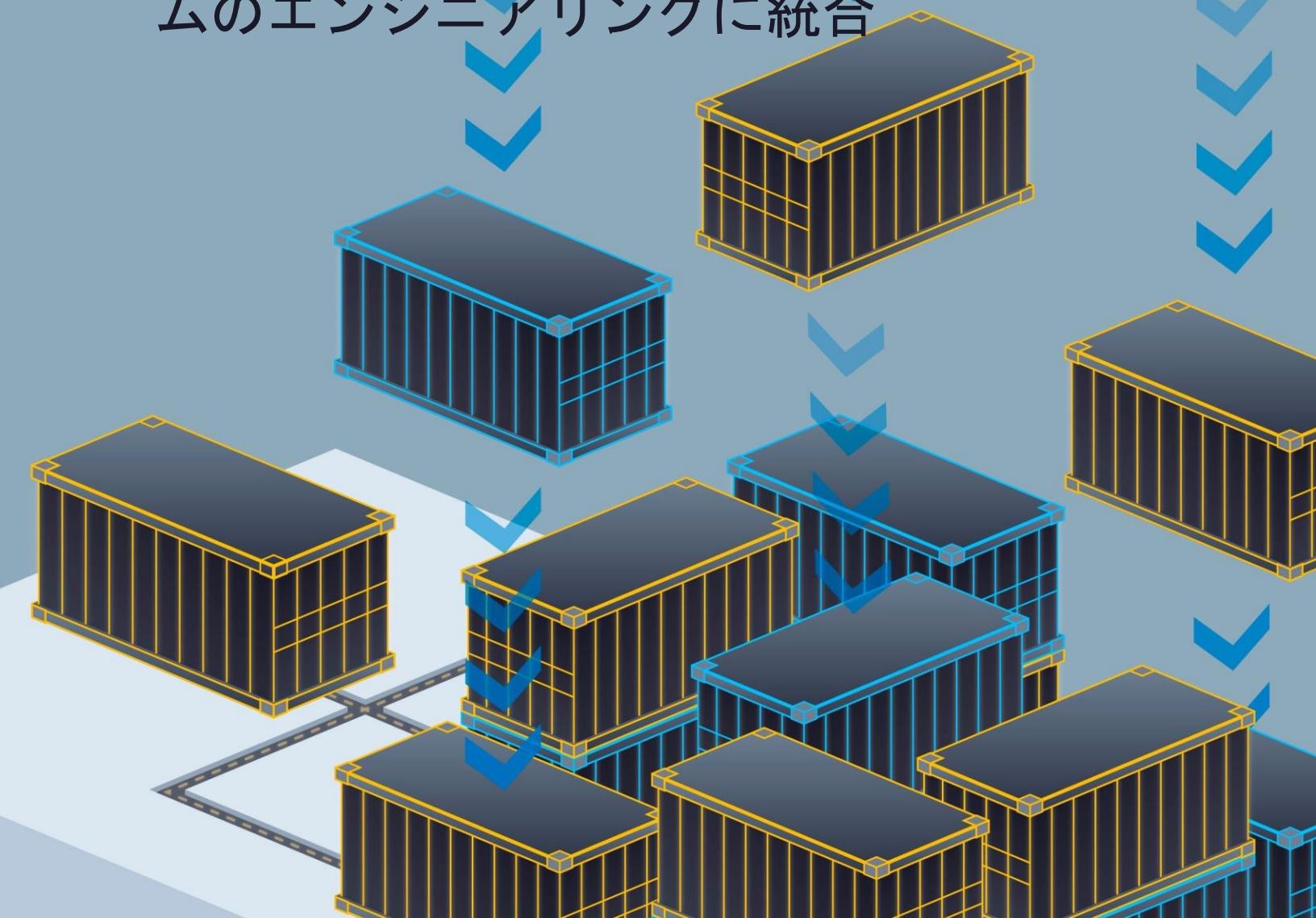
CSAJC cloud security alliance SM Japan Chapter

CSA cloud security alliance®

# 安全なマイクロサービスアーキテクチャ実装のためのベストプラクティス

## マイクロサービスのセキュリティに関する考慮事項を信頼できるセキュアシステムのエンジニアリングに統合



The permanent and official location for Cloud Security Alliance Application Containers and Microservices research is:
https://cloudsecurityalliance.org/research/working-groups/containerization/

# Acknowledgments


## Editor:

Anil Karmel

## Team Leaders/Authors:

Marina Bregkou
Aradhna Chetal
Mark Yanalitis
Michael Roza

## Authors:

Yeu Wen Mak
Vishwas Manral
Ramaswamy Chandramouli
Shachaf Levy
James Turrell
William Gafford
Alex Rebo
Jane Odero Greene
Atul Chaturvedi
Michael Green
Amit Maheshwari
Vrettos Moulos
Ricardo Ferreira

