AWS Dev Day Tokyo 2017

～マイクロサービスを設計するすべての開発者に送る～
クラウド時代のマイクロサービス設計徹底解説！

2017/5/31

鈴木雄介

グロースエクスパートナーズ株式会社　執行役員

日本Javaユーザーグループ　会長

AWS Dev Day Tokyo 2017

マイクロサービス化設計入門

~~～マイクロサービスを設計するすべての開発者に送る～~~

~~クラウド時代のマイクロサービス設計徹底解説！~~

2017/5/31

鈴木雄介

グローバスエクスパートナーズ株式会社　執行役員

日本Javaユーザーグループ　会長

自己紹介

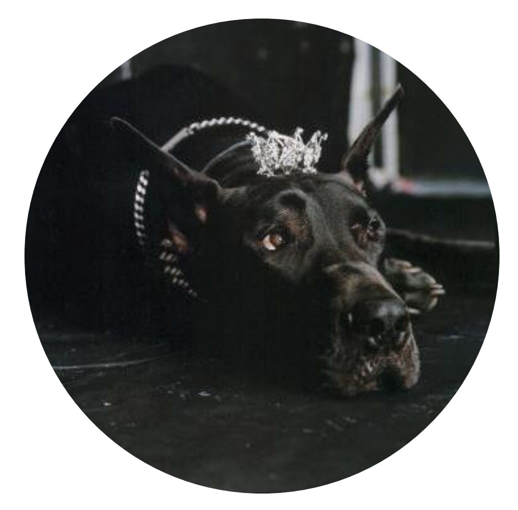
鈴木雄介

- グロースエクスパートナーズ（株）
  - 執行役員/アーキテクチャ事業本部長
  - http://www.gxp.co.jp/
- 日本Javaユーザーグループ
  - 会長
  - http://www.java-users.jp/
- SNS
  - http://arclamp.hatenablog.com/
  - @yusuke_arclamp

主な対象と前提

- 既存でモノリシックなシステムが存在
  - SoR系（モード1）とSoE系（モード2）のシステムが混在し、それぞれに連携している
- マイクロサービスにこれから取り組む/取り組み始めた
- いきなりマイクロサービス化することは技術的にも組織的にも容易ではない気がしている
- 具体的な作り方は話しません
  - アイコンのみ付けてます

アジェンダ



- なぜマイクロサービスか
- マイクロサービス化設計
  - サービス分割
  - メッセージング
  - 無停止デプロイ
  - レジリエンス
  - テスト戦略 など
- まとめ

なぜマイクロサービスか

なぜマイクロサービスか

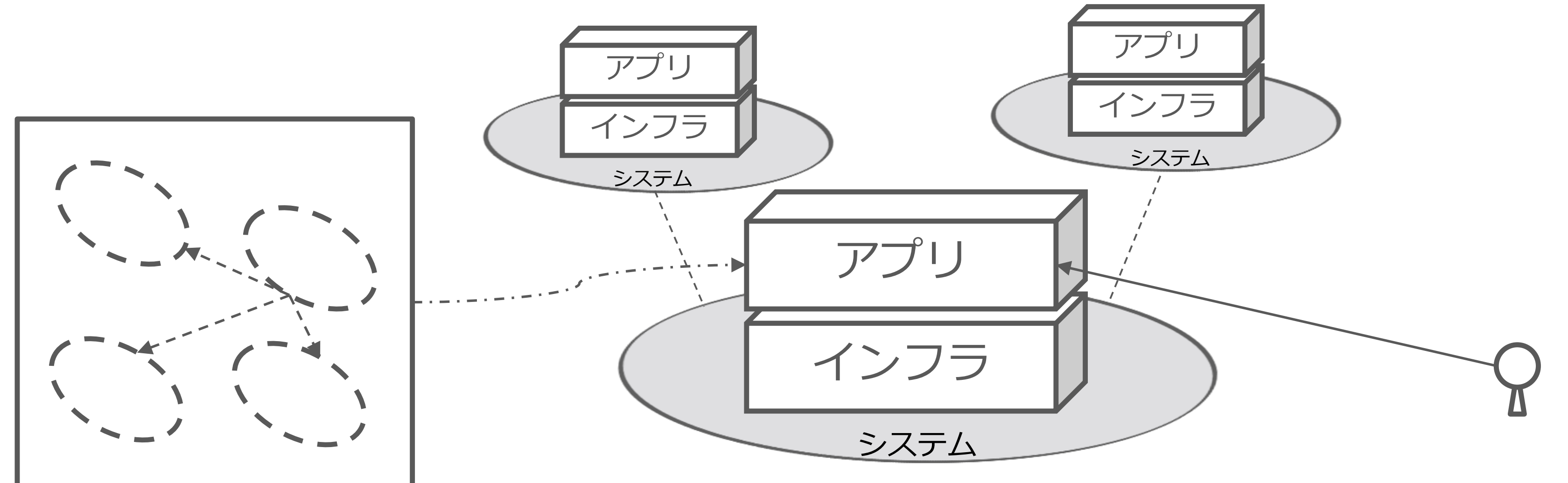
ウェブサービスの状況

- システム全体は巨大化していく
  - システム全体の中には様々な機能（SoR/SoE）がある
  - それぞれが独自のライフサイクルを持っている
- でも、ビジネス環境の変化には素早く適応したい
  - 機能やリソースの変更を素早く、無停止で行いたい

従来のアプローチ

モノリシックにおける戦略

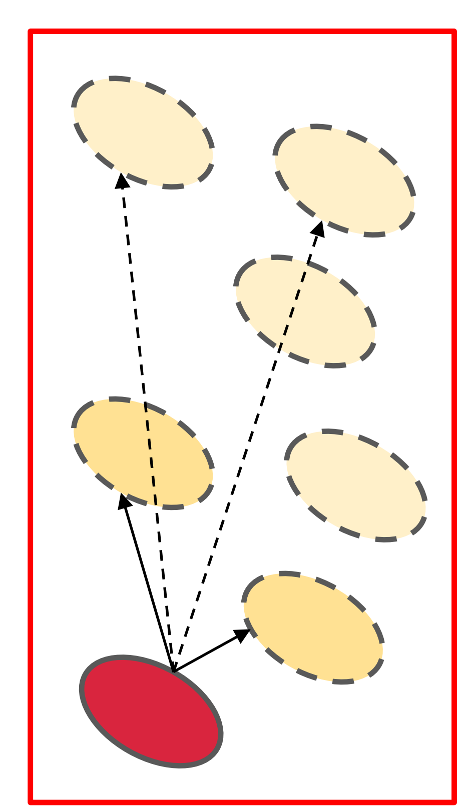
- アプリ内の適切な機能分割設計
  - » アプリ内を分割してコード変更の影響範囲を限定する
    - ▸ クラス設計パターン、マルチティア、UI/Logic/DAOなど
  - » そのアプリを1つのノードとして配備し、それぞれ連携



