

マイクロサービスアーキテクチャ

採用の技術資料

モノリシックからマイクロサービスへの移行ガイド

開発チーム・経営層 向け

モノリシック

従来のアプローチ



サービスA

サービスB

サービスC

サービスD

マイクロサービスアプローチ

目次

このプレゼンテーションでは、モノリシックアーキテクチャからマイクロサービスへの移行に関する重要なトピックをカバーします。

対象者

👥 開発チーム

👔 経営層

現状と課題

- 🚩 **現在のモノリシックアーキテクチャの課題**
スケーリング、保守性、デプロイ

マイクロサービスの概念

- 💡 **マイクロサービスの基本概念と利点**
経営層向け概要
- 🔗 **マイクロサービスの技術的利点**
開発チーム向け詳細
- ⚖️ **モノリシックとマイクロサービスの比較**
アーキテクチャ分析

移行戦略

- 📊 **移行アプローチ - 全体像**
戦略とロードマップ
- 📋 **移行アプローチ - 段階的実装**
実践的な手順

技術スタック

- 🐳 **Docker**
- 🏠 **API Gateway**
- 🌐 **Kubernetes**

課題と今後

- 🛡️ **想定されるチャレンジと対策**
リスク管理と緩和策
- 🚩 **まとめと今後のアクション**
次のステップ

現在のモノリシックアーキテクチャの課題

一つの巨大なアプリケーションとして構築された現行システムが直面する問題点



スケーラビリティの制約

システム全体を一度にスケールする必要があり、特定機能のみの拡張が困難



開発の複雑性

コードベースが巨大化し、修正・機能追加の影響範囲が予測困難



デプロイの遅延

小さな変更でも全体をテスト・デプロイする必要があり、リリースサイクルが長期化



技術スタックの制約

アプリケーション全体で同一技術を使用せざるを得ず、最適な技術選択が困難

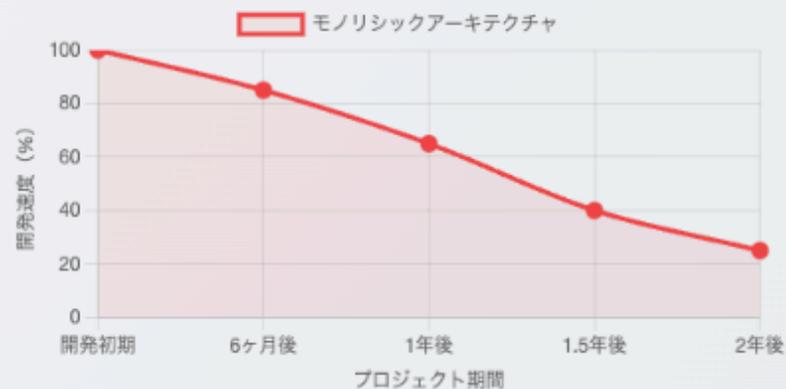


障害リスクの集中

一部の問題が全体に波及するリスクが高く、システム全体の信頼性低下につながる



プロジェクト進行に伴う開発速度の低下



マイクロサービスの基本概念と利点

経営層の視点から見たビジネス価値

マイクロサービスとは

個々のビジネス機能を担当する「小さな独立したサービス」に分割されたアーキテクチャ。各サービスは独立して開発・デプロイ・スケーリングが可能。

経営層視点での主要メリット



ビジネスの俊敏性向上

変化する市場ニーズに素早く対応する能力が強化



市場投入時間の短縮

新機能を迅速にリリースし、競争優位性を確保



サービス可用性の向上

一部障害がシステム全体に影響しない堅牢なアーキテクチャ



スケーリングの柔軟性

需要の高いサービスのみをスケールし、コスト効率を最適化



