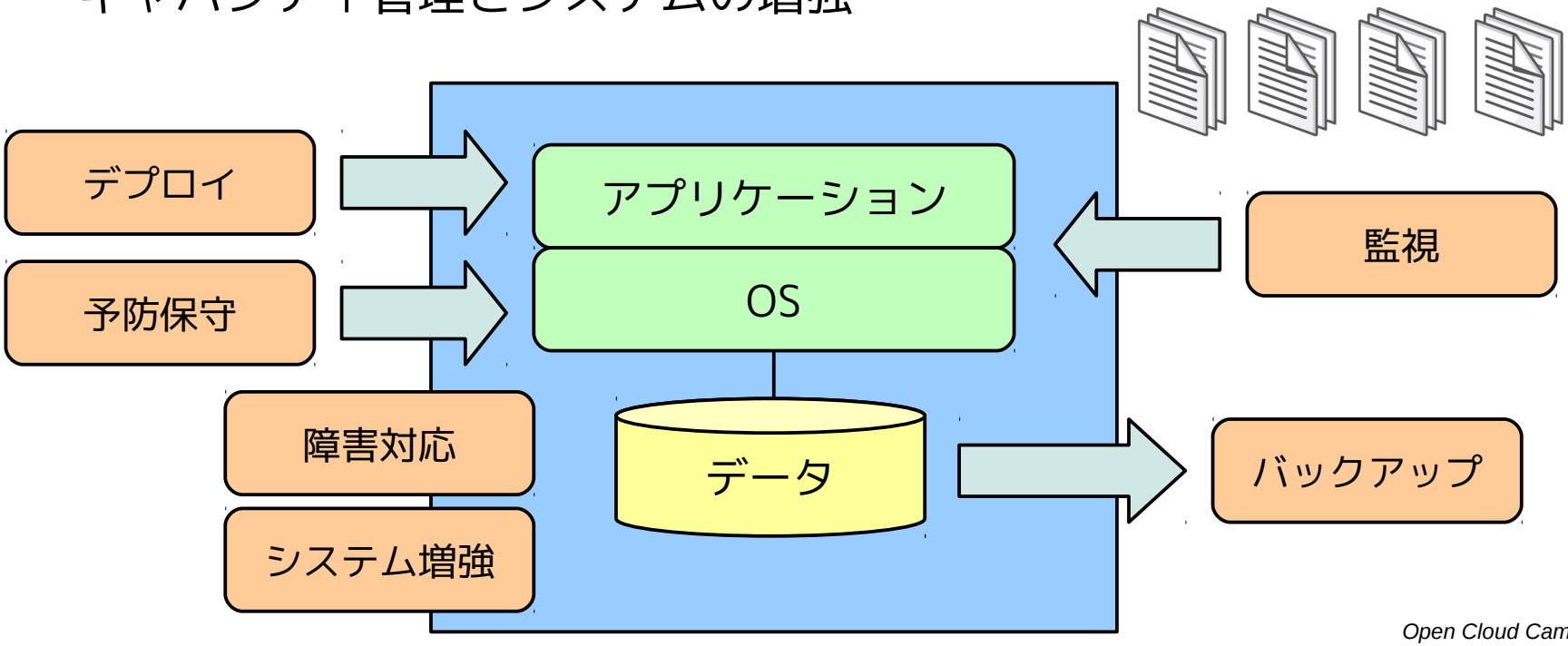


システム運用業務の現状

運用手順書による
手作業が前提

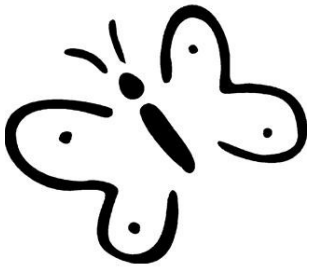
- 主な運用業務
  - アプリケーション配備（デプロイ）
  - 予防保守
  - システム監視と障害対応
  - バックアップ
  - キャパシティ管理とシステムの増強

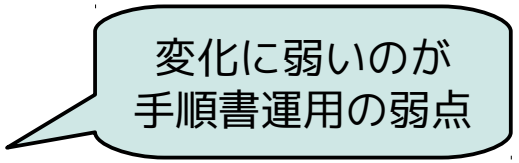
- OS導入手順書
- アプリケーション導入手順書
- パッチ適用手順書
- バックアップ手順書
- 障害対応手順書
- etc...

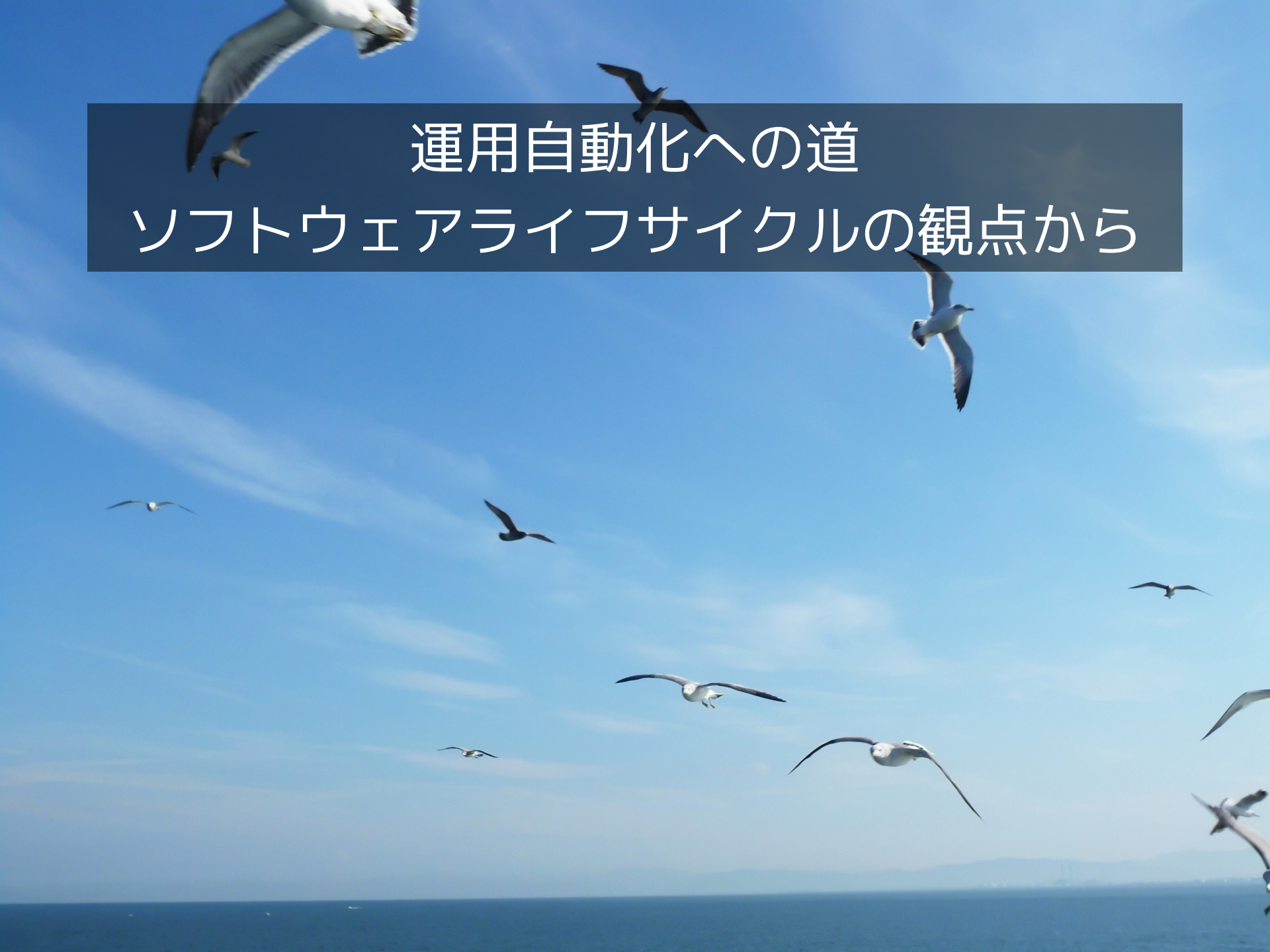


手順書ベースの運用の課題

- 運用中のソフトウェアの変更に時間/コストがかかる
  - ビジネス要件とそれを支える技術、双方の変化の速度が上がり、アプリケーション、ミドルウェアなどの頻繁な更新が必要に。
  - しかしながら、ソフトウェア変更に伴う手順書の書き換え、検証、手順変更の周知徹底などに時間とコストがかかり、変更を気軽に行うわけにはいかず・・・。
- 環境の複雑化への対応が困難
  - 仮想化によりシステム環境が複雑化し、動的な変更・変化も可能に。
  - しかしながら、それに伴う運用手順も複雑になり、作業時間が長くなる、作業ミスによる問題発生が起きやすくなるなどの課題が・・・。