従来のアプローチ

モノリシックの弊害

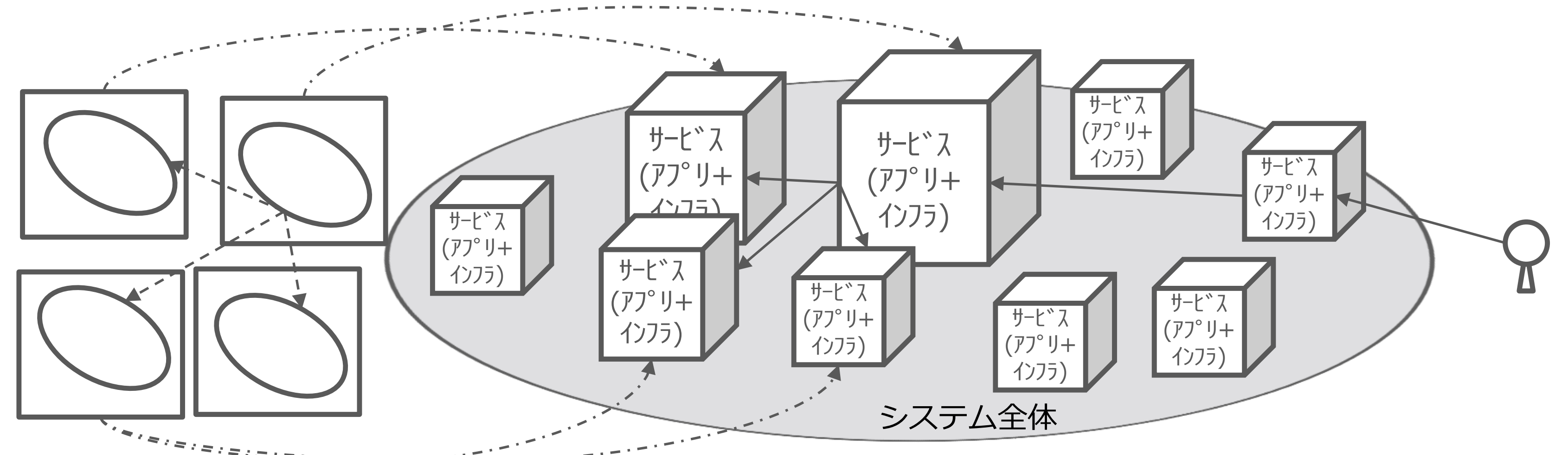
- 実行時の影響範囲が見えにくい
  - 設計時の分割だけでは実行時の依存関係を把握することは困難
  - 部分への性能劣化が全体に波及しやすい
  - 影響調査とリグレッションテストに工数を消費
- 巨大化すると難易度があがる
  - 個別変更でも全体調整
  - 部分変更でもシステム全体をリリース



マイクロサービスのアプローチ

サービス化による戦略

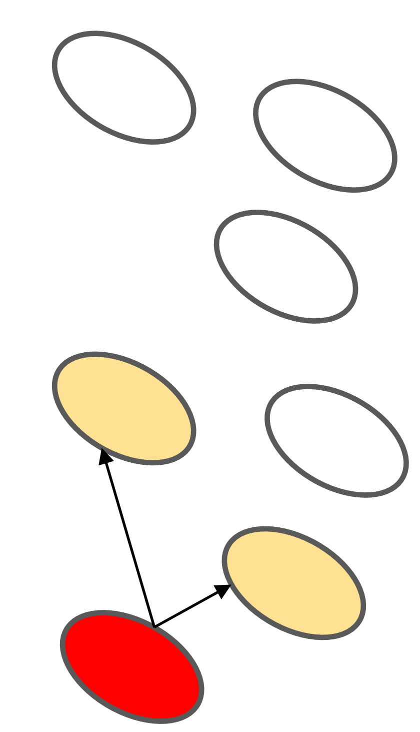
- アプリを機能ごとにサービスで分割していく
  - » サービス＝稼働状態にあるアプリケーション
    - ▸ 機能だけではなく非機能も切り分けられている
  - » サービスは個別ノードに配備し、ゆるやかに連携している



マイクロサービスのアプローチ

マイクロサービスアーキテクチャ

- 実行時の影響範囲が限定しやすい
  - APIを通じたサービス同士の依存関係が明確
  - サービスが独立しており、他とは疎結合
- 巨大化してもサービス単位で管理
  - サービス単位に好きなタイミングで仕様変更、リソース変更ができる
    - 「速さ」ではなく「独立性」が重要



なぜマイクロサービスか

マイクロサービス化の目的と手段

- システム全体が巨大になろうとも、機能別に好きなタイミングで機能やリソースの変更を行いたい
- そこで、機能をサービス化してノード分割し、それらを管理することにした
  - クラウドによる仮想化技術の普及が管理ノードの増加を後押し
  - 自動化によりインフラ構築コストを0にすることができる

マイクロサービス化設計

マイクロサービス化設計

マイクロサービス化設計における考慮点

- サービス分割
- 横断的関心事の分離
- データの分離
- メッセージング
- 無停止デプロイ
- レジリエンス
- テスト戦略
- チーム体制

マイクロサービス化設計

サービス分割

サービス分割

サービスの分割はどうあるべきか？

- 増：細かく分割するほど機能単位での再利用性が高まる
- 減：ノード間連携のオーバーヘッドは大きく、管理も難
  - » 分散コンピューティングの限界

- よって、適切な大きさ/数に分割する必要がある
  - » 先進的な企業で数百サービス×マルチノード＝数千ノード

サービス分割

サービスの粒度は「サービスによる」

- ナノ： 0-3人月程度の単機能なAPI
- マイクロ：3-20人月程度で複数のAPIを提供

- 既存のシステム単位に近いサービス粒度
  - ミニ：20-100人月程度。3-6人でメンテ
  - 中規模：100-300人月程度。7-10人がかりでメンテ
  - 大規模：中規模以上

サービス分割

サービスはドメイン単位にする？

- ドメインとは？
  - ドメイン≒業務（利用者視点での関心事のカタマリ）
  - 商品登録ドメイン、発注ドメイン、物流ドメイン…
- ドメインはボトムアップで発見すべき
  - 実践してフィードバックによって学んでいくしかない
    - 例：コードにして検証して改善する
  - ただ、それだけだと辛い

サービス分割

サービス観点で分割する

- 業務や利用の観点で変更が発生する要因をくくる
  - 「変更要因」のタイミングや理由が同じモノをくくる
  - 同時に変更が発生するものは一緒にしておけばいい
- 作り手の観点ではないことに注意が必要
  - 従来のモノリシックにおける分割とは観点が異なる
    - 例：UI/ロジック/DAOのような技術特性での分割
    - もちろん、サービス内部は作り手の観点でよい
  - ただし、技術的な限界は存在する

サービス分割

変更発生要因が分離しうる要素

No	要因	例
1	ユーザー種	コンシューマーか、社内部門か
2	目的	探したいか、注文したいか
3	利用プロセス	個別処理か、一括処理か
4	利用状況	PC利用か、モバイル利用か
5	ユーザー思考	容易にか、細かくか
6	ビジネス要件	商品登録か、物流か

- フロントエンド(SPA)とバックエンドの分離などはユーザー種が原因と考えることができる

サービス分割

いかに技術制約を超えていくか

- サービス分割は「分けれるものを分ける」だけ
  - 変更の発生要因の違いは常に分割点になりうる
- 問題は分割時の技術制約をどうするか
  - ここから後の話の