技術的負債の低減

サービス毎の段階的な技術刷新が可能となりメンテナンスコスト削減

モノリシック vs マイクロサービス

すべての機能が一つのアプリケーション

モノリシック

認証サービス

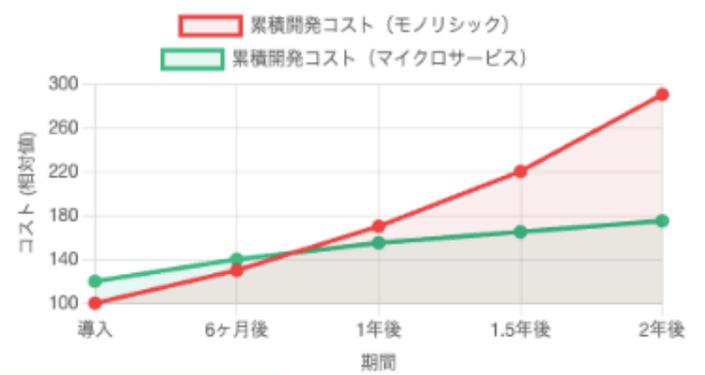
決済サービス

在庫サービス

注文サービス

マイクロサービス

マイクロサービス投資効果



💡 変化する市場環境での競争力を維持するために、ビジネスの俊敏性が重要です

マイクロサービスの技術的利点

開発チーム視点での実装メリットと技術詳細

独立したデプロイサイクル

各サービスを個別にデプロイ可能。リリース頻度向上と安全性確保の両立。

- CI/CDパイプラインを各サービスごとに最適化

障害隔離と回復力

サーキットブレーカーやバルクヘッドパターンによる障害の局所化。

```
@CircuitBreaker(  
  name = "paymentService",  
  fallbackMethod = "fallbackPayment"  
)  
public PaymentResponse processPayment() {  
  // 通常の決済処理  
}
```

データ分散管理

サービスごとの独立したデータストアにより、拡張性向上とパフォーマンス最適化

セキュリティの細分化

マイクロサービスごとに異なるセキュリティポリシーを適用可能

モニタリングと可観測性

分散トレーシング、メトリクス収集、ログ集約によるシステム全体の可視化

最適な技術スタックの選択

各マイクロサービスに最適な言語・フレームワークを選択可能。

- Java
- Node.js
- Python
- 多様なDB

API設計の自由度

REST、GraphQL、gRPCなど目的に応じた通信方式の選択が可能。

- REST API
- GraphQL
- gRPC
- WebSockets

- API Gateway による統合と管理

アーキテクチャ比較：モノリシック vs マイクロサービス

モノリシックアーキテクチャ



単一のコードベースに
すべての機能が含まれる

VS

マイクロサービスアーキテクチャ



小規模で独立したサービスの集合

各側面での比較

🔧 スケーラビリティ	× 全体をスケールする必要あり	✓ 個別サービスを独立してスケール可能
🚀 デプロイ	× 全体を一度にデプロイ	✓ 個別サービス単位でデプロイ可能
</> 複雑性	✓ シンプルな構成・開発	× 分散システム特有の複雑さ
🛡️ 障害分離	× 1箇所の障害が全体に影響	✓ 障害が局所化される
👥 チーム編成	× 大規模チーム、調整複雑	✓ 小規模チームの自律性
🔧 メンテナンス	× 時間経過で複雑化・硬直化	✓ 個別更新・柔軟性維持