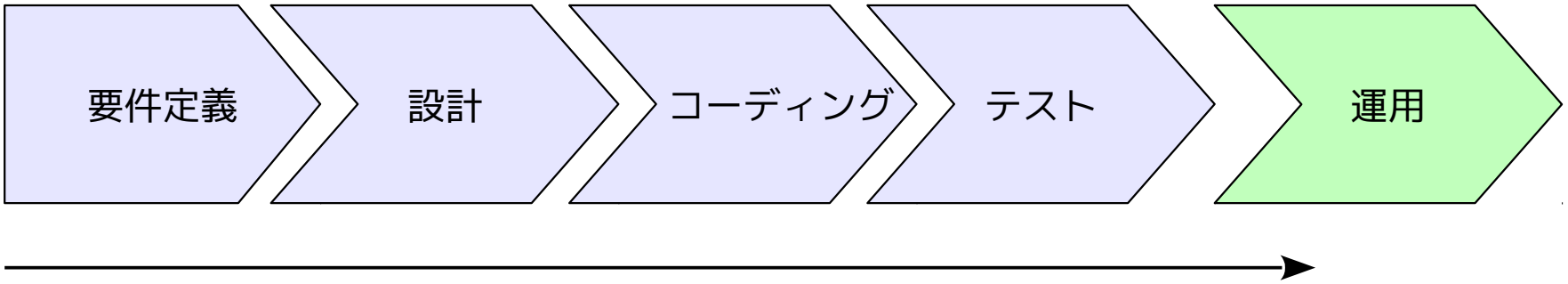




運用自動化への道
ソフトウェアライフサイクルの観点から

ソフトウェアの変更要求が高まる背景

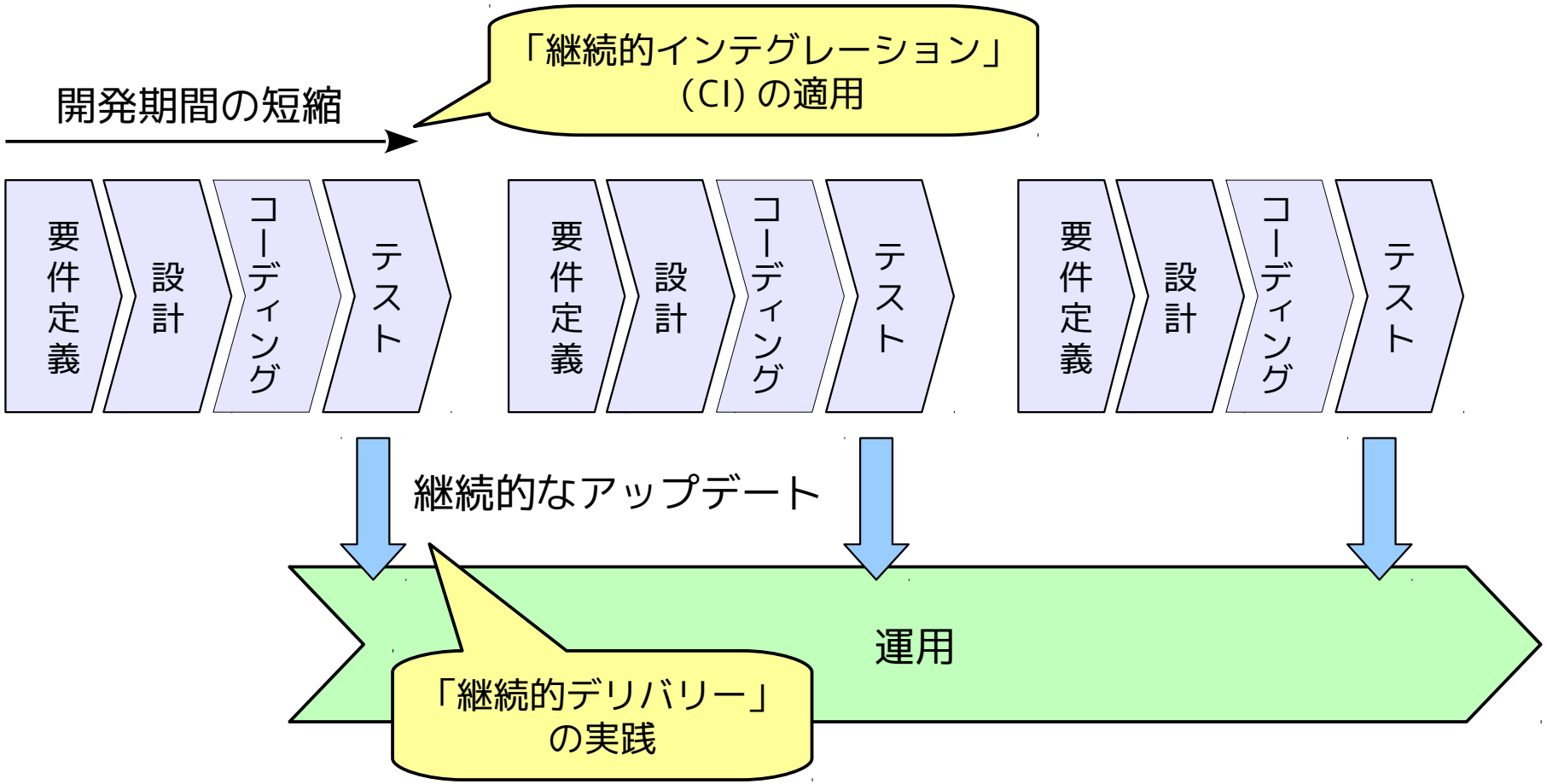
- ビジネス要件の変化と技術の進歩に開発速度が追いつかない現状



要件定義の際に決めたシステムが・・・　運用開始時は既に時代遅れに！

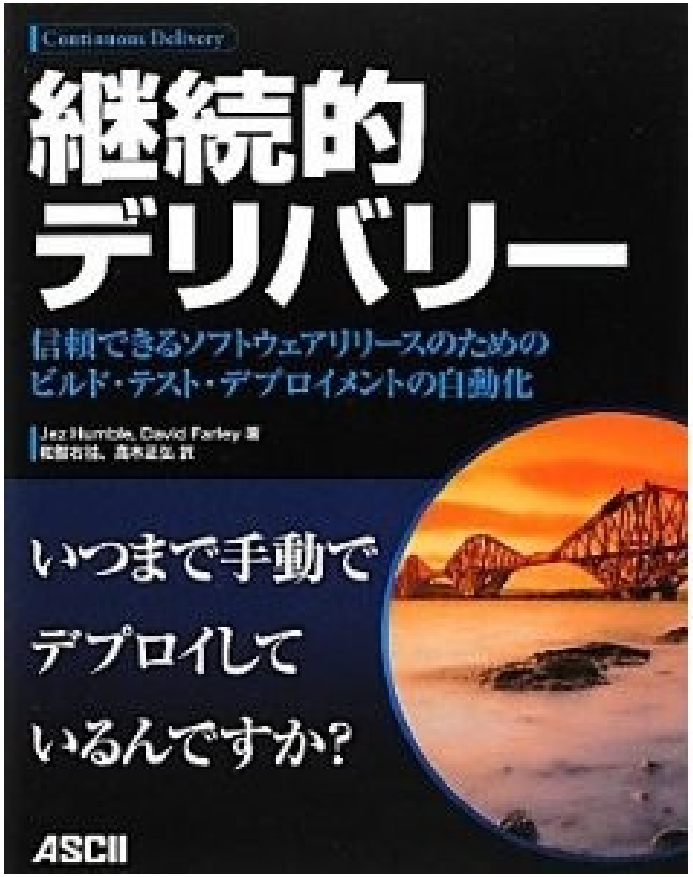
ソフトウェアの変更要求が高まる背景

- 開発期間の短縮と継続的なアップデートが必要に。

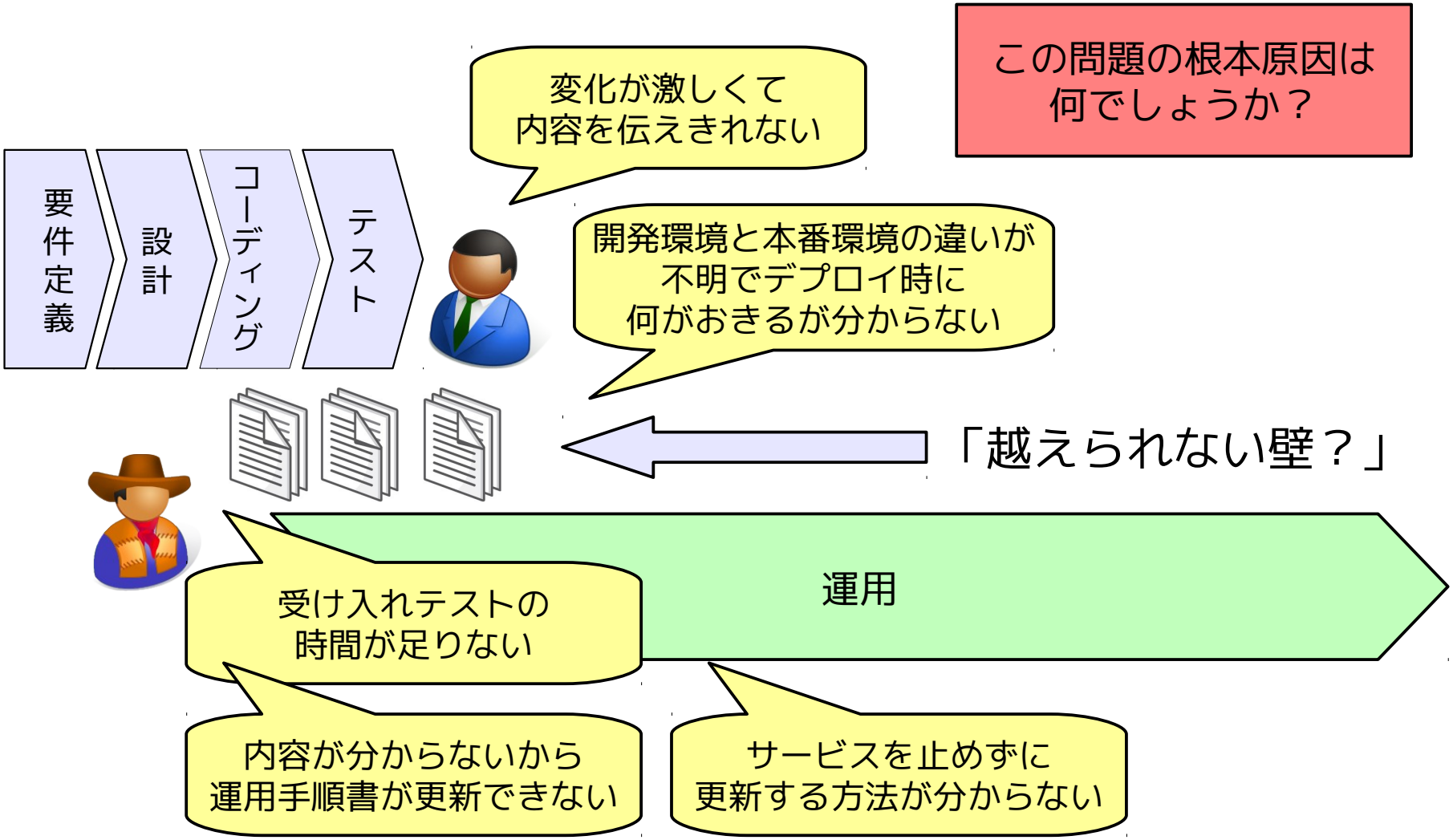


参考書籍

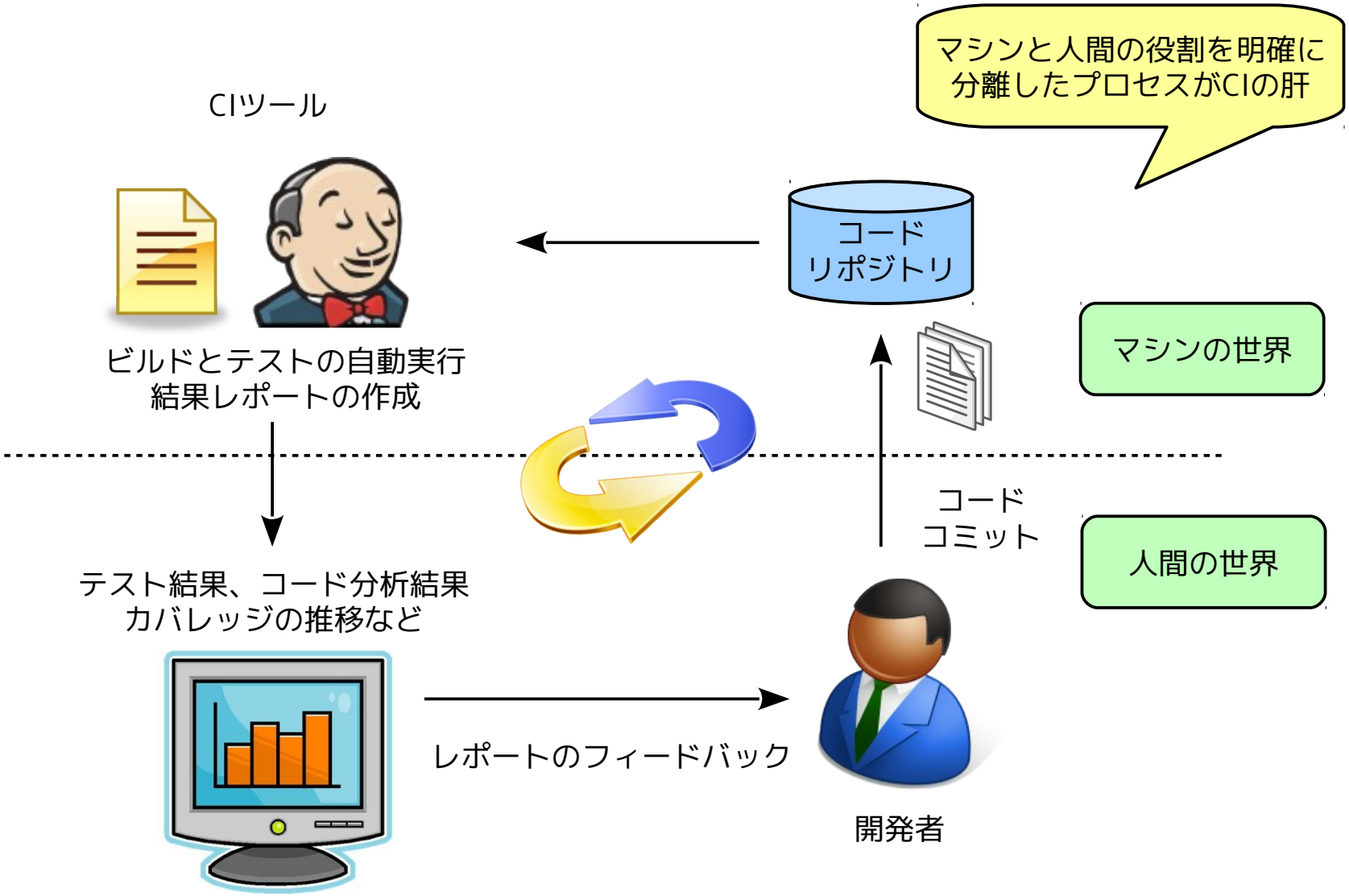

継続的
インテグレーション入門
開発プロセスを自動化する7つの作法
「自動ビルド」で
ソフトウエアを常に
正しく動作する状態に保つ


Continuous Delivery
継続的
デリバリー
信頼できるソフトウェアリリースのための
ビルド・テスト・デプロイメントの自動化
いつまで手動で
デプロイして
いるんですか?
ASCII