マイクロサービス化設計

横断的関心事の分離

横断的関心事の分離

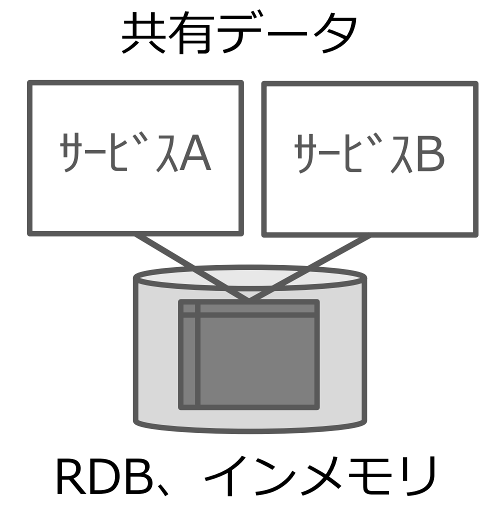
「横断的機能」の分離

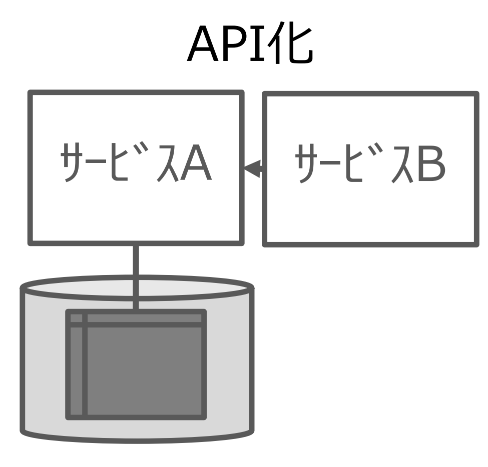
- わかりやすい例は認証認可、ログ、設定管理
  - 複数のサービスで共有する機能
  - 複数のサービスを俯瞰的に見る機能
- サービス分割のために横断的関心事の早期分離が必要
  - いわゆる「プラットフォーム」の機能を拡充させる

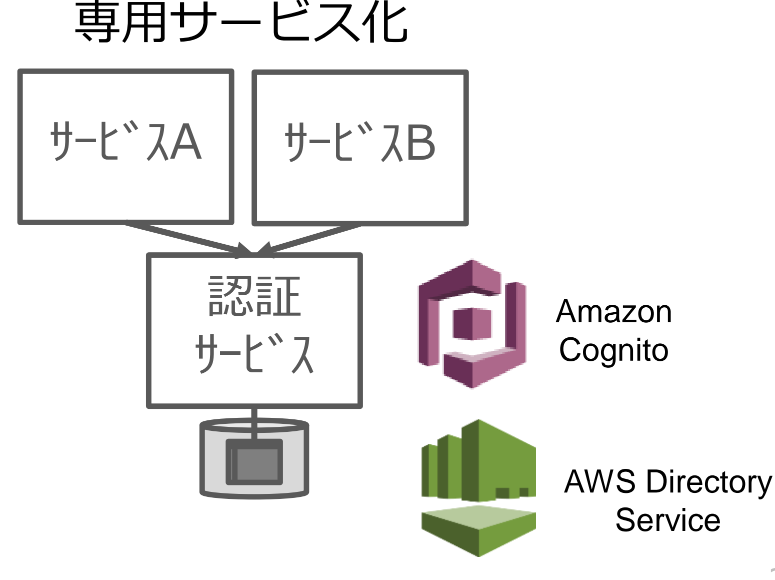
認証認可

認証機能な段階的な移設

- 共有データ：AとBの機能仕様が共有される必要がある
- API化：サービスAにサービスBの都合が含まれる
- 専用サービス化：横断的関心事として切り出される



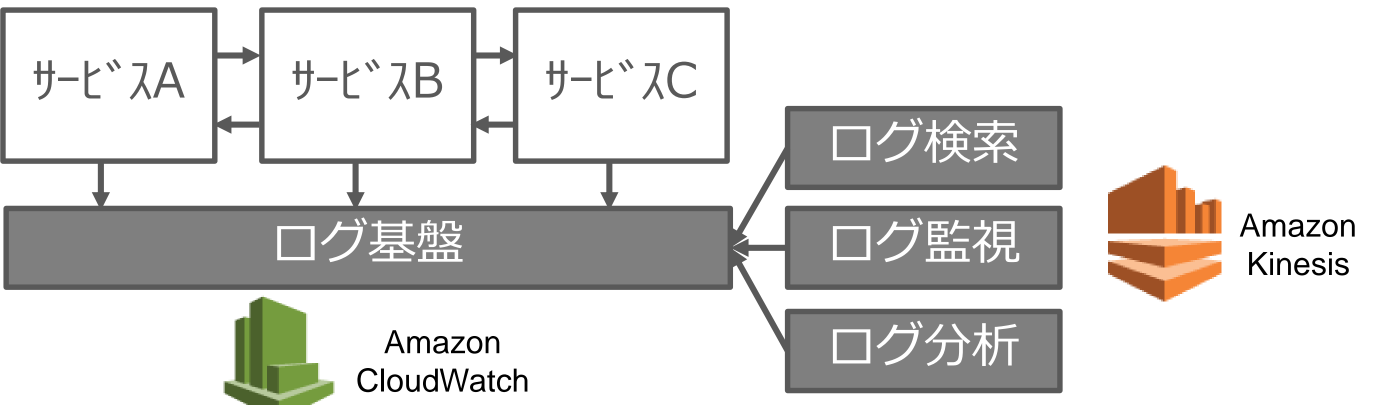




ログ基盤

リクエストの分散トレース

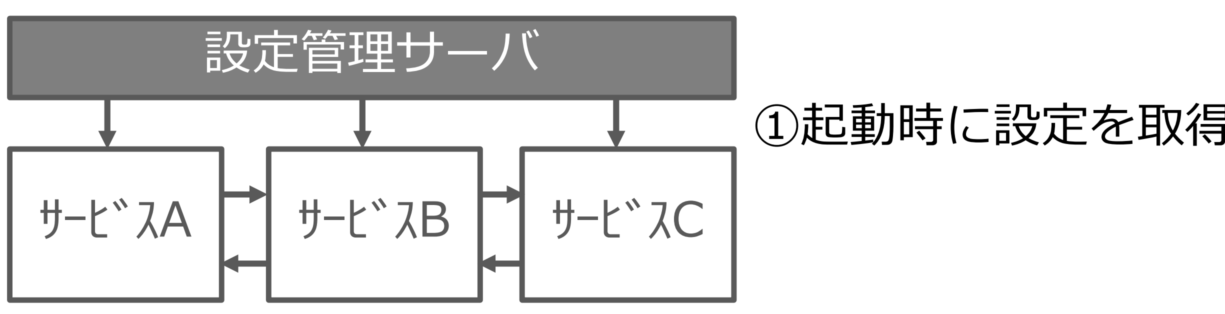
- サービスをまたがったトレーサビリティの確保
  - リクエストやサービス間通信にIDを発行する
  - 各サービスからはログ基盤にログを送信する
- 監視、分析
  - さらにログをベースにした異常検知、アクセス/メトリクス分析



設定管理

設計情報の集中管理

- 環境に依存する設定情報を集中管理する
  - 各サービス内に設定ファイルを配置するのではなく、ネットワーク経由で取得する
  - Spring Cloud Config



メニュー | amazon web services | 基調講演ライブ中継 | 無料利用枠 | 製品 | ソリューション | 詳細 | 日本語 | アカウント | コンソールへサインイン »

