クラウド・ネイティブ・コンピューティング最新技術トレンドのご紹介

マイクロサービス概要
コンテナ概要
オーケストレーション(Kubernetes)概要

2018年　3月　13日
日本アイ・ビー・エム株式会社
IBMクラウド事業部．クラウド・テクニカル・セールス
樽澤 広亨

この資料の目的

- クラウド・ネイティブ・コンピューティングの構成要素を理解する
- クラウド・ネイティブ・コンピューティングを実現するIBM Cloud製品・機能を理解する
- マイクロサービスの概要と技術的ポイントを理解する
- Dockerコンテナーの概要と技術的ポイントを理解する
- コンテナー・オーケストレーション (Kubernetes)の概要と技術的ポイントを理解する

内容

- クラウド・ネイティブ・コンピューティング概要
- マイクロサービス概要
- マイクロサービス実践のポイント
- Dockerコンテナ概要
- Kubernetes概要
- まとめ

クラウド・ネイティブ・コンピューティング概要

クラウド化の対象ドメイン

SoR
Systems of Record

確実な運用

ミッション・クリティカルな
アプリケーション運用管理

SoE
Systems of Engagement

スピーディな変革

マーケット開発のための
開発,デプロイ,フィードバック

API化
レガシー・モダナイゼーション

クラウド化

- **SoEのクラウド化**
  - マーケットを先取りするスピードと柔軟性の実現
- **SoRのリノベーション**
  - ミッション・クリティカルのデジタル化によるエコ・システム創造

IBM Cloud 導入戦略

戦略	対象	動機	方策	成果物
クラウドネイティブ	✓ SoE	✓ アプリ新規開発 ✓ スモール・スタート ✓ 週次月次のリリース	✓ PaaS ✓ アジャイル開発 ✓ 継続的デリバリー ✓ マイクロサービス	クラウドネイティブアプリ
リフト&シフト	✓ SoE ✓ SoR	✓ システム更改 ✓ コスト削減	✓ まずIaaS、後 PaaS ✓ アプリのクラウド対応 ✓ 後クラウドネイティブ化	クラウド対応/ネイティブアプリ
分析&リフト	✓ SOR	✓ 既存資産の有効活用 ✓ コスト削減	✓ IaaS または PaaS ✓ アプリのクラウド対応 ✓ アプリのAPI対応	クラウド対応アプリ

変化に柔軟スピーディ

信頼性 コスト削減

新規開発から、メインフレーム上の既存資産の有効活用まで～

3つの戦略で幅広いニーズに柔軟に応えます。

Systems of Engagementに求められる要件

スピード
~素早いITシステム開発~

柔軟性
~容易なシステム変更・保守~

スピードと柔軟性を兼ね備えた
- IT基盤：クラウド・プラットフォーム
- 業務：**クラウド・ネイティブ・アプリ**

⇒ クラウド・ネイティブ コンピューティング

クラウド・ネイティブ・コンピューティングとは

オープンソース・ソフトウェア・スタックを用いた…

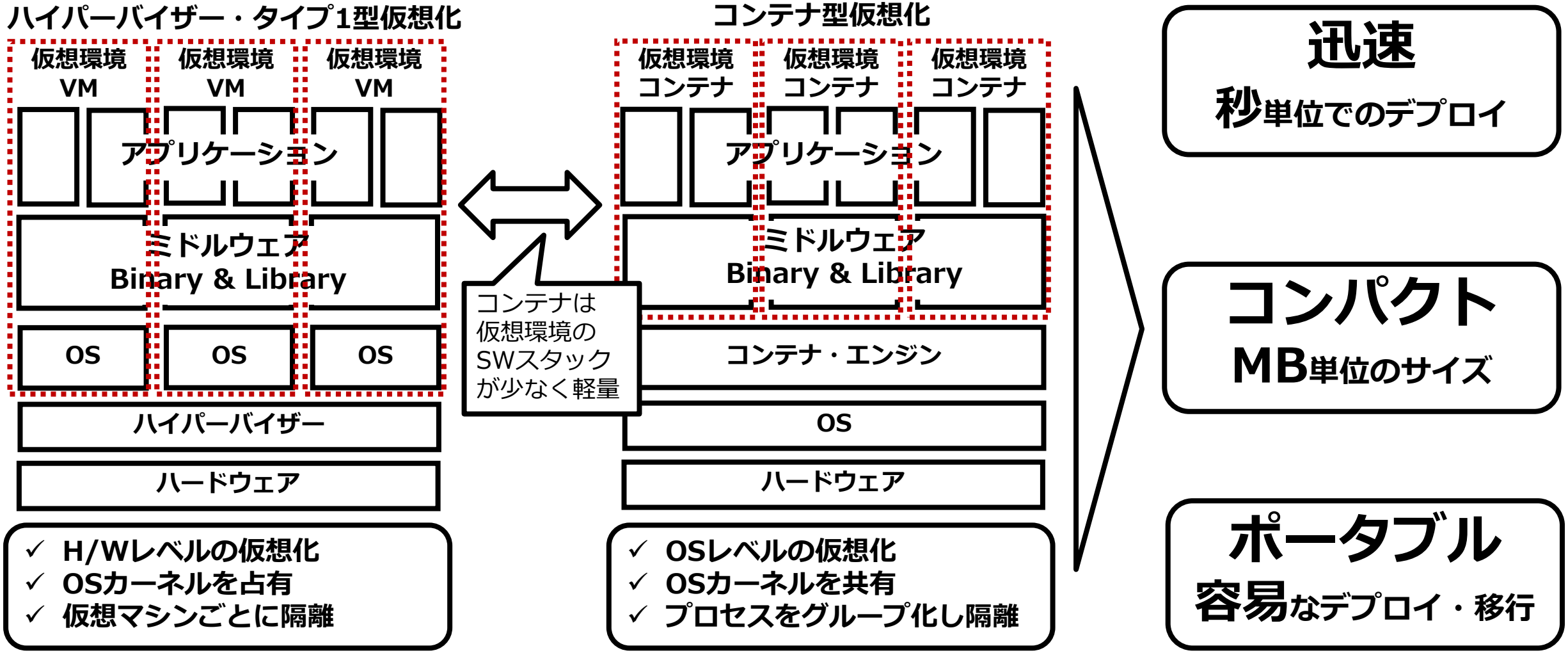
- ✓ コンテナ化
- ✓ 動的なオーケストレーション
- ✓ マイクロサービス指向

Cloud Native Computing Foundation FAQより

Cloud native computing uses an open source software stack to be:
1. Containerized. Each part (applications, processes, etc) is packaged in its own container. This facilitates reproducibility, transparency, and resource isolation.
2. Dynamically orchestrated. Containers are actively scheduled and managed to optimize resource utilization.
3. Microservices oriented. Applications are segmented into microservices. This significantly increases the overall agility and maintainability of applications.

コンテナ

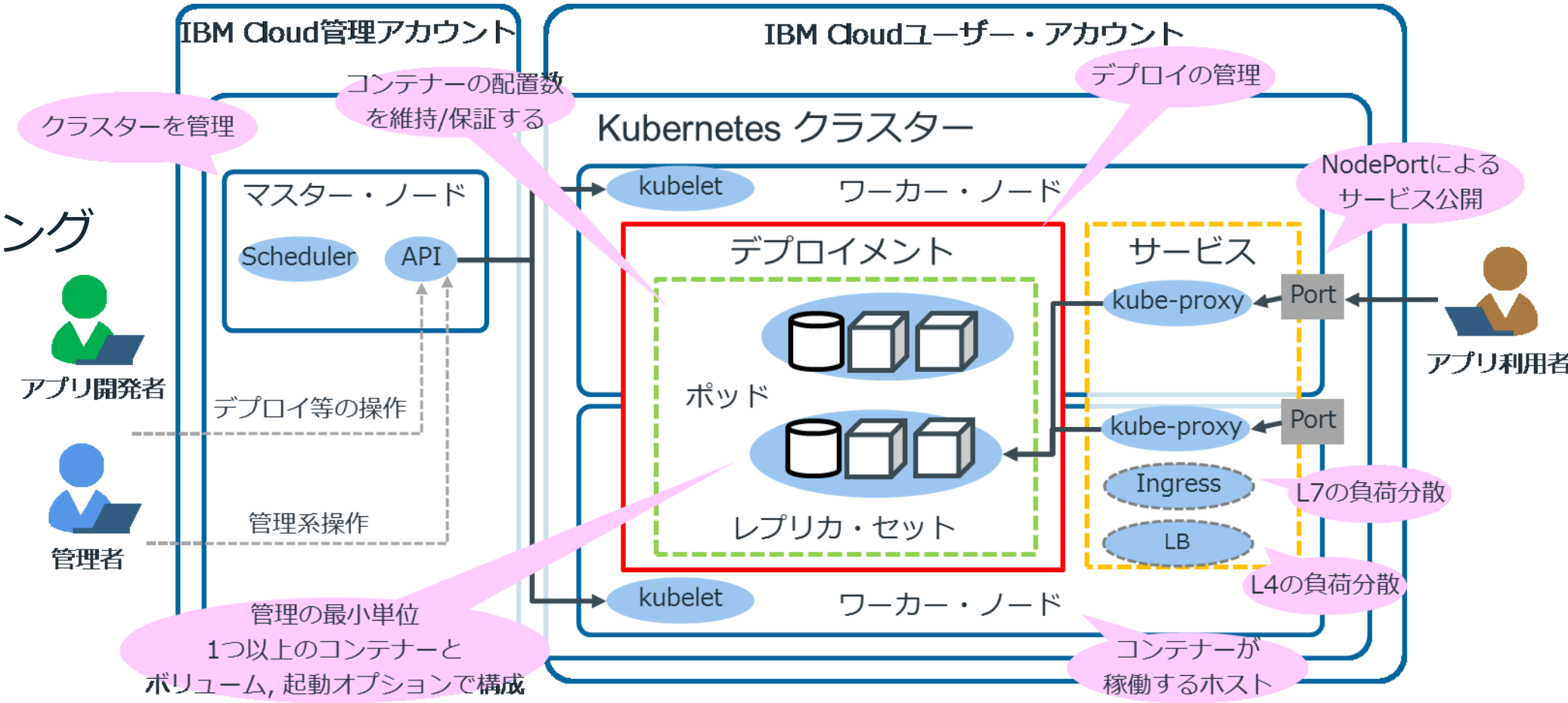
仮想化の技術トレンドは、ハイパーバイザーからコンテナに移行しています
- **“速い”,”小さい”, “ポータブル”がコンテナ技術の強みです**



Kubernetes ～ コンテナのオーケストレーション・フレームワーク

不特定多数のコンテナ・クラスターの運用管理を担うのがKubernetesです
エンタープライズ・システムのコンテナ基盤機能を支えます

- デプロイ
- ルーティング
- スケーリング
- スケジューリング
- メンテナンス



Microservices：概要

クラウド・ネイティブ・アプリの”つくり”を規定するのがMicroserviceです
高頻度のメンテ＆変更に耐えうる柔軟な構造をアプリに与えます

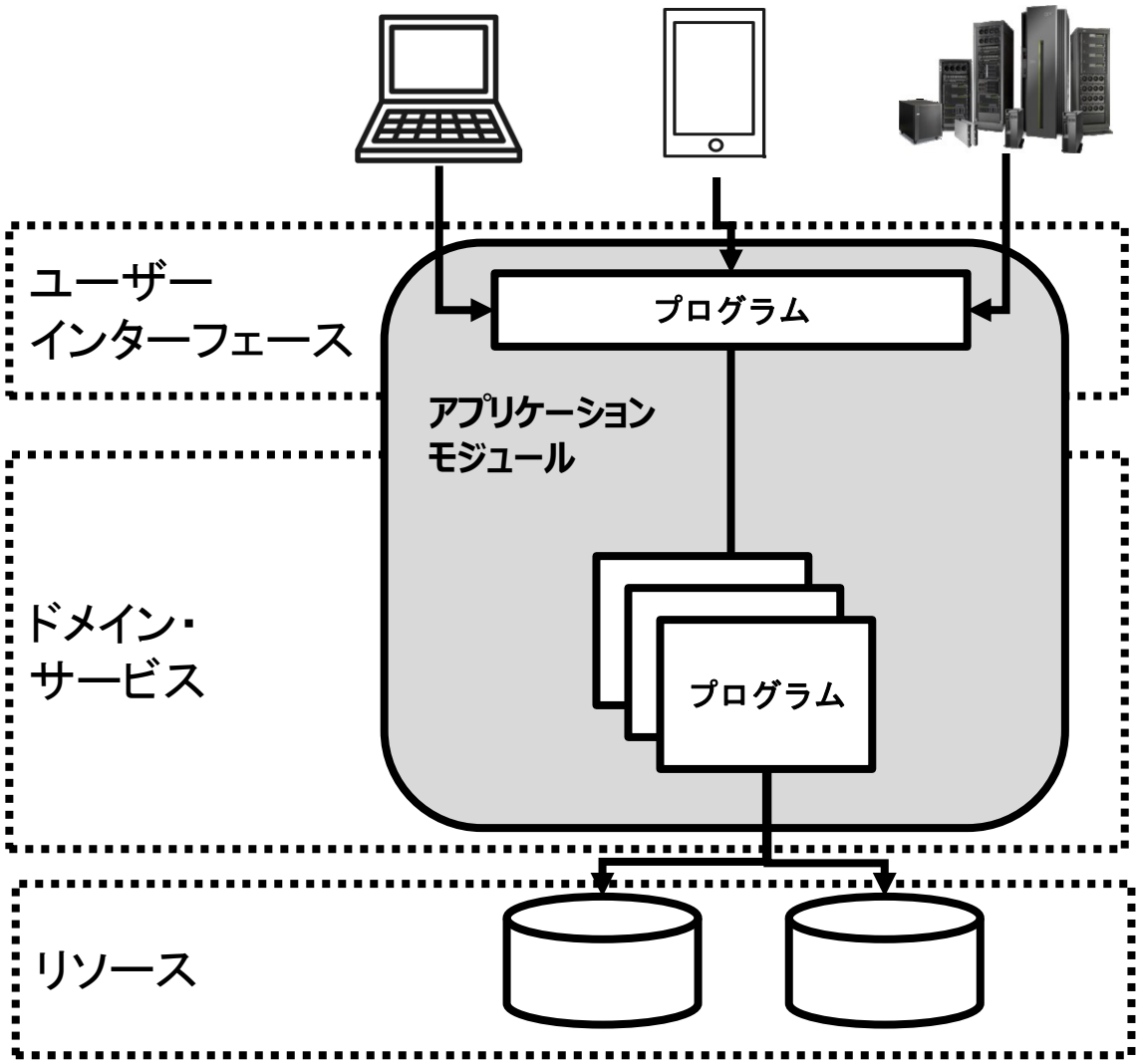
■ クラウド・ネイティブ・アプリケーション開発・運用のベスト・プラクティス
- ◆アーキテクチャー
- ◆開発・運用チーム・フォーメーション
- ◆システム・ライフサイクル

■ 動機
- ◆アプリケーション個別の保守を可能にする柔軟なモジュラー構造(サービス)の実現

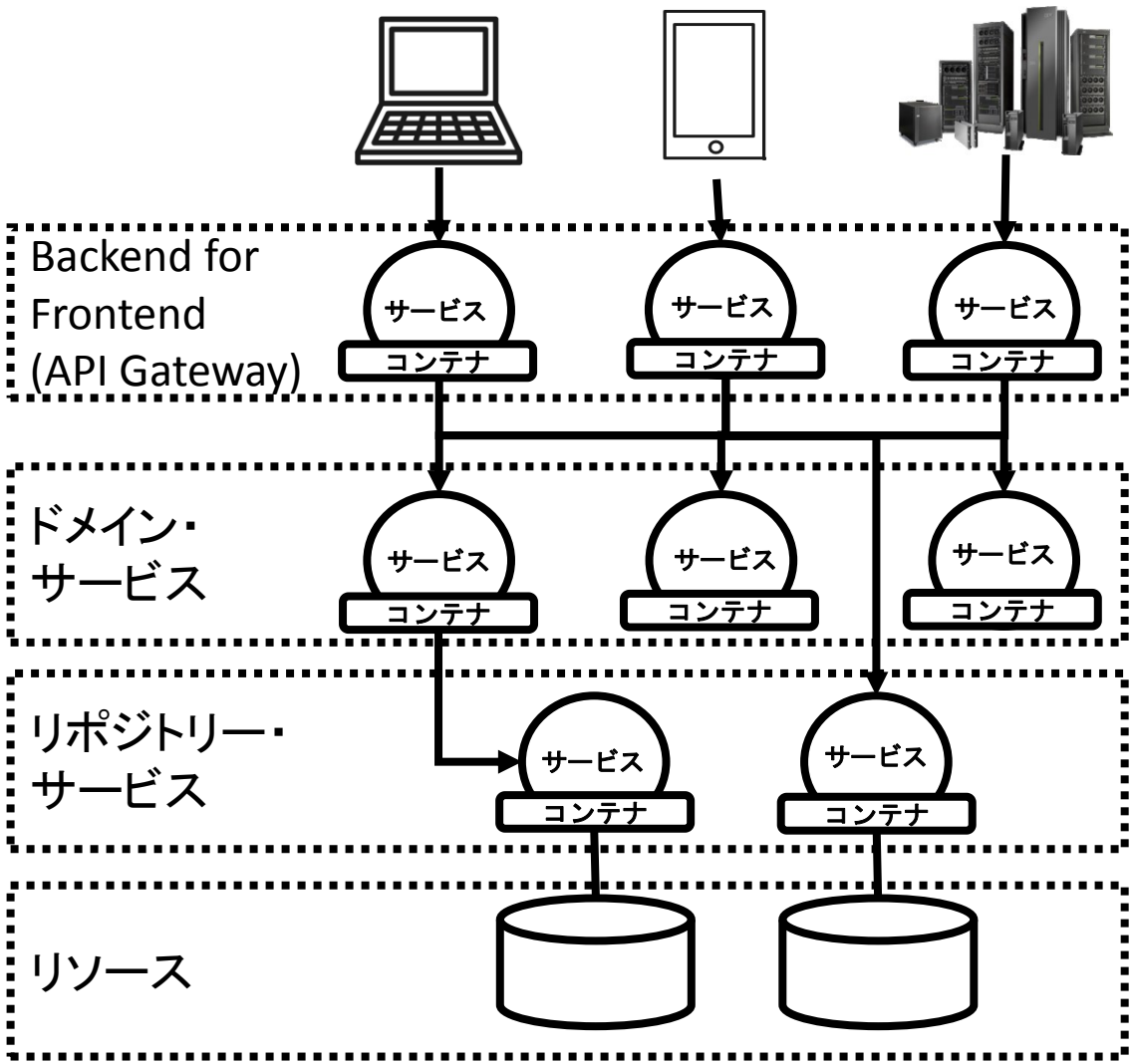
■ マイクロサービス・アーキテクチャ・スタイル
- ◆**小さなサービスを組み合わせて**, 一つのアプリケーションを開発する
- ◆各サービスは, **それぞれ独立したコンテナで動作**する
- ◆各サービスは, **RESTやメッセージングのような軽量な仕組みで通信**する
- ◆各サービスは, 完全に**自動化された仕組みで, それぞれ個別にデプロイ**される
- ◆サービスは, それぞれ**異なるプログラミング言語で実装**できるし, **異なるデータ・ストレージを利用**できる

Microservices：アーキテクチャー・イメージ

モノリス (1枚岩)アプリケーション構造

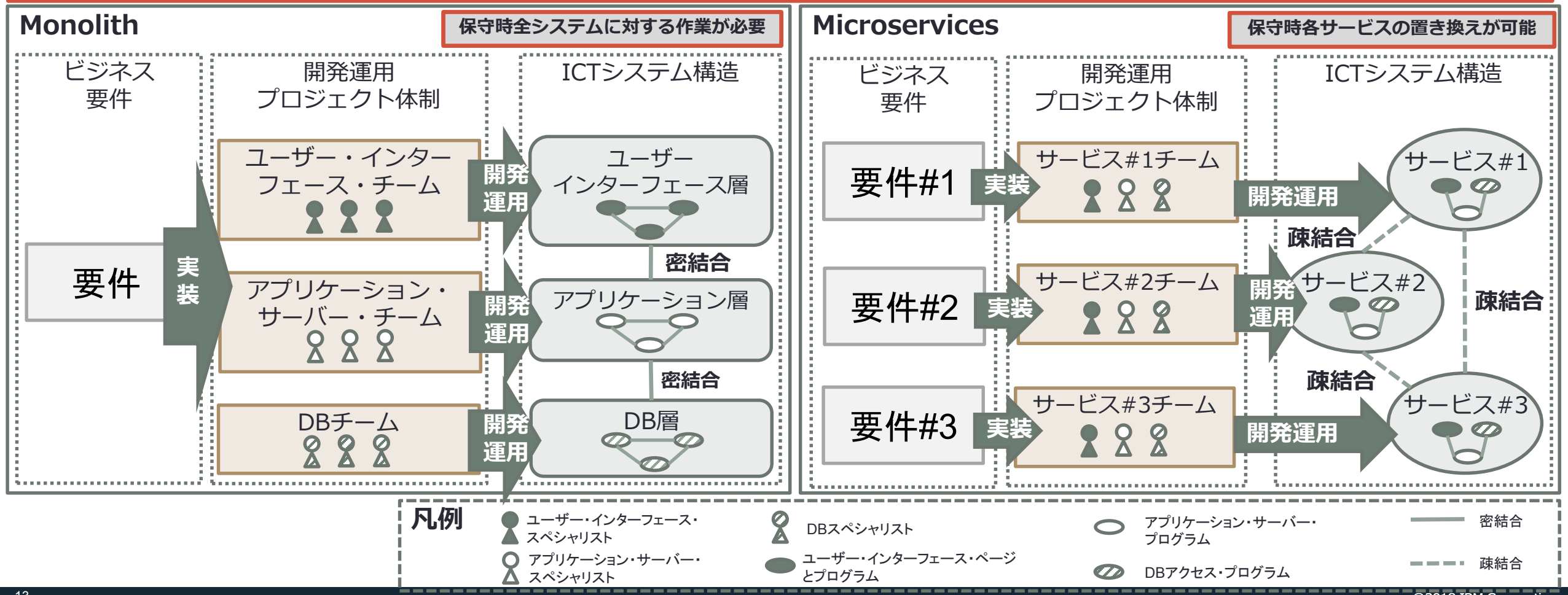


マイクロサービスに基づいたアプリケーション構造



Microservices：開発・運用チーム・フォーメーションの関係

- コンウェイの法則：システム構造は, プロジェクト体制を反映する
- Microservices：
  - ビジネス目的に基づいてチーム編成
  - システムは、独立して置き換え可能なサービスで構成される