## CSA Staff:

Hillary Baron
Marina Bregkou

## Reviewers:

Marina Bregkou
Rohit Sansiya
Michael Roza
Vrettos Moulos
Prabath Siriwardena
Suhas Bhat
Carlos Villavicencio Sánchez
Ricardo Ferreira
David Cantrell

# 目次

# 日本語版提供に際しての告知及び注意事項

本書「安全なマイクロサービスアーキテクチャ実装のためのベストプラクティス」は、Cloud Security Alliance (CSA)が公開している「Best Practices in Implementing a Secure Microservices Architecture」の日本語訳です。本書は、CSA ジャパンが、CSA の許可を得て翻訳し、公開するものです。原文と日本語版の内容に相違があった場合には、原文が優先されます。翻訳に際しては、原文の意味および意図するところを、極力正確に日本語で表すことを心がけていますが、翻訳の正確性および原文への忠実性について、CSA ジャパンは何らの保証をするものではありません。
この翻訳版は予告なく変更される場合があります。以下の変更履歴（日付、バージョン、変更内容）をご確認ください。

変更履歴

| 日付 | バージョン | 変更内容 |
|---|---|---|
| 2020 年 11 月 17 日 | 日本語版 1.0 | 初版発行 |



# CSAジャパン成果物の提供に際しての制限事項

日本クラウドセキュリティアライアンス（CSA ジャパン）は、本書の提供に際し、以下のことをお断りし、またお願いします。以下の内容に同意いただけない場合、本書の閲覧および利用をお断りします。

１．　責任の限定

CSA ジャパンおよび本書の執筆・作成・講義その他による提供に関わった主体は、本書に関して、以下のことに対する責任を負いません。また、以下のことに起因するいかなる直接・間接の損害に対しても、一切の対応、是正、支払、賠償の責めを負いません。

(1) 本書の内容の真正性、正確性、無誤謬性
(2) 本書の内容が第三者の権利に抵触しもしくは権利を侵害していないこと
(3) 本書の内容に基づいて行われた判断や行為がもたらす結果
(4) 本書で引用、参照、紹介された第三者の文献等の適切性、真正性、正確性、無誤謬性および他者権利の侵害の可能性

２．　二次譲渡の制限

本書は、利用者がもっぱら自らの用のために利用するものとし、第三者へのいかなる方法による提供も、行わないものとします。他者との共有が可能な場所に本書やそのコピーを置くこと、

利用者以外のものに送付・送信・提供を行うことは禁止されます。また本書を、営利・非営利を問わず、事業活動の材料または資料として、そのまま直接利用することはお断りします。

ただし、以下の場合は本項の例外とします。

(1) 本書の一部を、著作物の利用における「引用」の形で引用すること。この場合、出典を明記してください。
(2) 本書を、企業、団体その他の組織が利用する場合は、その利用に必要な範囲内で、自組織内に限定して利用すること。
(3) CSA ジャパンの書面による許可を得て、事業活動に使用すること。この許可は、文書単位で得るものとします。
(4) 転載、再掲、複製の作成と配布等について、CSA ジャパンの書面による許可・承認を得た場合。この許可・承認は、原則として文書単位で得るものとします。

３．　本書の適切な管理

(1) 本書を入手した者は、それを適切に管理し、第三者による不正アクセス、不正利用から保護するために必要かつ適切な措置を講じるものとします。
(2) 本書を入手し利用する企業、団体その他の組織は、本書の管理責任者を定め、この確認事項を順守させるものとします。また、当該責任者は、本書の電子ファイルを適切に管理し、その複製の散逸を防ぎ、指定された利用条件を遵守する（組織内の利用者に順守させることを含む）ようにしなければなりません。
(3) 本書をダウンロードした者は、CSA ジャパンからの文書（電子メールを含む）による要求があった場合には、そのダウンロードしまたは複製した本書のファイルのすべてを消去し、削除し、再生や復元ができない状態にするものとします。この要求は理由によりまたは理由なく行われることがあり、この要求を受けた者は、それを拒否できないものとします。
(4) 本書を印刷した者は、CSA ジャパンからの文書（電子メールを含む）による要求があった場合には、その印刷物のすべてについて、シュレッダーその他の方法により、再利用不可能な形で処分するものとします。

４．　原典がある場合の制限事項等

本書が Cloud Security Alliance, Inc.の著作物等の翻訳である場合には、原典に明記された制限事項、免責事項は、英語その他の言語で表記されている場合も含め、すべてここに記載の制限事項に優先して適用されます。