コンピューティング
- Amazon EC2
- Amazon EC2 Container Registry
- Amazon EC2 Container Service
- Amazon Lightsail
- Amazon VPC
- AWS Batch
- AWS Elastic Beanstalk
- AWS Lambda
- Auto Scaling
- Elastic Load Balancing

ストレージ
- Amazon Simple Storage Service (S3)
- Amazon Elastic Block Storage (EBS)
- Amazon Elastic File System (EFS)
- Amazon Glacier
- AWS Storage Gateway
- AWS Snowball
- AWS Snowball Edge
- AWS Snowmobile

データベース
- Amazon Aurora
- Amazon RDS
- Amazon DynamoDB
- Amazon DynamoDB Accelerator (DAX)
- Amazon ElastiCache
- Amazon Redshift
- AWS Database Migration Service

移行
- AWS Application Discovery Service
- AWS Database Migration Service
- AWS Server Migration Service
- AWS Snowball
- AWS Snowball Edge
- AWS Snowmobile

ネットワークとコンテンツ配信
- Amazon VPC
- Amazon CloudFront
- Amazon Route 53
- AWS Direct Connect
- Elastic Load Balancing

開発者用ツール
- AWS CodeStar
- AWS CodeCommit
- AWS CodeBuild
- AWS CodeDeploy
- AWS CodePipeline
- AWS X-Ray
- AWS コマンドラインインターフェイス

管理ツール
- Amazon CloudWatch
- Amazon EC2 Systems Manager
- AWS CloudFormation
- AWS CloudTrail
- AWS Config
- AWS OpsWorks
- AWS Service Catalog
- AWS Trusted Advisor
- AWS Personal Health Dashboard
- AWS コマンドラインインターフェイス
- AWS マネジメントコンソール
- AWS マネージドサービス

人工知能
- Amazon Lex
- Amazon Polly
- Amazon Rekognition
- Amazon Machine Learning

分析
- Amazon Athena
- Amazon EMR
- Amazon CloudSearch
- Amazon Elasticsearch Service
- Amazon Kinesis
- Amazon Redshift
- Amazon QuickSight
- AWS Data Pipeline
- AWS Glue

セキュリティ、アイデンティティ、コンプライアンス
- AWS Identity and Access Management (IAM)
- Amazon Inspector
- AWS Certificate Manager
- AWS CloudHSM
- AWS Directory Service
- Amazon Cloud Directory
- AWS Key Management Service
- AWS Organizations
- AWS Shield
- AWS WAF
- AWS Artifact

モバイルサービス
- AWS Mobile Hub
- Amazon API Gateway
- Amazon Cognito
- Amazon Pinpoint
- AWS Device Farm
- AWS Mobile SDK

アプリケーションサービス
- AWS Step Functions
- Amazon API Gateway
- Amazon Elastic Transcoder
- Amazon AppStream

メッセージング
- Amazon Simple Queue Service (SQS)
- Amazon Simple Notification Service (SNS)
- Amazon Pinpoint
- Amazon Simple Email Service (SES)

ビジネスの生産性
- Amazon Chime
- Amazon WorkDocs
- Amazon WorkMail

デスクトップとアプリケーションのストリーミング
- Amazon WorkSpaces
- Amazon AppStream 2.0

ソフトウェア
- AWS Marketplace

IoT
- AWS IoT プラットフォーム
- AWS Greengrass
- AWS IoT ボタン

コンタクトセンター
- Amazon Connect

ゲーム開発
- Amazon GameLift
- Amazon Lumberyard

マイクロサービス化設計

データの分離

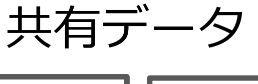
データの分離

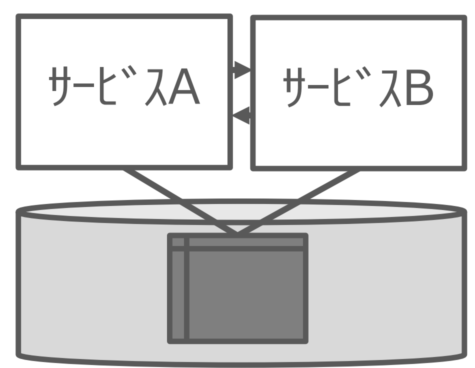
データは各サービスが管理する

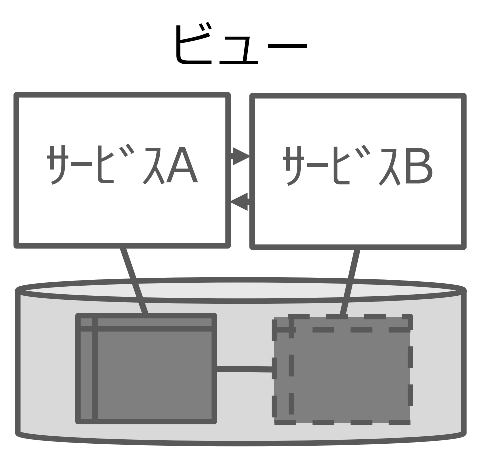
- データをサービス間で共有することは避けるべき
  - » テーブル構造の変更が強制的に同期されるから
- とはいえ、データの特性や種類に応じて方式を選択
  - » RDBの機能は有効活用。ただし、2フェーズコミットを避ける
  - » 不整合の許容範囲を決めることが重要
    - ▸ 許容できないなら疎結合化は困難になる
  - » メッセージングの仕組みとの組み合わせで検討

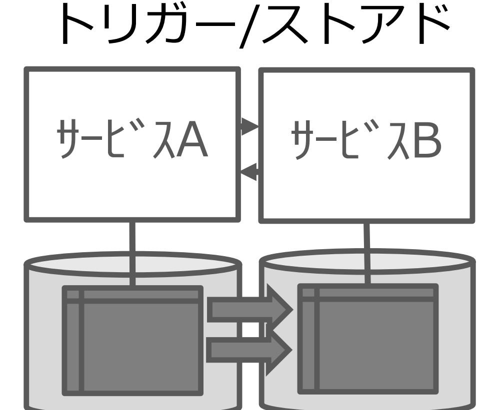
データ分離

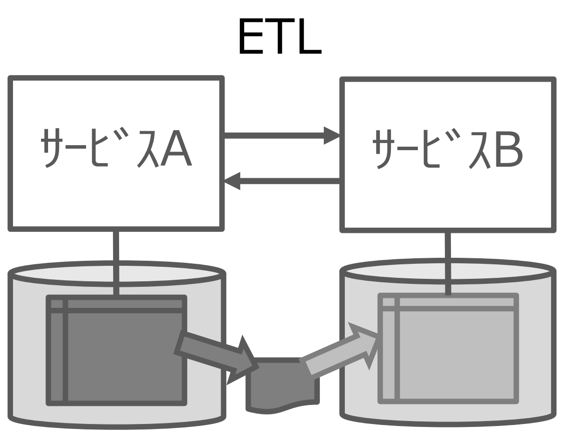
- 共有データ：テーブルを共有する。変更同期が必要
- ビュー：参照公開前提だが、変更同期が必要
- トリガー/ストアド：変更同期は不要。非機能同期が必要
- ETL：全て非同期になる