Microservices：システム・ライフサイクル

プロジェクトではなく製品として捉える
～フィードバックを受けながら, One Teamが継続的に”開発・運用”～

プロジェクトの例：ダム

開発局面 → ゼネコン

運用局面 → 官公庁・電力会社等

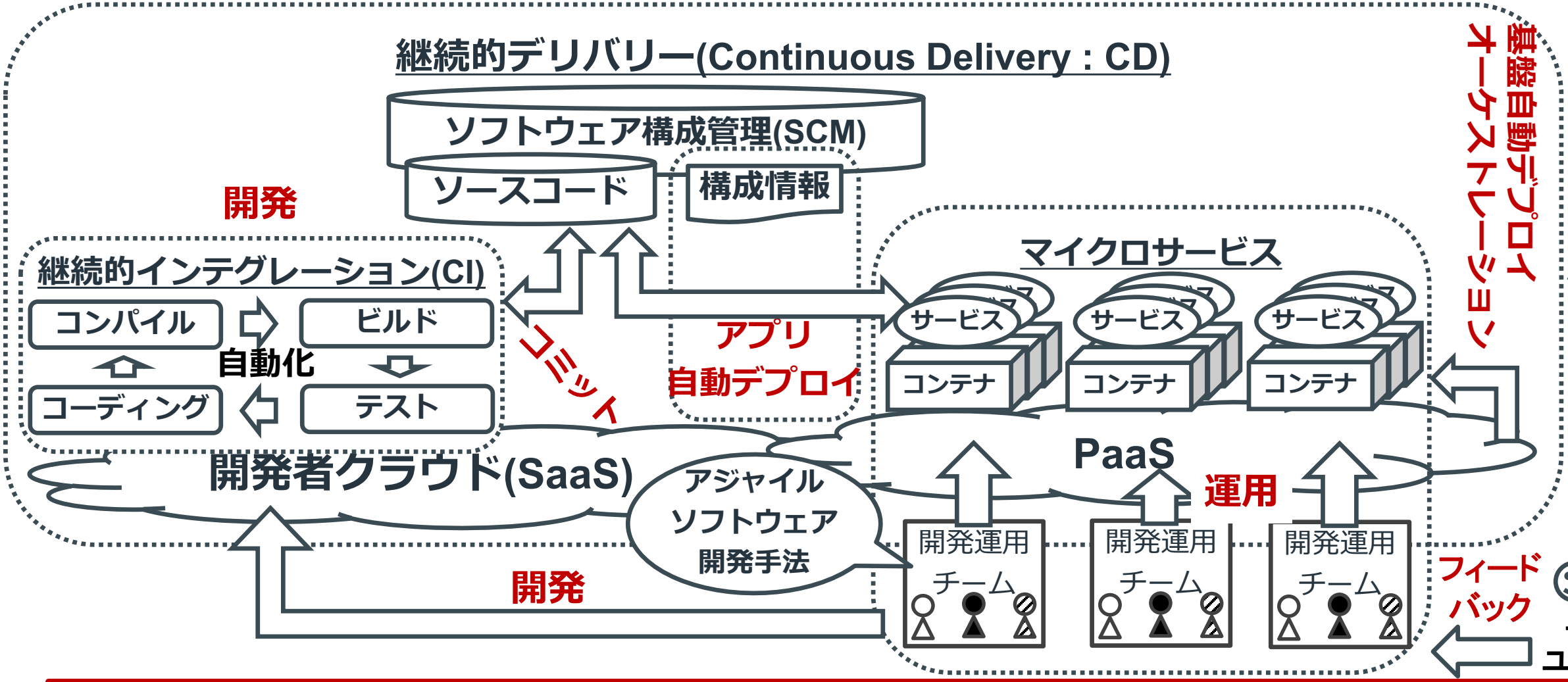
Microservicesの開発・運用スタイル

製品の例：自動車

開発・運用局面 → 開発チーム

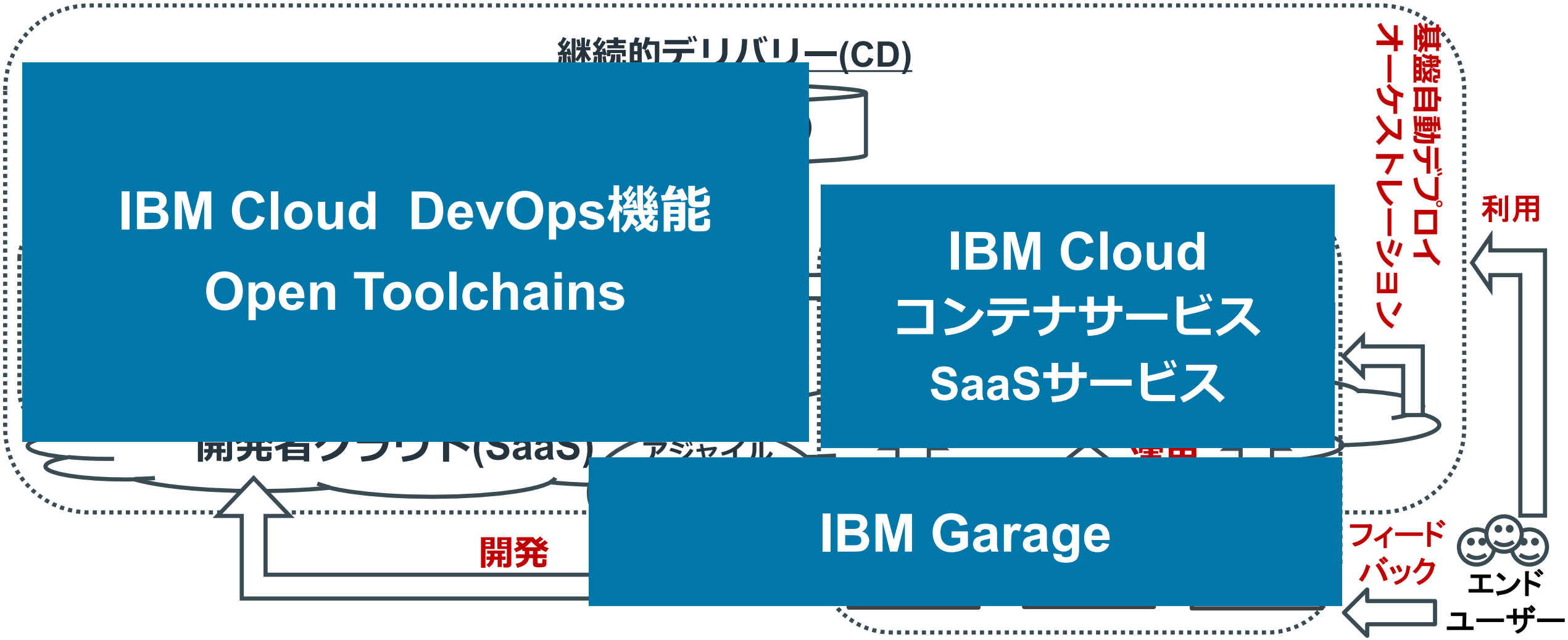
開発・運用局面 → 開発チーム

Microservicesのデリバリー・スタイル：継続的デリバリー (CD)



- アプリケーションの開発プロセスとデプロイ，基盤のデプロイを自動化
- システム・リリースのスピード・アップ，コスト抑制，品質向上を実現

IBMクラウド・ネイティブ開発・運用ソリューション全体像



IBM Cloudはクラウド・ネイティブ開発をフルカバレッジで支援します

IBM Cloud

IBM Cloudとは・・・
多様なニーズに応えるIBMのクラウド・ソリューション・プラットフォーム

お客様開発のアプリケーションとサービス

IBM Cloud Platform

運用管理

セキュリティ

オーケストレーション
Kubernetes

DevOps
Open Toolchains

サービス

API管理 | Watson | IoT | Analytics | ブロックチェーン | モバイル | …

アプリプラットフォーム(WAS, Node, …)

Java | WebSphere | node | php | Go | python | Ruby

Cloud Foundry

コンテナ

サーバーレス
OpenWhisk | Cloud Functions

インターネット
The Weather Company 気象
ビデオ配信
電子マネー決済
V.me by Visa
メール一斉配信
電話
モバイル通知
健康/運動

オンプレミス
既存IT資産
UNIX系分散Sys
メインフレーム
オフコン

IBM Cloud Infrastructure

お客様ご提供IaaS

凡例
IBMによる提供・運用管理
お客様による開発・構築・運用管理
IBM/お客様による提供・運用管理

IBM Cloudの特徴

- オープンソース製品技術(OSS)の積極的採用によるベンダー・ロックインの排除
- データ暗号化、VPN/専用線サービスによるセキュア・プラットフォーム*
- お客様による柔軟な運用**

スピーディな
アプリケーション環境構築

Polyglot
-多言語サポート-
Java, JavaScript, Python, Swift, PHP, …

150種類以上の豊富な
サービス (API)
AI, IoT, Blockchain, …

ハイブリッド・クラウド対応
- **IBM Cloud Public** - Public Cloud
- **IBM Cloud Dedicated** – Single Tenant
- **IBM Cloud Private** - Private Cloud

Open Toolchains
-オープン・スタンダードな
DevOpsサポート-

IBM Garage
-破壊的アイディアの創造-

* 専用線サービスとはDirectLinkサービスのことです。VPNとDirectLinkをご利用になる場合、IBM Cloud Dedicatedのご契約が必要です。
** IBM Cloud Privateにおいてはお客様による運用が可能です。IBM Cloud PublicとIBM Cloud Dedicatedにつきましては、IBMが運用いたします。

マイクロサービス概要

Microservices：概要

クラウド・ネイティブ・アプリの”つくり”を規定するのがMicroserviceです
高頻度のメンテ＆変更に耐えうる柔軟な構造をアプリに与えます

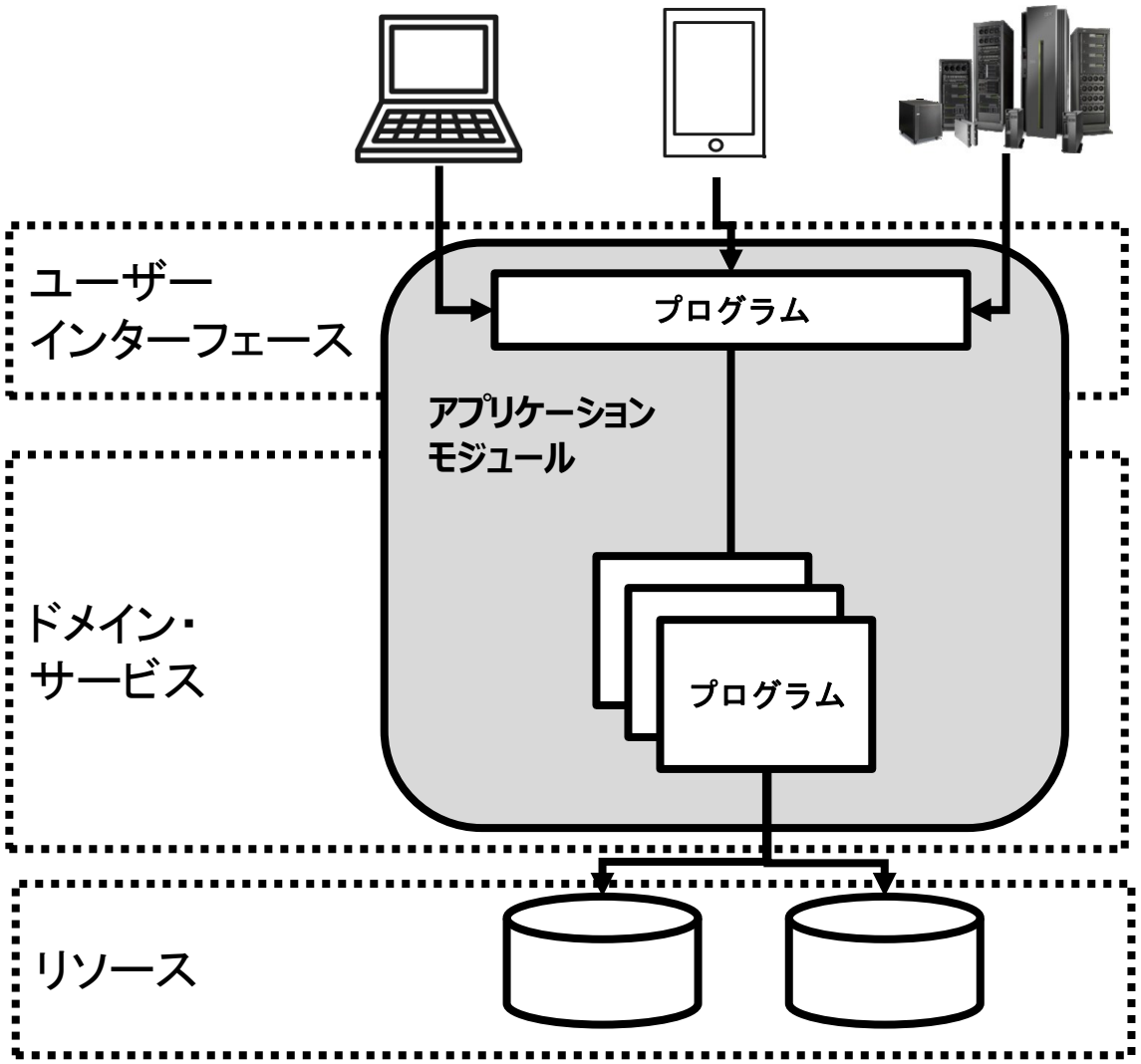
■ クラウド・ネイティブ・アプリケーション開発・運用のベスト・プラクティス
- ◆アーキテクチャー
- ◆開発・運用チーム・フォーメーション
- ◆システム・ライフサイクル

■ 動機
- ◆アプリケーション個別の保守を可能にする柔軟なモジュラー構造(サービス)の実現

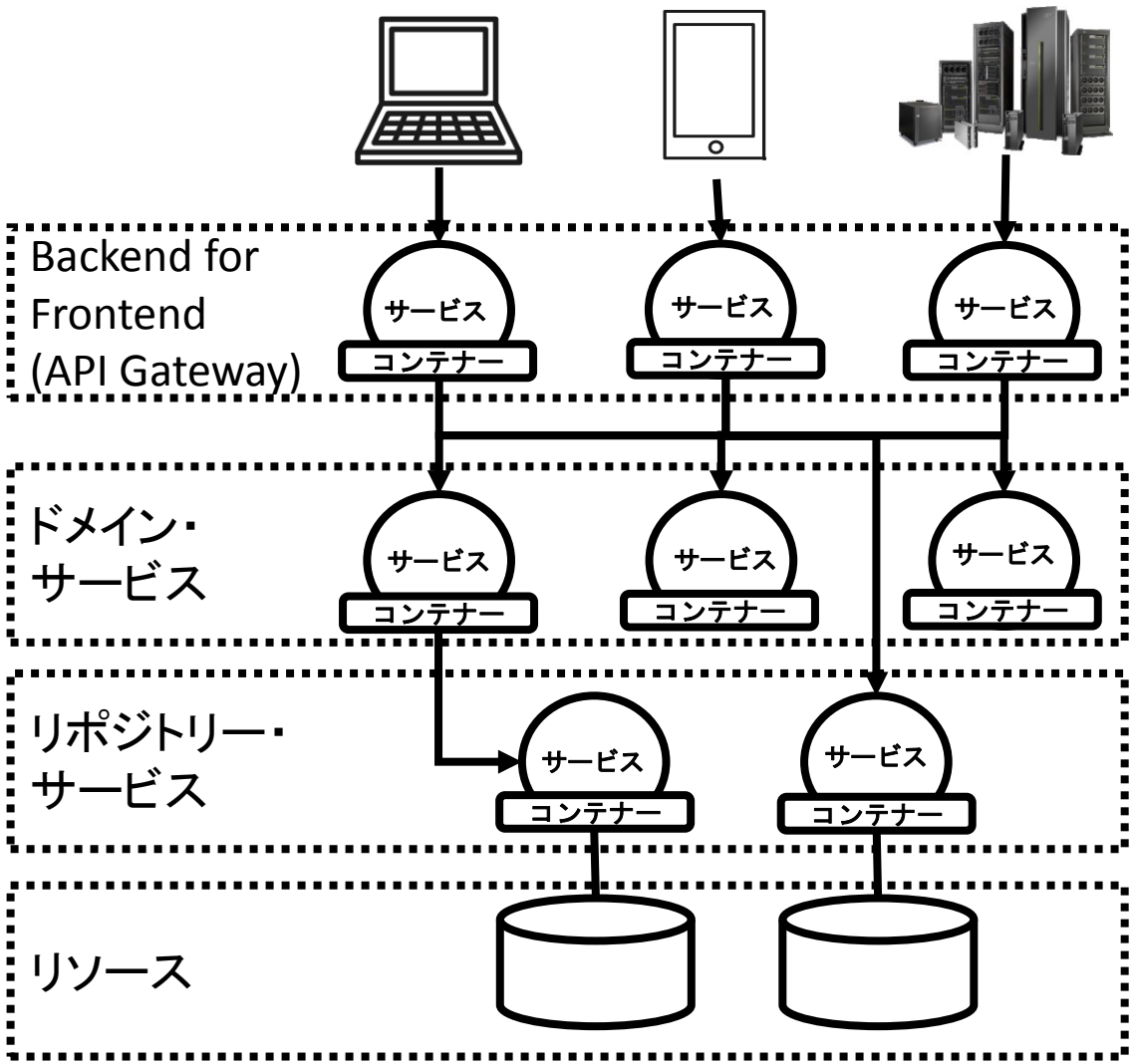
■ マイクロサービス・アーキテクチャ・スタイル
- ◆**小さなサービスを組み合わせて**, 一つのアプリケーションを開発する
- ◆各サービスは, **それぞれ独立したコンテナーで動作**する
- ◆各サービスは, **RESTやメッセージングのような軽量な仕組みで通信**する
- ◆各サービスは, 完全に**自動化された仕組みで, それぞれ個別にデプロイ**される
- ◆サービスは, それぞれ**異なるプログラミング言語で実装**できるし, **異なるデータ・ストレージを利用**できる

Microservices：アーキテクチャー・イメージ

モノリス (1枚岩)アプリケーション構造

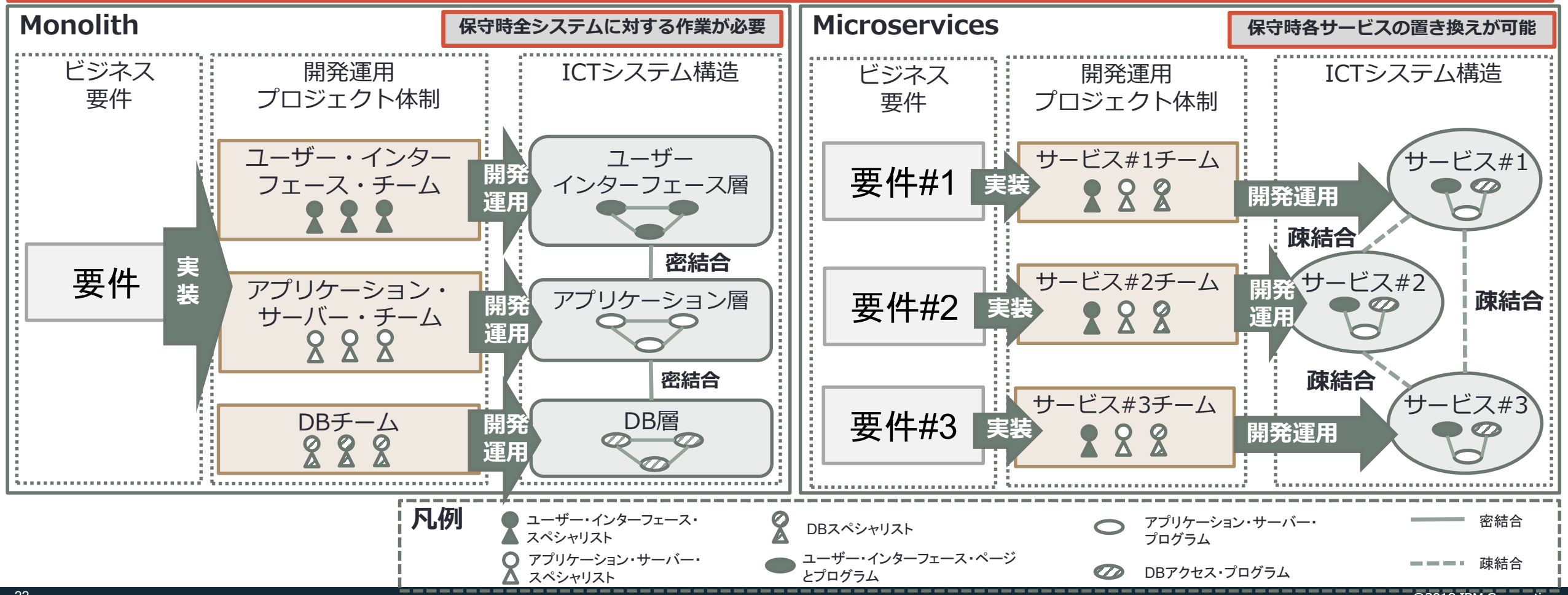


マイクロサービスに基づいたアプリケーション構造



Microservices：開発・運用チーム・フォーメーションの関係

- コンウェイの法則：システム構造は, プロジェクト体制を反映する
- Microservices：
  - ビジネス目的に基づいてチーム編成
  - システムは、独立して置き換え可能なサービスで構成される



Microservices：システム・ライフサイクル

プロジェクトではなく製品として捉える
～フィードバックを受けながら, One Teamが継続的に”開発・運用”～

プロジェクトの例：ダム

開発局面 → ゼネコン

運用局面 → 官公庁・電力会社等

Microservicesの開発・運用スタイル

製品の例：自動車

開発・運用局面 → 開発チーム

開発・運用局面 → 開発チーム

マイクロサービス実践のポイント

MicroservicePremium～マイクロサービス、始める？始めない？

don't even consider microservices unless you have a system that's too complex to manage as a monolith.

