システム運用の基本と戦略

2024年度 新卒研修

株式会社サイバーエージェント AI事業本部

高橋 駿（たかはし しゅん）

23卒 バックエンドエンジニア

所属：AIオペレーション室

業務：AWS, DevOps

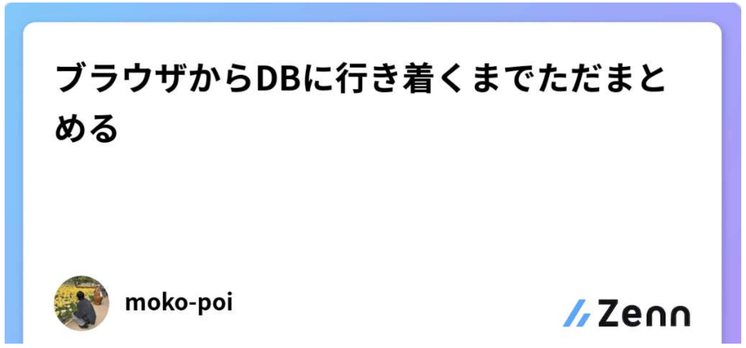
趣味：ディズニー, 邦ロック

- CA BASE CAMP 2023 登壇
- SRE Kaigi 2025 コアスタッフ

@1341Shun

新卒1年目がECSにCanary Releaseを導入し信頼性を高めた話～PipeCD～

PipeCDにおけるECSのCanary Releaseに対して、ListenerRule対応のコントリビュートした話



ブラウザからDBに行き着くまでただまとめる

目的

- 運用に対しての引き出しを増やす
- クラウドネイティブ化の歴史について知る
- DevOpsやSREなどのモダンな運用戦略について知ってもらう

Contents

1

はじめに

CyberAgent

運用したことありますか？

開発はしたことあるけど、運用は？

正社員だと**責任感**や**ビジネス影響**、**コスト管理**なども業務

開発では完成がゴールだが、**運用では継続的なサービス提供がゴール**

良い運用がビジネスの成功を支える

ビジネスの持続可能性、**顧客の信頼**、
そして**最終的な利益**に影響を与える

良い運用は単にシステムを動かすこと以上の価値を持つ

運用に求められるスキル

ネットワークの構成、トラフィックの流れ、サーバーやOSについての理解

運用は日常業務だけでなく、**問題発生時の対応**も大事
技術力と共に、**高い当事者意識**を持つことが重要

→クラウド技術は急速に進化している
基本となる技術理解があれば、新しい変化に迅速に適応し、最適な選択が可能

最新技術を学ぶ際は、その**歴史**や**背景**を
知ることでより**深い理解**と**適切な応用**が可能になる

2

システム運用

システム運用とは

システムが**リリース**されてからサービス提供が**終了**するまでの間、
システムを安定的に稼働させ続けるために**システムを維持・管理**すること

機能要件と非機能要件

機能要件: システムが実行すべき具体的な機能と操作

例：ログイン機能、決済データの分析

非機能要件: パフォーマンス、セキュリティ、スケーラビリティなど、システムの品質を定義する要件

例：システム障害時に3秒以内に復旧する、検索ボタンを押したら3秒以内に表示する

定義することによって、プロジェクトの**スコープ**を明確にし、
予算(コスト)や**リソースの計画**、**リスク管理**を効率的に行うために必要になる

定義しなかった場合…

仮に「システム障害時に3秒以内に復旧する」という要件を定義していなかった場合

・障害発生→復旧手順や人員配置が疎かになる→復旧が遅れる→ユーザーに迷惑をかける

機能要件はイメージしやすい

→　検索機能が欲しい、ログイン機能が欲しい

非機能要件はイメージが難しい

→　検索ボタンを押してから何秒経過したら、ユーザーは不便を感じるのか？

ユーザー目線で考え、ユーザーが不利益や不便を感じないようにするにはどうしたら良いか考える

運用の種類

業務運用

基盤運用

運用管理

業務運用

ユーザーと**システム管理者**などが**円滑に業務**を行えるようにする運用

例:サポートデスクの運用, 貸し出しPCなど…

基盤運用

アプリケーションが稼働するための**土台（基盤）**となる

システムが**安定して稼働**できるようにする運用

例：パッチ管理, バックアップ, リストア…

補足：CIU

CIU（CyberAgent group Infrastructure Unit）は、CyberAgentグループ全体のインフラを支える組織

Cycloudというブランドでプライベートクラウドを展開しており、
IaaSとしての**OpenStack**やKaaSである**AKE**など様々なサービスを提供しています

また、AWS/GCPを利用している事業部のサポートや、
動画配信を始めとするスタジオインフラも担当しており、
強みであるインフラを主軸に幅広い活動をしています

CIU CyberAgent group Infrastructure Unit

https://it.cyberagent.group/team/ciu/

運用管理

システム運用に関わる全ての人が**守るべきルール**と

基準を明確にし、**運用の一貫性と効率性**を確保する

業務運用、基盤運用で統一された運用基準を設定し、一貫性の向上するのが目的

3

基盤運用

CyberAgent

基盤運用は、組織の全ての基礎となる**システム、ハードウェア、**

ソフトウェア、ネットワークインフラの管理を担う

基盤運用の原則とプラクティスは、どんなドメインやサービスにも適用可能

具体的な基盤運用

- アカウント運用
- ログ運用
- 監視
- パッチ適用
- バックアップ/リストア運用
- コスト最適化

アカウント運用

ユーザーの**アクセス権限の管理**と**セキュリティの保守**が目的

アカウントの作成、更新、削除、および権限レベルの設定

認証：識別情報と本人しか知り得ない情報をセットで本人確認すること
認可：認証されたアカウントに対して、サービス利用権限を付与すること

アカウントへの権限付与のパターン
1. アカウントに権限を付与し、複数人でそれを利用する
2. 個々のアカウントを作成し、それぞれに必要な権限を付与

→適切なアカウント管理により、
不正アクセスとデータ漏洩のリスクを最小限に抑える

補足：認証/認可基盤PERMAN

簡単にいうと**認証/認可基盤**
難しい言葉でいうと、**Identity Governance & Administration(IGA)**に分類されるシステム

さまざまな機能を提供している

- SAML2(AWS, Google, Azure, Slack, GitHub, その他色々)連携
- OpenID Connect
- WebAuthnによるパスワードレス認証
- Google Authenticatorなどの認証アプリやSMSなどによるMFA

…

内製化の理由

- 社員数に対してのIDaaSを使用するコストが高い
- 独自の要求に対応できる柔軟性が高い

https://developers.cyberagent.co.jp/blog/archives/33264/

ログ運用

システムとユーザー活動の監視が主な目的

種類：操作ログ、認証ログ、アクセスログ、イベントログ、通信ログ、設定変更ログ、エラーログ

ログ運用で検討すべきこと
1. **ログ設定**：取得/管理するログのリストアップ、ログ出力設定、ログローテーション
例：DEBUG, INFO, WARNING, ERROR, CRITICAL

2. **ログ転送**：ログ転送先の検討、ログ転送設定
例：CloudWatch Logs, FireLens(Fluentd, fluentbit),

3. **ログ保管**：ログ保管先の検討、ログ保管期間の検討
例：CloudWatch Logs, S3

4. **ログ利用**：ログの閲覧、ログ分析、目的に沿った形でログを利用
例：Kibana, Grafana, Snowflake

→ 効果的なログ管理により、異常な行動やシステムエラーの早期発見が可能になる

パッチ適用

パッチとは
・OSやソフトウェアを部分的に修正・更新をするための追加分のプログラムのこと
パッチ適用とは
・パッチとして必要な部分だけをダウンロードして適応する対応のこと