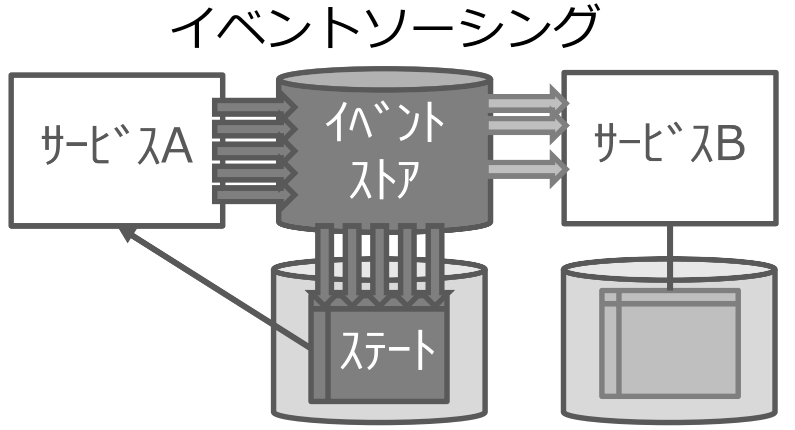






AWS
Glue

- イベントソーシング
  - データのステートではなくデータへのイベントを共有する
    - 非同期ではあるが、時間的ズレを小さくできる
  - CQRS/コマンドクエリ責務分離
    - コマンド：更新などのロジックを含む処理
    - クエリ：検索などの単純な処理



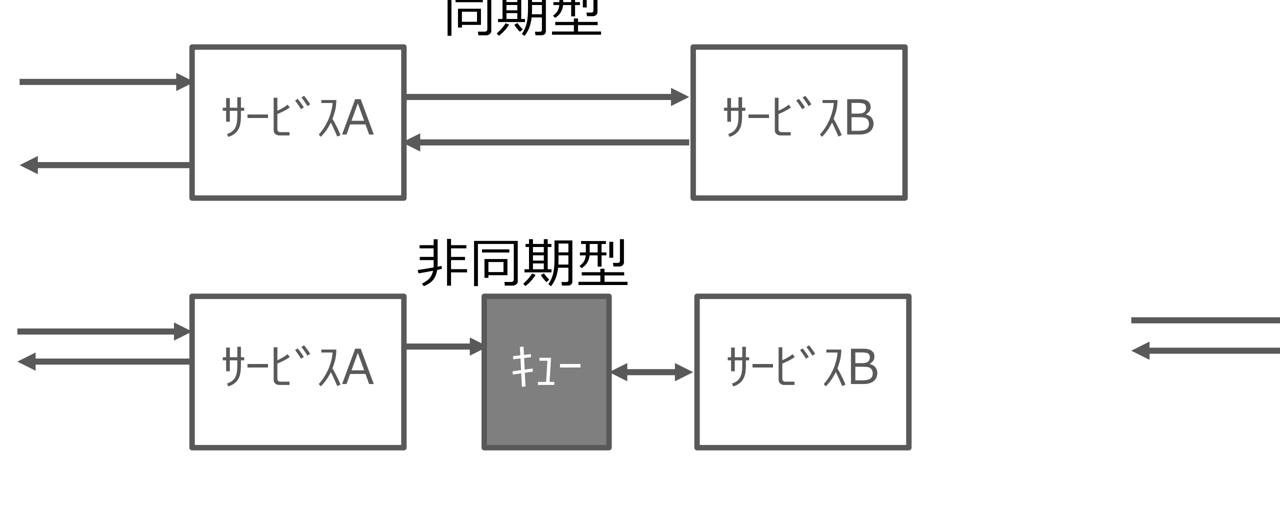
マイクロサービス化設計

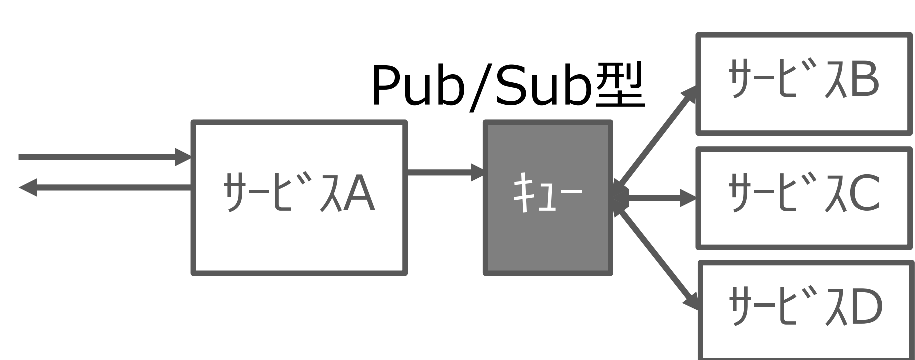
メッセージング

メッセージング

サービス同士を連携させる

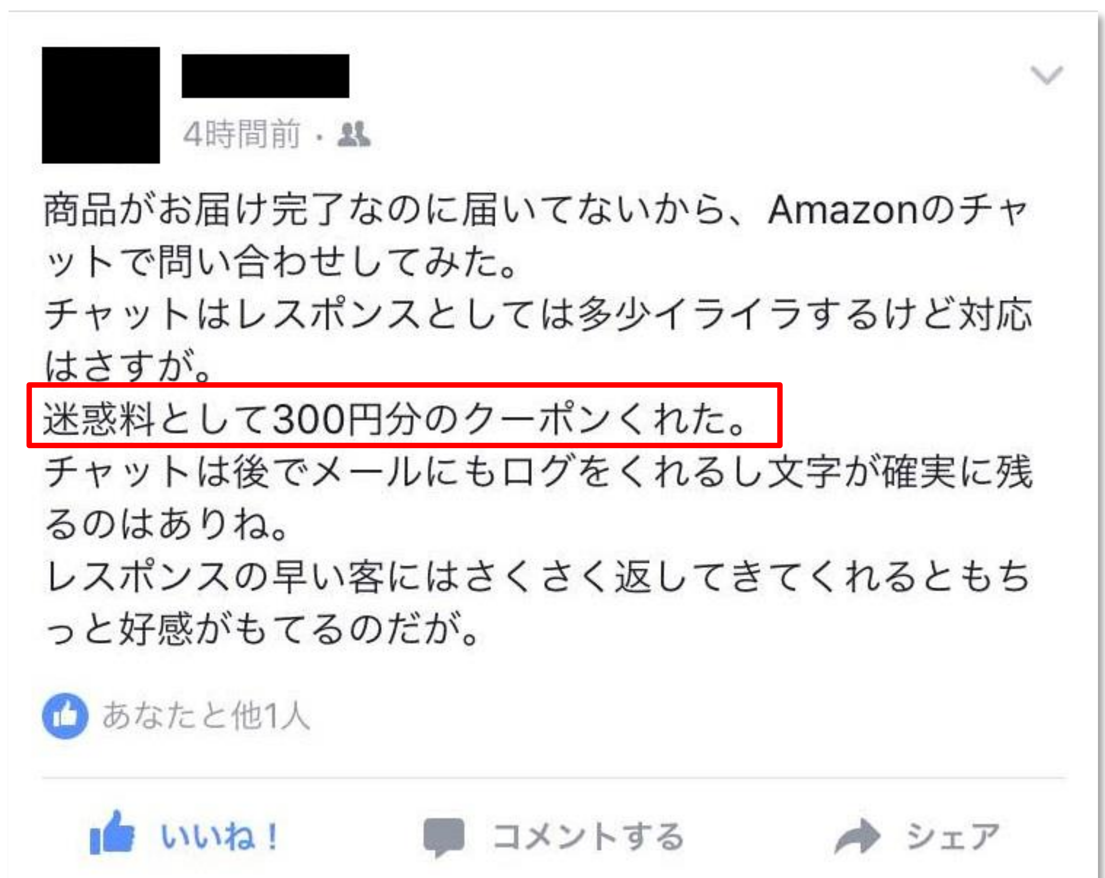
- RESTful API over HTTP
  - もっともシンプルで分かりやすい実装
- メッセージキューによる非同期化
  - 機能同士の非機能を分離することができる