５．　その他

その他、本書の利用等について本書の他の場所に記載された条件、制限事項および免責事項は、すべてここに記載の制限事項と並行して順守されるべきものとします。本書およびこの制限事項に記載のないことで、本書の利用に関して疑義が生じた場合は、CSA ジャパンと利用者は誠意をもって話し合いの上、解決を図るものとします。

その他本件に関するお問合せは、info@cloudsecurityalliance.jp までお願いします。

# 日本語版作成に際しての謝辞

「安全なマイクロサービスアーキテクチャ実装のためのベストプラクティス」の日本語訳は、CSAジャパン会員の有志により行われました。
作業は全て、個人の無償の貢献としての私的労力提供により行われました。なお、企業会員からの参加者の貢献には、会員企業としての貢献も与っていることを付記いたします。
以下に、翻訳に参加された方々の氏名を記します。（氏名あいうえお順・敬称略）

伊賀　誠
太田　吏城
勝見　勉
神保　冬和子
高瀬　一彰
成川　達也
成田　和弘
三浦　貢造
満田　淳
諸角　昌宏

# 1.0 マイクロサービスアーキテクチャの概要

この章では、マイクロサービスアーキテクチャの進化の背景、過去のアーキテクチャと比較した利点と、構成とセキュリティの観点からの新しい課題について説明します。

アプリケーションシステムの最も初期のアーキテクチャは、アプリケーション全体が単一プロセスとして実行されるように設計され、”サーバ”と呼ばれるリソース集約型コンピューティングプラットフォームでホストされる”モノリス”です。アプリケーションは異なるモジュールとして構成されている場合がありますが、どのモジュールでも、アプリケーション全体の再コンパイルと再配備が必要です。モジュール間の通信は、ローカルプロシージャ/関数呼び出しによって実行されます。

アプリケーションアーキテクチャの次の進化は、”サービス指向アーキテクチャ”（SOA）です。 SOA では、ソリューションの全範囲（ビジネスプロセスのサポートなど）は、サービスと呼ばれる複数の部分またはコンポーネントに分割されます。このアプローチにより、アプリケーション全体ではなく特定のサービスに操作を制限できるため、アプリケーション全体の開発、保守、および展開が容易になります。ただし、サービスが連携して必要なソリューションを提供するには、エンタープライズ・サービス・バスや通信プロトコル（Webサービスなど）のような重いミドルウェアを使用する必要があります。これらのミドルウェアコンポーネントのセキュリティを確保することは、複雑なプロセスになります。さらに、これらのミドルウェアコンポーネントによって提供される接続の性質により、インターフェースなどの個々のサービスの属性を厳密に制御し、サービス間の密接な結合を作成する必要があります。

マイクロサービスアーキテクチャの設計は、個々のマイクロサービスが RESTなどの軽量プロトコルを使用して相互に通信できるようにすることで、SOA の制限に取り組むことを目的としています。さらに、個々のマイクロサービス間の疎結合による展開と独立したスケーラビリティに加えて多様性を考慮して、個々のマイクロサービスはそれらに最適なプラットフォームで開発できます。ただし、このアプローチには、コンポーネント数の増加による攻撃対象面の増加や、場所が変更されるサービスインスタンスの動的な性質に対応したセキュアなサービス検出方法など、新しいセキュリティ上の課題があります。

NIST SP800-180は、マイクロサービスを”アプリケーションコンポーネントを、いかなるベンダー、製品、技術からも独立した標準的な通信プロトコルや 明確に定義されたAPIを利用して相互に通信する、自己完結したサービスの疎結合のパターンに、アーキテクチャ的に分解した結果生まれる基本要素”と定義しています。マイクロサービスは、サービスではなく機能を中心に構築され、通常はアプリケーションコンテナ内に展開されます[NIST SP800-180]。

マイクロサービスアーキテクチャ（MSA）は、N 個のマイクロサービスの集まりで構成され、”システム オブ システムズ”として一緒に動作します。このアーキテクチャは、ビジネスまたは技術的なニーズに応じてサービスを独立して進化させることにより、開発の俊敏性とビジネスの柔軟性を実現します。ただし、ほとんどの組織におけるマイクロサービスへの移行において、シングルノードシステムの運用から複雑なマルチノードの”システム オブ システムズ”という分散アーキテクチャへの大幅な移行が行われるため、このアーキテクチャは大きな課題ももたらします。これは、分散システム開発の利点と制限の両方をもたらします。これは、従来のモノシリックシステムアーキテクチャでの作業に慣れている開発チームと運用チームにとって意外なことかもしれません。このアプローチは、分散システムが直面する同じセキュリテ