→ **計画的なパッチ適用作業が必要**
事前準備：検証環境でパッチ適用、作業連絡・作業日程の調整、手順書の作成
作業当日：サーバーにログイン、パッチのダウンロード、パッチの適用作業

※AWSのほとんどのマネージドサービスでは、OSやミドルウェアの管理はAWS側で行われる
例：Amazon RDS、Amazon ECS、AWS Fargate、Amazon EKS

しかし、責任共有モデルに定義されているIaaSに分類されるAWSサービスでは、
OS以上の管理はユーザー側に任されている（ユーザーの責任）
例：Amazon EC2：インスタンスタイプやOSをユーザーが選択する

バックアップとリストア

バックアップ
・データの消失、破損に備えてあらかじめデータを別の保管場所に保存しておくこと
リストア
・バックアップから復元すること

→データを復旧してシステムを継続的に稼働し続けるために必要

【取得方法】
1. オフラインバックアップ
・システムを停止した状態でバックアップを行う方法
・システム停止が可能な時間帯（**バックアップウィンドウ**）とバックアップ取得に必要な時間を把握することが必要

2. オンラインバックアップ
・システムを稼働させている状態でバックアップを行う方法
・業務時間外のデータ更新・利用者が少ない時間帯に取得するなど

補足：RDSとAuroraの機能

自動の日次バックアップ機能

・バックアップウィンドウとバックアップ保有期間は任意で指定可能
・バックアップ保持期間は1日から35日まで

スナップショット

・運用者が手動で任意のタイミングで取得するバックアップ
例：変更作業を行う前に手動でスナップショットを取得する

→　どちらもS3バケットへのエクスポートが可能

Auroraのリストア機能

・**ポイントタイムリカバリ**：特定の時点のDBクラスターをリストアする機能（新しいクラスター）
・**バックトラック**：既存のクラスターをある特定の時点の状態に巻き戻す機能
（Amazon Aurora MySQL 互換エディションのみ対応）

コスト最適化

運用コストを効率的に管理し、**不必要な支出**を削減すること

コスト要因：ハードウェア、ソフトウェア、人件費、保守費用...

削減手法：パフォーマンスチューニング、自動化、サーバレス....

クラウドにおけるコスト管理の観点

1. 料金体系の理解
2. AWS利用料の把握と不要リソースの削除
3. 適切なAWSサービスおよびスペックの選択
4. 負荷状況に応じたリソースのスケーリング
5. 継続的コスト最適化の実行

→クラウドの利用料金は**従量課金制**が採用されている
そのため**不要なリソースの把握と削減**が重要

4

監視

CyberAgent

なぜ**監視**が必要なのでしょうか？

監視の必要性

システムの**パフォーマンスを維持**し、潜在的な**問題を早期に発見**し対処するために不可欠

最終的には**ユーザー体験の向上**と**運用コストの削減**を実現

監視のアンチパターン

ツールに依存

・「何をやるか」よりもツールに焦点を当てすぎている
・銀の弾丸となる監視ツールはないため、ツールを増やしたり、時には自作するのも良い

役割としての監視

・監視は特定の人だけが担うものではなく、アプリ、インフラ、ネットワーク担当各自が監視を担う

監視は役割ではなくスキル

チェックボックス監視

・「これを監視してます」と誰かに言うための監視
・その監視を行う明確な理由がない場合は止めて良い

監視を支えにする

・監視しているからといって安心するのは良くない
・監視は状況の改善にはならないため、**サービス自体を安定させる必要がある**

https://www.oreilly.co.jp/books/9784873118642/

→　手段と目的を混同しがち！！

監視は問題を予防し、迅速に対応するための**手段**であり

単にシステムが動作しているかを確認するための道具ではない

監視のデザインパターン

- **組み合わせ可能な監視を使う**

  → 監視システム全体のカスタマイズ性と拡張性を向上させ、
  特定のニーズに応じた専門的な対応が可能になる

  各コンポーネントを個別に扱う：データ収集、データストレージ、可視化、分析とレポート、アラート

- **ユーザ視点での監視から手掛ける**

  → 監視の目的は、ユーザーの体験を直接的に改善すること

  例：HTTPのレスポンスコード、リクエスト時間、**ユーザーに直接影響を与えるメトリクスを最優先にする**

- **作るのではなく買う**

  → 高機能な監視ツールを使用することで、**コストと時間を節約**し、製品や機能の改善に集中できる

アラート

・緊急性に応じてSMSやPagerDutyを使用し、日常の通知は社内チャットで行う
・**アラートの内容や修復の手順**が分かるリンクを入れる
・対応が自動化できるものは自動化するべき

オンコール

・エンジニアを含めて**オンコールローテーション**を行う
・オンコールデータを分析し、**アラートの設定を最適化**する

インシデント管理

・サービスの特性に合わせてカスタマイズされたITILフレームワークを使用
・**役割分担を事前に明確**にして迅速な対応を行う

※ITIL(アイティル)とは、ITサービスマネジメントの成功事例をまとめ、体系化したガイドラインのこと

5

クラウドネイティブ

クラウドネイティブとは

クラウド環境で最新のアプリケーションを構築、デプロイ、および管理するソフトウェアアプローチ
高度にスケーラブルで、柔軟性があり、回復力のあるアプリケーションを構築できるのが特徴

ビジネスに対するメリットは？

効率性
・DevOpsやCDなどのアジャイルなプラクティスを活用し、デベロッパーは、自動化ツール、クラウドサービス、設計を使用して、スケーラブルなアプリケーションを迅速に構築できます

コスト削減
・物理インフラストラクチャの先行投資とメンテナンスに投資する必要がなくなります

可用性
・回復力と可用性の高いアプリケーションを構築でき、機能の更新によってダウンタイムを
発生することのなく、アプリのリソースをスケールアップして、優れた価値を提供できる