各アーキテクチャの特性比較



どのような状況に向いているか

🔧 モノリシック向き

- 小規模なアプリケーション
- 少人数の開発チーム
- プロトタイプ・MVP開発
- リソースが限られている場合

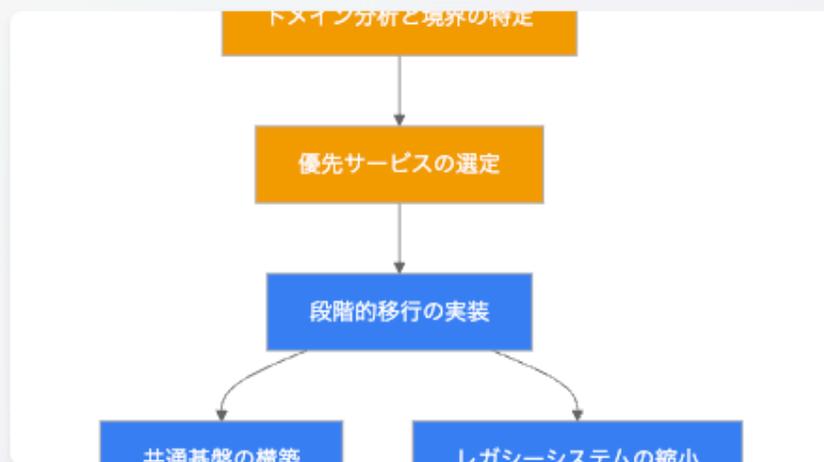
🚀 マイクロサービス向き

- 大規模なエンタープライズシステム
- 複数チームでの並行開発
- 高可用性が求められるシステム
- 段階的な技術更新が必要なシステム

移行アプローチ - 全体像

モノリシックからマイクロサービスへの段階的な移行戦略

移行戦略のフレームワーク



✂ スラングラーパターン

既存システムを徐々に「絞め殺す」ように段階的に置き換えていく手法

🏗 ブランチバイ抽象化

抽象レイヤーを導入し、実装を新旧で切り替え可能にする手法

🗄 DMSパターン

データベース、マイクロサービス、スキーマの段階的移行手法

🔗 ドメイン駆動設計

ビジネスドメインに基づきサービス境界を定義する方法論

フェーズ別アプローチ

1

準備・分析フェーズ

- 🔍 現行システム分析とボトルネック特定
- 🌟 ドメイン境界と分割候補の特定
- 🛠 技術スタックと環境の選定

📅 期間：2-3ヶ月

2

パイロット実装フェーズ

- 🏗 最初のマイクロサービス抽出と実装
- ✂ CI/CD・コンテナ基盤の構築
- 🔗 API Gateway・サービスメッシュの導入

📅 期間：3-6ヶ月

3

段階的拡大フェーズ

- 🔗 順次サービス抽出と並行運用
- 🗄 データ分割・移行の継続
- 📈 モニタリングと可観測性の強化

📅 期間：6-18ヶ月

4

最適化・完全移行フェーズ

- 📈 パフォーマンス最適化
- 🗄 レガシーシステムの完全撤去
- 🔗 継続的な改善と運用体制確立

📅 期間：継続的



成功の鍵

ビジネス価値の高い機能から段階的に移行し、各フェーズで検証と改善を繰り返すアプローチが重要です

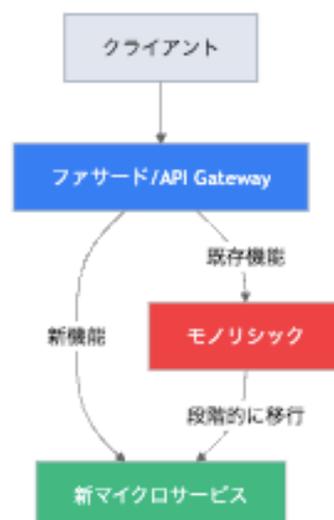
移行アプローチ - 段階的実装

ストラングラーパターンを活用した実践的マイクロサービス移行手法

ストラングラーパターン実装

段階的移行の流れ

徐々に新システムに機能を移行



段階的移行のタイムライン例

- 月1: 環境準備
API Gateway導入、認証サービス抽出
- 月3: 優先機能移行
カタログサービス抽出・旧機能と並行稼働
- 月6: コア機能移行
注文・決済サービス抽出、データ分離
- 月9-12: 完全移行
残機能移行、モノリシック縮小
- 月18: 最適化
モノリシック廃止、マイクロサービス最適化