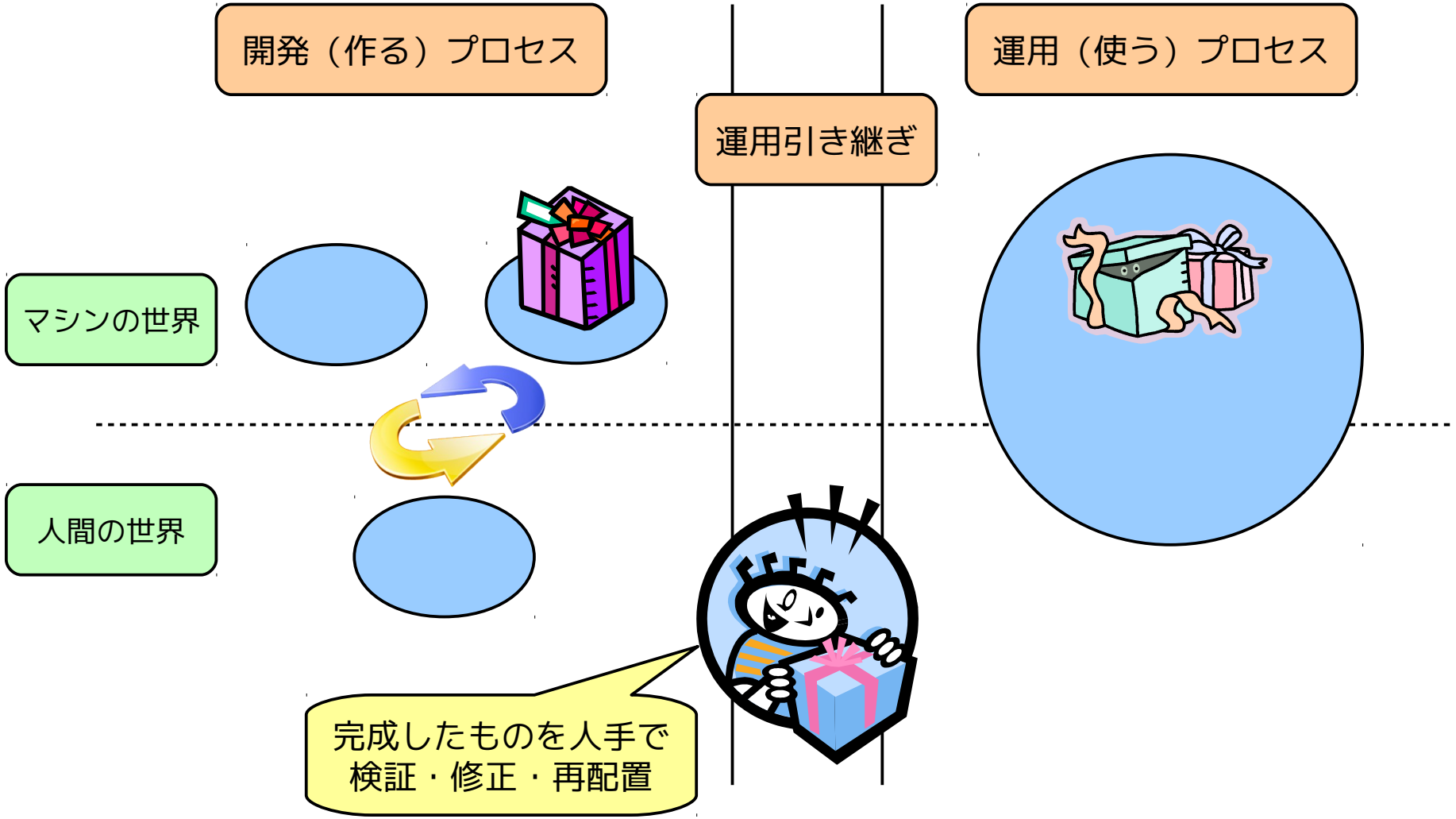
継続的デリバリーの課題は開発と運用の「プロセスの断絶」



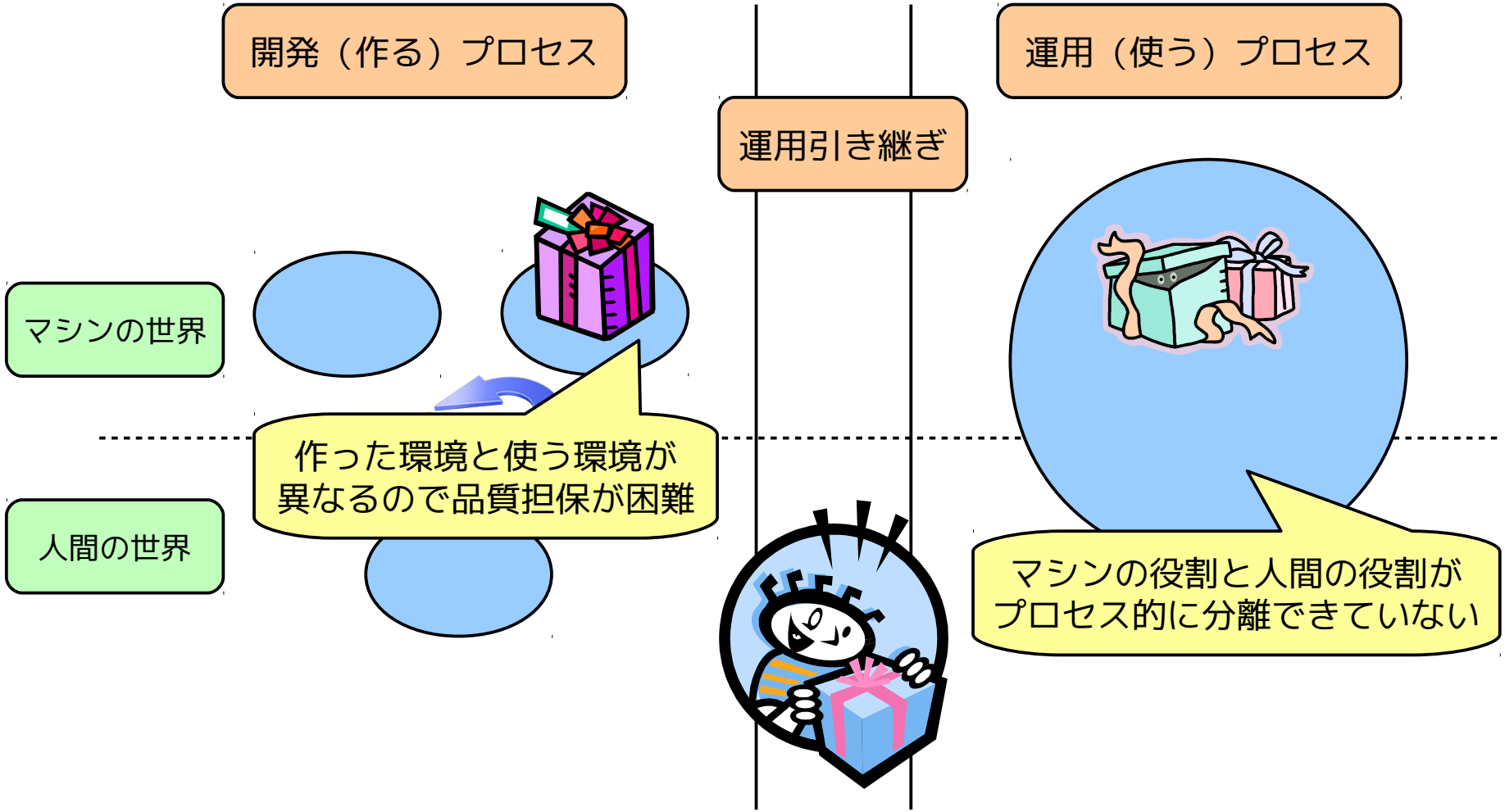
CIプロセスにおける「マシン」と「人間」の役割分担



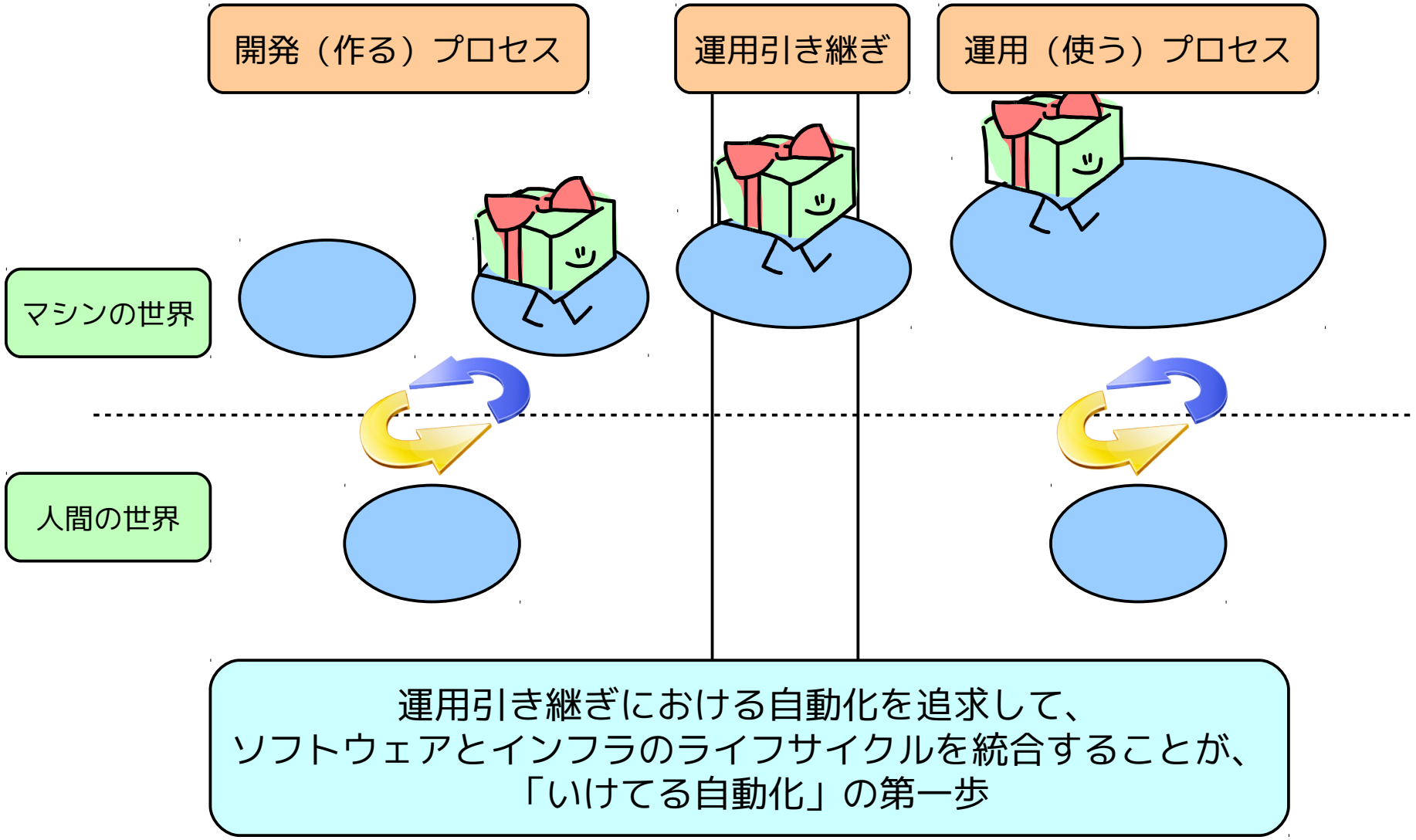