https://martinfowler.com/bliki/MicroservicePremium.html

基本方針

対象システムが複雑である場合：マイクロサービス化を検討する

対象システムが単純である場合：マイクロサービス化には向いていない

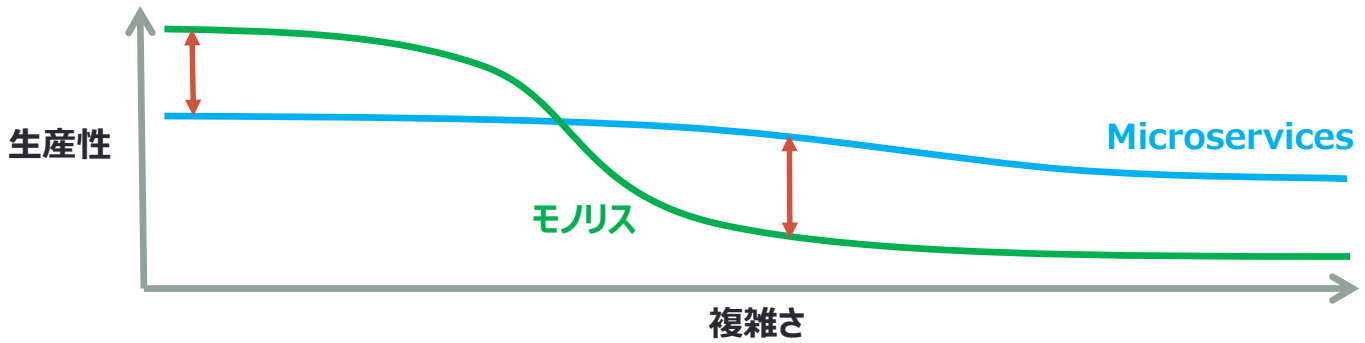
■ 複雑さの例：

◆大規模開発チーム, マルチ・テナンシー, 多様な連携モデル, スケーラビリティ, 複数のビジネス機能

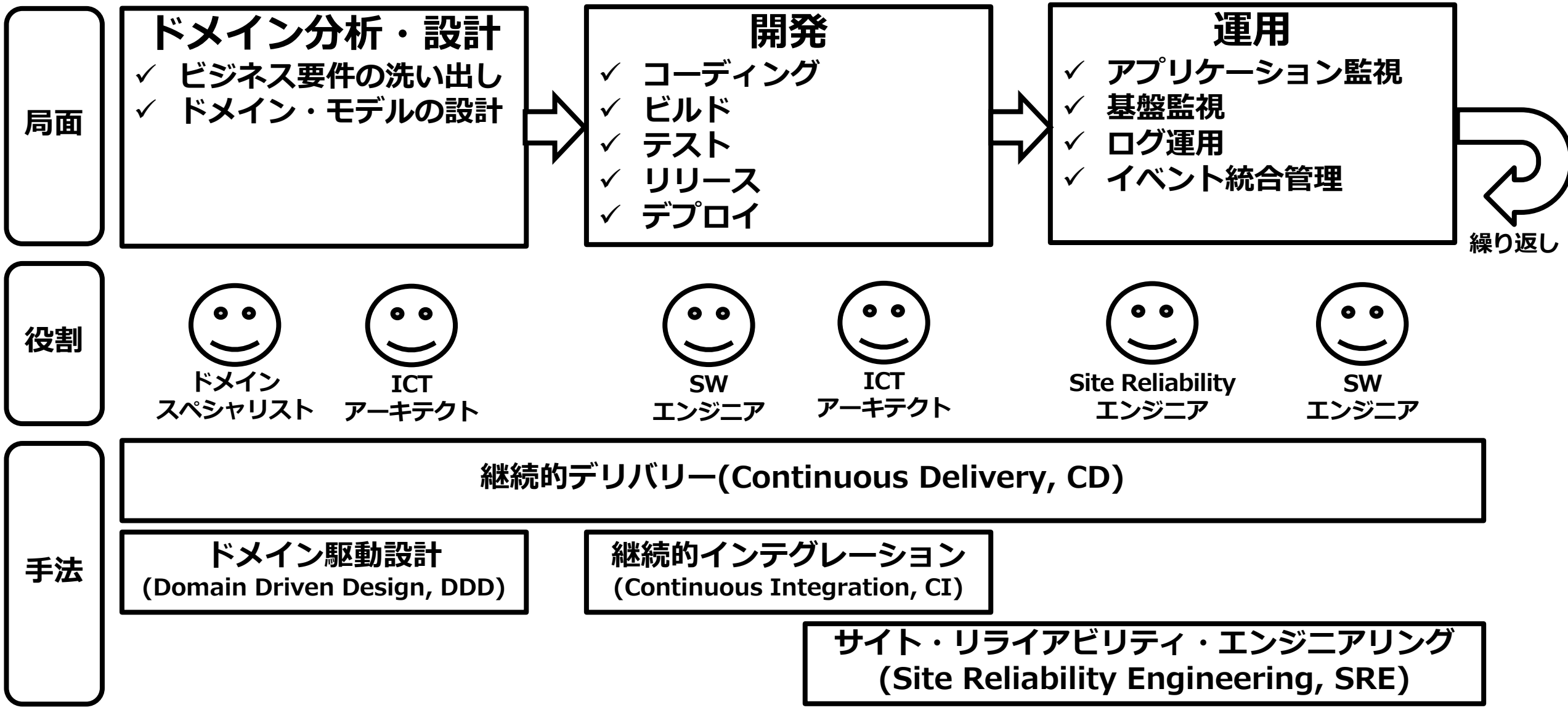
■ 単純なシステムに対するマイクロサービス化は、高いコストに帰結
～”MicroservicePremium”

◆マイクロサービス化のための取り組み・実装が、”無駄な”コストに繋がる…

●自動デプロイメント, 分散システム運用監視, 結果整合性, ビジネス分析, サービス設計, …



開発・運用の流れと役割



サービス化どう進める?

粒度の議論はやめましょう

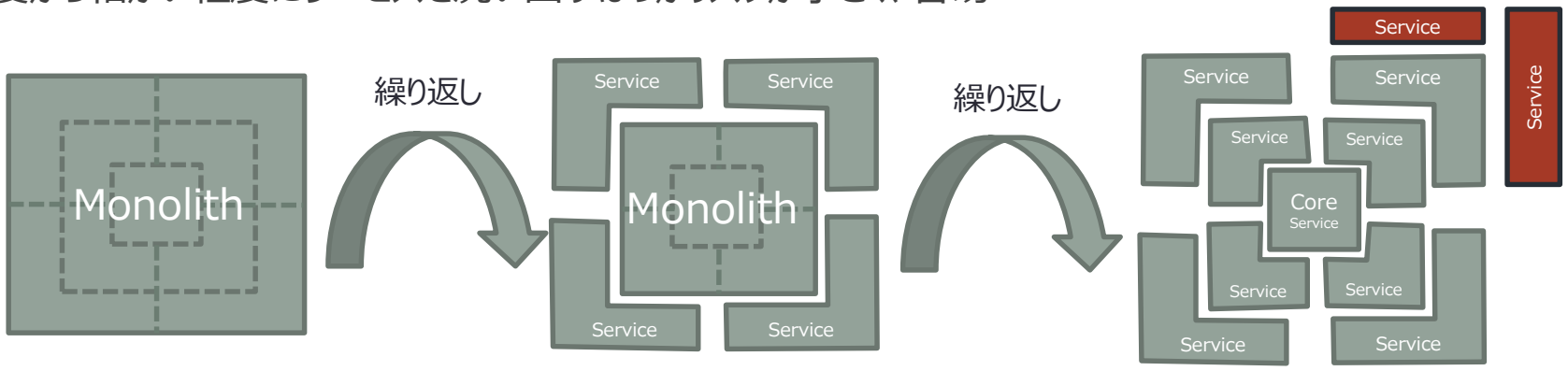
- 机上の粒度の議論は不毛であると結論が出ている。
- PowerPointや議論ではなく、プログラミングによる実践が、クラウド・ネイティブの流儀である。

サービス化どう進める?：ビジネス指向と繰り返し開発

前提/文脈	● 素早いビジネス立上げ ● 柔軟なビジネス変更	● 1回の分析・設計で サービス最適化は不可能
方針	ビジネス指向	開発・運用の 繰り返し
方策	✓ 業務観点でサービスの 切り出し ✓ ドメイン駆動設計	✓ アジャイル開発 ✓ モノリス・ファースト
期待される 効果	素早いビジネスのローンチ& トランスフォーメーション	サービス粒度の最適化

サービス化どう進める?：モノリス・ファースト

- モノリス・スタイルのアプリケーションを段階的にマイクロサービス・スタイルに置き換えるプラクティス
  - ◆初回のアプリケーション開発時：従来通りモノリス・スタイルで開発
  - ◆繰り返し開発/保守時：マイクロサービス・スタイルで開発
  - ◆http://martinfowler.com/bliki/MonolithFirst.html
- 動機
  - ◆サービス化の実践
  - ◆アジャイル開発の実践
- 効果
  - ◆ソフト・ランディングで現実的なサービス化が可能
    - ●粗い粒度から細かい粒度にサービスを洗い出すほうがリスクが小さく、容易

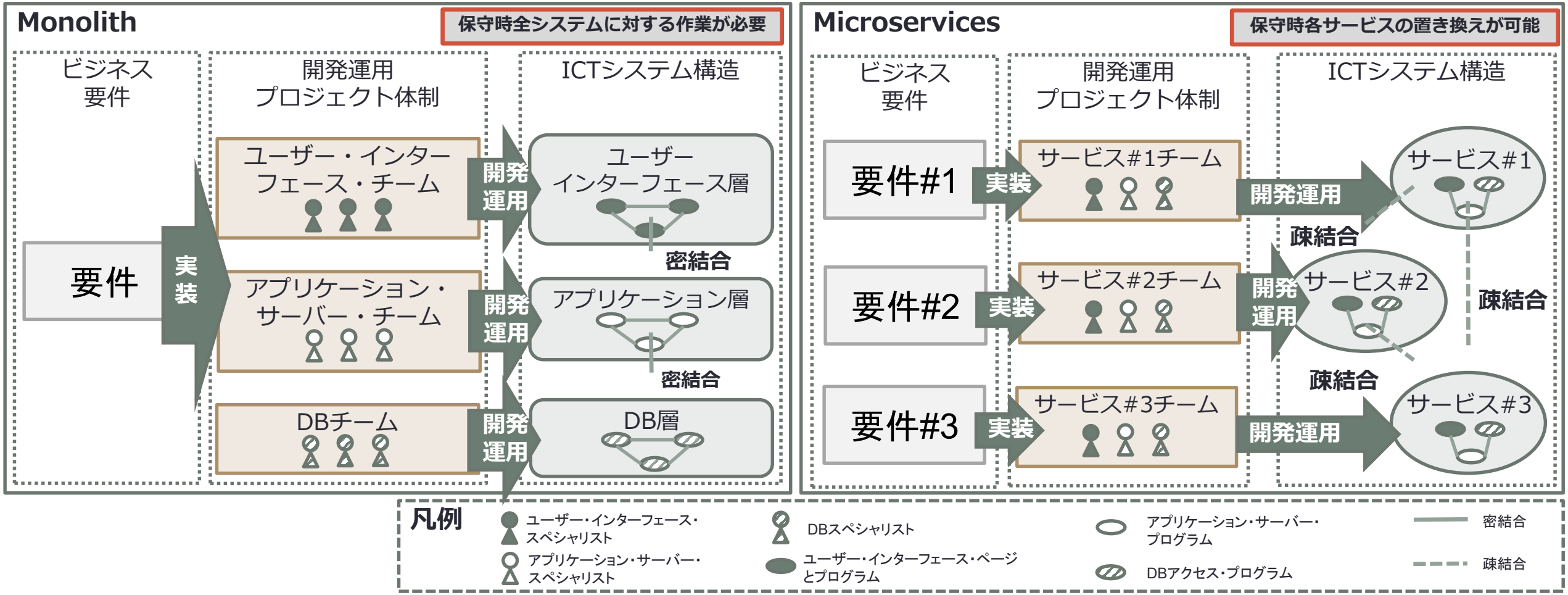


- ➢ モノリスとしてアプリケーションを開発
- ➢ 一部機能をサービスとして分離
- ➢ 機能をサービスとして分離
- ➢ 新機能をサービスとして追加開発

サービス化どう進める?：チーム・フォーメーション

■チーム・フォーメーションとサービスの相関関係

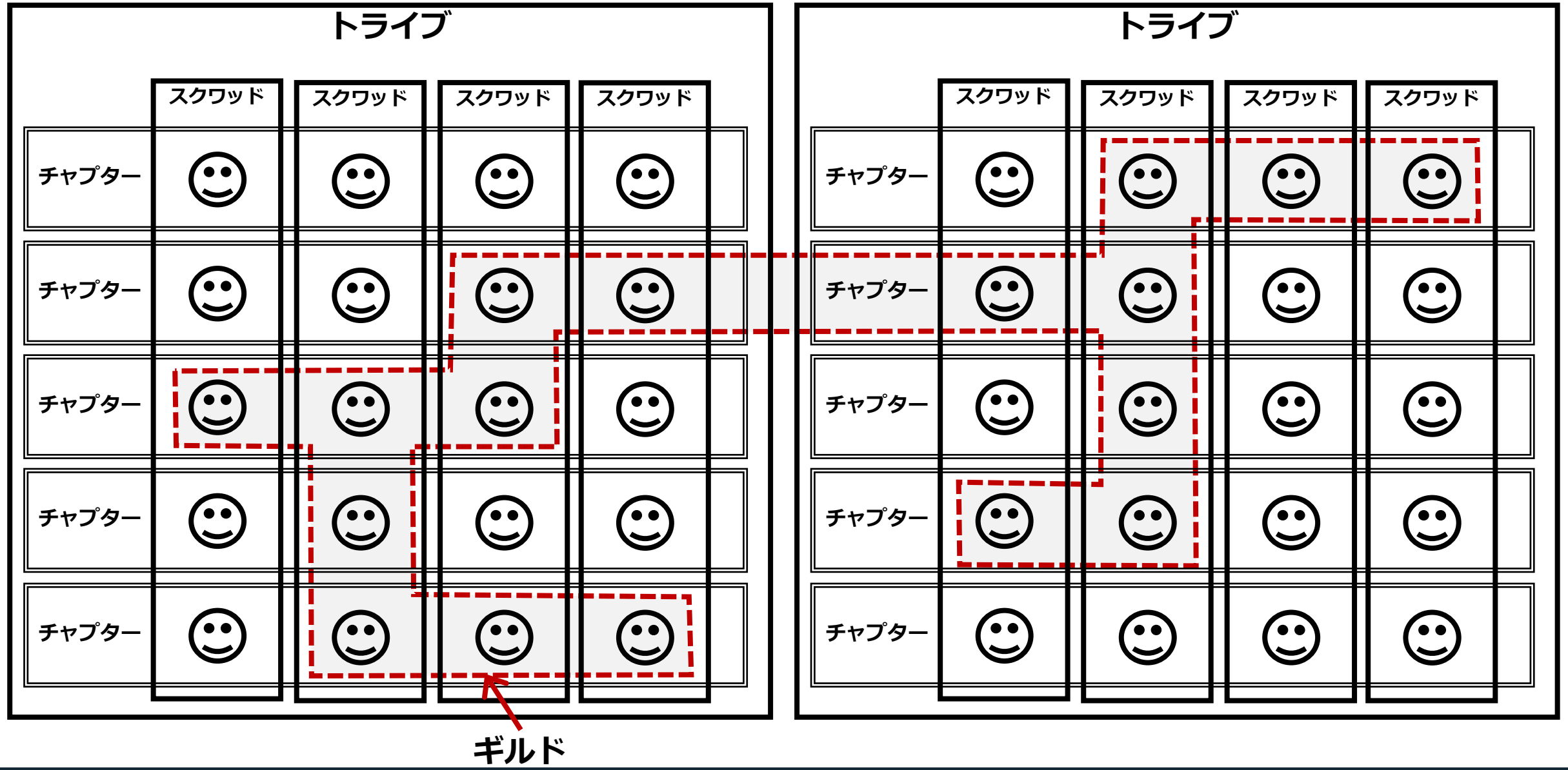
◆コンウェイの法則：サービス単位のチーミング



サービス化どう進める?：チーム・フォーメーション

- チーム・スケールの目安
  - 2枚のピザ・ルール
  - Spotify
    - スクワッド (Squad)：約 8名
    - トライブ (Tribe)：50 Squads (≒400名)

[参考] サービス化どう進める?：Spotifyのチーム編成



[参考] サービス化どう進める?：Spotifyのチーム編成

- トライブ (Tribe)
  - ◆一番大きな組織体。ある特定の独立した対象(製品、サービス等)に責任範囲とする。
- スクワッド (Squad)
  - ◆個別のビジネス要件、まとまった機能 (Spotifyではこれを”Product”と呼ぶ)の開発・運用に責任を持つ。
  - ◆8名程度。
- チャプター (Chapter)
  - ◆各スペシャルティ毎に設定される人事上の組織体。
  - ◆各スクワッド・メンバーは、人事上はチャプターに属す。
- ギルド (Guild)
  - ◆特定のスペシャルティをテーマとする組織横断のコミュニティ。
  - ◆自身が属すトライブ, スクワッド, チャプターを問わず参加可能。

サービスの構造

■ 構成要素

◆境界

- ●ビジネス対象を特定する範囲
  - ➢ どのデータを対象とするか、ではない
  - ➢ ビジネス範囲とサービス範囲は同一となる

◆サービス(文脈またはコンテクスト境界)