https://aws.amazon.com/jp/what-is/cloud-native/

従来の組織体系

開発チーム（Dev）：
設計、開発、テスト、リリース　→ **機能開発に責任を持つ**

運用チーム（Ops）：
運用、保守、障害対応　→ **稼働率に責任を持つ**

DevとOpsのサイロ化

サイロとは
「独立している」状態のことで、開発側・運用側のチーム同士がうまく連携できていない状態を指します

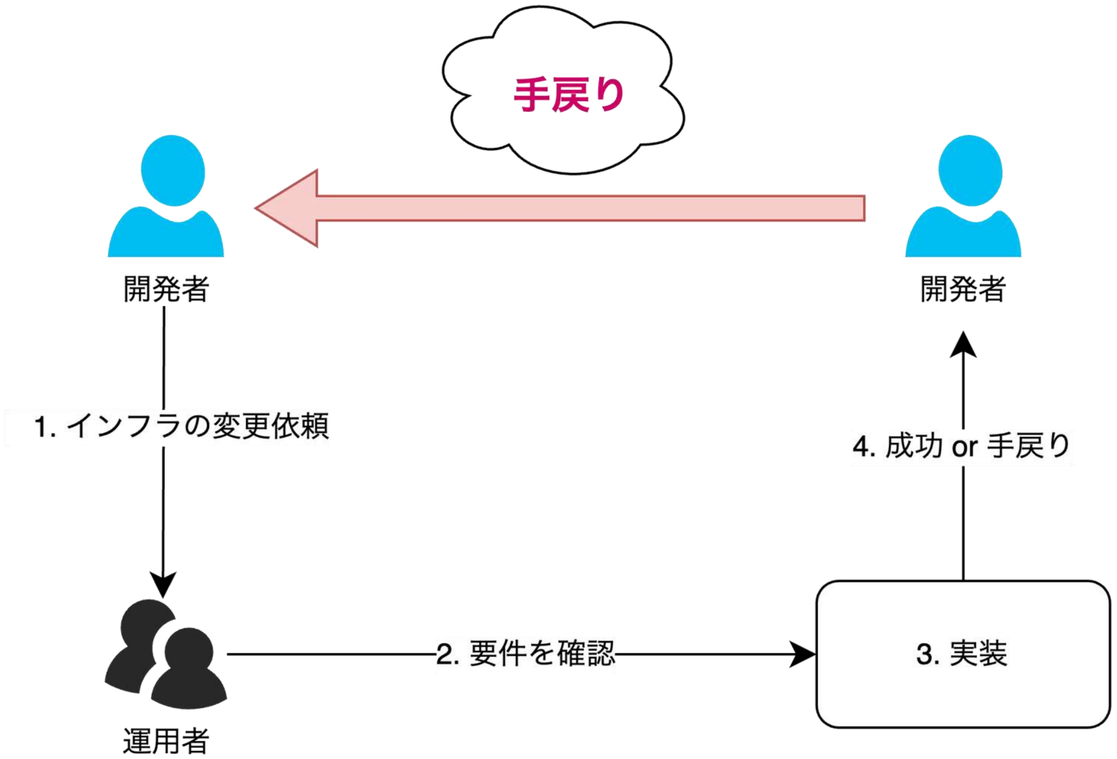
例：
・Devチームが新しいバージョンのミドルウェアやライブラリのインストールを要求
→できるだけ頻繁に新機能をリリースしたい

・Opsチームがセキュリティや運用ポリシーを理由に、それを許可しない
→ できるだけ稼働率を高めながらリリースしたい

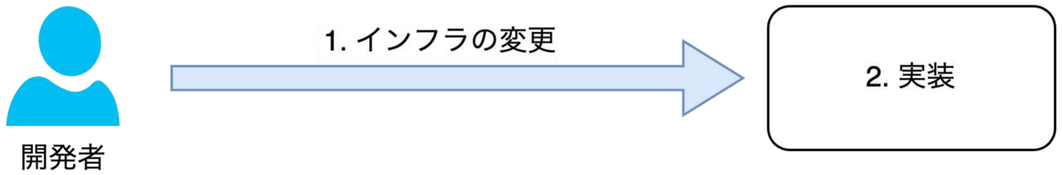
意見の食い違いが積み重なることで、チーム間の溝が深まる**「サイロ化」**が発生する

https://sreake.com/blog/sre-vs-devops/

現実：運用者にインフラ実装を依頼する



理想：開発者がインフラを実装



役割の**明確な分離**により、**コミュニケーション不足**と**目標の相違**が生じ、

これが**プロジェクトの遅延**や**品質問題**の原因となる

ユーザーに価値を届けるのが遅くなる

→　このような問題を解決するために生まれたのが、**DevOps**という考え

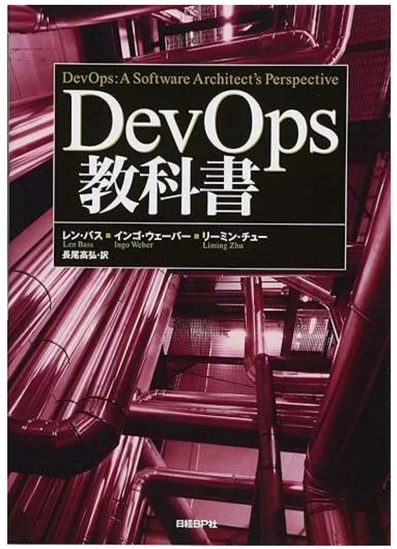
DevOps

DevOpsの定義

「**DevOps**は、高品質を保ちつつ、**システムに変更をコミットしてから**
その変更が通常の本番システムに組み込まれるまでの
時間を短縮することを目的とした一連の実践である」

ようするに..
- **頻繁にリリースする**
- **稼働率を高める**

→ *相反する両方の側面を達成できるように考えていくこと*



https://www.amazon.co.jp/DevOps教科書-レン・バス/dp/4822285448

DevOpsの4つの原則

ソフトウェア開発ライフサイクルの自動化

・自動化によって1日に何度もコードをリリースするようなレベルでの開発の高速化を可能にする（CI, CD, IaC）

コラボレーションとコミュニケーション

・開発担当者や運用担当者をはじめとしたチームが一体となり、協力できるかどうかが大きく影響する

継続的な改善と無駄の最小化

・継続的に改善することや無駄を最小化することに注力する

ユーザーニーズの重視とフィードバックループの最短化

・ユーザーのニーズを中心に開発を進める

https://www.pagerduty.co.jp/blog/what-is-devops

DevOpsではない例

- 手作業での一貫性のないソフトウェアのデプロイ
- 不十分な手作業での品質保証
- 障害時のロールバック計画がない
- リリースを行うのに管理者の手動での承認必須

DevOpsの例

- ブランチ戦略
- フィーチャーフラグ
- CI （Continuous Integration）
- CD（Continuous Delivery）
- IaC
- GitOps

https://note.com/cyberz_cto/n/n6ddc1d143c88

ブランチ戦略

開発者が同じコードに同時に作業する際に競合が起こり、
不安定なコードが本番環境に影響することがある

適切な**ブランチ戦略**を採用することで、機能ごとに開発とテストを分離し、
コードの安定性を保ちながら迅速なリリースとバグ修正が可能になり、
開発プロセスの**品質と速度が向上**する

ブランチ戦略の種類

- Git Flow
- トランクベース戦略

https://scrapbox.io/shufo/Gitブランチ戦略比較

Git Flow

ブランチベース開発のアプローチで、**develop**ブランチと**master**ブランチが中心
機能開発は**feature**ブランチで行い、完成後developにマージ、安定後masterにリリース

例：feature → develop → master

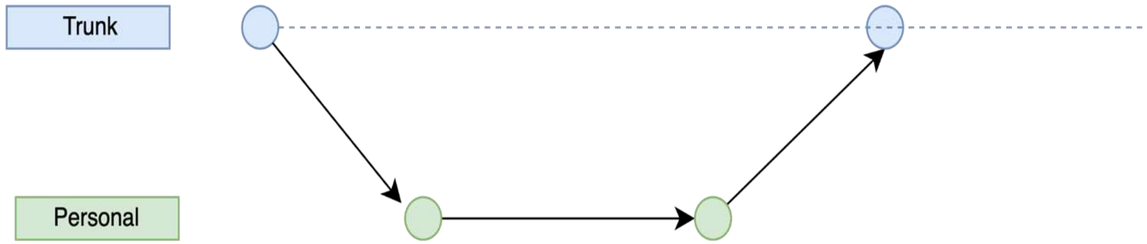
→ リリースサイクルがゆっくりで問題ない場合、調整したい場合
保守運用フェーズの場合に利用されることが多い