Amazonのクーポン

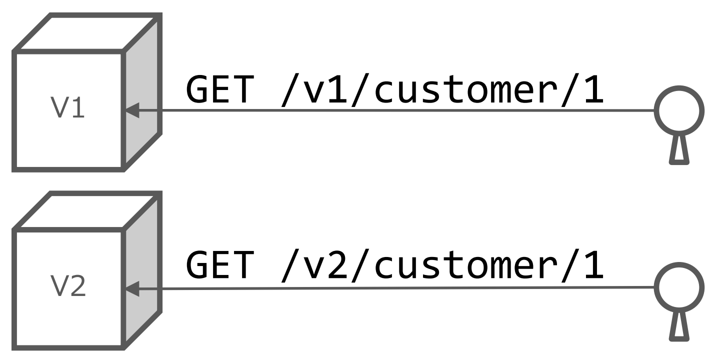
- 非同期による不整合の許容
  - サービス間の疎結合を重視する
  - 運用でリカバリするほうがメリットが大きければクーポンもあり



APIバージョニング

サービス間のバージョン管理 1/2

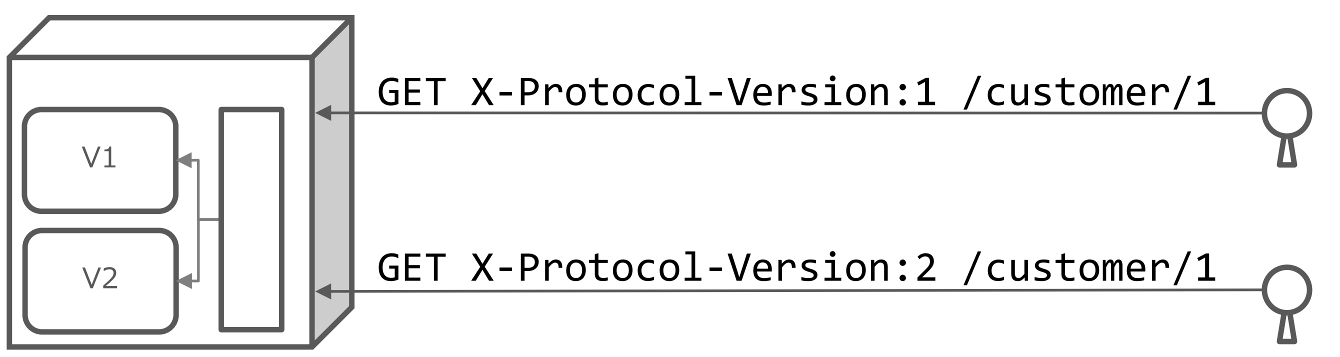
- URLベース
  - URLにサービスのバージョンを表現してしまう
  - 単純で実装しやすいが、古いサービスの維持が必要になる



APIバージョニング

サービス間のバージョン管理 2/2

- 互換性維持
  - 複数バージョンの機能を維持し、パラメタで判別
    - 例：HTTPヘッダの値（X-Protocol-Version）
  - バージョン切替は、
    - 返却データ形式であればMVCのビュー切替など
    - ロジックであればアダプタパターンなど



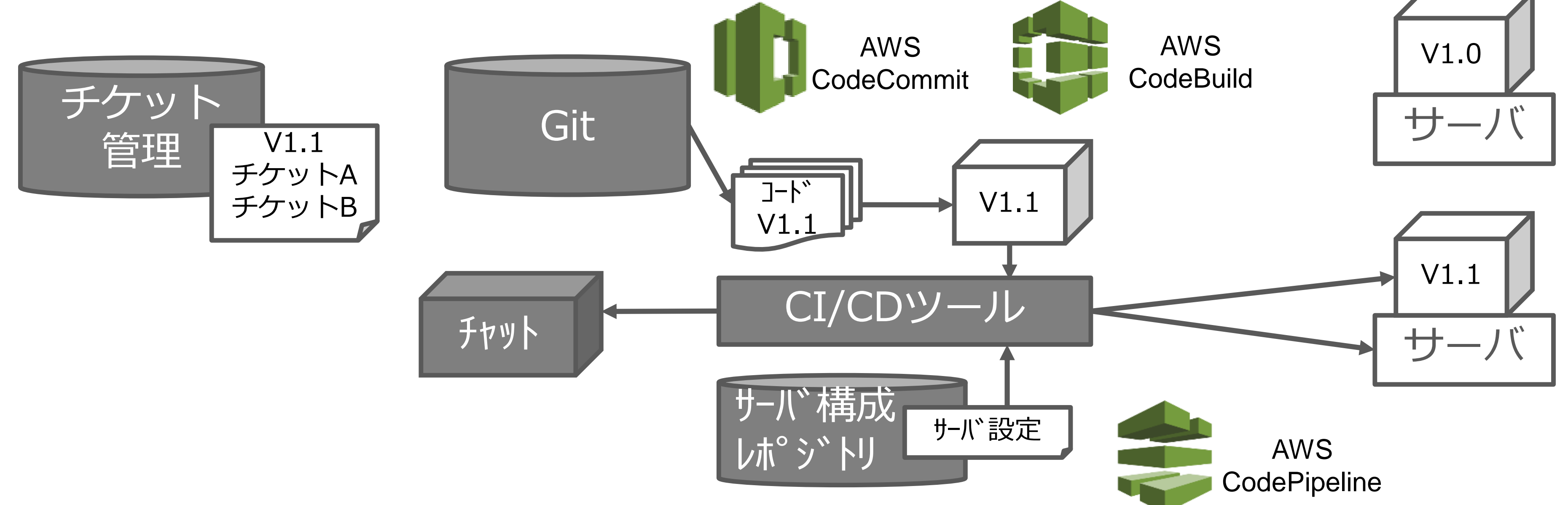
マイクロサービス化設計

無停止デプロイ

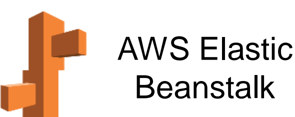
無停止デプロイ

Continues Deploy

- コードがサービスになるまでの一連を自動化
  - » コードチャックアウト、ビルド、テスト、パッケージング
  - » サーバ構築、サーバ設定、デプロイ、起動、起動確認