- ●実装すべきビジネス要件
- ●インターフェースとモデルを包含する

◆インターフェース

- ●プログラミング・インターフェース

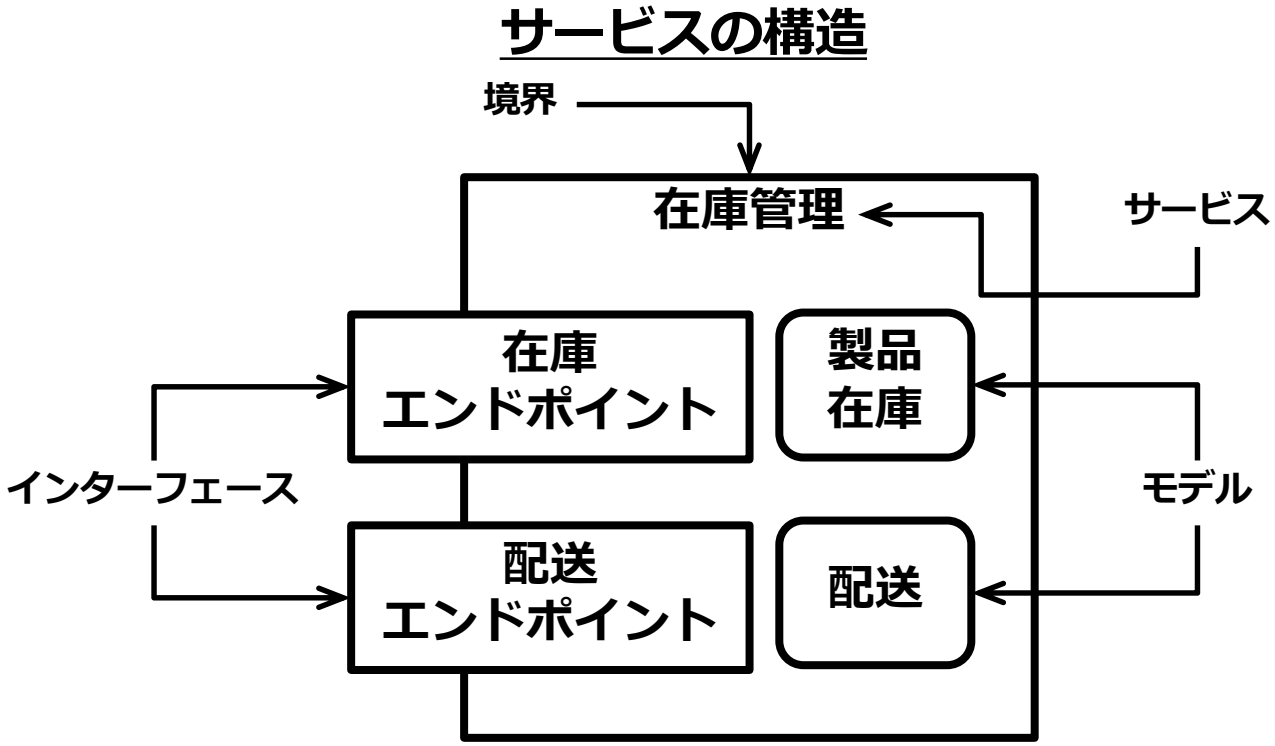
◆モデル

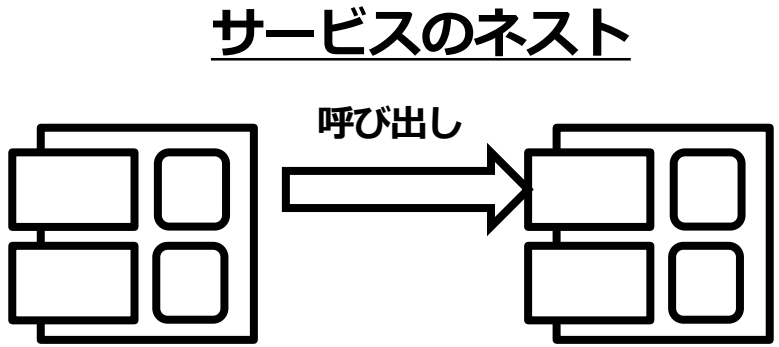
- ●データ

■ 設計上のポイント

◆サービスのネストは許容される

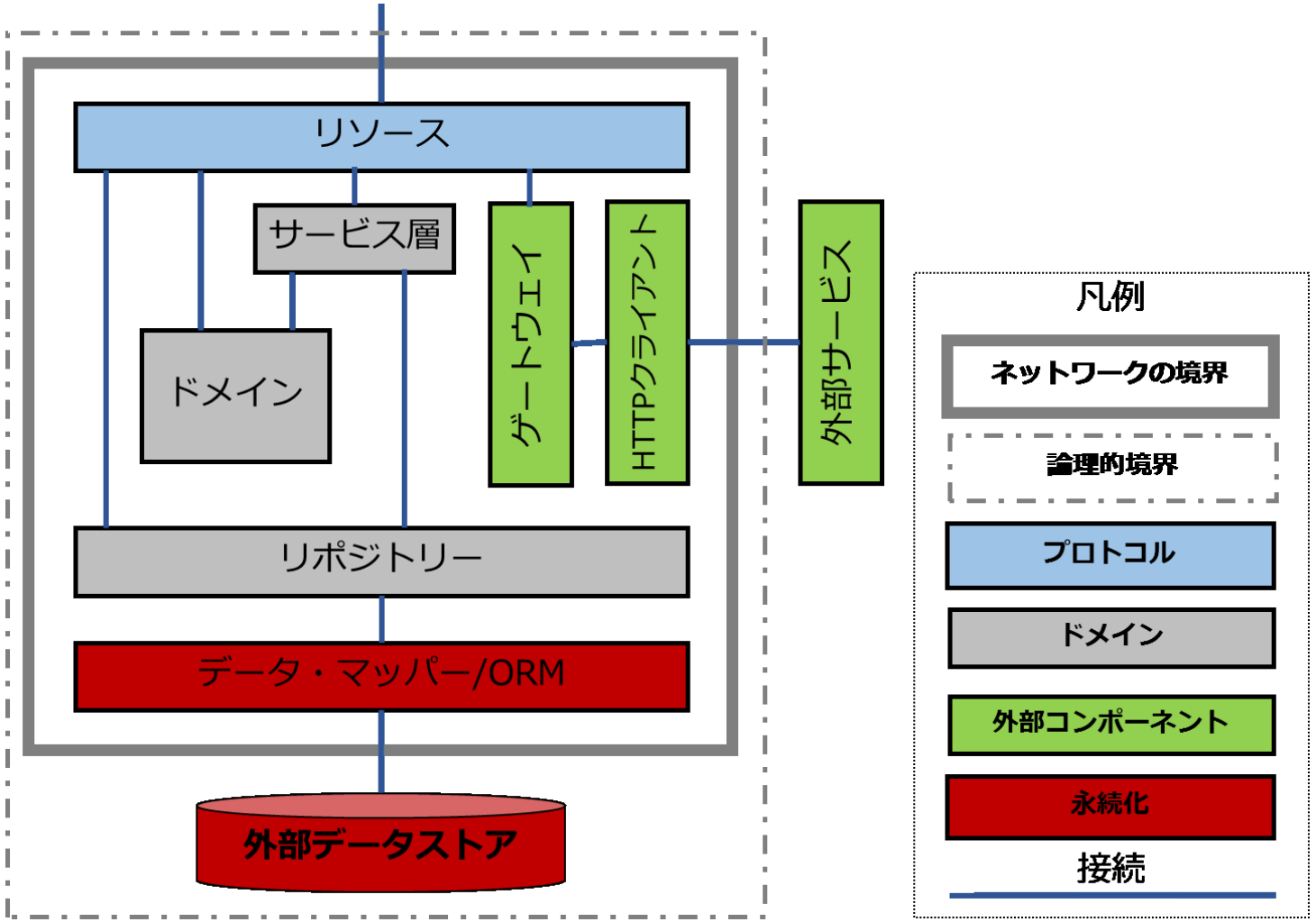
- ●ネスト：サービスが他のサービスを呼び出すこと





サービスのレイヤー構造

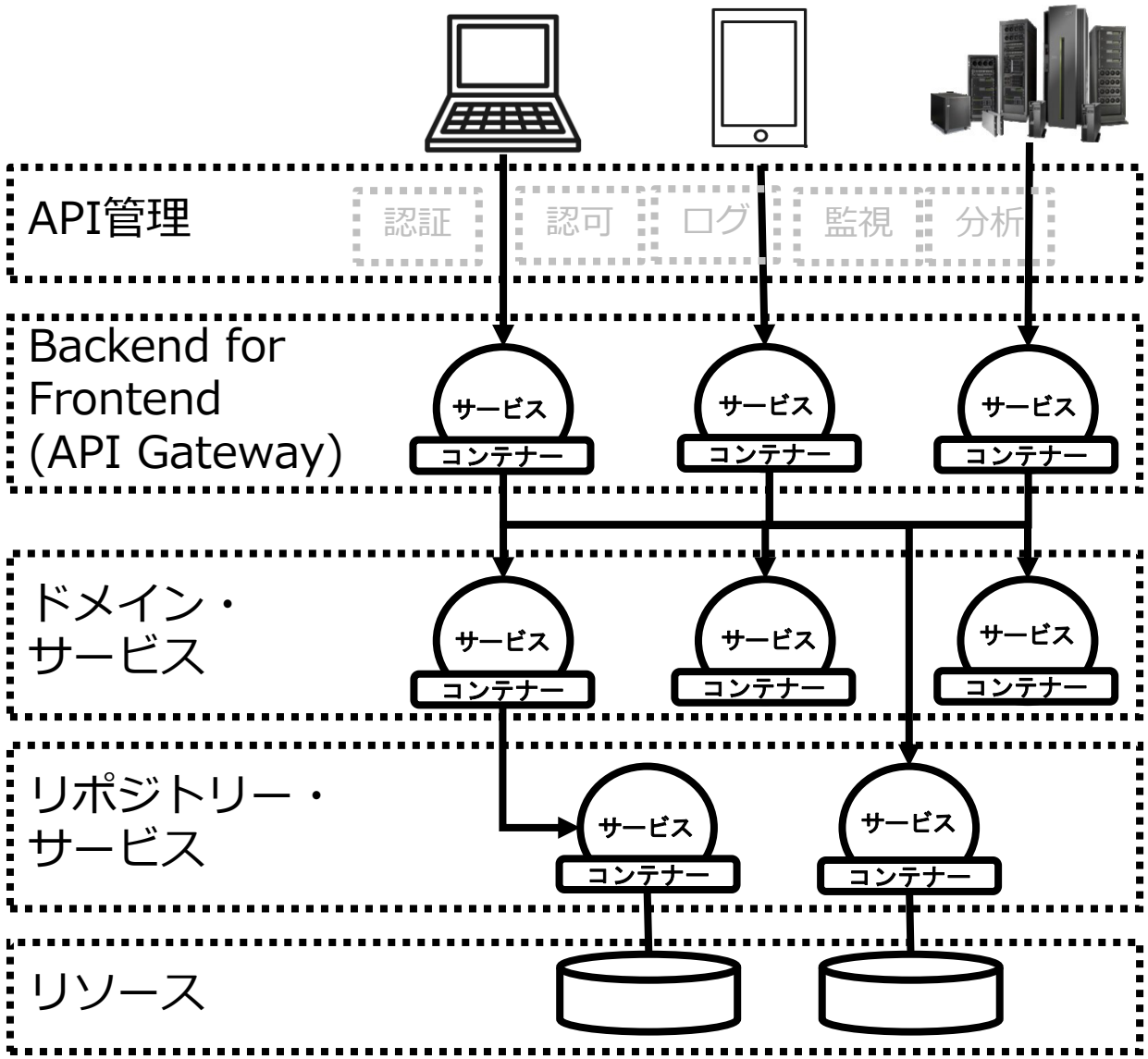
■ マイクロサービス・アーキテクチャー・アプリケーションの内部構造



https://martinfowler.com/articles/microservice-testing/

サービスのレイヤー構造

■ マイクロサービス・アーキテクチャー・アプリケーションの構造



- ゲートウェイ
- Façade

✓ クライアント向けAPIエンド・ポイント
✓ クライアントに応じたデータ/プロトコル変換
✓ サービス呼び出しのアグリゲーション

- ビジネス・ロジック

✓ 業務処理の実装

- リソース・アクセス

✓ データ, メッセージング等リソース・アクセス

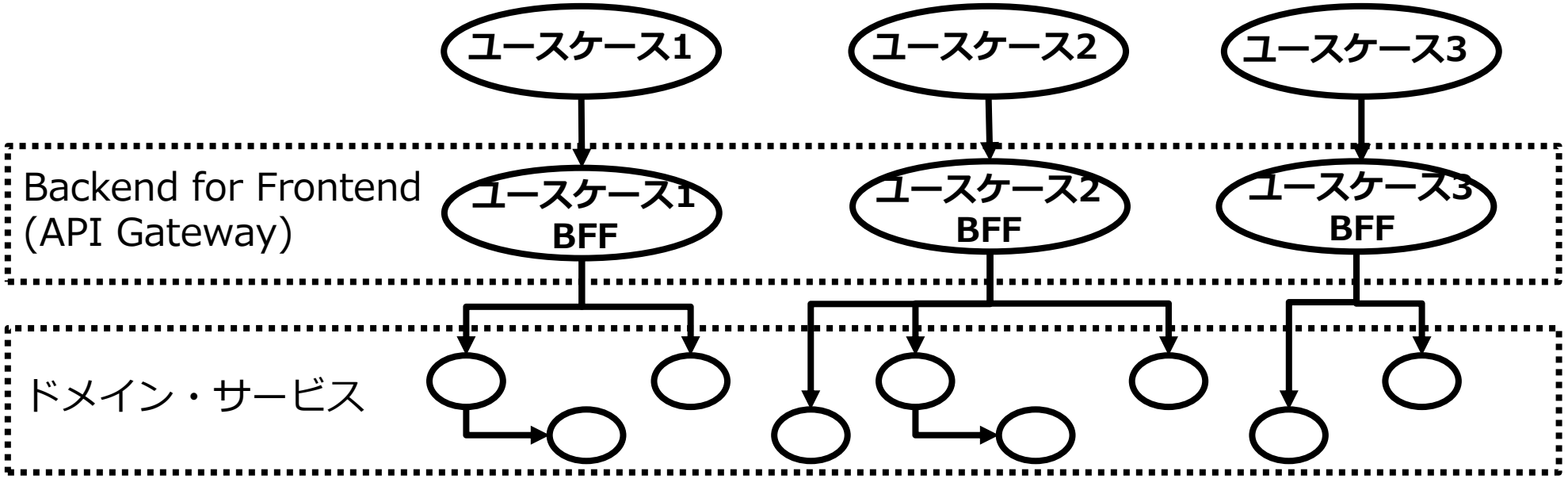
BFF/API Gatewayの粒度

2つの観点からBFF/API Gatewayの設置単位を検討する
- **ユースケース**
- **クライアントのタイプあるいは種類**

■ユースケースの観点での検討

◆1ユースケース毎に、1BFFを推奨

◆複数ユースケースに1BFFを設置した場合、BFFの粒度が大きくなり、メンテナンスのボトルネックとなる



BFF/API Gatewayの粒度

■クライアントのタイプあるいは種類の観点での検討

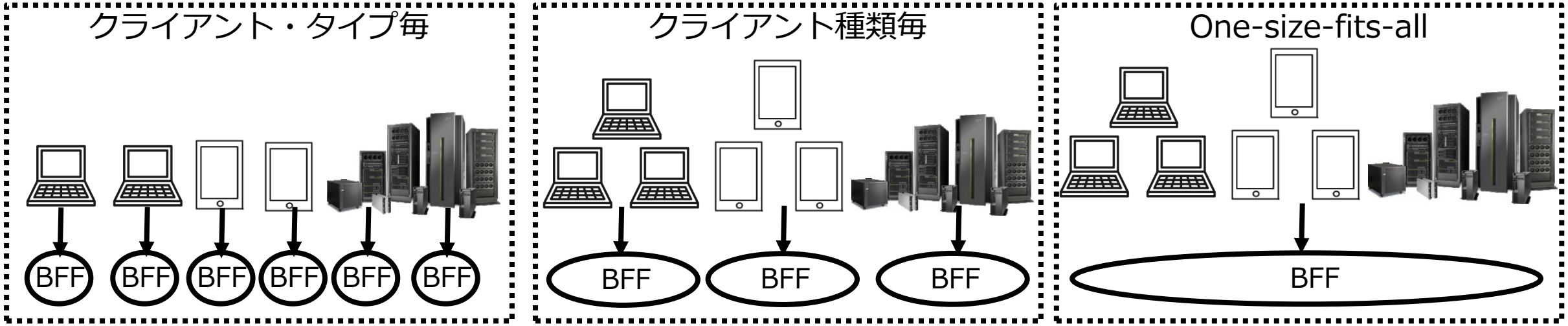
◆原則クライアント種類毎に1BFF設置を推奨

◆クライアント・タイプ毎に1BFF設置した場合

●BFFの種類が多くなりすぎ、メンテナンスのボトルネックになる懸念がある

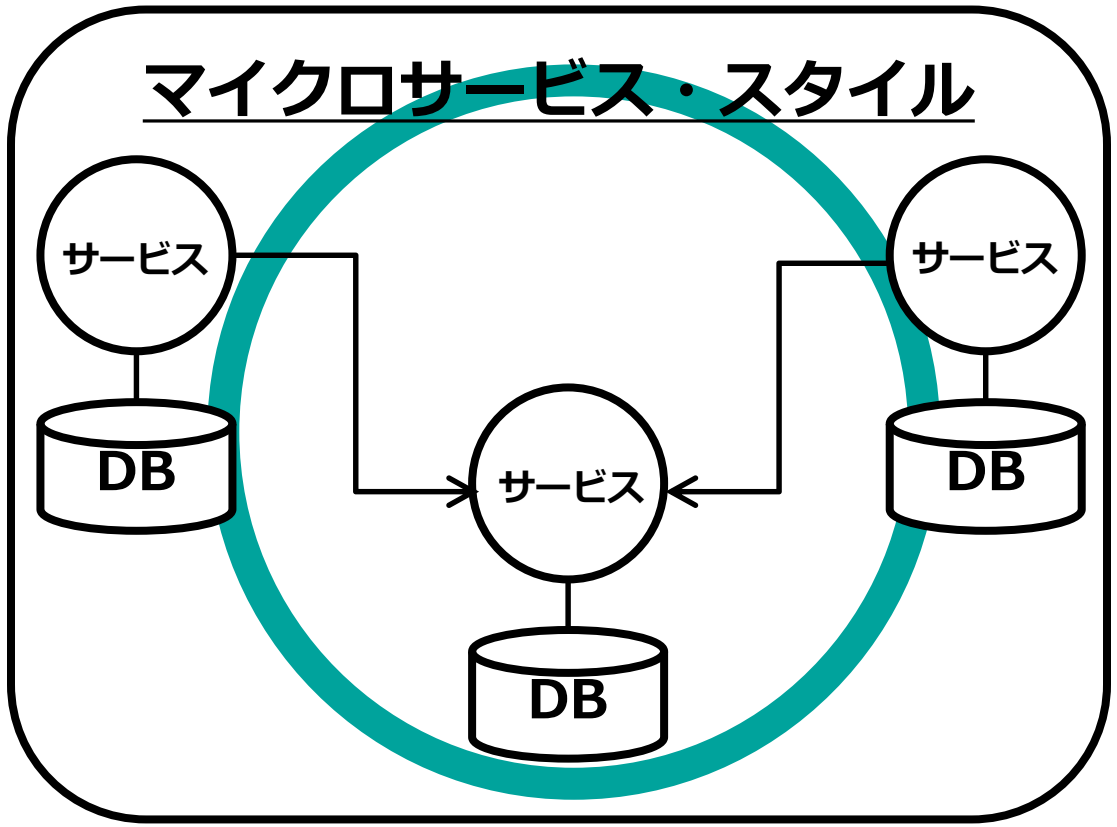
◆One-size-fits-allの場合

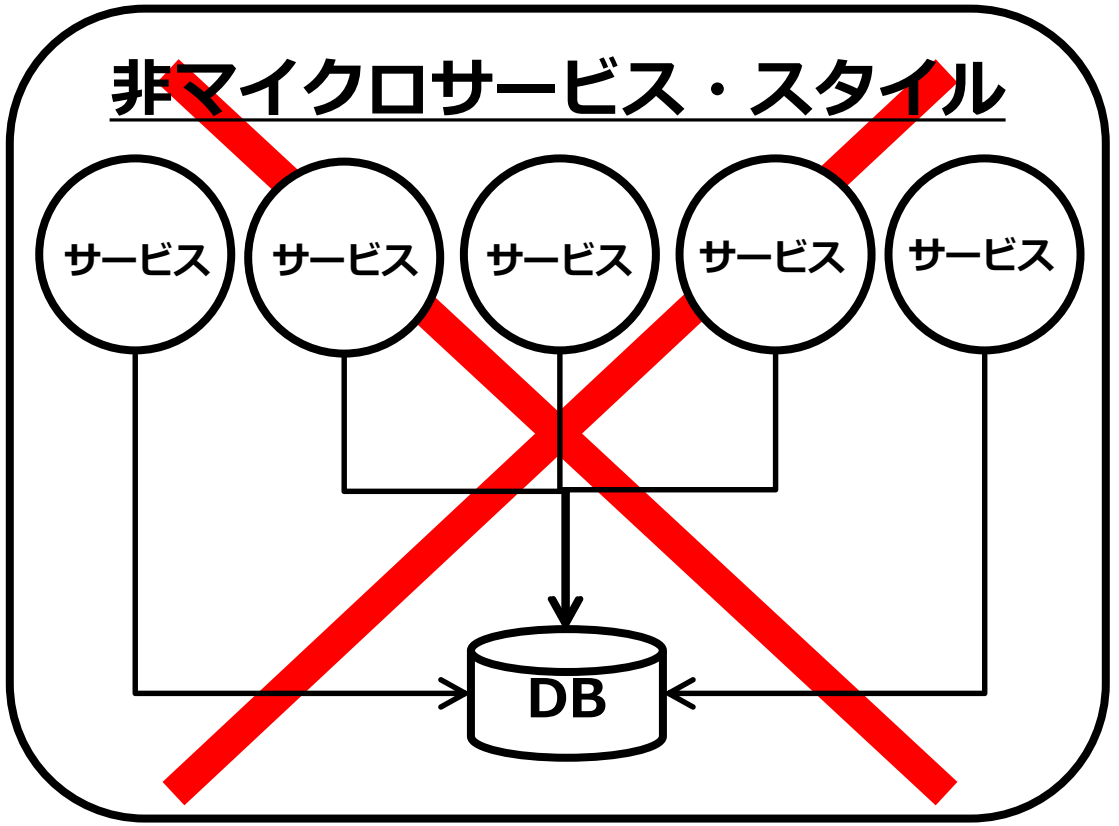
●BFFが大きくなりすぎ、メンテナンスのボトルネックになる懸念がある



データ・アクセス

■ サービスを介してデータベースにアクセスする





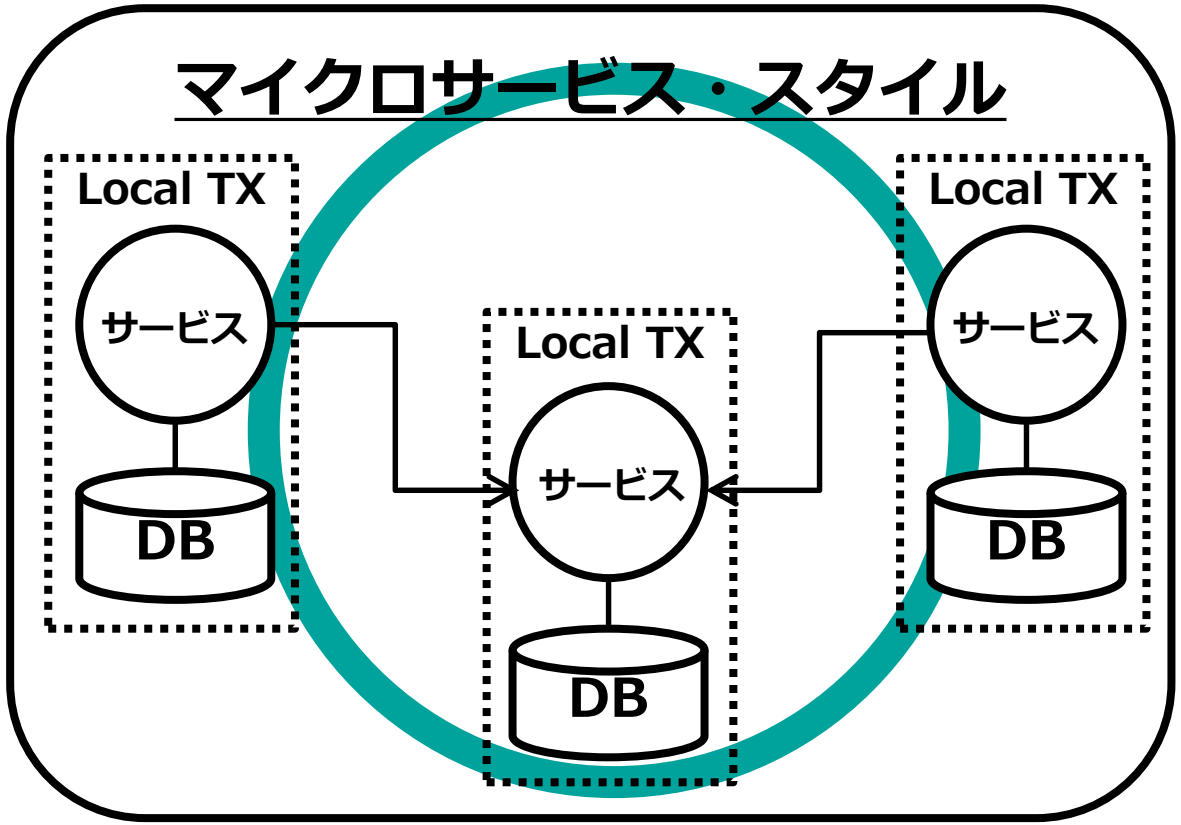
DBのメンテナンスの影響を最小化し、サービスへの影響を極力排除する

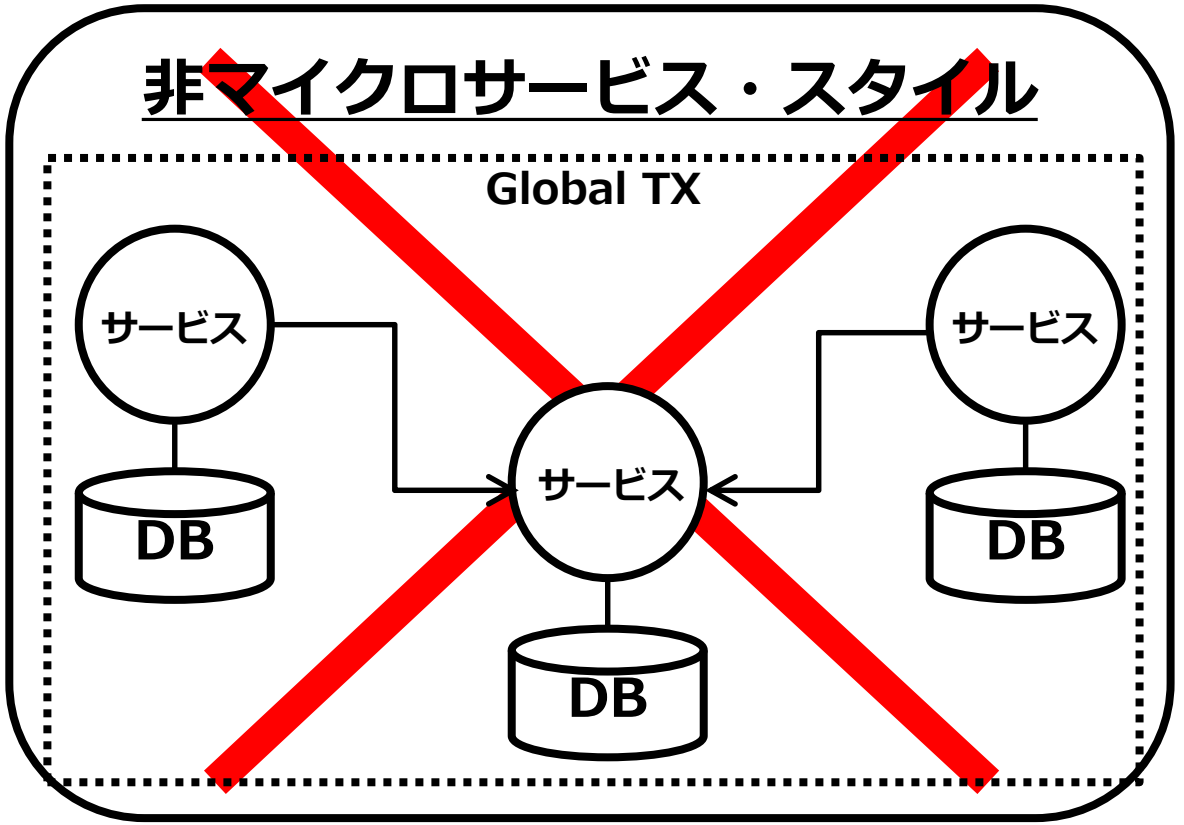
トランザクション処理

■ 原則ローカル・トランザクション処理

◆分散トランザクションは推奨されない

Local TX：ローカル・トランザクション
Global TX：グローバル・トランザクション





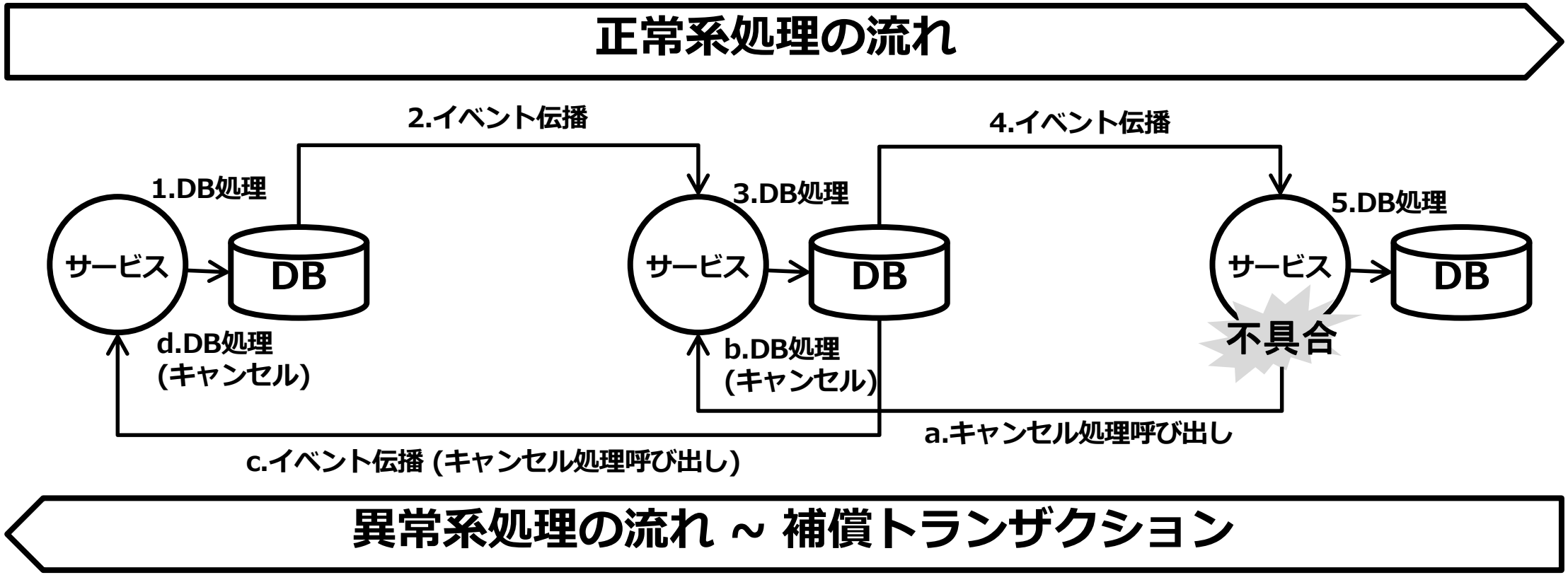
- 実装をシンプルに保ち、スピーディな障害対応に備える
- サービス間の疎結合を維持する

トランザクション処理：複数リソースの同期処理

■ Saga パターン

◆複数リソースの同期を取るデザイン・パターン

◆ローカル・トランザクション, イベント, 補償トランザクションを活用



IBM Cloud Functionsのイベント駆動処理に最適な処理パターン…

データ&トランザクション処理：データをどこに配置する?

文脈：データをどこに配置するか？
制約：パブリック・クラウドにはデータを配置したくない。
要件：セキュア & ハイ・パフォーマンス。
解決策：キャッシュを使いましょう！！！

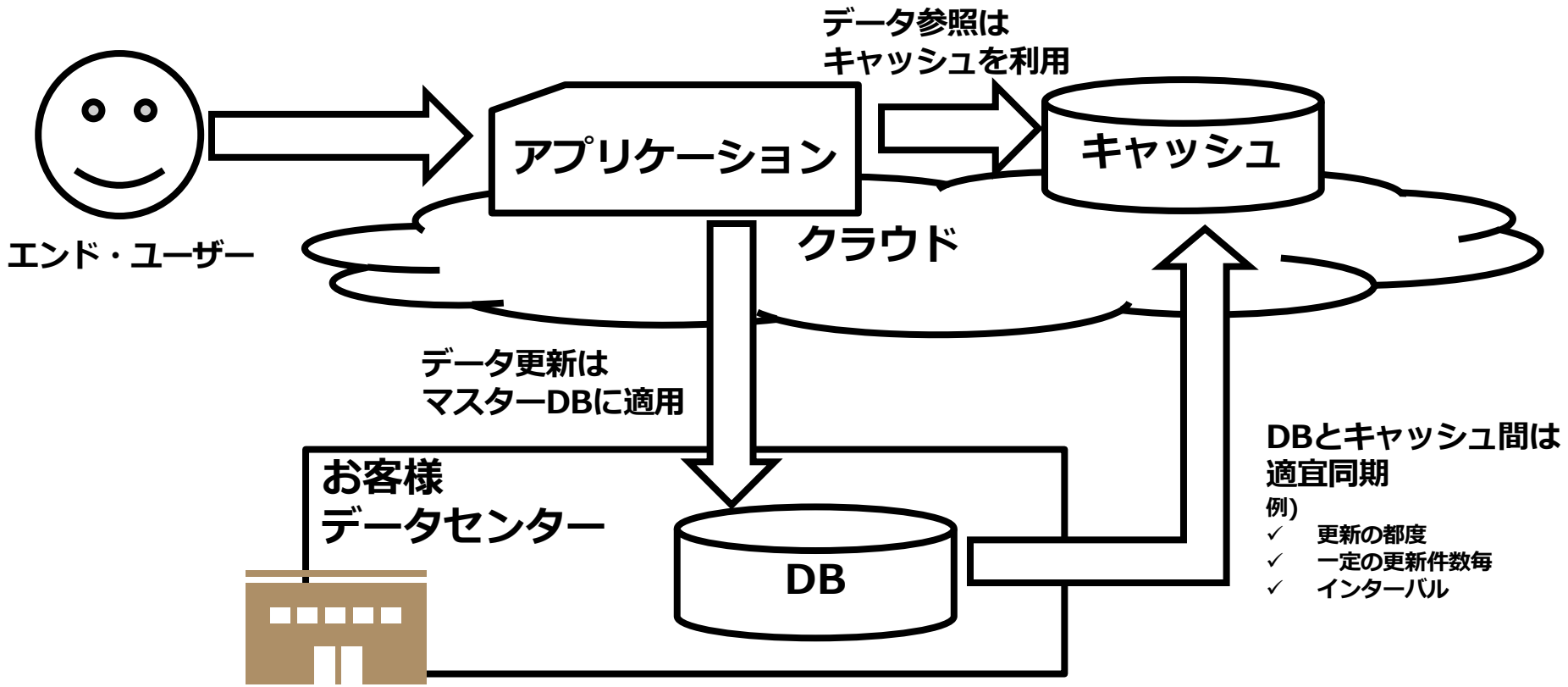
で、一体どうする???

参照系データと更新系データの分離

- 参照系データ：クラウド上のキャッシュ