課題点
- developとmasterの間に大きな差が出ることがあり、統合が複雑化する
- masterへのマージ時には、再度の全体テストが必要になる
- 大規模な変更が一度にリリースされることが多く、リスクが増大（ビックバンリリース）

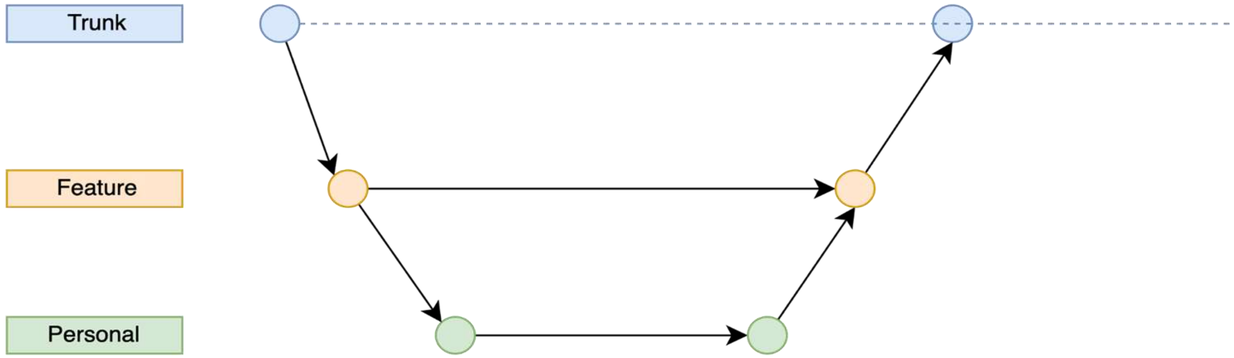
トランクベース戦略

メインライン（masterブランチやtrunkブランチ）を中心とした開発手法
開発者は短いサイクルでメインラインに頻繁にコミットを行うスタイル

フィーチャートグル
- 未完成またはテスト中の機能はフィーチャートグルを使用して有効化・無効化する
- 機能を本番環境にデプロイしながらも限定的に公開することができる



①小さなバッチでの作業を行い、作業を1日に1回以上マージ
(ブランチの作業時間は数時間以内にする)



②機能が全て揃った後にトランクにマージ

→フィーチャートグルやカナリヤリリース、CIによる自動テストなどの整った体制が必要

https://dora.dev/devops-capabilities/technical/trunk-based-development/

フィーチャーフラグ

静的または動的に機能のON/OFFを切り替える手法
利用することでアプリケーションをリリースすることなく機能のON/OFFを制御する
→　つまり**デプロイ**と**リリース**を**切り分けることができる**

```
if featureFlag() {
    // Do someting
} else {
    // Do someting
}
```

ユースケース
- 限定的なリリース、トランクベース開発、A/Bテストなど

CA発のOSS「**Bucketeer**」がある
AWSだと「**Amazon CloudWatch Evidently**」などがある

https://developers.cyberagent.co.jp/blog/archives/35021/

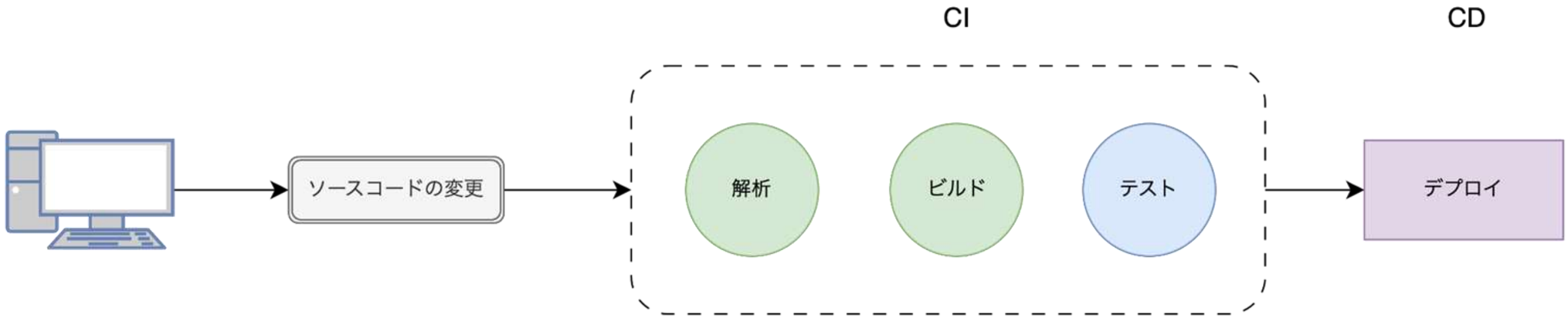
CI/CD

CI（Continuous Integration: 継続的インテグレーション）

・開発者が自分のコード変更をした際に自動化されたビルドとテストを実行するソフトウェア開発の手法
・主な目的は、バグを早期に発見して対処すること、ソフトウェアの品質を高めること、そしてソフトウェアの更新を検証してリリースするためにかかる時間を短縮すること

CD（Continuous Delivery: 継続的デリバリー）

・コード変更が発生すると、自動的に構築、テスト、および実稼働環境へのリリース準備が実行されるというもの
・すべてのコード変更が、ビルド段階の後にテスト環境または本番環境、あるいはその両方にデプロイされます



https://aws.amazon.com/jp/devops/what-is-devops/

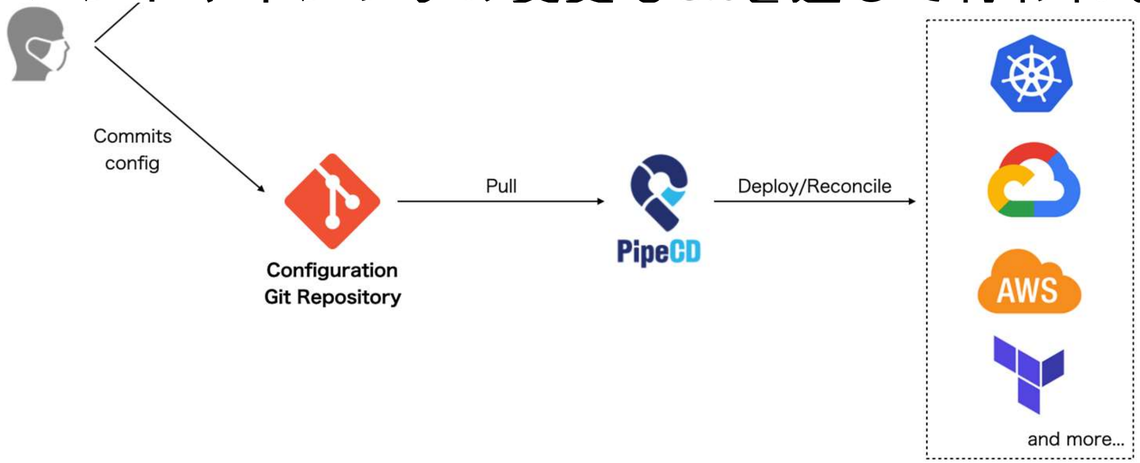
IaC, GitOps

IaC (Infrastructure as Code):
・インフラストラクチャーをコードとして管理する手法
・環境の構築がコードによって定義されているため、一貫性があり、エラーが少なくなる
・またコードがそのまま文書として機能し、状態の追跡と管理が容易になる