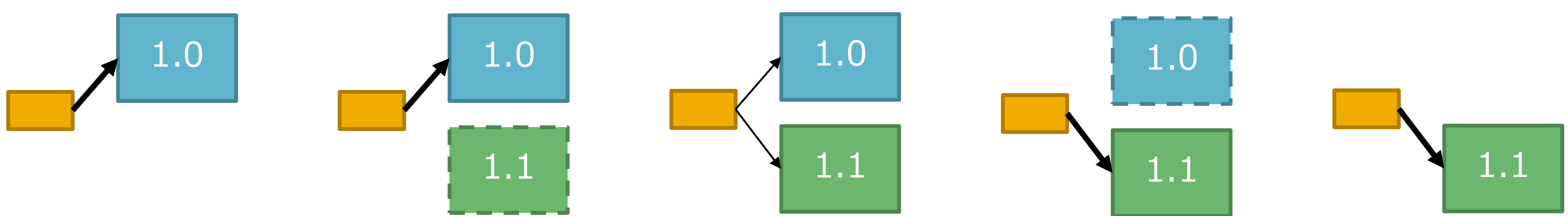


無停止デプロイ



ブルーグリーンデプロイメント

- 瞬断せずにリリースを行うための手法
  - 1.新verを重複して構築
  - 2.ルーティングを制御して新verを解放
  - 3.旧verを削除



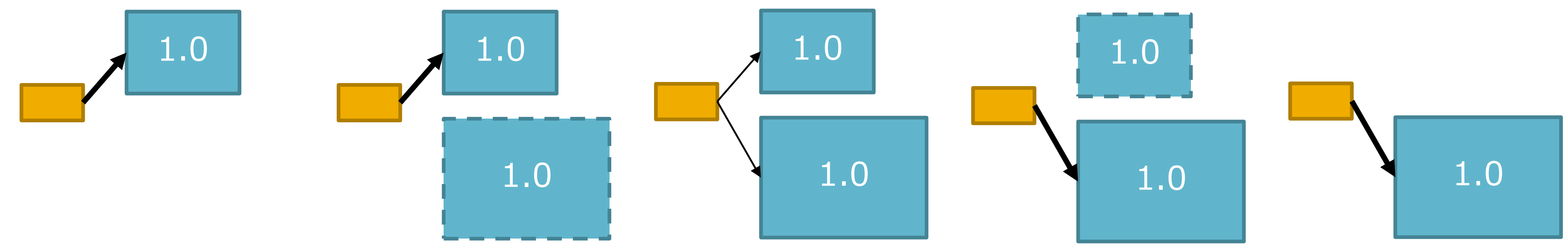
別名：カナリアリリース

- カナリア＝炭鉱で天井に吊しておく鳥
  - ガスは空気より軽いので一番最初に危険に気付く
- カナリア＝最初に新バージョンにアクセスするユーザー
  - 何らかの障害が発生したら切り戻しを行う
  - 少しの犠牲によって大部分を救うことができる

無停止デプロイ

無停止スケールアップ

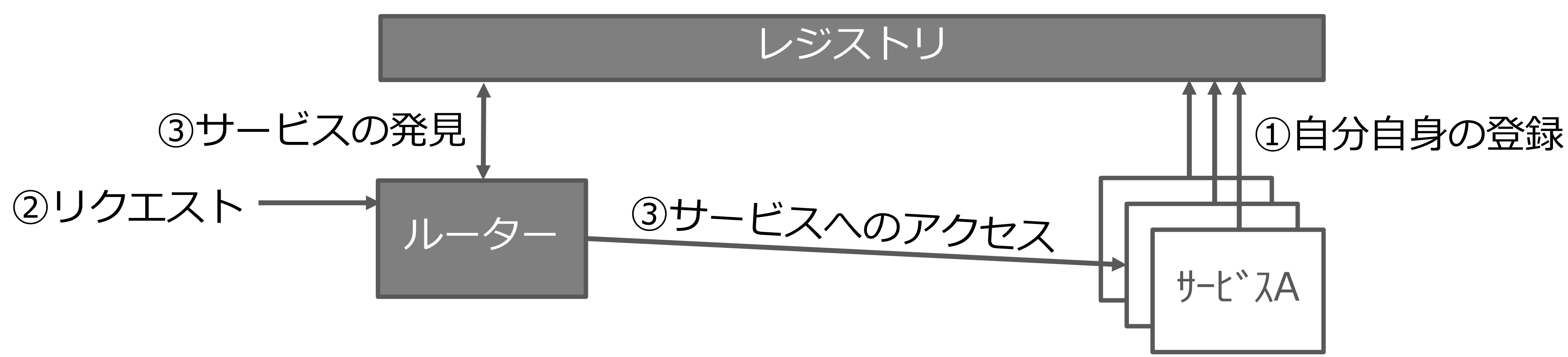
- 瞬断せずにリソース増強する手法
  - 1.新しいインスタンスを重複して構築
  - 2.ルーティングを制御して新しいインスタンスを解放
  - 3.古いインスタンスを削除



インテリジェントルーター

サービスディスカバリとルーティング

- 現在有効なサービスを判断するための方式
  - 新しく起動したサービスは自分の居場所を自分で登録する
  - レジストリに問合せてサービスを発見し、ルーティングする
  - Netflix Eureka、Zool



マイクロサービス化設計

レジリエンス

レジリエンス

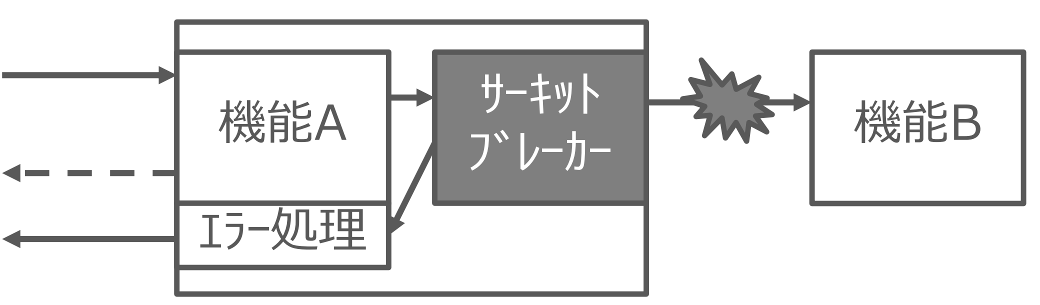
マイクロサービスにおける可用性

- 障害が発生しないようにする、ではない
  - レジリエンス＝復元力
  - 個別サービスの障害でシステム全体が停止しないようにする
  - 従来の「停止しない部分を積み重ねる方式」は限界
- 階層型障害を避ける事が重要

サーキットブレーカー

自分のことは自分で守る

- サービス呼び出し時の異常処理対応
  - 障害発生時のパターン化
  - タイムアウト、自動リトライ、デフォルト値、値のキャッシュ、例外処理記述など
  - Netflix hystrix、Spring Retry