段階的移行の実践ポイント

- 👤 **小さく始める**
リスクの低い周辺機能から始め、成功パターンを確立
- 🔄 **ロールバック計画**
各段階で問題発生時の戻し戦略を用意

- 👥 **エンドユーザー通時的に**
ユーザー体験を損なわない移行計画の実施
- 👤 **チーム再編**
マイクロサービス志向のチーム構成への移行

- 📊 **計測と検証**
移行前後のパフォーマンスを比較・検証
- 🔍 **観測性確保**
分散トレーシング・ログ集約の早期導入

サービス抽出の優先順位付け

最優先 境界が明確なサービス

ビジネス上の責務が明確に分かれている機能から抽出（認証、決済など）

📌 リスクが低く、成功体験を得やすい機能を先行

中優先 ビジネス価値の高い機能

変更頻度が高く、ビジネス競争力に直結する機能を優先的に抽出

📌 早期に価値を示せる機能を選定

実例 APIゲートウェイ活用例

```
// API Gateway設定例 (Kong, NGINX等)
location /api/auth/ {
    proxy_pass http://auth-service/; # 新マイクロサービス
}

location /api/catalog/ {
    # A/Bテストで徐々に移行
    if ($random_number < 0.3) {
        proxy_pass http://new-catalog-service/;
    }
    else {
        proxy_pass http://monolith/catalog/; # 既存モノリシック
    }
}
```

データ戦略 データ分離アプローチ

- 1 **データベースビュー**
初期段階で読み取り専用ビューを活用
- 2 **データ同期**
CDC/イベントを使った二重書き込み
- 3 **完全分離**
データの完全な所有権移行

使用技術 - Docker概要

コンテナ技術でマイクロサービスをパッケージング

Dockerとは

アプリケーションとその依存関係を「コンテナ」としてパッケージ化し、環境を問わず一貫して実行できるプラットフォーム



コンテナの特徴

軽量で起動が速く、ホストOSのカーネルを共有しながらも隔離された実行環境を提供

Dockerの主要コンポーネント



Docker Engine

コンテナの実行環境



Dockerfile

イメージ構築の定義ファイル



Docker Image

コンテナの実行に必要なスナップショット



Docker Compose

複数コンテナ定義・管理ツール



Docker Hub

イメージ共有リポジトリ



Docker Network

コンテナ間通信の管理

マイクロサービスにおけるDockerのメリット



環境の一貫性

開発・テスト・本番環境の差異を最小化し、「自分の環境では動く」問題を解消



迅速なデプロイ

マイクロサービス単位の独立した構築・デプロイが容易になり、CI/CDパイプラインとの統合も簡素化



リソース効率

軽量のコンテナ技術により、サーバーリソースを効率的に活用し、高密度のデプロイが可能



技術スタック多様化

各マイクロサービスに最適な言語・ライブラリの選択が可能に



スケーラビリティ

需要に応じたコンテナの水平スケーリングが容易に実現



分離とセキュリティ

サービス間の適切な分離によりセキュリティ向上と障害影響の局所化

コンテナと仮想マシンの比較



Dockerfile例

```
FROM node:16-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