ツール: Terraform, AWS CloudFormation, Ansible

GitOps
・Gitリポジトリを使用してインフラストラクチャーとアプリケーションの構成管理を行う手法
・Gitが単一の信頼できる情報源となり、デプロイメントやインフラの変更もGitを通じて行われる

ツール: Argo CD, Flux, PipeCD



※CA発のOSS「PipeCD」がある

https://pipecd.dev/docs-v0.45.x/overview/

DevOpsを実践することにより、
稼働率を下げずに、リリース頻度が高くなる

サービスの稼働率を高めるには？

SRE

DevOpsとSREの違い

“class SRE implements interface DevOps”

https://sre.google/workbook/how-sre-relates/?_fsi=XoYJpNx5

→つまり「DevOpsの実現方法の一つがSRE」

DevOpsでは**「何を」**すべきかを重視
SREでは**「どのように」**すべきかを重視

SRE（Site Reliability Engineering）

> *So SRE is fundamentally doing work that has historically been done by an operations team, but using engineers with software expertise, and banking on the fact that these engineers are inherently both predisposed to, and have the ability to, substitute automation for human labor.*

引用：https://sre.google/in-conversation/

オペレーションを**ソフトウェアの問題**として扱う手法

→要するに**運用の問題**を**ソフトウェアで解決**するということ

そもそもReliability（信頼性）とは？

信頼性の高いシステムを設計する際には、**可用性を確保することが不可欠**

可用性とは**「システムが利用できる状態であり、システムが継続稼働できること」**

サービスを利用できることが**売上、利益**につながる
サービスが使えない=**売上、利益が減る**

よくある勘違い

SRE=職種（ポジション）✕□

SREは**ロール**であり、**技術**であり、**概念**である
開発者も、管理者も、SREを理解すると良い

→ 可用性が担保されて、初めてユーザーはサービスを利用できるから

https://qiita.com/cocoa-maemae/items/d4f5a403f94e81da798f

SREを実践する上で必要な知識

- 可用性
- 信頼性
- SLI/SLO/SLA
- エラーバジェット（エラー予算）
- ポストモーテム
- トイル

可用性

「システムが継続して稼働できる度合いや能力のこと」

[可用性対応表]

可用性	最大利用不可能時間 (年間)	アプリケーションのカテゴリ
99%	3 日と 15 時間	バッチ処理、データの抽出、転送、ロードのジョブ
99.9%	8 時間 45 分	ナレッジ管理、プロジェクト追跡などの社内ツール
99.95%	4 時間 22 分	オンラインコマース、POS
99.99%	52 分間	動画配信、ブロードキャストのワークロード
99.999%	5 分間	ATM トランザクション、通信のワークロード

→**サービスの目的**、**影響度**、**ユーザーの期待**を考慮し、**適切な可用性レベルを選定するのが大事**

https://docs.aws.amazon.com/ja_jp/wellarchitected/latest/reliability-pillar/availability.html

可用性

適切な可用性レベルの選定
すべてのサービスに同じレベルの可用性を適用する必要はない
例：クリティカルな金融システムは99.99%、内部用の文書管理システムは99.9%で十分

コストと可用性のトレードオフ
高い可用性はより多くのインフラ投資と継続的な保守が必要になる
99.9%から**99.99%**への向上は、**大幅なインフラ投資**、**より高度な冗長性設計**、**継続的な保守管理**が必要

可用性向上のコストがそのサービスから得られる利益を超えないようにする
→費用対効果を考慮して、サービスごとに最適な可用性レベルを定めるのが大事

補足：
Amazon S3では、99.9999999% (イレブンナイン) の**耐久性**と、99.9% の**可用性**が保証されている

可用性＝システムが稼働し続けることを保証するレベル
耐久性＝データを失わないことを保証するレベル

https://aws.amazon.com/jp/builders-flash/202202/awsgeek-s3/?awsf.filter-name=*all

信頼性

いかなるシステムにおいて**最も重要な機能は信頼性**

- **「ユーザーに使ってもらえる」**状態にする
- 監視が信頼性を決めるのではなく、**ユーザーが決めるもの**

ログや監視などの実績値では問題がなくても、
ユーザーが不満に思っていたらそれは信頼性に欠けている言える

例：CPU値が100%でもサービスが使えていればユーザーは満足、逆に余裕があっても使えなければ不満

SLI/SLO/SLA

SLI (Service Level Indicator)：サービスレベル指標
予め決めたユーザーの利用するサービスが、ユーザーが許容できる範囲で完了するまでの指標

例：DSPがSSPから入札要請を受けてから、広告の入札単価を提示するまでの時間
（SSPでは0.1秒以内でレスポンスが帰ってこないDSPをそのリクエストでは除外する仕様）

$$\mathrm{SLI}（レスポンス時間）= \frac{要求に対して0.1秒以内に応答した入札数}{総入札要求数} \times 100$$

SLO(Service Level Objectives)：サービスレベル目標
SLIで設定した値以内で完了したサービス提供回数が全てのサービス提供回数のうち、どのぐらいの割合で提供できればいいかという目標値

例：目標: DSPが受けた入札要請の少なくとも99%に対して、0.1秒以内に応答する

SLO（レスポンス時間）= 99%

SLA (Service Level Agreement): サービスレベル契約
SLOが達成できなかった場合に、ユーザーに対して何らかの補償などを約束する取り決め

エラーバジェット（エラー予算）

許容できるサービスの**ダウンタイムやエラーの量**を定量的に表したもの

サービスのSLO（Service Level Objectives、サービスレベル目標）を基にして定義され、
SLOを超えるサービスの不具合が「許容範囲」内であるかどうかを測定します

→　**SLO - SLI = エラーバジェット**

エラーバジェットが「余っている」場合
・新機能の**リリース**や**実験**、**課題解決**を行う

エラーバジェットが「なくなった」場合
・新機能の**リリースを控え**、システムの安定性や信頼性、**可用性を改善する**必要がある

ポストモーテム

システム障害や**インシデント発生後**に行われる**分析のプロセス**

何が起こったのか、なぜ起こったのかを理解し、将来同様の問題を防ぐための改善策を特定する

→　ポストモーテムは、**問題を解決しようとするのではなく**、
問題から学び、組織のプロセスやシステムを改善する機会である

※ インシデントが完全に解決した後に取り組む課題

ステップ
1. インシデントの記録 → 2. 影響の評価 → 3. 根本原因の分析 → 4. 解決策と改善策の議論
→ 5. 報告書の作成 → 6. フォローアップ

注意点

・定期的なレビューを行うこと

・ミスをした人の追求をしない（原因を追求すること）

https://speakerdeck.com/kazumax55/bei-earehahuan-inasi-xiao-lu-de-nainsitentodui-ying-womu-zhi-susrenoqu-rizu-mi

具体例：ポストモーテム

あるオンラインショッピングサイトで、重要なプロモーション期間中にシステム障害が発生し、数時間サービスが利用できなくなったとします

1. 障害発生の時間、影響を受けたサービスの範囲、ユーザーからの報告数などを記録
2. 売上損失、ユーザー満足度の低下、ブランドへの悪影響など、ビジネスに与えた影響を評価
3. サーバーの過負荷が原因であることが判明、特定のプロモーションが想定以上のトラフィックを引き起こし、リソースが不足したためだった
4. 短期的には追加のサーバーリソースを確保し、長期的にはトラフィック予測の見直しと自動スケーリング機能の導入
5. 分析の結果と提案