開発、引き継ぎ、運用プロセスの現状イメージ



開発、引き継ぎ、運用プロセスの現状イメージ



開発、引き継ぎ、運用プロセスの理想イメージ



自動化の追求で開発と運用の断絶を乗り越える

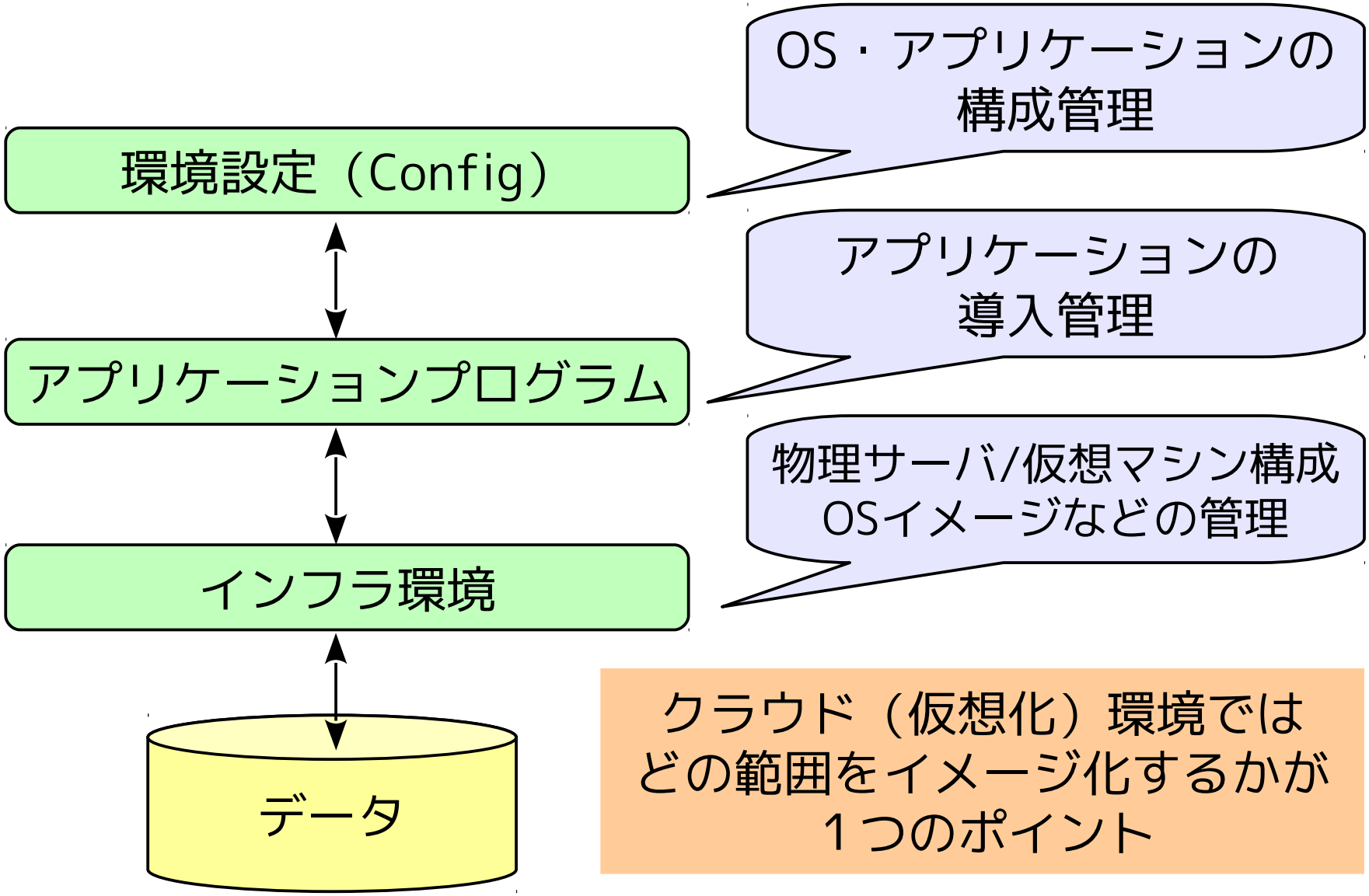
- 運用引き継ぎのプロセスを可能な限り自動化して、「デプロイ」の苦痛を軽減することが「継続的デリバリー」実現の鍵