- 更新系データ：任意のセキュア・ロケーションのマスターDB
- マスターDBとキャッシュは適宜同期

データ&トランザクション処理：データをどこに配置する?

■ 参照系データと更新系データの分離の実装モデル



セキュア

- マスターDBを安全なお客様データセンターに配備

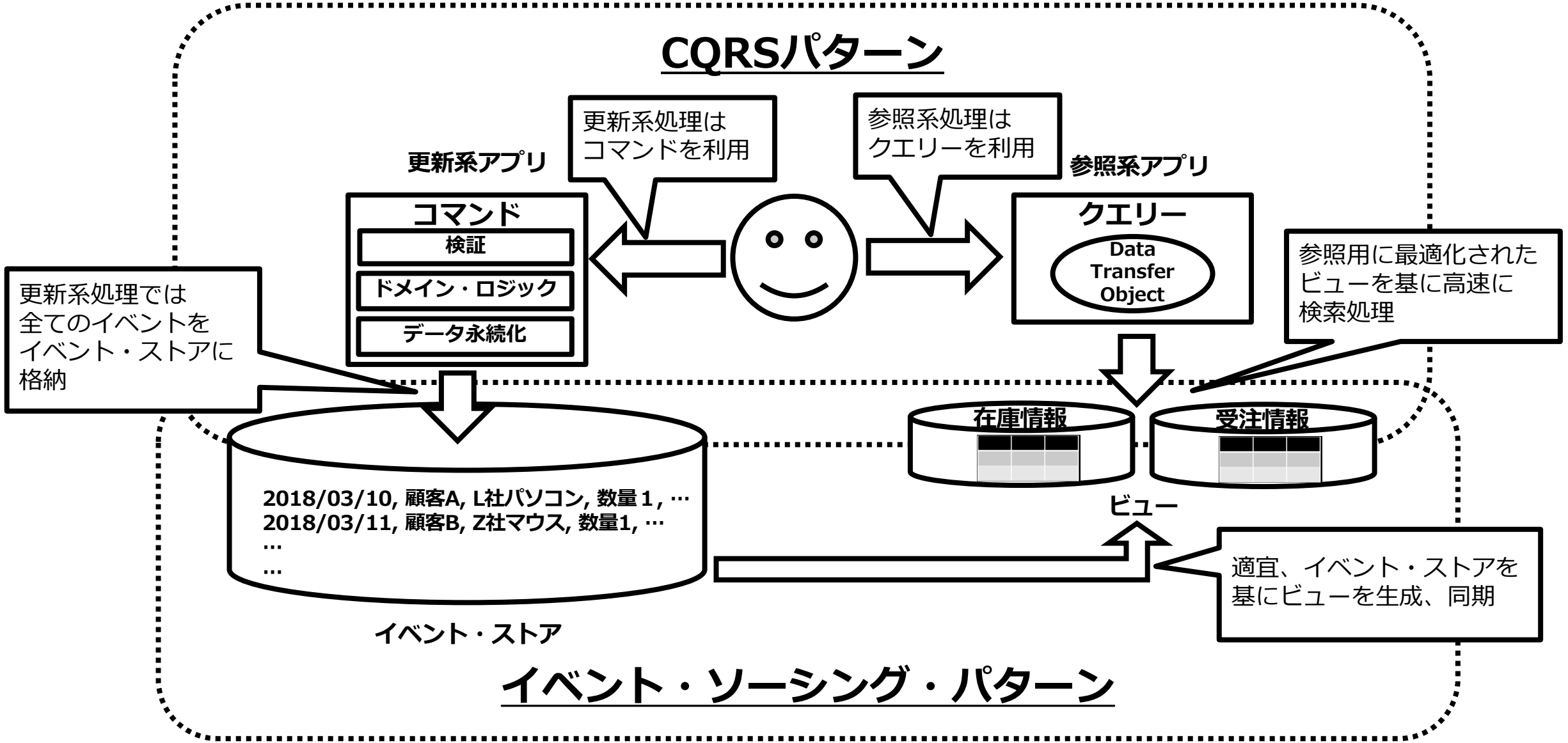
パフォーマンス最適化

- 高頻度の検索要求にはクラウド上のキャッシュで素早く処理
- クラウド-DC間のネットワークは、比較的低頻度の更新処理に限定

[参考]データ&トランザクション処理： CQRSとイベント・ソーシング

- CQRSとイベント・ソーシングは、クラウド・ネイティブ・アプリのデータ&トランザクション処理で用いられるデザイン・パターン
- CQRS (Command Query Responsibility Segregation):コマンドクエリ責務分離
  - ◆コマンド(データ更新)とクエリ(データ参照)に、異なるデータ・モデルを利用する
  - ◆期待される効果：
    - ●パフォーマンス最適化
    - ●コマンド, クエリそれぞれの処理に対する、データ・モデルの最適化
    - ●セキュリティとアクセス管理の単純化
- イベント・ソーシング
  - ◆業務処理に纏わる一連の属性情報をイベントとして、イベント・ストアに書き込む
    - ●イベントの発生を受けて、関連する業務を駆動する
  - ◆イベント・ストアを基に適宜ビューを生成・同期し、参照処理に充てる
  - ◆期待される効果：
    - ●パフォーマンス最適化
    - ●ドメインに関する完全な監査性・証跡性

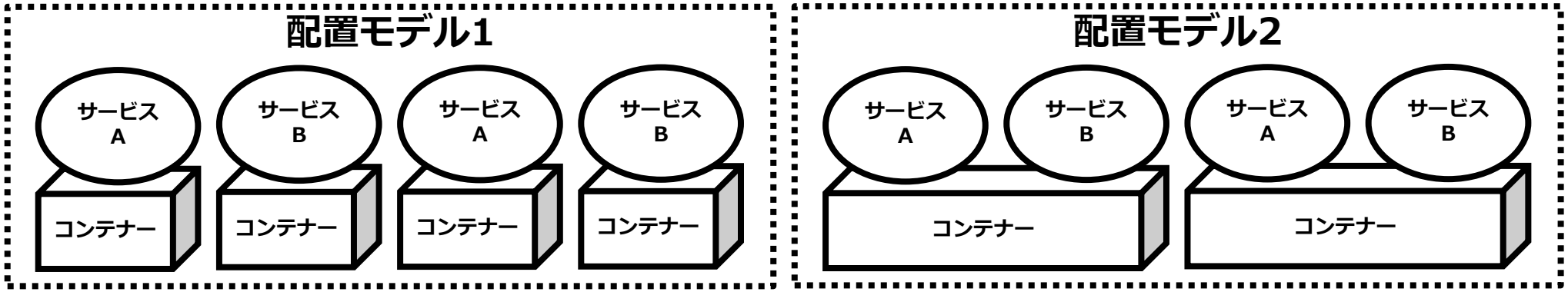
[参考]データ&トランザクション処理：CQRSとイベント・ソーシング



マイクロサービスの配置パターン

1コンテナーあたり、1つのサービスをデプロイする

- 各サービス毎(≒ビジネス毎)のメンテナンスが可能。
- よりきめ細かなスケーラビリティ調整が可能。

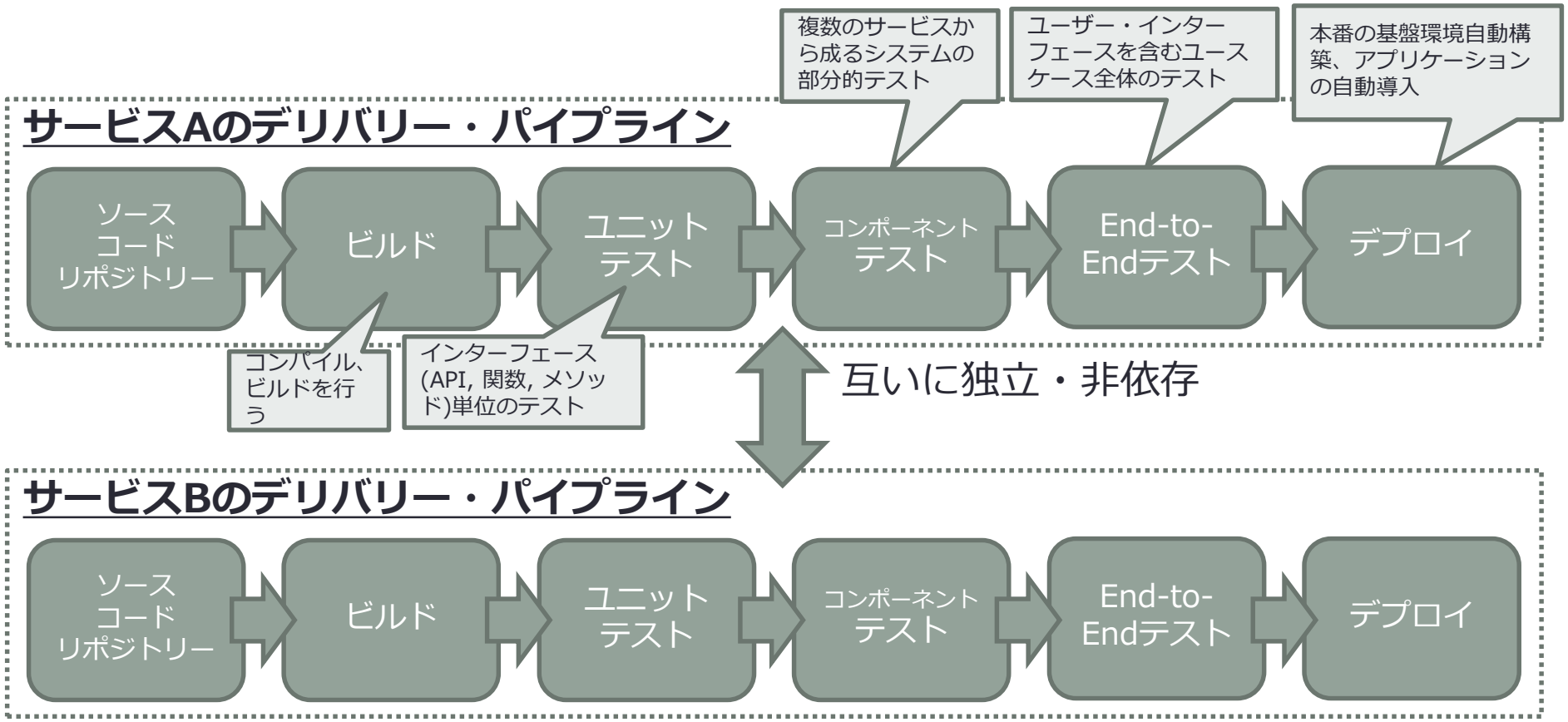


	配置モデル1 コンテナーあたり1サービス	配置モデル2 コンテナーあたり複数サービス
スケーラビリティ	◎ サービス毎にスケール可能。	○ コンテナー単位でのスケールのみ。
メンテナンスの柔軟性	○ サービス単位での変更が可能。	△ 複数サービスが同一基盤上に存在するため、 ルーティング等トリッキーな運用が求められる。

デリバリー・パイプライン設計

サービス単位に、デリバリー・パイプラインを設ける

- 各サービス毎(≒ビジネス毎)のデプロイ、メンテナンスが可能。
- 範囲が小さいため、デプロイ失敗時の影響を局所化し、リスクを最小限に抑制



Dockerコンテナ概要

Dockerコンテナ概要

■ 仮想環境を実現するソフトウェア・ソリューション

- ◆Linuxカーネル機能を利用したコンテナ型仮想化 (≒OSレベルの仮想化)を行う
- ◆Docker, Inc.(旧dotCloud, Inc.)が開発し、2013年3月にリリース開始
- ◆Apache License 2.0ライセンスの元、オープン・ソース・ソフトウェアとして配布

■ 特徴

- ◆軽量：MB単位のコンテナ・イメージ
- ◆スピーディ：秒単位のデプロイ時間
- ◆ポータブル：異なるホスト・マシンに持ち運び可能
- ◆標準：Open Container Initiative (OCI) への貢献 ～runC, containerd～
- ◆カスタマイザブル：ミドルウェア(バイナリー, ライブラリー)の構成最適化が可能

■ 関連製品技術

- ◆Docker Hub：コンテナ・イメージのレジストリー・サービス
- ◆Docker Compose：複数コンテナの構成定義・運用ツール
- ◆Docker Machine：Dockerエンジン(docker-machine)の導入・運用ツール
- ◆Docker Swarm：コンテナ・クラスターのオーケストレーション・ツール
- ◆Kubernetes：コンテナ・クラスターのオーケストレーション・ツール