```
docker run -p
8080:3000 my-service
```

```
docker build -t
my-service .
```

使用技術 - Kubernetes概要

コンテナオーケストレーションでマイクロサービスを効率的に管理

Kubernetesとは

コンテナ化されたアプリケーションのデプロイ、スケーリング、管理を自動化するオープンソースのコンテナオーケストレーションプラットフォーム



Kubernetesの役割

多数のコンテナの配置・負荷分散・自動復旧・スケーリングを管理し、宣言的な状態管理を実現

主要コンポーネント



Control Plane

クラスター全体を管理



Node

コンテナを実行する物理/仮想マシン



Pod

最小デプロイ単位



Service

Pod群にアクセスするための抽象化



Deployment

Pod数・状態を宣言的に管理



Ingress

HTTP/Sルーティング管理

マイクロサービス管理に重要なリソース



名前空間 (Namespace)

サービスごとの論理的な分離とリソース制限



ConfigMap / Secret

設定情報と秘密情報の外部管理



Horizontal Pod Autoscaler

負荷に応じた自動スケーリング

マイクロサービスにおけるK8sの利点

自己修復機能

障害時に自動検知・再起動でサービス可用性向上

スケラビリティ

各マイクロサービスを独立してスケール

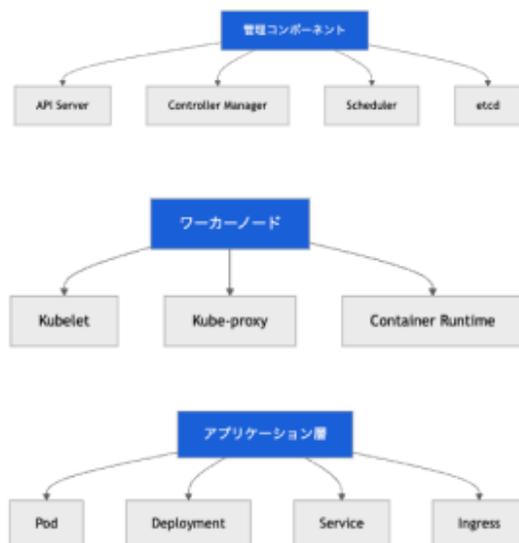
ゼロダウンタイム更新

ローリングアップデートでサービス継続

セキュリティ分離

ネットワークポリシーでマイクロサービス間通信制御

Kubernetesアーキテクチャ



設定例 (マイクロサービス用)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: payment-service
  template:
    metadata:
      labels:
        app: payment-service
    spec:
      containers:
        - name: payment-service
          image: mycompany/payment-service:1.0
          ports:
            - containerPort: 8080
          resources:
            limits:
              cpu: "500m"
              memory: "512Mi"
          readinessProbe:
            httpGet:
              path: /health
              port: 8080
```

使用技術 - API Gateway概要

マイクロサービスの玄関口としての統合ポイント

API Gatewayとは

マイクロサービスへのアクセスを一元管理するエントリーポイントとして機能し、クライアントとサービス間の仲介役を担うコンポーネント



API Gatewayは複数のマイクロサービスへのリクエストを統合し、単一のエンドポイントを提供



主要機能

- ルーティング**
リクエストを適切なマイクロサービスに転送
- 認証・認可**
アクセス権限の一元管理
- 負荷分散**
トラフィックを複数インスタンスに分散
- レート制限**
過剰リクエストからの保護
- プロトコル変換**
REST、GraphQL、gRPC間の変換
- モニタリング**
API使用状況の監視と分析

実装例 (Kong Gateway)

```
# サービス定義
services:
- name: auth-service
  url: http://auth-service:8080
  routes:
  - paths:
    - /api/auth
    strip_path: true

- name: order-service
  url: http://order-service:8080
  routes:
  - paths:
    - /api/orders
    strip_path: true

# レート制限プラグイン
plugins:
- name: rate-limiting
  config:
  minute: 60
  policy: local
```

マイクロサービスでの利点

- クライアント側の簡素化**
複雑なバックエンド構造を隠蔽
- セキュリティ統合**
共通ポリシーを一箇所で適用
- サービス疎結合**
クライアントとサービス間の依存性低減
- 横断的関心事の集約**
ログ、監視、認証を一元管理

主要なAPI Gateway製品

Kong OSS / Enterprise	AWS API Gateway クラウドマネージド	Spring Cloud Gateway Java環境向け
NGINX 高性能/軽量	Azure API Management Azureサービス	Tyk OSS / Enterprise

想定されるチャレンジと対策 →

マイクロサービス移行で直面する課題と解決アプローチ



まとめと今後のアクション

主要な成功要因

- ✓ 段階的な移行アプローチ
- ✓ ビジネス価値を最大化する機能優先
- ✓ クロスファンクショナルチーム体制
- ✓ CI/CD・自動化の徹底
- ✓ 経営層のコミットメントと理解

今後のアクションプラン

- 1 PoC プロジェクト始動**
境界明確な単一サービスを選定 (2-3ヶ月)
- 2 共通基盤整備**
CI/CD、監視、ログ集約基盤の構築
- 3 技術トレーニング**
コンテナ技術、クラウドネイティブ開発研修
- 4 段階的移行開始**
優先度の高いサービスから順次移行

最終目標

「変化に強く、安定性と拡張性を兼ね備えた次世代システム基盤の実現」

ビジネス俊敏性

技術的柔軟性

運用効率向上