トイル

手動で反復的かつ**自動化可能**な、**価値を生み出さない作業**のこと

→トイルは、**チームの効率性を低下**させ、
技術的な成長や**新しいプロジェクトへの障害**になるため、できるだけ**減らすべきもの**

～特徴～

- **反復的**: 同じ作業を何度も繰り返す必要がある
- **手動**: 自動化されていないため、人の手による介入が必要
- **スケーラブルでない**: システムが成長するにつれて、トイルの量は比例して増加する
- **戦術的**: 短期的な問題解決にはなるが、長期的な価値はほとんどまたは全くない
- **自動化可能**: 作業の性質上、プロセスを自動化することでトイルを減らすことが可能

SREの職務内容

システムの信頼性と性能の監視
・システムの健全性の確認、KPIの定義と監視、パフォーマンスボトルネックの特定など

障害発生時の迅速な対応
・ダウンタイムを最小限にし、ユーザー影響を低減する
・障害のアラート対応を行い、後にポストモーテム分析を通じて再発防止

サービスレベル目標（SLO）、サービスレベル指標（SLI）の定義と監視
・サービスの期待される品質の明確化、顧客約束を明確にするためSLIを定義、それに基づいて監視のダッシュボードを作成し、SLOの達成率を調整する

自動化の推進
・手動のエラーを削減、運用効率を向上させる

これらを実践することで
サービスの稼働率を高めることができる

→サービスが巨大化したら何が起きるか

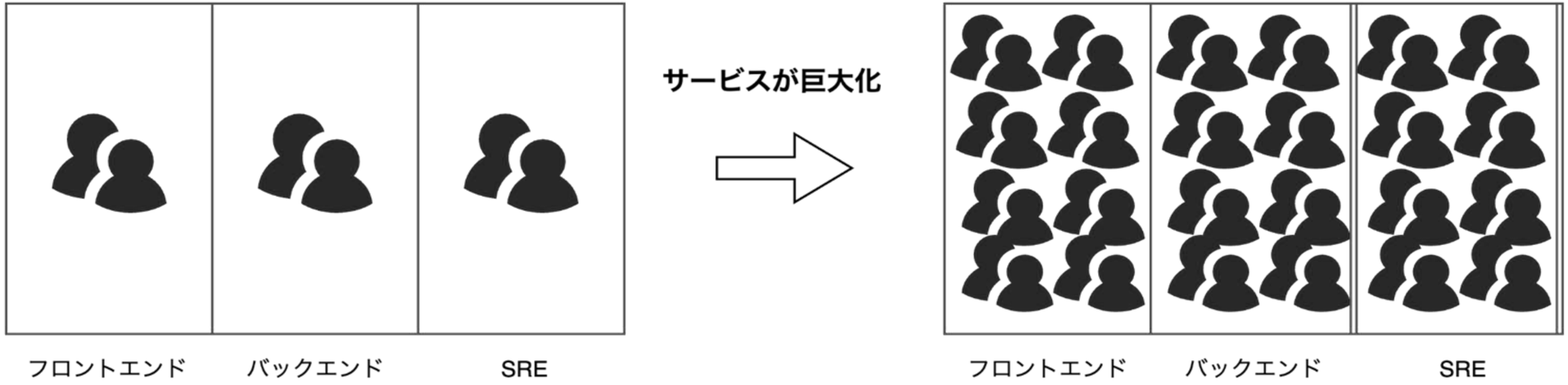
更なる課題

リンゲルマン効果とは
・集団で共同作業すると、人数の増加とともに1人当たりの仕事効率が低下する

サービスが拡大するにつれ、
フロントエンド、バックエンド、SRE、QA
各チーム間で新たな**サイロ**が発生する

一つの機能をリリースする際、**すべてのチームの共同作業**が必須となる

その他の課題点
・コミュニケーションコストが大きい
・リンゲルマン効果によりメンバーの士気やパフォーマンスが低下
・チーム間で優先順位が共有されなくなる…



解決の一つとして...

マイクロサービスアーキテクチャの採用することで、

スケーラビリティと柔軟性を高め、新たな**サイロを防ぐ**

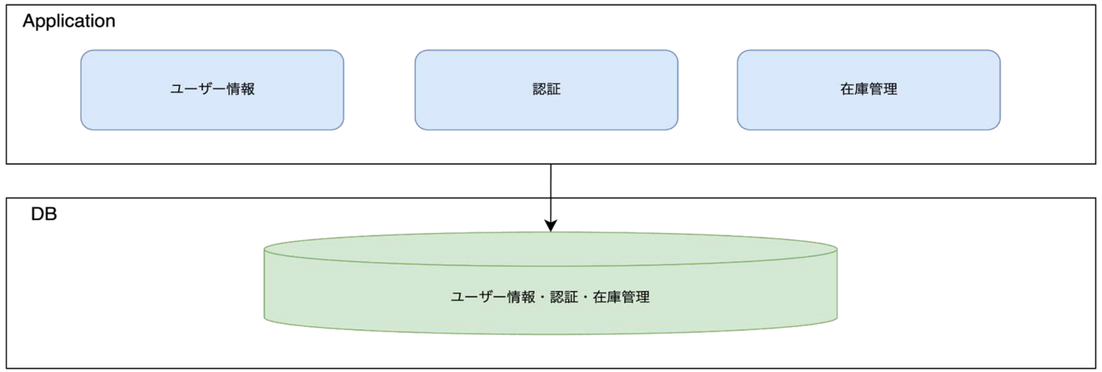
→ チームを**クラウドネイティブ化**

Microservices Architecture

代表的なアーキテクチャ

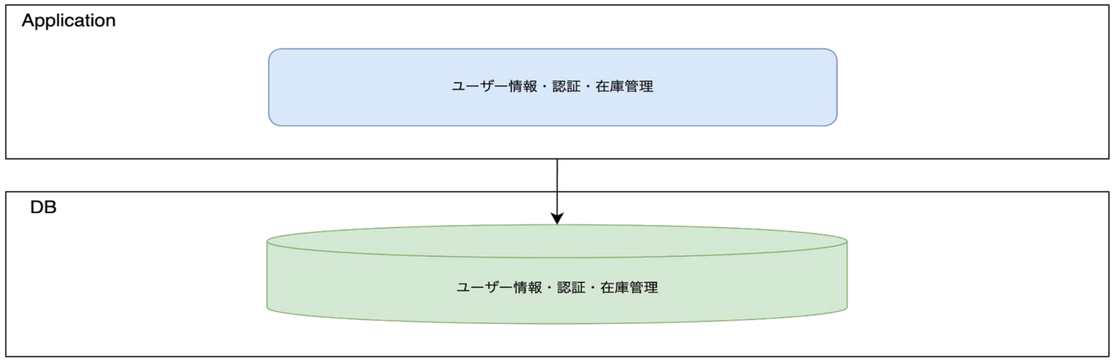
モノリシック
一元管理、スケーリングが困難、更新の複雑性、障害時の影響範囲が広い