1. デプロイ処理を自動化して、開発、テスト、本番、すべて同じプロセスでデプロイしたアプリケーションを使用する。
   ⇒ 環境差異を小さくして、「環境の違いで起きる問題」の発生を防止
2. 受け入れテストなど、運用引き継ぎに伴う作業を可能な限り自動化して、さらには、既存サービスに影響を与えずに自動更新する仕組みを導入する。
   ⇒「安全で容易、そして頻繁なデプロイ」を可能に
3. 運用業務を手順書による手作業から、プログラムによる自動化に置き換え。
   ⇒ 手順書のメンテナンスと運用作業の負荷を軽減

開発と運用を一体化したプロセスを策定して、
ソフトウェアとインフラのライフサイクルの統合を
実現することが最終的な目標（いわゆる「DevOps」）

アプリケーションデプロイ自動化の現状

システムデプロイにおける管理対象



クラウド（仮想化）環境では
どの範囲をイメージ化するかが
１つのポイント

システム構築自動化の現状

- クラウド/仮想化環境での自動化3大パターン

1. 仮想アプライアンス（ゴールデンイメージ）方式
   - アプリケーション導入済みの環境をマシンイメージ化して利用。
   - 巨大なテンプレートファイルの保守管理、インフラ間の可搬性が課題。

仮想化環境でよく利用される方法

2. JEOS(Just Enough Operating System)方式
   - 最小限のOS環境をマシンイメージ化して利用。アプリケーションの導入・設定は、別途、ツールで自動化。

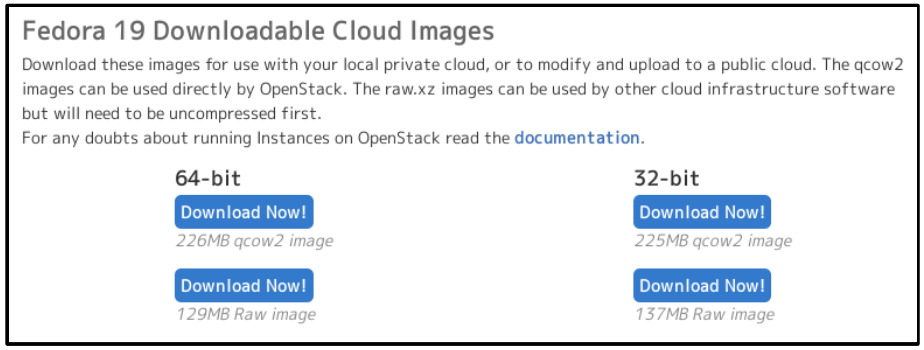
クラウドで主流になりつつある手法

3. 自動インストール方式
   - OSのインストールからアプリケーションの導入・設定まで、すべての作業を自動化して適用。

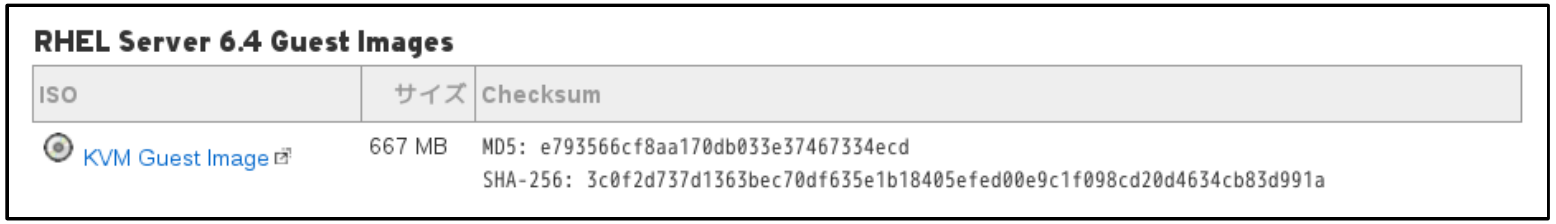

JEOS方式の自動化ツール

JEOSの作成方法

- Oz（仮想マシンイメージの作成に特化したOSインストールツール）の利用
  - OpenStack用イメージ作成のデファクトツール
    - https://github.com/clalancette/oz/wiki
- ディストリビューション標準のJEOSイメージを利用
  - FedoraのOpenStack対応イメージ
    - http://fedoraproject.org/en/get-fedora-options#clouds



  - RHEL6.4のOpenStack対応イメージ（要RHN ID）
    - https://rhn.redhat.com/rhn/software/channel/downloads/Download.do?cid=16952



アプリケーション導入、システム設定の自動化ツール

- Puppet/Chef
  - パッケージ、サービス、ファイル、ユーザ/グループなど、規定の「リソース」について、指定の状態に自動構成するツール
  - 「希望する状態」を指定すると、現状からその状態に変更するコマンドをツールが自動判別して実行
  - リソース間の依存関係を指定することで、設定の順序を制御可能

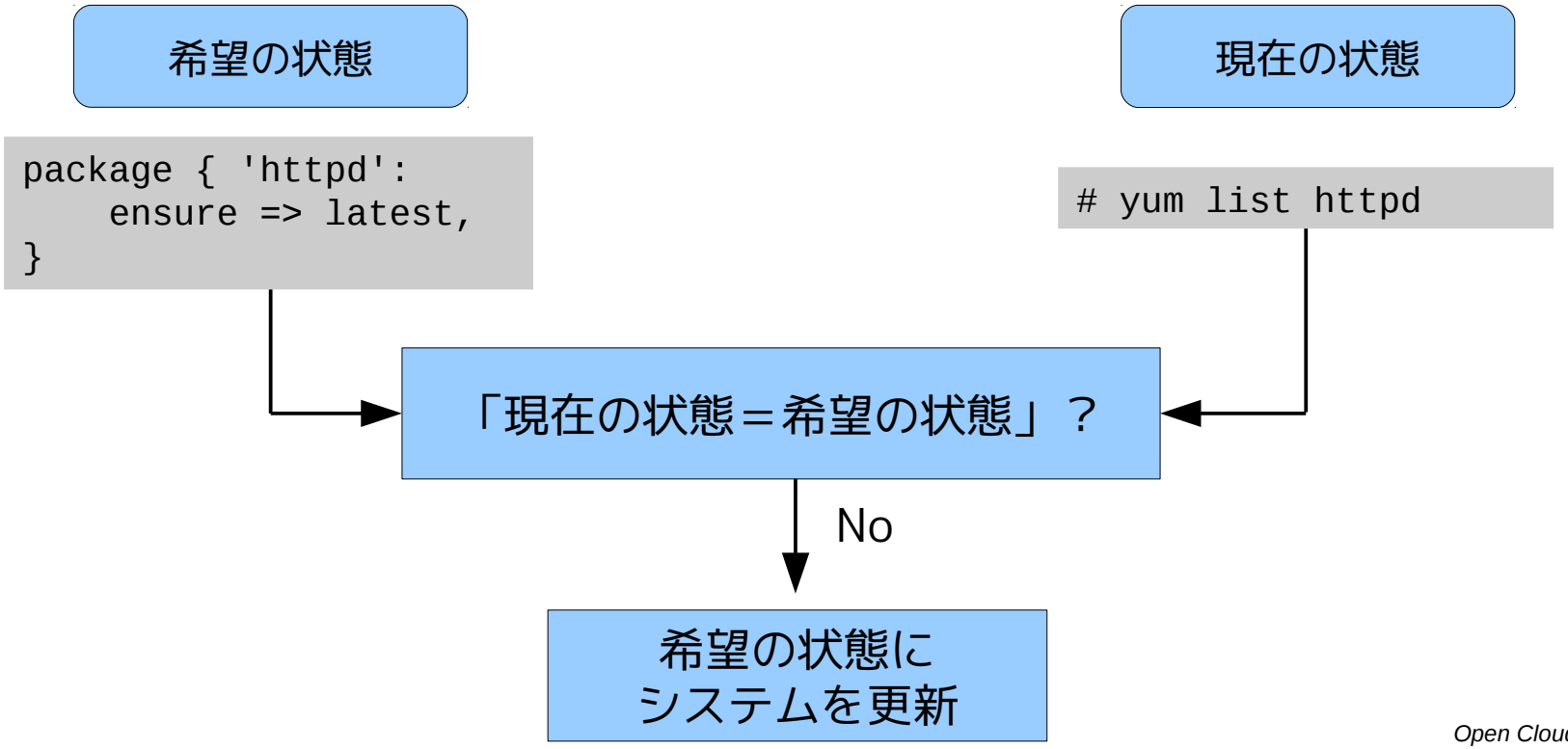
httpdを導入するPuppetマニフェストの例

```
package { 'httpd':
  ensure => latest,      # 最新バージョンを導入
}

service { 'httpd':
  ensure     => running, # サービスを起動
  enable     => true,    # サーバー起動時の自動起動を設定
  hasrestart => true,    # 「service xxxx restart」が利用可能
  hasstatus  => true,    # 「service xxxx status」が利用可能
}

file { '/var/www/html/index.html':
  owner => 'apache',     # ファイルオーナー
  group => 'apache',     # ファイルグループ
  mode  => '0600',       # アクセス権
  content => '<h1>Hello, World!</h1>',  # ファイルの内容
}

exec { 'fw-http':
  path    => '/usr/sbin',       # コマンドのパス
  command => 'lokkit -s http',  # 実行するコマンド
}

Package['httpd']
  -> File['/var/www/html/index.html']
  -> Service['httpd']
  -> Exec['fw-http']
```