コンテナ仮想化を実現する基礎技術

cgroups

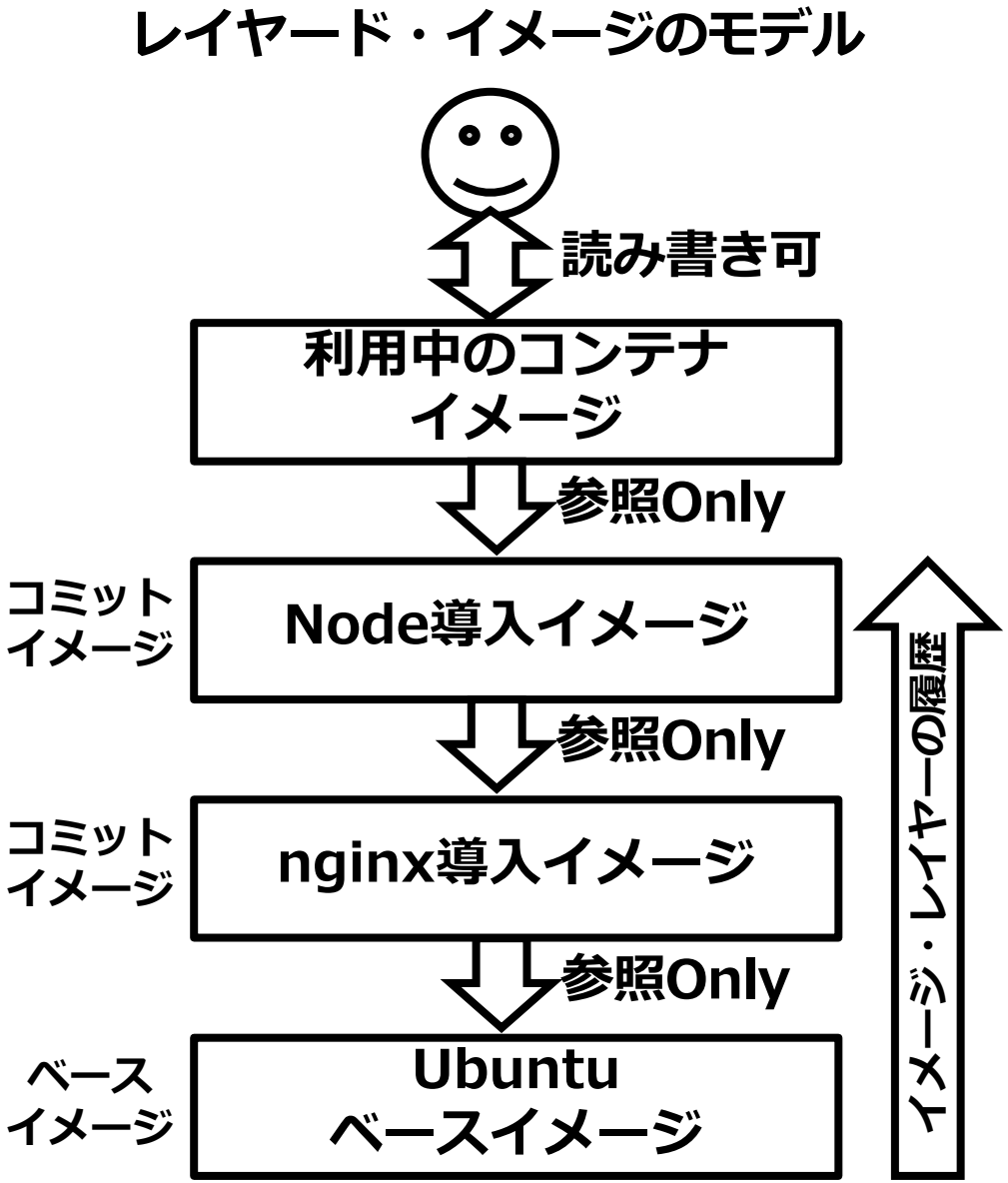
- プロセス/スレッドのグループに対するリソースのアロケーションを制御する仕組み。
- cgroupsが対象とするリソースの例：
  - CPU使用量, メモリーやスワップの使用量, デバイスへのアクセス権, プロセスの開始・停止, ネットワーク制御, 他

namespaces

- 独立した個別の名前空間を基にしたネーミングで、区画化・隔離を図る仕組み。
- namespacesの対象の例：
  - プロセス, ネットワーク, プロセス間通信, マウント, UID, UTS

aufs, devicemapper等ストレージ・ドライバー

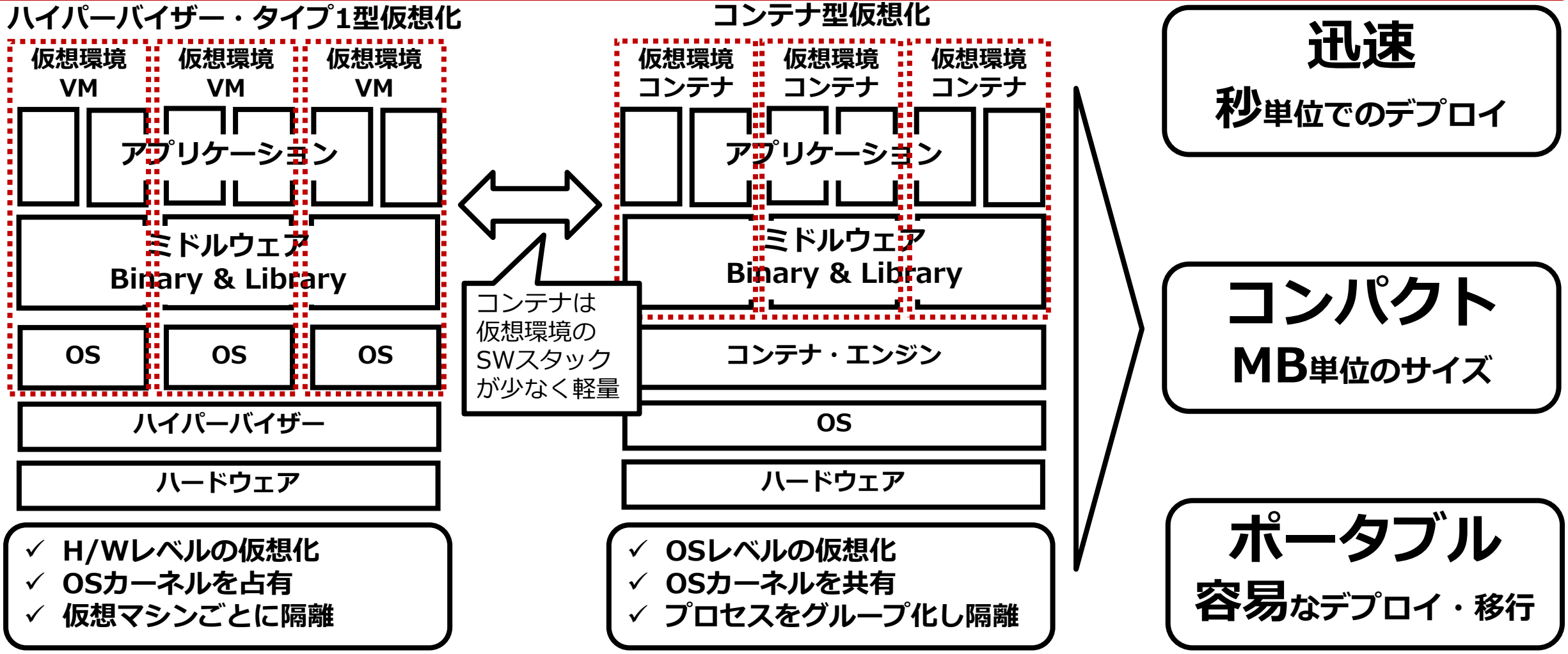
- コンテナ稼動基盤となるLinuxのファイル・システムをベースに、各コンテナ独自のファイル・システムを実現する仕組み。
- ファイル・システム・イメージをレイヤー化し、各コンテナでは差分を上位レイヤーに書き込む。



なぜコンテナなのか？

“速い”,”小さい”, “ポータブル”がコンテナ技術の強みです、しかし…

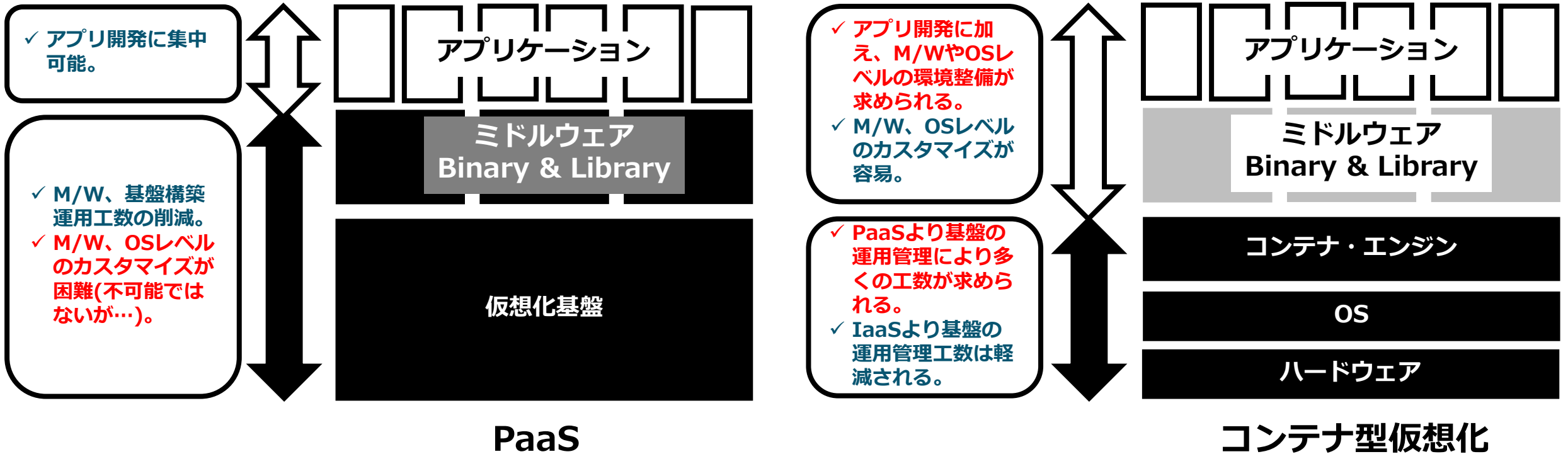
- **Cloud Foundry等PaaSも同様の特長を有します**



なぜコンテナなのか?

コンテナのエッセンスは、PaaSのシンプルさとIaaSの柔軟性のいいとこ取り

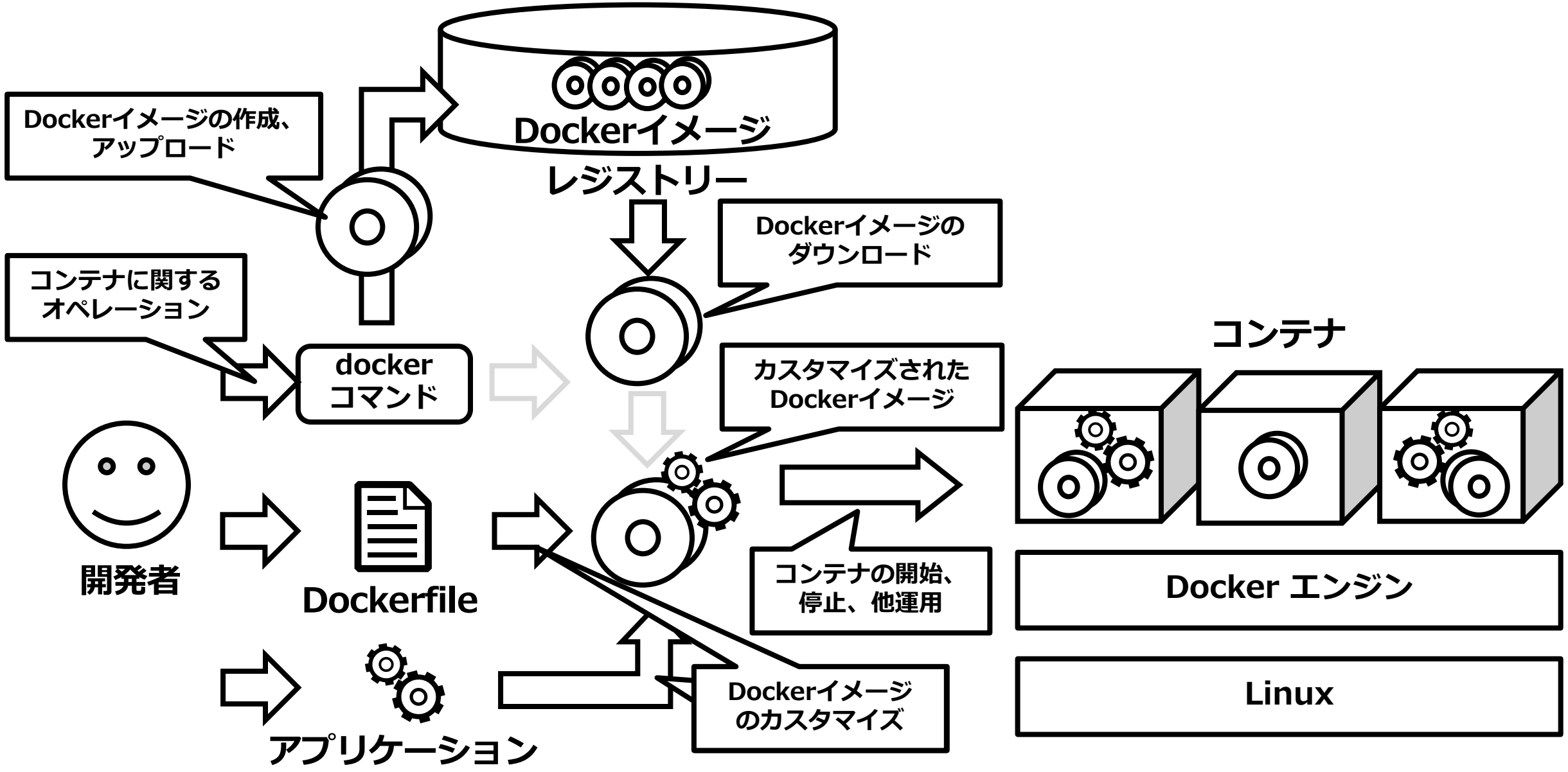
- **PaaSには無い”容易で柔軟なカスタマイズ性”がコンテナの売り**



凡例

- クラウド・サービス・カスタマーの責任範囲。
- クラウド・サービス・プロバイダーの責任範囲。
- クラウド・サービス・プロバイダー/カスタマー双方が関与する範囲。
- ✓ メリット
- ✓ デメリット

Dockerコンテナ全体像



Dockerfile

- Dockerイメージをビルドする際の仕様・要領を規定するテキスト・ファイル
  - ◆基になるDockerイメージに対して、任意のソフトウェアの導入や構成が可能
- DockerfileとDockerイメージによって、ポータビリティが実現される
- シンプルなDockerfileの例
  - ◆WebSphere Liberty (IBM のJava Web アプリケーション・サーバー) にユーザー作成のアプリケーションを導入
    - ●Dockerイメージ名：websphere-liberty
    - ●アプリケーション・パッケージ名：BookCatalog.war

Dockerイメージ”websphere-liberty”を基にする

```
FROM websphere-liberty:webProfile7
ADD BookCatalog.war /config/dropins/
```

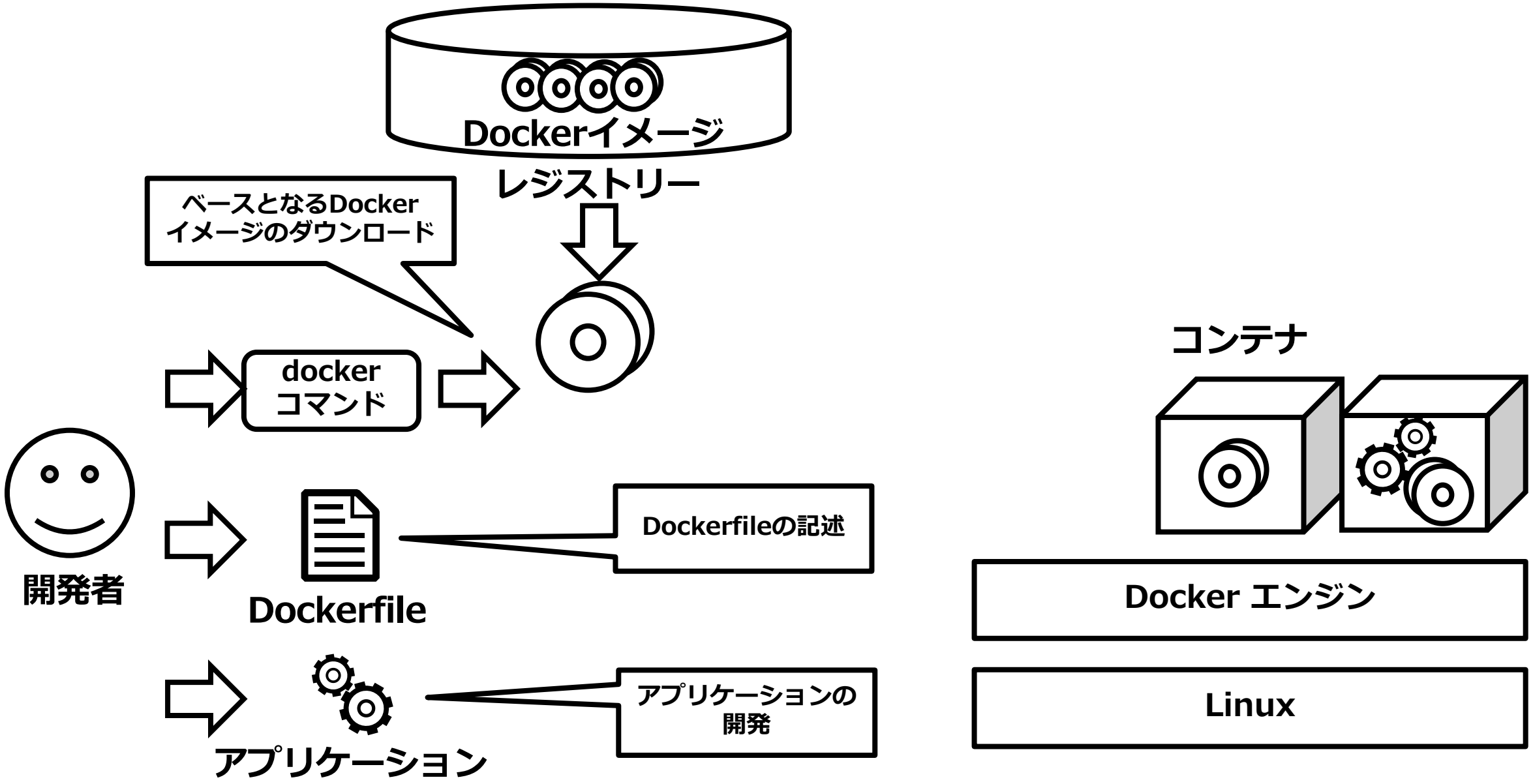
BookCatalog.warを/config/dropins/にコピーすることで、WebSphere Libertyにアプリケーションがデプロイされる

Dockerfile

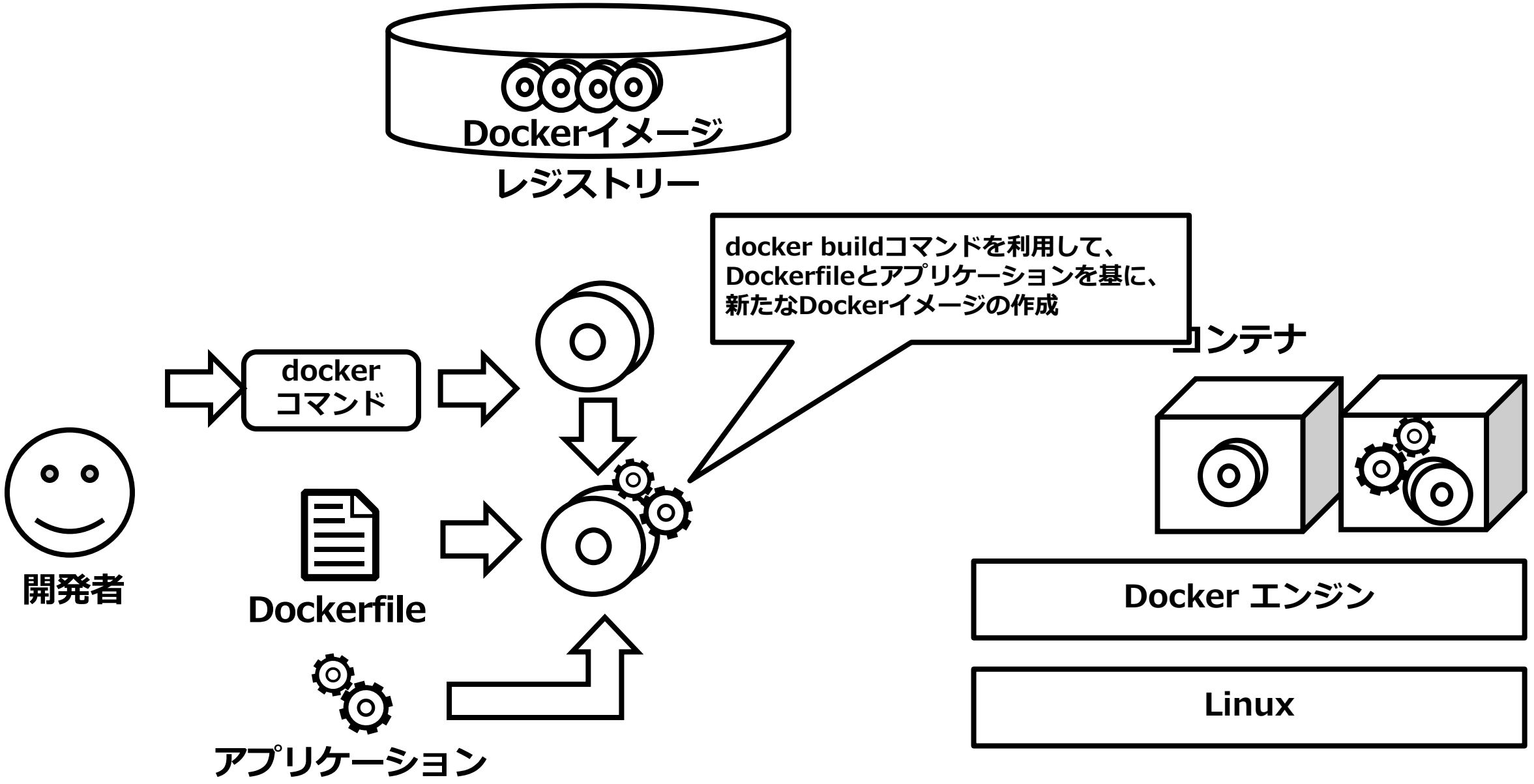
■ Dockerfileで利用できるインストラクションの例

インストラクション	要約
FROM	ベース・イメージの指定。
RUN	任意のコマンドの実行。
CMD	実行時のコンテナに対する、コマンドや引数の省略時値の設定(ビルド時には機能しない)。
LABEL	Dockerイメージに対して“キー＝値”形式でラベルを設定。
EXPOSE	Dockerコンテナがリスンするポート番号の構成。
ENV	“キー＝値”形式で環境変数を構成。
ADD	ファイル、ディレクトリー、外部サイトを、Dockerコンテナのファイル・システムに追加。
COPY	ファイル、ディレクトリーをDockerコンテナのファイル・システムにコピー。
ENTRYPOINT	Dockerコンテナが呼び出すコマンドと引数を構成(ビルド時には機能しない)。
VOLUME	外部ボリュームのマウント・ポイントの指定。
USER	ユーザー、グループの設定。
WORKDIR	作業ディレクトリーの指定。
ONBUILD	Dockerイメージのビルド完了後に実行されるインストラクションの指定。