マイクロサービス化設計

テスト戦略

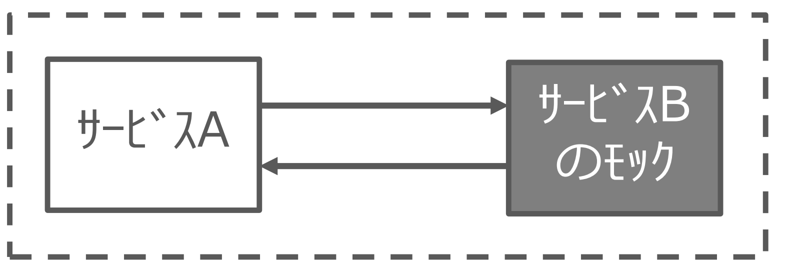
テスト戦略

マイクロサービスにおけるテスト戦略

- サービス間テスト
  - サービス内のテストは閉じている
  - 他のサービスを呼び出し部分のテスト手法
- 本番環境のテスト
  - レジリエンスのテスト

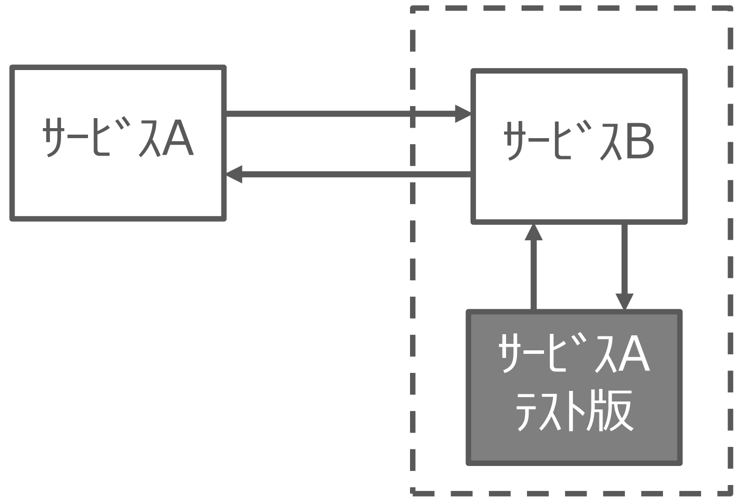
テスト戦略

- Test Doubles
  - 本物を動かすのが大変なのでスタブやモックを使ってテストする
  - コンシューマー側でプロバイダの想定する挙動を実装し、それを使ってテストする
  - プロバイダの変更が発生するとモックの変更が必要になる



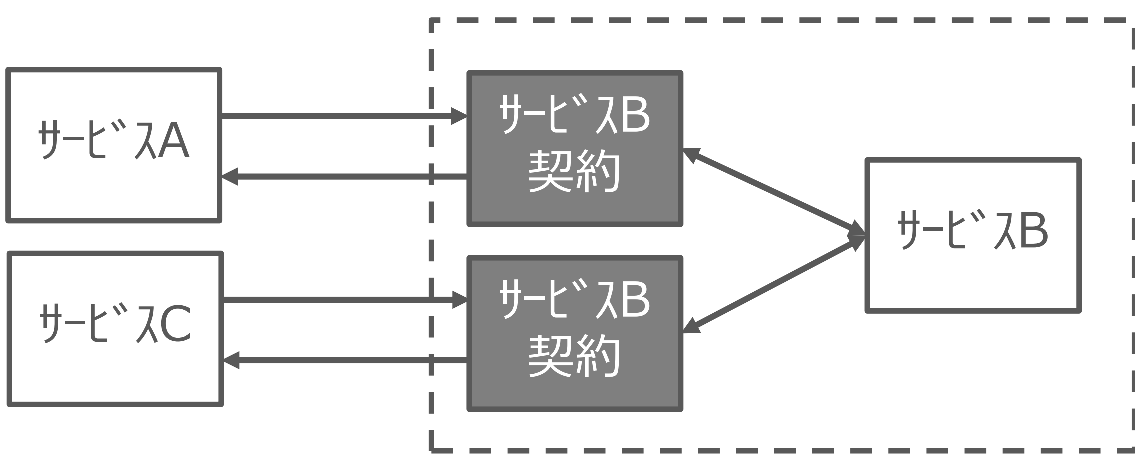
テスト戦略

- ダークカナリア（秘密のカナリア）
  - 本番環境上に開発者しかアクセスできないテスト版をリリース
  - 本物を使ってテストする（ただし、できないものもある）



テスト戦略

- Consumer-Driven Contract testing
  - コンシューマーが要求するAPI挙動を契約として定義し、プロバイダが契約を検証して破壊的変更を防ぐ
  - pact-jvm



テスト戦略

本番環境での"障害注入テスト"

- Netflix's "Failure Injection Testing"（FIT）
  - ランダムにサーバーやデータセンターに障害を発生させ、障害復旧が自動的に行われることを検証
  - Netflix Chaos Monkeyなど
- レジリエンスの検証は本番環境でテストする
  - コードのテストではなく、サービスのテスト

マイクロサービス化設計

チーム体制

チーム体制

コンウェイの法則

- サービス構成とチーム構成を同一にする
  - 「アーキテクチャは組織に、組織はアーキテクチャにしたがう」
  - チーム構成＝サービス構成＝ドメイン構成
  - チームにサービスの自治権を与える
- プラットフォームを管理するチームは必要
  - 横断的関心事の管理チーム

まとめ

なぜマイクロサービスか

マイクロサービス化の目的と手段

- システム全体が巨大になろうとも、機能別に好きなタイミングで機能やリソースの変更を行いたい
- そのために機能別にサービス化を行い、機能の独立性を高めていった
  - 機能間の依存性を管理し、可能な限り疎結合にしていく

マイクロサービス化設計

マイクロサービス化設計における考慮点

- サービス分割
- 横断的関心事の分離
- データの分離
- メッセージング
- 無停止デプロイ
- レジリエンス
- テスト戦略
- チーム体制

マイクロサービス化設計

今日、話ができていないこと

- コンテナ（Docker）
  - » コンテナオーケストレーション

- 非同期ストリーム処理
  - » HTTPの限界を超えていく
- サーバレス

- マイクロサービス向けフレームワーク
- その他、もろもろ

マイクロサービス化設計

マイクロサービス向けフレームワーク

- Spring Cloud
  - Spring Cloud Config
  - Spring Cloud Netflix
  - Spring Cloud Bus
  - Spring Cloud Cluster
  - Spring Cloud Consul
  - Spring Cloud Security
  - Spring Cloud Sleuth
  - Spring Cloud Data Flow
  - Spring Cloud Stream
  - Spring Cloud Task
  - Spring Cloud Zookeeper
  - Spring Cloud for AWS
  - Spring Cloud Connectors
  - Spring Cloud Contract

マイクロサービス化にむけて

明日からでも取り組むべき

- サービスの分割は、いまからでもできる
  - 今できないものが未来になってできることはない
  - 技術の制約を超えるには「不整合の受入れ」が必要
- 現時点をマイクロサービス LV1だと捉える
  - 大規模サービス＋共有データ＋ETLからのスタートでよい
  - どこを切り出していけばいいのかを考える

マイクロサービス化にむけて

マイクロサービス化成熟レベル

Level	名称	状況
1	ほぼモノリシック	数個の大規模サービスが共有データ/ETL連携
2	マイクロサービスの初期段階	複数のサービス群がAPI連携 部分的なプラットフォーム提供
3	マイクロサービス	プラットフォームの整備 CI/CD＋インフラ自動化 インテリジェントなルーティング処理
4	高度なマイクロサービス	高度なサービスやインフラの管理 イベントソーシング ストリーム処理
5	先進的マイクロサービス	マイクロサービスに関する技術開発やOSSの提供

AWS Dev Day Tokyo 2017

マイクロサービス化設計入門

2017/5/31

鈴木雄介

グロースエクスパートナーズ株式会社　執行役員

日本Javaユーザーグループ　会長