リソースの依存関係

参考：Puppet/Chefの本質は「ビルトインの現状確認機能」

- 作業手順（コマンド）を並べたスクリプトは、前提環境が変わると利用不能
- 「現在の状態」に応じて必要なコマンドを判断して「希望の状態」を実現するのが構成管理ツールの本質（「冪等性」はあくまで結果的に得られる性質）
- リソースの種類ごとに「現状確認コマンド、設定変更コマンド」などの”Intelligence”がビルトインされている事がPuppet/Chefの価値



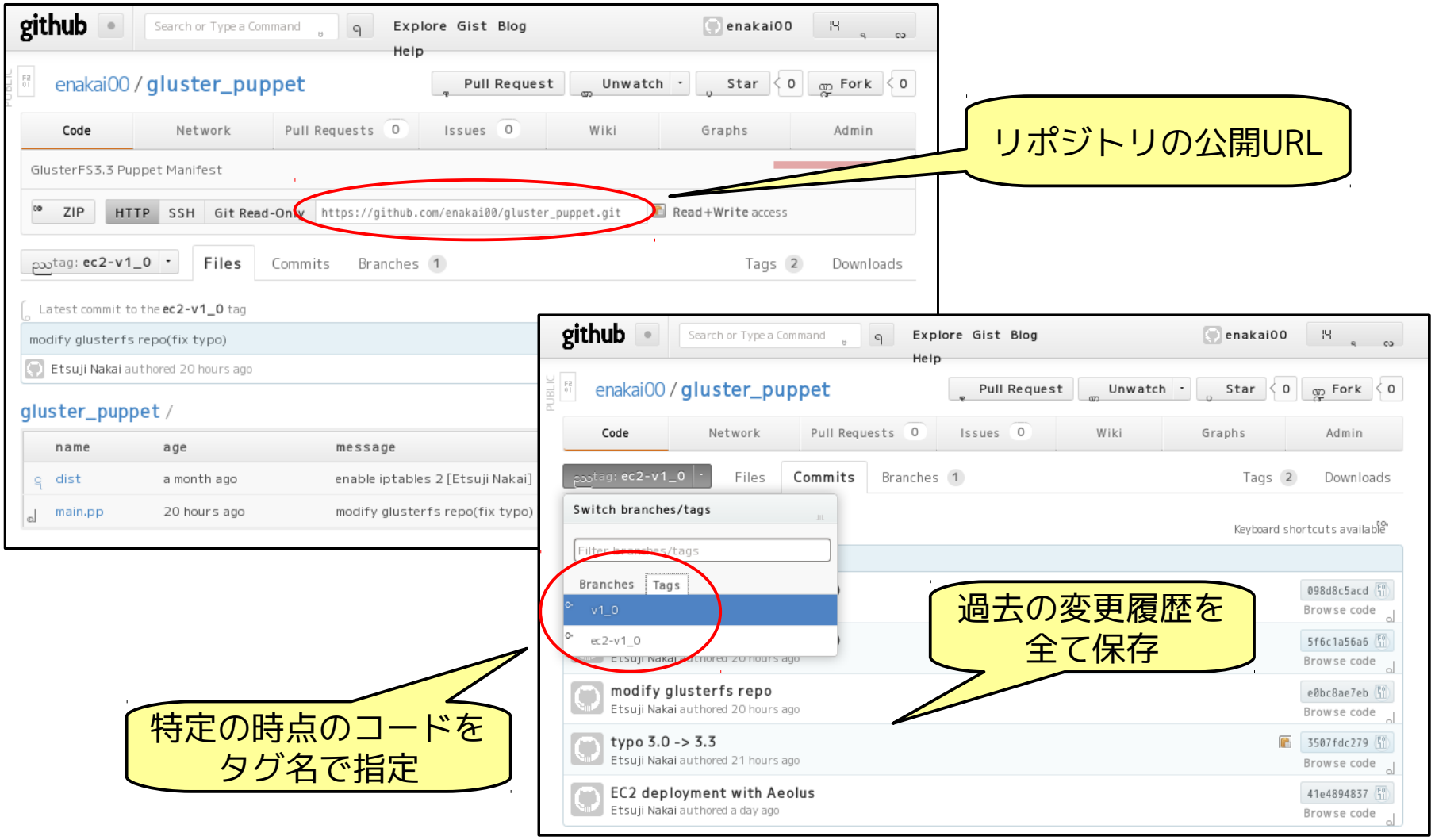
設定ファイルのバージョン管理

- Git / Github
  - プログラムソースコードの分散バージョン管理システム。元々はLinux Kernelの開発用に作成された。インターネット上にコードリポジトリを作成して無料で利用できるサービス「Github」の登場により、利用者が増加した。
  - ここでは、Puppetの設定ファイル（マニフェスト）をGithubにおいて、KickstartのPostスクリプトから、指定バージョンのマニフェストのダウンロードと適用を実行している。

Githubからマニフェストを取得して適用するスクリプトの例

```
#!/bin/sh -x

yum -y install puppet git

GitRepository=https://github.com/enakai00/apache_puppet
ConfigTag=f18
RepoName=${GitRepository##*/}
RepoName=${RepoName%.git}

mkdir -p /tmp/gittmp
cd /tmp/gittmp
git clone $GitRepository
cd $RepoName
git checkout $ConfigTag
export FACTER_manifest_dir="/tmp/gittmp/$RepoName"
puppet apply main.pp
```

Githubのスクリーンショット



カスタマイズ・スクリプト（User Data）による自動化

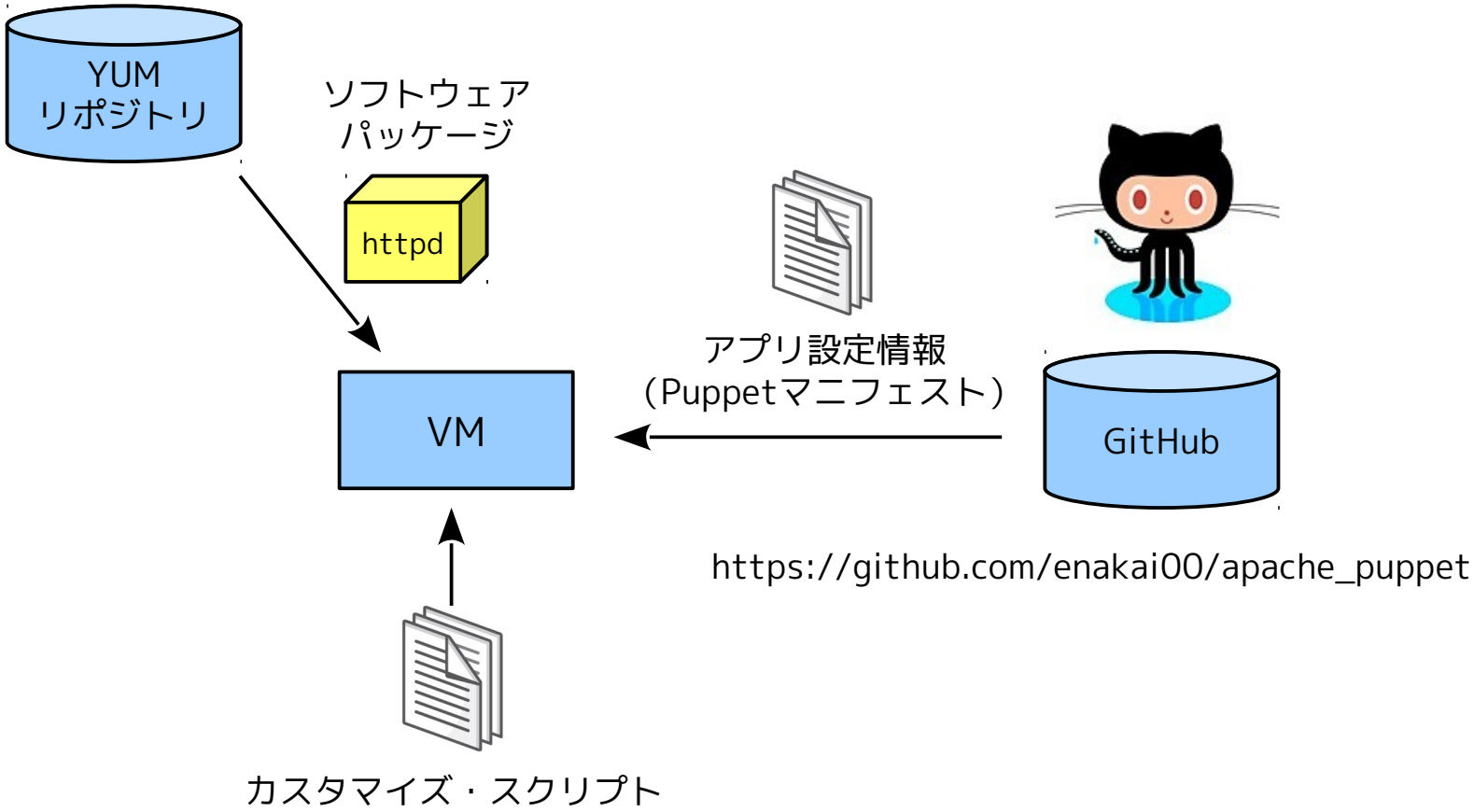
- OpenStackではマシンインスタンス起動時に「カスタマイズ・スクリプト（User Data）」を与えると任意のテキストをメタデータとしてゲストOSに受け渡すことができます。
- Cloud-Initは、カスタマイズ・スクリプトを解釈して、さまざまな自動化を実現します。
  - 下図はシェルスクリプトを渡して、「/etc/motd」を設定しています。
  - この他にもCloud-Init独自の構文で、処理内容を指示することができます。