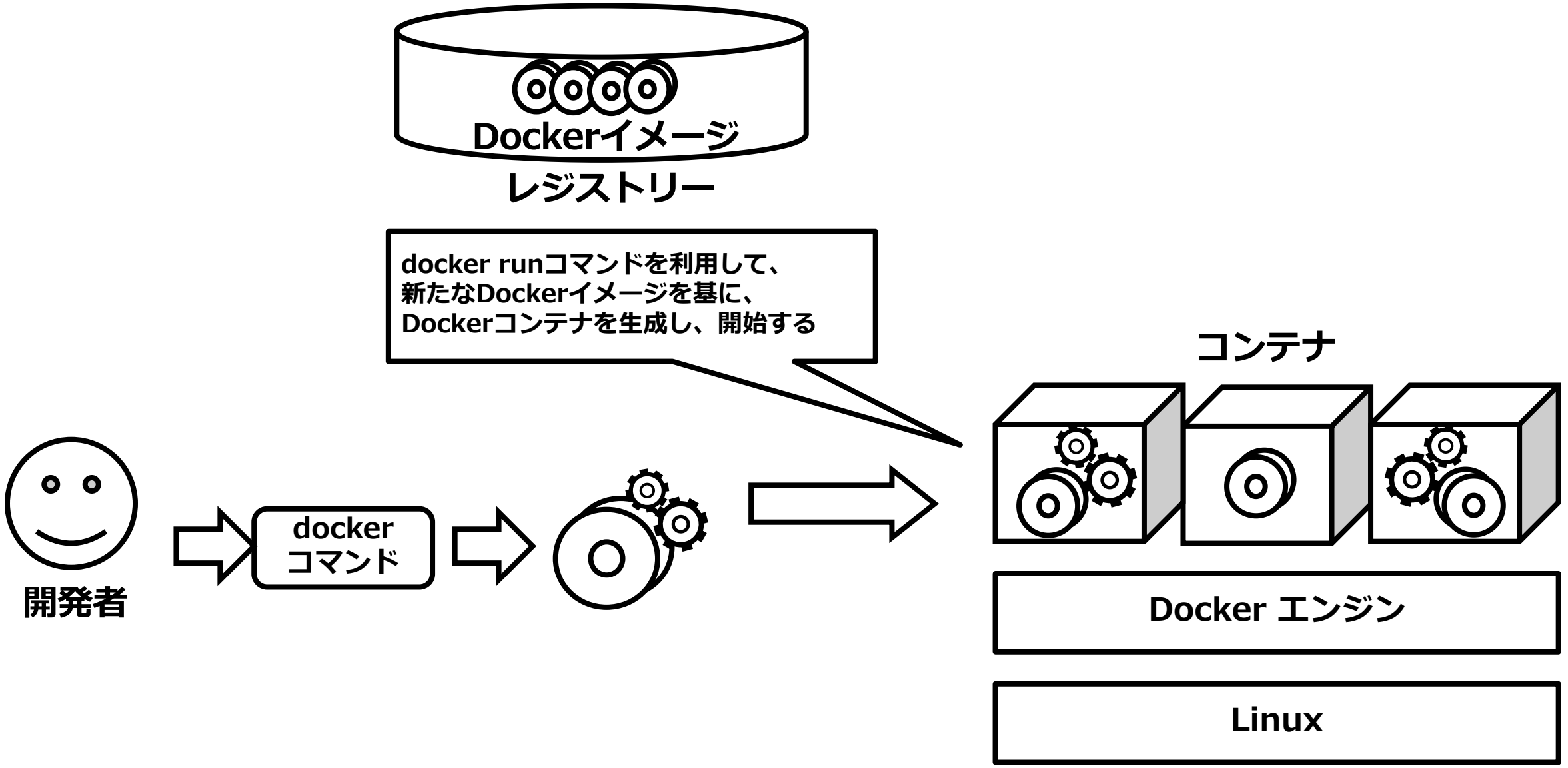
Dockerコンテナを利用したアプリケーション開発の流れ 1



Dockerコンテナを利用したアプリケーション開発の流れ 2

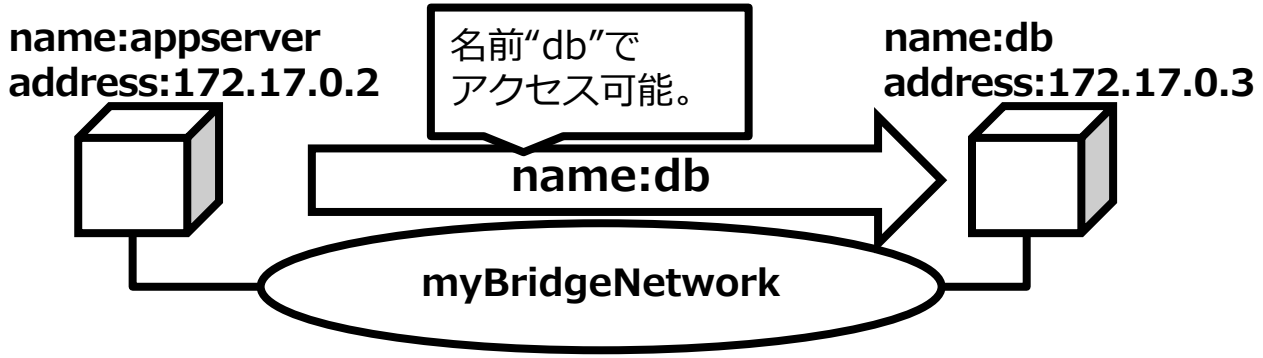


Dockerコンテナを利用したアプリケーション開発の流れ 3



Dockerコンテナ間通信

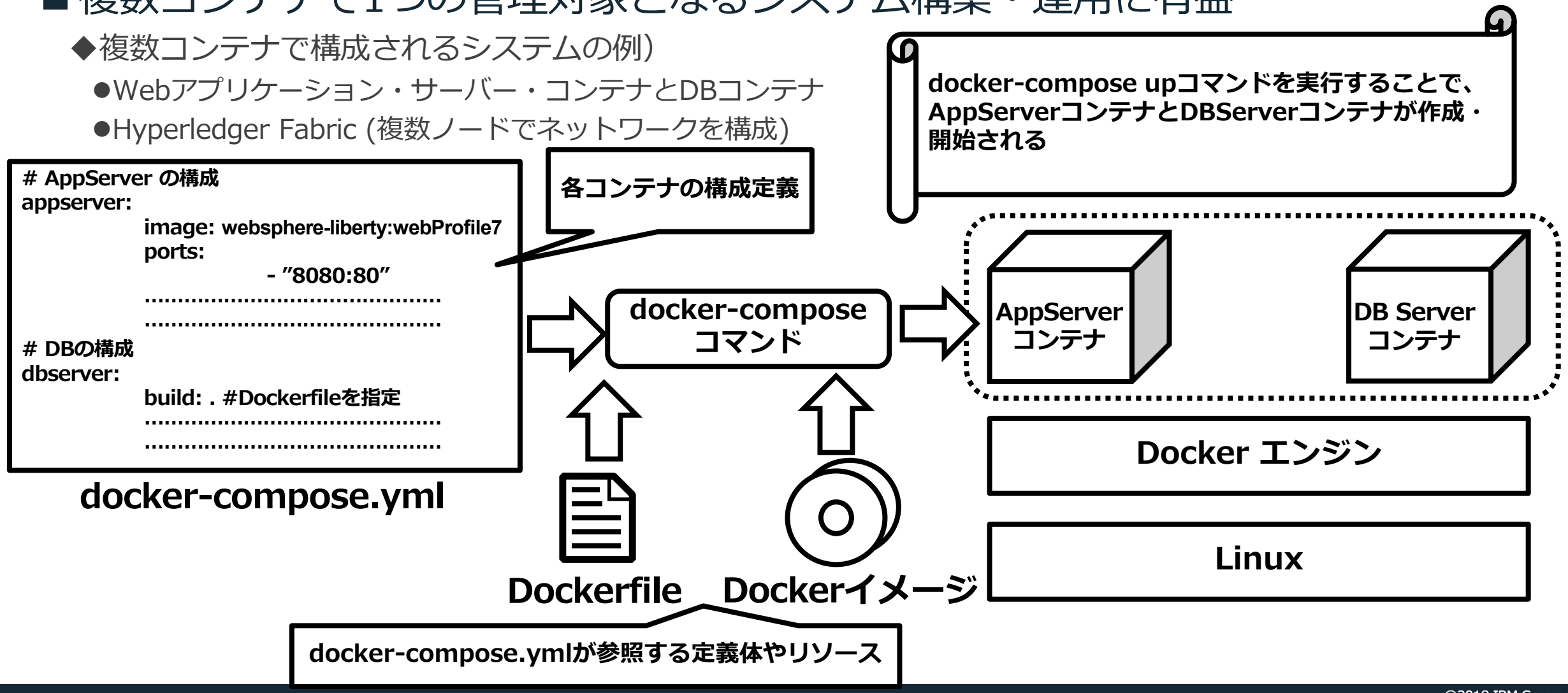
- コンテクスト：複数のコンテナ間で通信する必要がある。
  - ◆例) AppServerコンテナとDBコンテナ。
- 課題：通信相手のコンテナのアドレス解決。
  - ◆コンテナのデプロイ時に初めてコンテナのアドレスが確定する。
  - ◆アプリケーションが事前にコンテナのアドレスを知ることは出来ない。
- 解決策：Dockerネットワークの利用。
  - ◆ネットワークを作成：docker network create <networkName>
  - ◆コンテナをネットワークに参加させる：docker run --net <networkName> --name <containerName>
  - ◆コンテナ間通信では、アドレスの代替として<containerName>を利用可能となる。



注意：コンテナのリンク機能 (--linkフラグ)は非推奨(Legacy)となっています。
https://docs.docker.com/network/links/

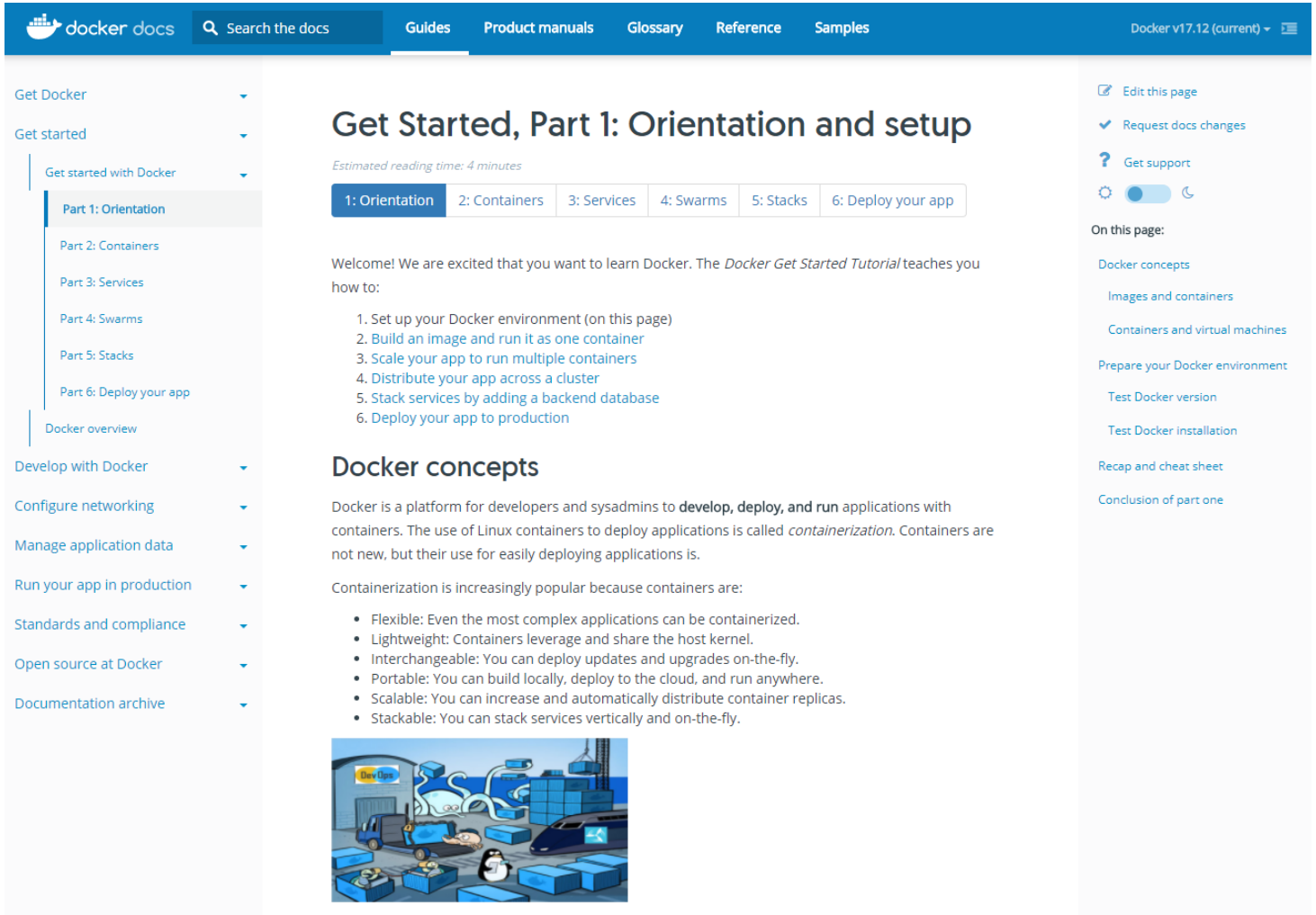
Docker Compose

- 1アクションで、複数のコンテナ環境の作成・運用を可能にするソリューション
- 複数コンテナで1つの管理対象となるシステム構築・運用に有益
  - ◆複数コンテナで構成されるシステムの例)
    - ●Webアプリケーション・サーバー・コンテナとDBコンテナ
    - ●Hyperledger Fabric (複数ノードでネットワークを構成)



体感するには

- Docker “Get started” をどうぞ
  - https://docs.docker.com/get-started/



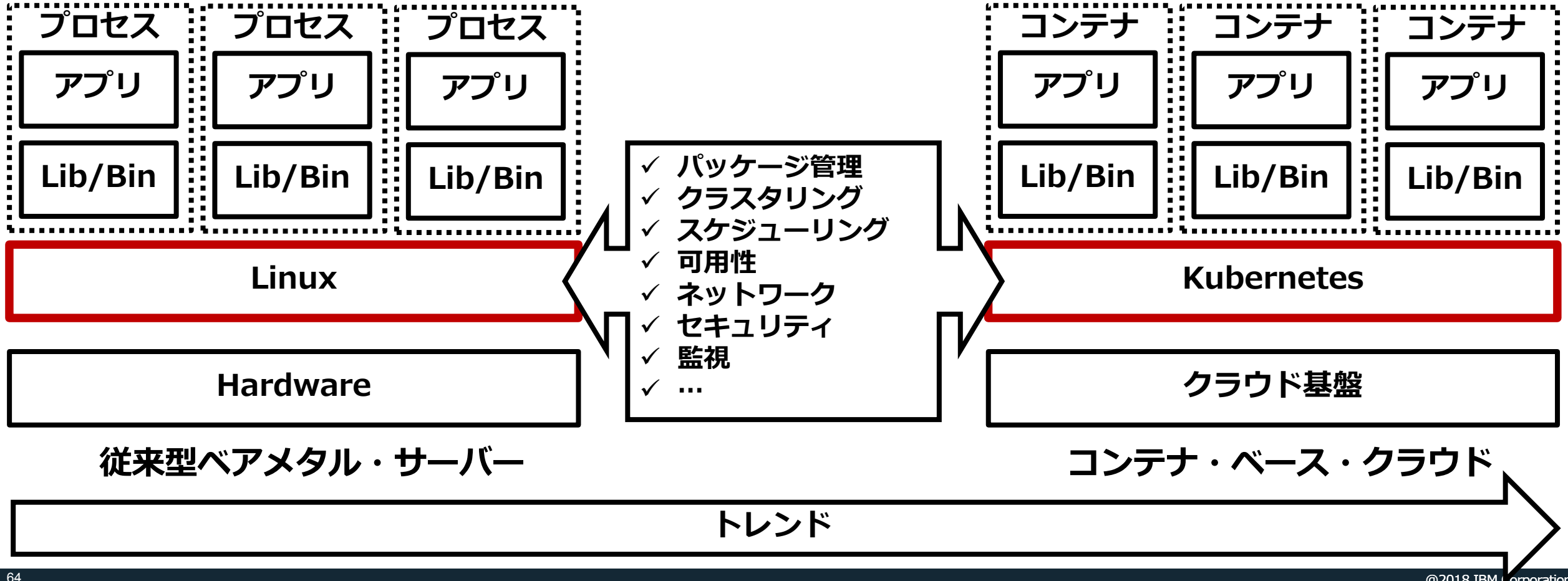
Kubernetes概要

KubernetesはクラウドにおけるLinuxになりつつある・・・

KubernetesはクラウドにおけるLinuxになりつつある・・・

クラウド・ネイティブ・アプリケーション運用に必要な基盤機能を提供するのが、Kubernetesです

- 従来型ベア・メタル・サーバーにおけるOS(例えばLinux)のように…

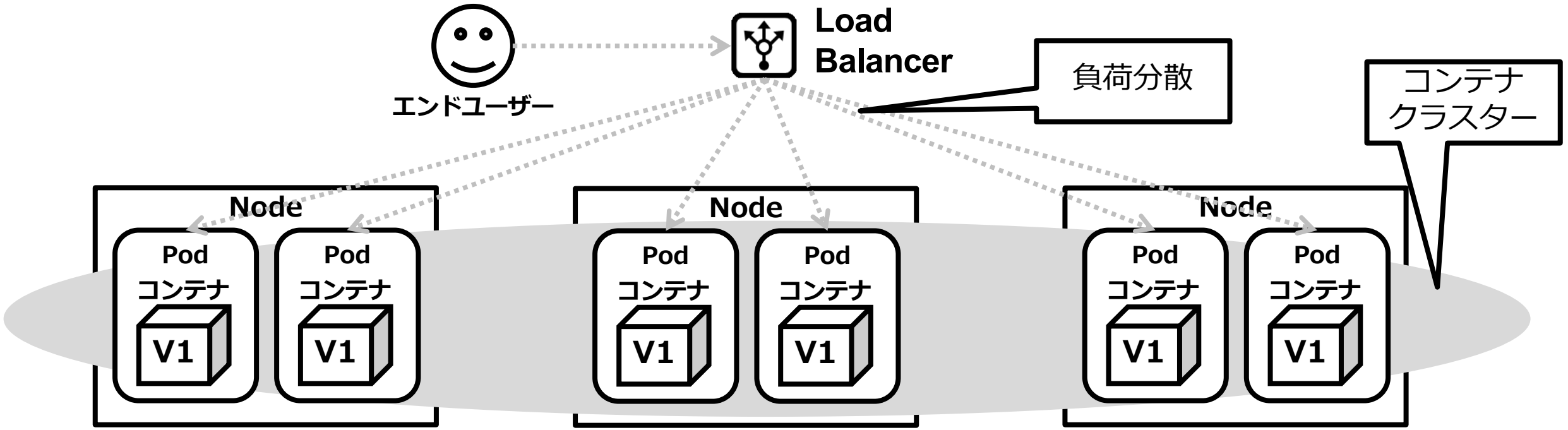


Kubernetes概要

- **“コンテナ環境オーケストレーション・ツール”**
- コンテナ化されたアプリケーションの
  自動デプロイ、スケーリング、運用管理
  のためのオープン・ソース・システム
  - ◆2000年初頭 Google,Inc. Borg Project
  - ◆2015年 CNCFよりKubernetes 1.0 リリース
  - ◆2018年2月時点 最新バージョンは1.9
- 機能
  - ◆自動化されたデプロイメント
  - ◆水平スケーラビリティ
  - ◆自動化されたロールアウトとロールバック (アプリの変更と変更の取り消し)
  - ◆ストレージ・オーケストレーション
  - ◆セルフ・ヒーリング (自己回復)
  - ◆サービス・ディスカバリーとロードバランシング
  - ◆バッチ処理

Kubernetes概要

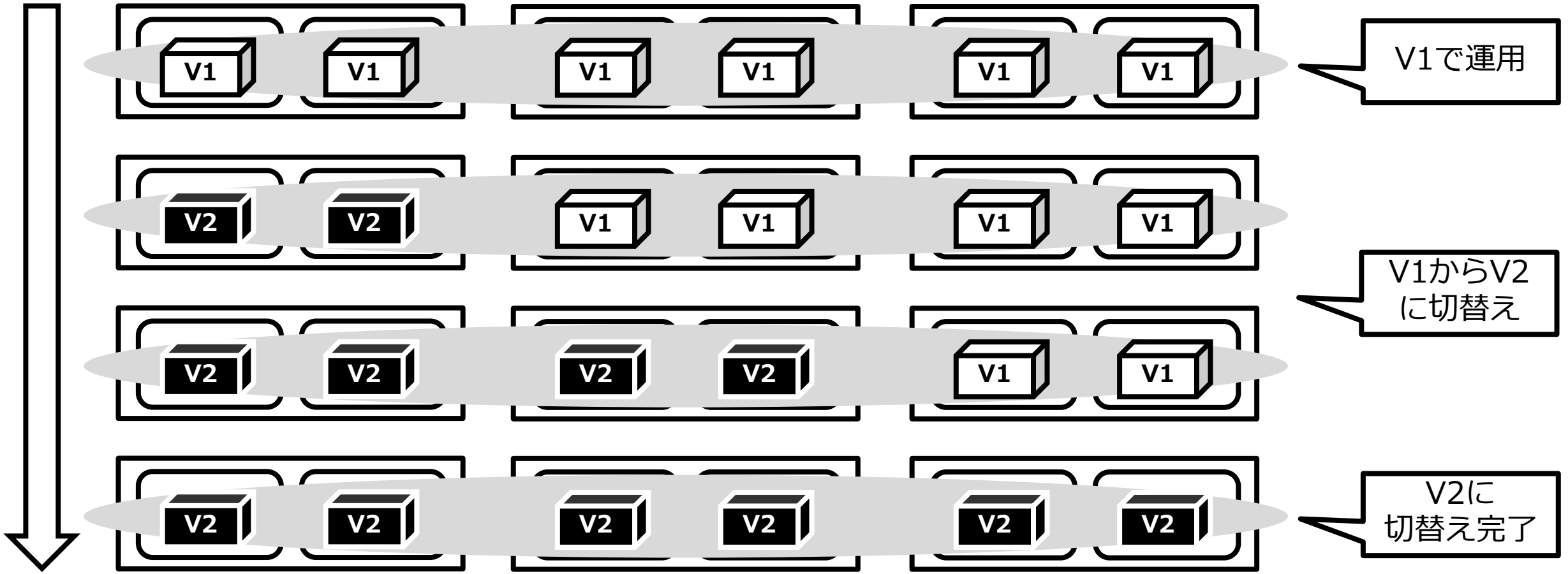
■ コンテナ・クラスターのコンセプト・モデル



水平スケーラビリティとオートスケール
✓ リクエスト量に応じたコンピューティング能力の最適化
クラスターに対する負荷分散とセルフ・ヒーリング
✓ 高い可用性の担保