サービス指向アーキテクチャ（SOA）



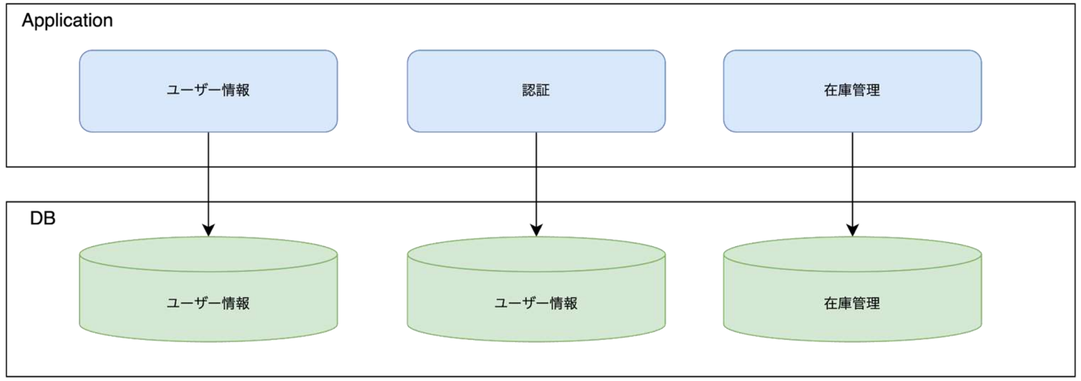
マイクロサービスアーキテクチャ
高い自律性、スケールの容易さ、デプロイの簡便性、
技術スタックの多様性、耐障害性、複雑な管理

モノリシック



サービス指向アーキテクチャ
サービスの再利用性、複雑な管理、個別のサービス更新が可能

マイクロサービス



マイクロサービスとは

ビジネスロジックを細分化して**個々のサービスとしたアーキテクチャ**

各サービスは特定のビジネス機能を担い、**独立して開発・デプロイ**される

サービスは独立しており、需要が高い部分だけを個別にスケールでき、
デプロイが迅速で、障害があっても、全体の稼働に影響が少なくリスクが低い

→　**そのためスケーラビリティと柔軟性が高くなる**

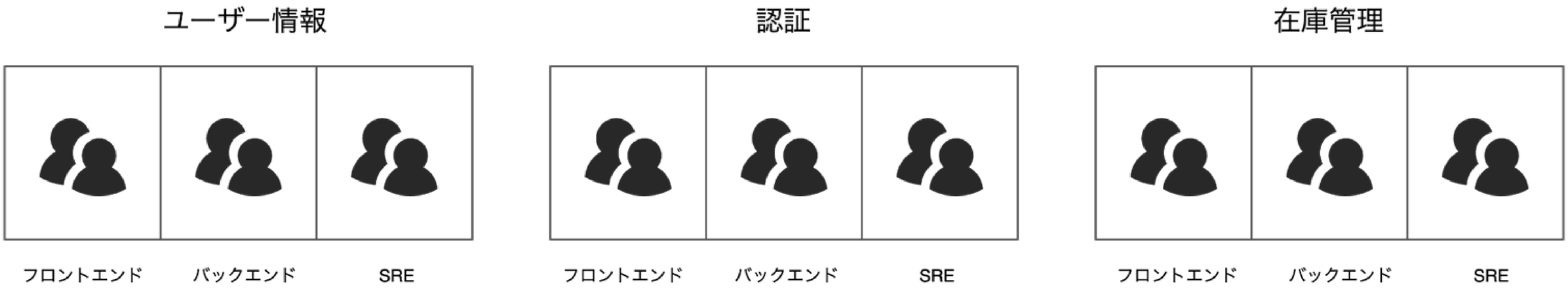
マイクロサービスアーキテクチャにおける究極的な成果物は**新たな組織**

自律的に機能する主体的な組織

従来の大規模なチームから、
マイクロサービスごとに専任の小規模なチームに再編成

少数構成の特徴
・チームは、E2Eでサービスのライフサイクルに対する責任を持つようになる
・自主的に機能するための権限と環境
・自ら課題を見つけ解決する
・コミュニケーションのオーバーヘッドを減らし、チーム内での情報共有がスムーズになる

最も重要なことは**「自律性と説明責任がある」**ということ



最適なチーム構成

Amazon: "You Build It, You Run It"
・自分たちで開発したマイクロサービスへのオーナーシップを強く持つ
・オーナーシップを持つことで、サービス品質がユーザー目線でもテクノロジー目線でも高くなる

"Two pizza rule" - 1つのチームは二つのピザで満足できる人数
具体的には6から10人程度

Netflix: Full Cycle Developers
・ソフトウェアのライフサイクルの全ての面に関わる
・具体的なライフサイクルには、設計、開発、デプロイ、運用、サポートまで含れます

→従来は、それぞれのサイクルを専門とするエンジニアがいたが、個々の効率は高かったものの、それぞれでサイロを生み出し、各エンジニアの**コミュニケーションコストが増大**、ライフサイクル全体としては**スピード低下**を招いていた

"Operate what you build" -機能を開発したチームが運用とサポートの責任も持つという原則

運用上の負荷も軽減することができるようになり、開発上の問題、キャパシティプランニング、アラートの問題、サポートといったことにもオーナーシップを持てるようになる

https://engineering.mercari.com/blog/entry/2018-12-01-200159/

チームの人数を小さくしたことによる問題

- **技術的な負担が増加**
- **チームごとの技術レベルが揃わない**
- **至る所で行われる車輪の再発明**

→ 各チームの**認知負荷（Cognitive Load）**が大きくなる

https://zenn.dev/kangetsu_121/articles/6b31565dda6053#認知負荷の定義%3A-定義の発展と三種の分類

認知負荷（Cognitive Load）

ユーザーが**情報を処理**したり**理解する際**にかかる**負荷**

人間の脳にワーキングメモリがあり、
その容量は限られていることから発生して作られた言葉

[負荷の種類]

学習対象そのものの複雑さによる負荷　→　仕組みで解決するべき
例：マイクロサービスアーキテクチャで複雑な分散合意の実装

業務や学習に無関係な余分な負荷　→　仕組みで解決するべき
例：ドキュメント不足による情報収集、手動による作業など

理解や学習が進行する際に発生する適切な負荷　→　少数チームではここに集中する
例：サービスの正確な理解、ドメイン理解

https://theelearningcoach.com/learning/what-is-cognitive-load/

仕組みで解決するべき..

- 学習対象そのものの複雑さによる負荷
- 業務や学習に無関係な余分な負荷

問題を低減する仕組み → **Platform Engineering**

Platform Engineering

Platform Engineeringとは

“先進的なテクノロジを活用したプラットフォームにより、アプリケーションのより迅速なデリバリとビジネス価値の創出を可能にする手法”
https://www.gartner.co.jp/ja/articles/what-is-platform-engineering

開発チームの**開発者体験と生産性を向上させることを目的**として、
セルフサービスで利用できる**プロダクトを設計・構築**する https://pages.awscloud.com/rs/112-TZM-766/images/20240229-platform-engineering-3-amazon_ecs_platformengineering.pdf

※ セルフサービス：開発チームが自律的に設計・開発を行えるようになり、開発生産性の向上につながるサービスのこと

Platform Engineeringが提供するもの

ウェブポータル
・プラットフォーム機能を提供するポータル

ゴールデンパス
・迅速なプロジェクト開発に役立つ巧みに統合されたコードと機能の**テンプレート構成**
例：CI/CDパイプラインのテンプレート、Terraform(IaC)のテンプレート、Kubernetes YAMLファイル…
→開発チームにオーナーシップを委ねるのが目的https://cloud.google.com/blog/ja/products/application-development/golden-paths-for-engineering-execution-consistency

アーティファクト
・開発プロセスで生成される成果物やそれらを管理する
例：コンテナイメージ、ライブラリ、ソースコード..
→効率的に生成、管理、配布することが目的