http://cloudinit.readthedocs.org/en/latest/

デモンストレーション

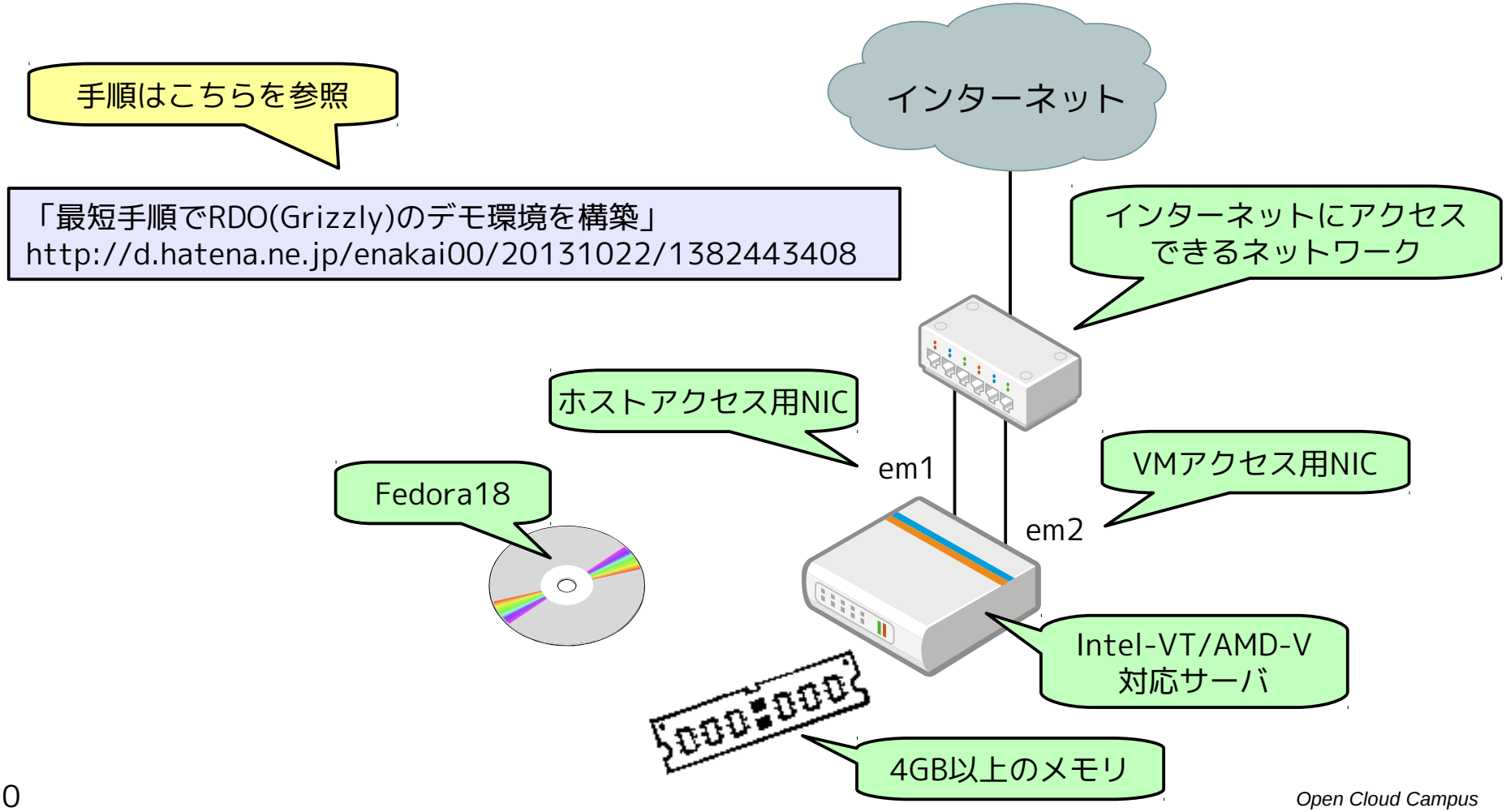
http://bit.ly/18w0Mp2

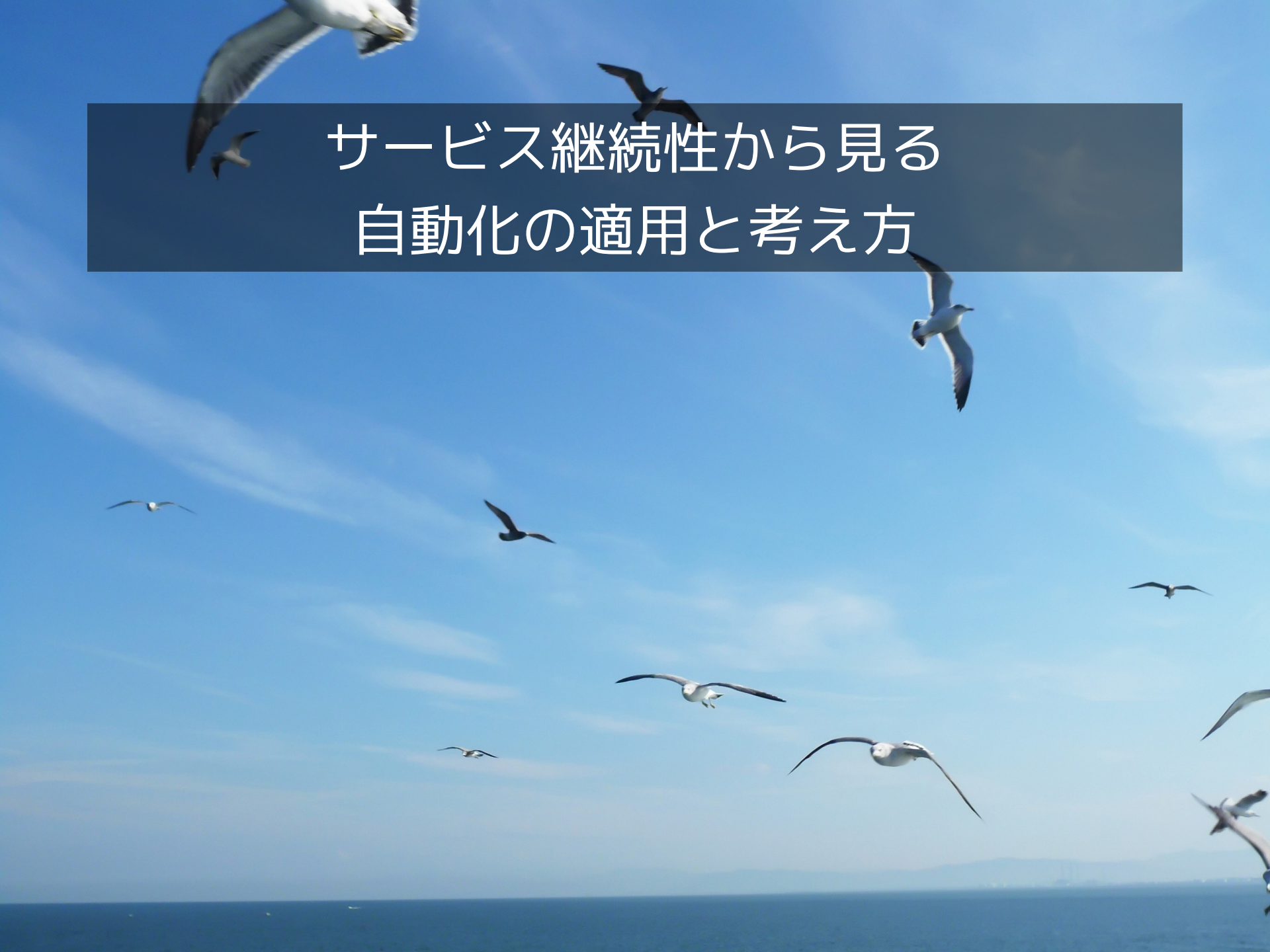
- カスタマイズ・スクリプトからGitHub/Puppetを連携させて、Webサーバを自動構築します。