Kubernetes概要

■ ローリング・アップデートのコンセプト・モデル



アプリ運用中のシームレスなアプリ・アップデートを実現

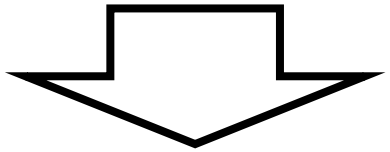
Kubernetes概要

- Kubernetesの稼働環境は多岐に渡る：
  - ◆ラップトップ, 仮想マシン, ベア・メタル・サーバー, …
- Kubernetes実装の選択肢の例
  - ◆比較的小規模のローカル・マシン向け：
    - ●Minukube
    - ●IBM Cloud Private Community Edition
    - ●…
  - ◆クラウド・サービス・プロバイダーによるソリューション：
    - ●IBM Cloud
    - ●Google Compute Engine
    - ●Azure
    - ●AWS
    - ●…
  - ◆オンプレミス・ターンキー・クラウド・ソリューション
    - ●IBM Cloud Private
    - ●Kubermatic
    - ●…

なぜKubernetesなのか？

Kubernetes provides a **container-centric** management environment. It orchestrates computing, networking, and storage infrastructure on behalf of user workloads. **This provides much of the simplicity of Platform as a Service (PaaS) with the flexibility of Infrastructure as a Service (IaaS), and enables portability across infrastructure providers.**

https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/



✓PaaSのシンプルさとIaaSの柔軟性を両立
✓基盤の違いを超えたポータビリティ

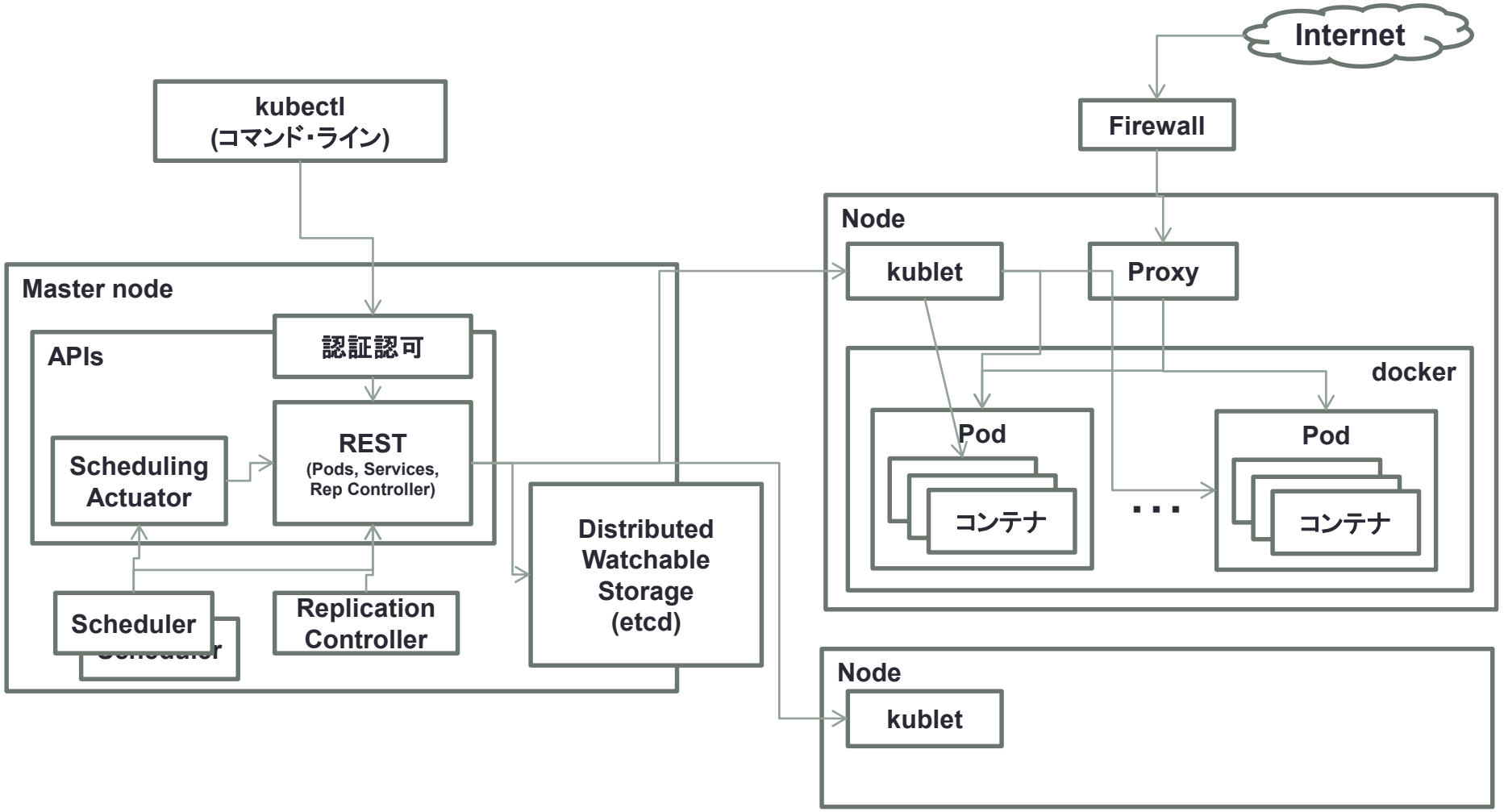
でもこれって、Kubernetesというよりは Dockerに起因するのでは…???

なぜKubernetesなのか？

優れたコミュニティ戦略

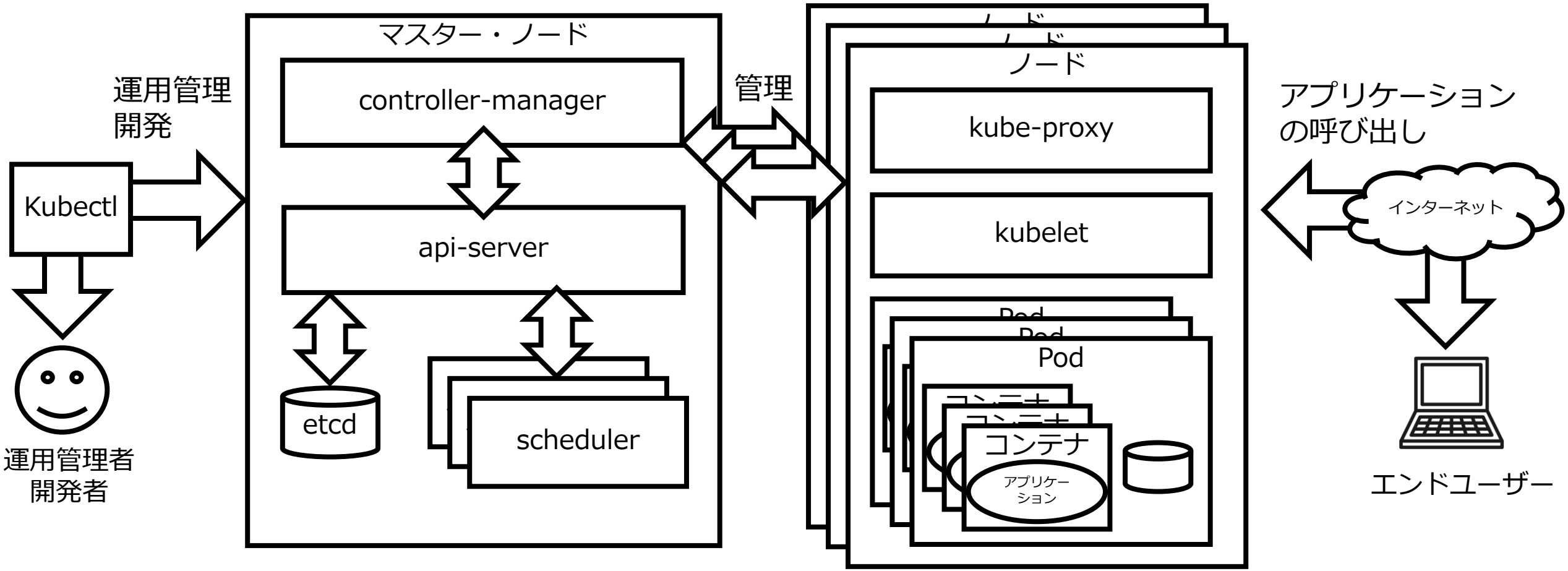
- ✓オープン・ソース
  - ⇨ベンダー・ロックイン排除
- ✓Google, Inc.における実績
  - ⇨リスク低減
- ✓Cloud Native Computing Foundation
  - ⇨マーケットにおけるマジョリティ

Kubernetes全体像



注意：
- ✓ **Kubernetes V1.0.3のアーキテクチャーを基に2015年に作成したダイアグラムです。**
- ✓ **最新の技術要素は含まれておりません。**

Kubernetes全体像

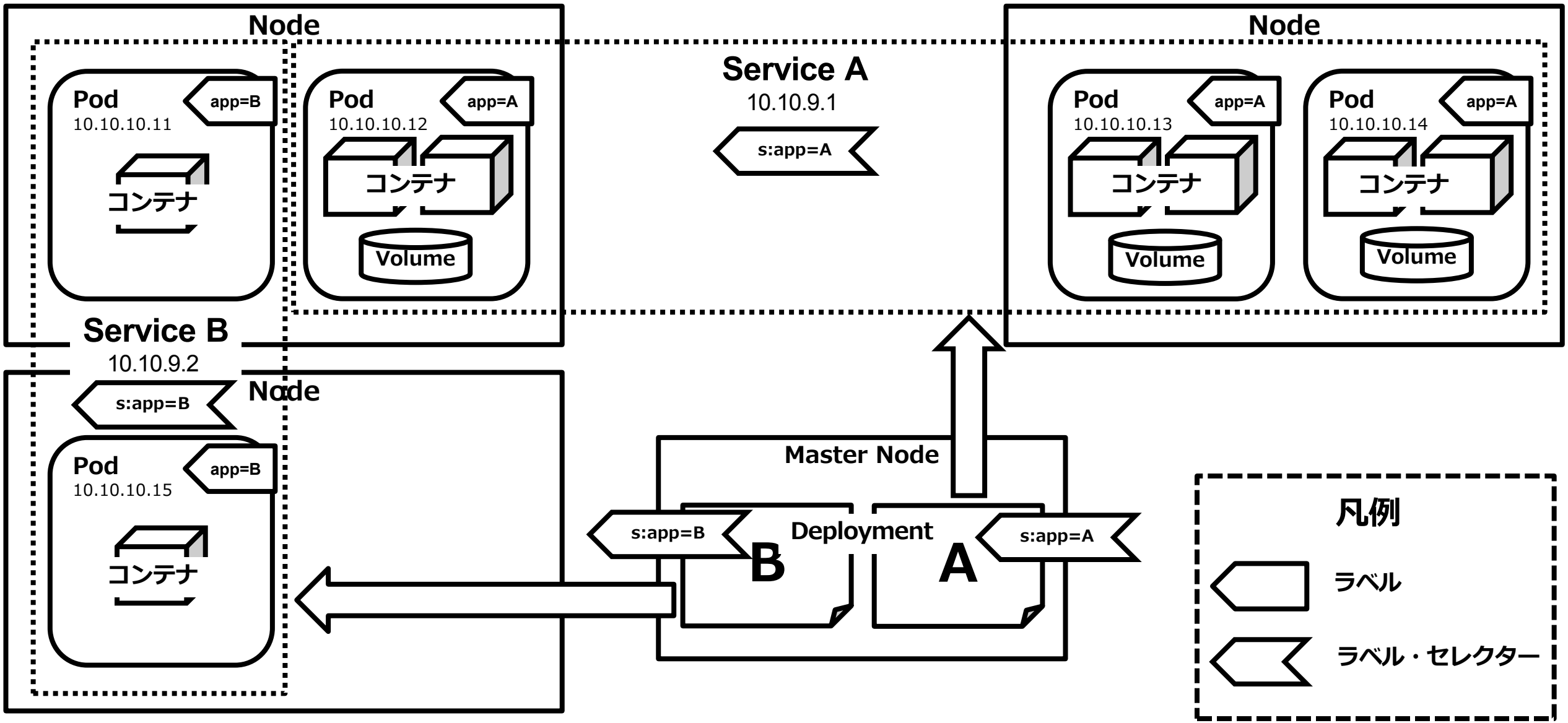


注意：
✓ **このダイアグラムは、説明の便宜上主要なコンポーネントに絞って描いています。**

Kubernetes全体像

- Kubectl：Kubernetes操作のためのコマンドライン・インターフェース。
- マスター・ノード：Kubernetes全体を運用するための管理ノード。
  - ◆kube-controller-manager：ノード, レプリカ等の各種管理作業を行う。
  - ◆api-server：Kubernetes操作のインターフェースとなるAPIサーバー。
  - ◆scheduler：新たに作成されたPodを割り当てるNodeを決定する。
  - ◆etcd :クラスターの管理データを保管するKVS。
- ノード：Pod & コンテナ・クラスターが稼動する処理ノード。
  - ◆kublet：マスター・ノードと連携し、ノードの管理作業を仲介するエージェント。
  - ◆kube-proxy：ルーティング、負荷分散を行うプロキシー・サーバー。
  - ◆Pod：Kubernetesの最小管理単位。コンテナが稼動する論理的サーバー。

Kubernetesクラスター構造のコンセプト



Kubernetesクラスター構造のコンセプト

■ Node

◆物理的あるいは仮想的なホスト・マシン。
◆Pod & コンテナ・クラスターが稼動する処理ノード。

■ Deployment

◆コンテナ・クラスター仕様を規定するオブジェクト。
◆Deploymentが規定する項目例：レプリカの数、Podのテンプレート(コンテナ、ボリューム)
◆各Pod等構成要素をLabelでまとめあげる。

■ Pod

◆Kubernetesにおける最小の管理単位。
◆1つ以上のコンテナ、0個以上のボリューム、プライベートIPアドレスを有す。
- ●関連性の高い複数のコンテナは1つのPodにデプロイすることが出来る。
  - ➢ ex. Webサーバー・コンテナとアプリケーション・サーバー・コンテナ

◆アプリケーションにとって”論理的なホスト・マシン”に相当する。
◆属性としてLabelを有す。

■ Service

◆1つ以上のPodから成るPodの論理的セットにアクセス手法を定義する概念。
◆Service仕様はYAMLまたはJSONで定義される。
◆各Pod等構成要素をLabelでまとめあげる。

Kubernetesの操作の流れ

1.コンテナ・イメージの作成 → dockerコマンド + α

コンテナ化されたアプリケーション

2.Nodeの作成・定義 → クラウド・サービス固有のUI

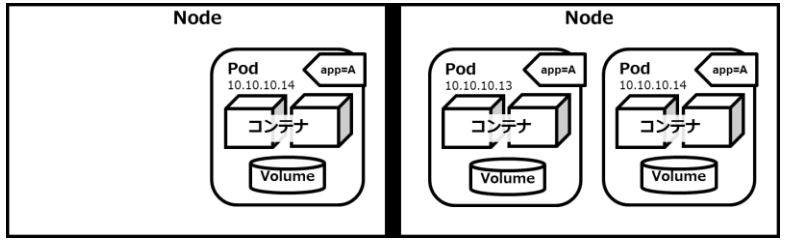
Podがデプロイされる Node

3.定義体の作成 → 任意のエディター

service.yml　deployment.yml

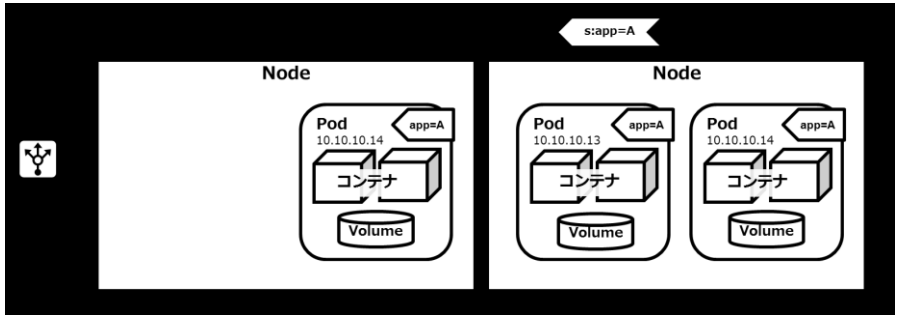
4.コンテナ・クラスターの作成 → Kubectl

deployment.yml



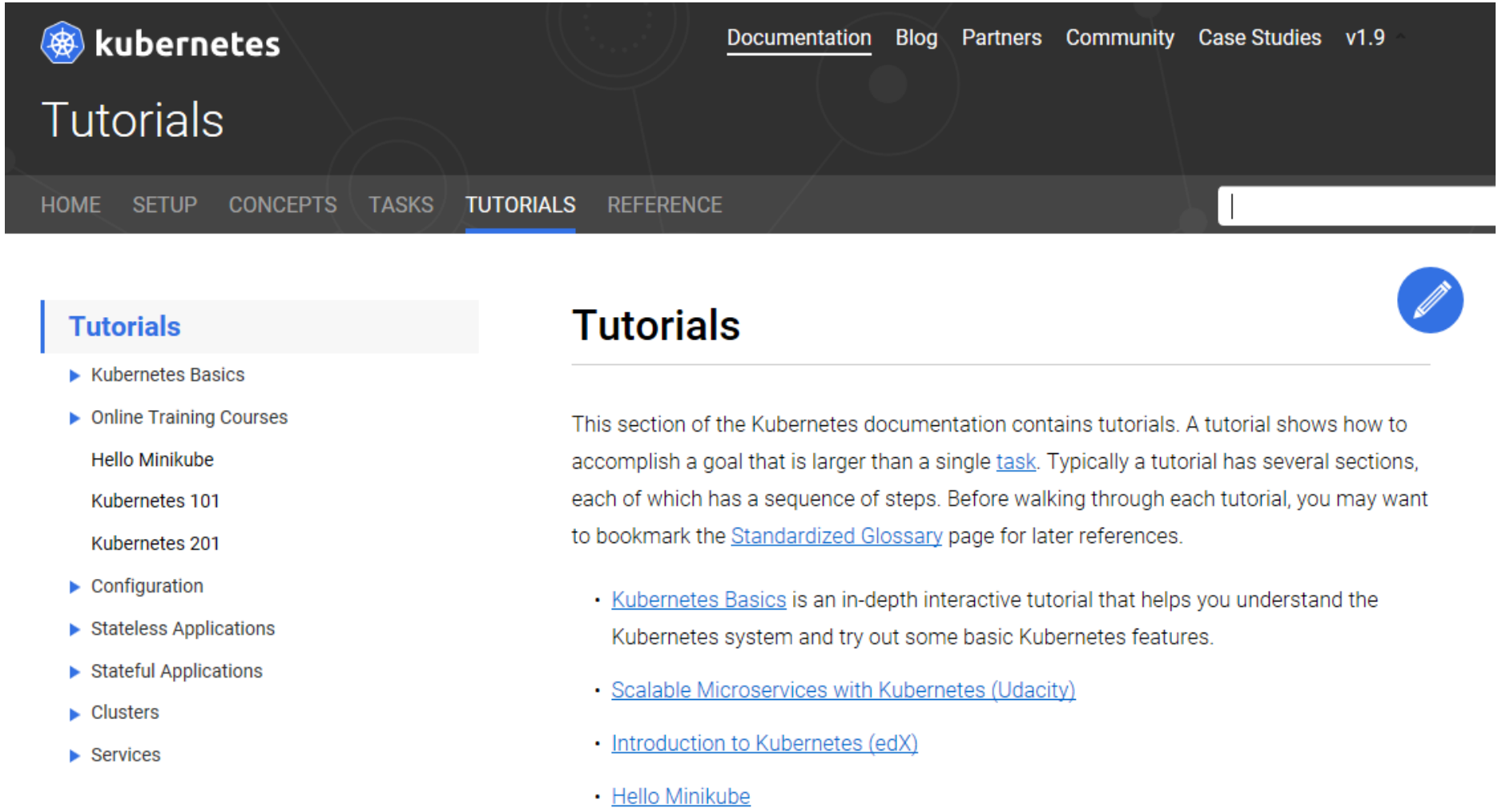
5.Serviceの作成 → Kubectl

service.yml



体感するには

- Kubernetes “Tutorials” をどうぞ
  - https://kubernetes.io/docs/tutorials/



IBM Cloud コンテナ・サービスの優位性

✓ハイブリッド・クラウド・レディ

パブリック・クラウド, シングルテナント, プライベート・クラウドで
コンテナ&Kubernetes利用可能

✓セキュア

プライベート・レジストリーの提供
Vulnerability Advisorによるコンテナ・イメージのセキュリティ・スキャン
Nodeに対して暗号化ディスクの提供

✓コンテナ化された実績あるミドルウェア

WebSphere App Server, MQ, Db2等ミッションクリティカル・システムにて
実績豊富なミドルウェアを、ハイブリッド・クラウド環境で実践可能

IBM Cloud コンテナ・サービスを始めるには

- IBM Cloud Container Service 概説
  - https://console.bluemix.net/docs/containers/container_index.html
- IBM Cloud Container Registry 概説
  - https://console.bluemix.net/docs/services/Registry/index.html
- IBM Cloud Container Service チュートリアル
  - https://console.bluemix.net/docs/containers/cs_tutorials.html

まとめ

まとめ

- デジタル・ビジネス促進のICT戦略の一つがクラウド・ネイティブ・コンピューティングの活用。
- クラウド・ネイティブ・コンピューティングとは、オープン・ソース・ソフトウェアを活かして、コンテナ、動的なオーケストレーション、マイクロサービスを実践する新たなICTのパラダイム。
- マイクロサービスは、変化に対応できる柔軟な構造を開発・運用チームとソフトウェアに与えるアーキテクチャー・スタイル。
- コンテナは、スピーディ、軽量、ポータブルな仮想化技術。
- Kubernetesは、コンテナ・クラスター運用管理のためのオーケストレーション・ツール。
- IBM Cloudは、コンテナ、Kubernetesをサポートするターンキー・ソリューション。

Ready for Cloud
Ready for Business