インフラストラクチャ
・開発チームがアプリケーションを効率的に開発、デプロイ、運用するための基盤技術の提供
例：コンピューティングリソース、ストレージ、モニタリング

共通基盤やパブリッククラウドとの違い

共通基盤
→従来のインフラ共通基盤では利用が強制されてしまう

パブリッククラウド
→様々なケースに対応する汎用的な技術を提供している

IDP（Internal Developer Platform）
・内部開発者プラットフォーム

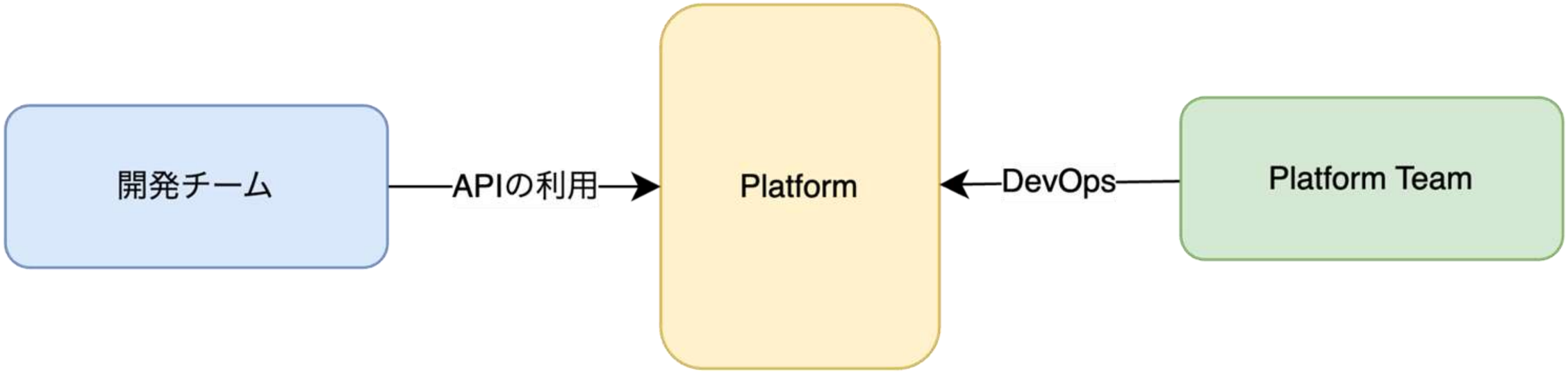
PlatForm EngineeringはProductとして提供する

・プラットフォームを使用するかどうかは**開発チームが選択**できるため、
必要に応じて最適なサービスを選択でき、組み合わせることができる

・またユーザー目線で価値ある機能を備えるため、継続的に改善される

認知負荷を減らしながら実現できる

それぞれのチームが**End-to-Endで説明責任と権限**を
持つことで、**サイロ化**を防ぐ



課題

Platform Engineeringで全体の生産性と信頼性を向上ができた

→ただ**局所的**で**特殊な用件**に対応するのは難しい場合がある

そこで **Embedded SRE**

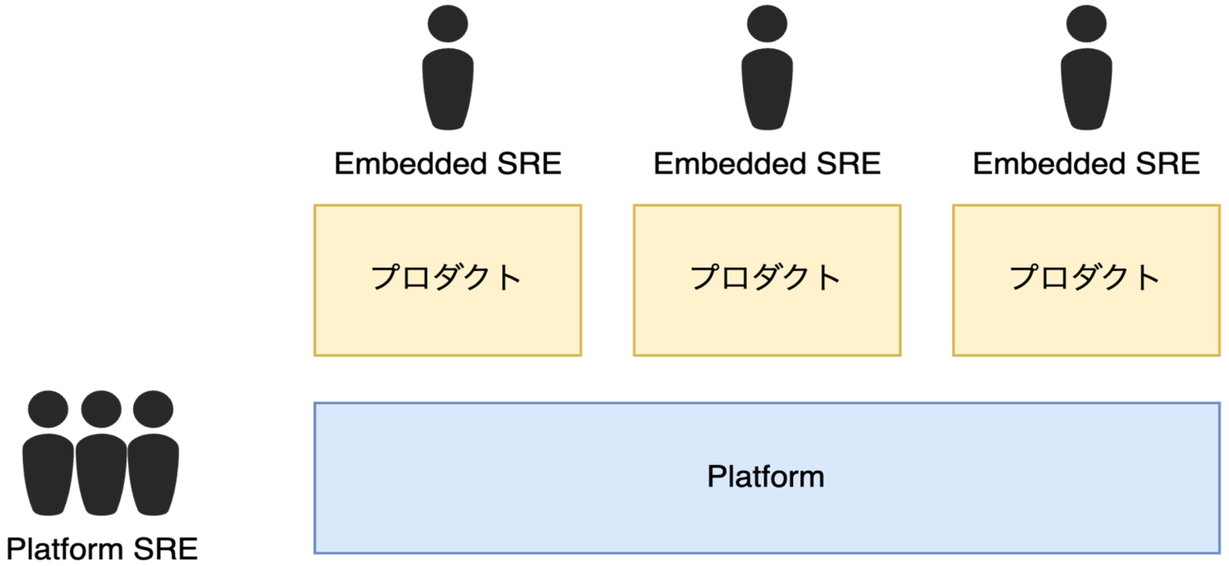
Embedded SREとは

サイロを生まないように一時的に**プロダクトチーム内にに配置されるSRE**

・プロダクトに深く関与し、**プロダクト固有の課題を理解して直接対応することが可能**

・**定期的なローテーション**をすることにより、異なるプロダクトチーム間での**ベストプラクティスと成功事例の共有**が行われ、組織全体の知見共有

・継続的なフィードバックループを確立し、**開発と運用の間のギャップを縮小**



https://chroju.dev/blog/types_of_sre_teams

Platformという**Product**を利用することによって

全体の80%程度の生産性や信頼性に関わる**認知負荷を解消**、

残りの**20%のドメインに局所的な問題を**Embedded SREが解決

クラウドネイティブの歴史

1.DevとOpsのサイロ化 → DevOps

2.サービスの稼働率 → SRE

3.サービス拡大につれ各チーム間でサイロ → マイクロサービス

4.技術的負担、チームごとの技術レベルが揃わない、

車輪の再発明 → Platform Engineering

6

最後に

CyberAgent

まとめ

- 開発するだけがエンジニアの仕事ではない
- 機能開発も重要だが、運用あってこそのプロダクトでもある
- ポジション関係なく、全員で運用する
- プロダクトに対して当事者意識を持つ（責任感、ビジネス影響、コスト意識）
- モダン技術を使えば良いというわけではない
- フェーズ・体制・規模などに合わせた、組織やチームに最適な運用を行う

References

- AWS運用入門 押さえておきたいAWSの基本と運用ノウハウ
- 障害対応入門記事まとめ～システム運用担当者になったらまず読むべき記事を厳選!～
- 入門 監視
- チームとプラットフォームをクラウドネイティブな開発に最適化する
- ちがいからみる Platform Engineering – クラウドネイティブ化に伴って生じた新たなチームトポロジー
- DevOps教科書
- DevOps実践ガイド
- AWSで実現するモダンアプリケーション入門 ～サーバーレス、コンテナ、マイクロサービスで何ができるのか
- SREエンタープライズロードマップ
- SRE サイトリライアビリティエンジニアリング
- サイトリライアビリティワークブック