（参考）RDOとPackstackでデモ環境を簡単構築

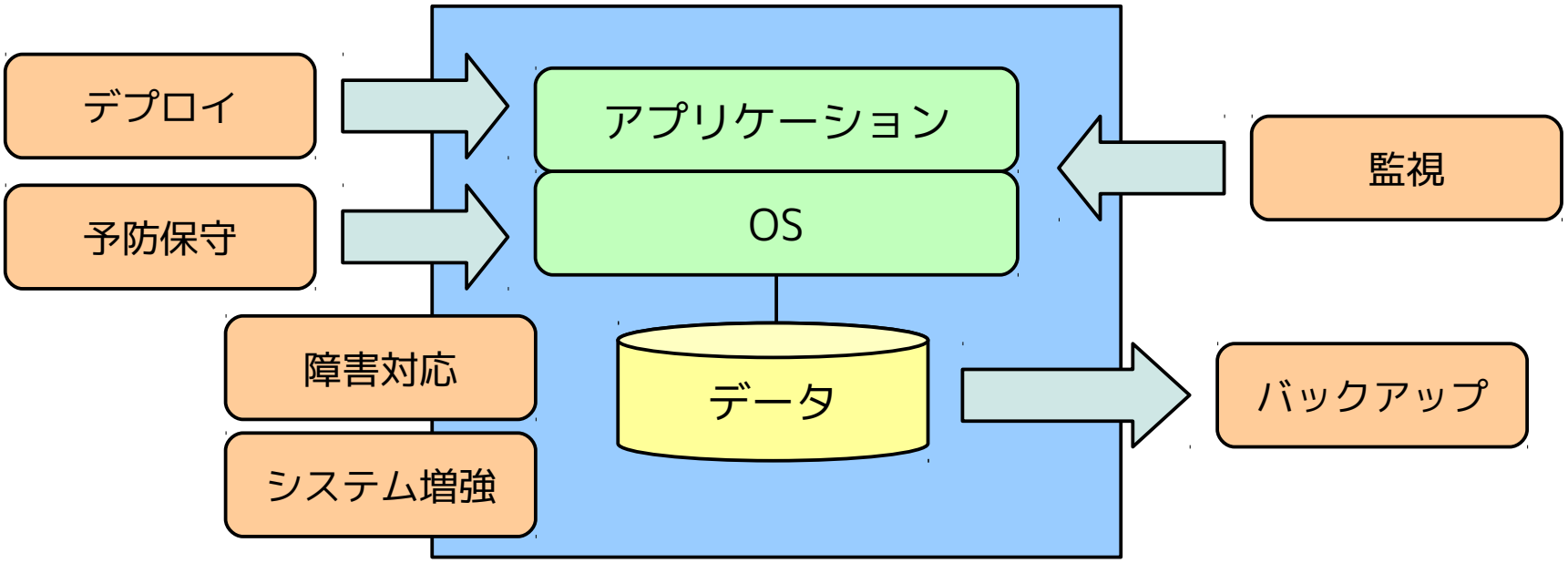
- 下図の道具があれば、オールインワン構成のデモ環境を簡単に構築できます。




サービス継続性から見る
自動化の適用と考え方

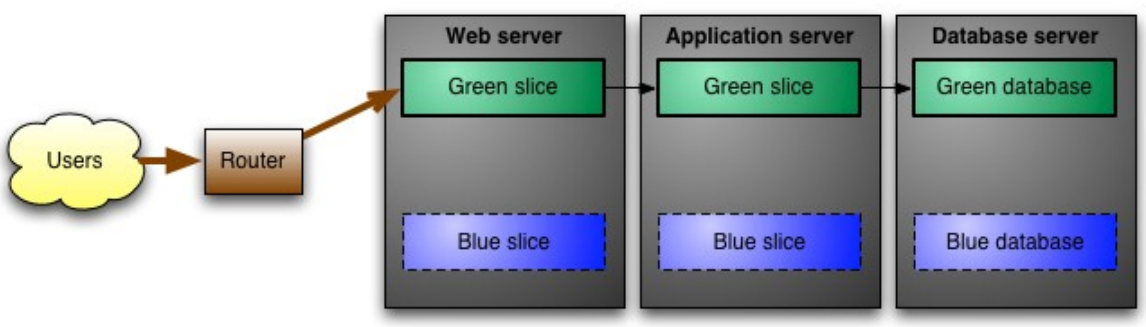
サービス継続性における自動化の役割

- サービス継続性の観点で適用できそうな自動化のエリア
  - キャパシティ不足時にシステムを自動増強
  - 既存サービスを停止せずに新規アプリケーションを自動デプロイ
  - 障害発生時にサービスを自動復旧



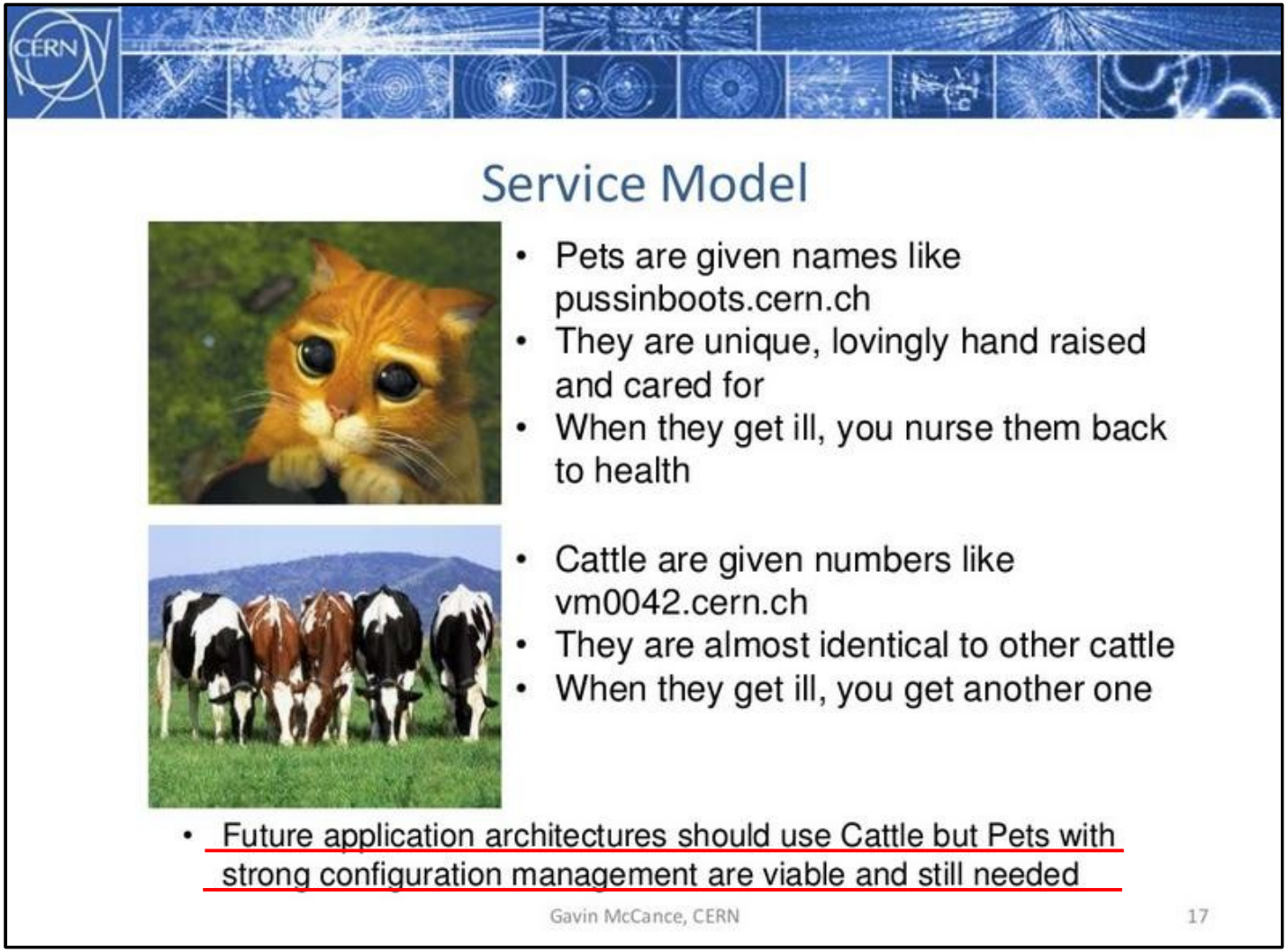
サービスを停止せずにバージョンアップする手法の例

- ブルーグリーン・デプロイメント
  - 本番環境を2種類用意して、旧バージョンと新バージョンを並行稼働させながら、切り替える手法。新バージョンに問題が起きた場合は、すぐに旧バージョンにフォールバックが可能。
    - クラウドでインフラ準備が自動化できるようになると、コスト的にも現実性がでてくる。
  - ユーザデータの複製などが課題。
    - アプリケーションのアーキテクチャから見直しが必要。



- カナリアリリース
  - 一部のユーザのみに新機能を提供して、問題がなければ、全ユーザに新機能を開放する
  - 機能レベルで小さなバージョンアップを繰り返すことが前提。機能変更に伴うデータ構造の変化の取り扱いが難しい。（とくに新機能使用ユーザと未使用ユーザは同じデータを共有する場合）
    - アプリケーションのアーキテクチャから見直しが必要。

システムアーキテクチャのパラダイムシフト



http://www.slideshare.net/gmccance/cern-data-centre-evolution

障害復旧の観点での自動化

- 自動化による「障害復旧」の発想転換
  - アプリケーション環境のデプロイが完全自動化されているのであれば、「壊れた環境を修復する」よりは、「新しい環境を作り直す」方が安上がり。（単純なインフラ障害は、すべて再デプロイで回復してしまう。）
  - アプリケーション実行環境とユーザデータを明確に分離して、再デプロイがデータに影響しないことが必要。
    - AWSなどは、この方向性を示唆するクラウドインフラ。（インスタンスが停止したらインスタンス上のデータは全て消える。永続データは、EBS、S3への外部保存が前提。）

いずれにしても、ソフトウェア自体が自動化を前提とした機能・アーキテクチャを持たないと実現は難しい

クラウドの自動化とソフトウェア開発の未来を一緒に考えて行きましょう！

中井悦司
Twitter @enakai00