ィの課題の多くを継承するため、マイクロサービスのセキュリティに直接的な影響があります。これらの課題には、分散認証の管理、各サービスメッセージの機密性と真正性の確保、単一のサービス内の複数のノードからのイベントログの適切なマージおよび監視、分散エンタープライズ全体でのサービス資格情報の動的な管理およびローテーション、絶えず変化する異種環境でのパッチ適用と脆弱性管理、企業全体の攻撃対象領域の拡大の管理等があります。

モノリシックアーキテクチャとマイクロサービスアーキテクチャの論理上およびアーキテクチャ上の違いについては、セクション 1.1 で説明します。２つのアーキテクチャ間の構成要素の違いを示すアプリケーションの例も提供されます。セクション 1.2では、SOA の展望を分析し、その特性、制約、および欠陥に焦点を当て、MSAがSOAの概念の一部を拡張し、それらの欠陥に対処する方法について説明します。

セクション 1.3 では、マイクロサービスアーキテクチャの利点を述べ、SOA の利点を享受する具体的なユースケースを提供します。MSA で特定されたセキュリティと構成の課題は、このドキュメントの後半部分で取り組まれる問題の基礎となります。

# 1.1 サービス指向アーキテクチャ

SOA の概念は、古くからあります。Erl [Erl,2005] のSOA の定義では、「適切なテクノロジープラットフォームを介して実現できる」アーキテクチャパターンとされています。サービス指向の基礎は技術ではなく機能の分離という理論にあり、大きな問題がより小さく個別の関連する部分に分割されサービスを形成します。SOA パラダイムは、IT 環境でこれらのサービスを計画、開発、管理するのに役立ちます。したがって、Erl [Erl,2005] で定義されている次のSOA定義を採用します：「SOA は、オープン、アジャイル、拡張可能、相互連携、組み立て可能なアーキテクチャであり、自律型、QoS能力、ベンダーの多様性、相互運用可能、発見可能、潜在的に再利用可能なサービスで構成される」。

それでは、「サービス」、「サービス指向」、「アーキテクチャ」の定義を明確にしましょう。

サービスは、明確な目的を持った独立したビジネスの機能です [Stojanovic et al., 2004]。サービスは、請求処理などの特定のビジネスプロセスをモデルにしています。

サービス指向、[Erl, 2005; Rosen et al., 2012]は、ソフトウェアコンポーネントに統合されたロジックの集まりです。このコンポーネントは、コンポーネントの利用主体にサービスとして提供されます。ここで、サービスの利用主体は他のサービスになることもできます。これらの独立したコンポーネントは、接続されると、「顧客アカウントの管理」などの特定のビジネスプロセスの完全なサポートを提供できます。

アーキテクチャは、サブシステムとソフトウェアシステムのコンポーネント、およびそれらの間の関係の説明です。

## 1.1.1 展望

SOAのライフサイクル（モデリング、アセンブリ、デプロイメント、管理）の展開を成功させるには、SOAガバナンスとして知られているコンテキストで行います。SOAガバナンスとは、人々が責任を遂行できるように、責任の連鎖とコミュニケーション、ポリシー、測定、制御メカニズムを確立するプロセスです。

- SOAガバナンスは自身で実施するプロセスであり、購入する製品ではありません[1]。 [Oracle 2013]
- SOAガバナンス自体には、計画、定義、有効化、測定という一連のフェーズがあります。
- SOAは以下の４つの基本サービスタイプを定義します ：

**ビジネスサービス**は、コアビジネスオペレーションを定義する粗削りのサービス（coarse-grained service）です。通常、XML、Web Services Definition Language（WSDL）か、Business Process Execution Language（BPEL）で表されます。

**エンタープライズサービス**は、ビジネスサービスによって定義された機能を実装します。ビジネス要求を満たすために、アプリケーションサービスとインフラストラクチャサービスに依存しています。

**アプリケーションサービス**は、特定のアプリケーションコンテキストにバインドされたきめ細かいサービス（fine-grained service）です。これらのサービスは、専用のユーザインターフェイスから直接呼び出すことができます。

**インフラストラクチャサービス**は、認証、監査、セキュリティ、ロギングなどの非機能タスクを実装します。アプリケーションサービスまたはエンタープライズサービスから呼び出すことができます。

**SOAの特性- 内部アーキテクチャ**

SOA 設計の 10 の原則[2]：

1. 相互運用性 – サービスは、加入者がサービスを使用できるようにする標準を使用する必要があります。これにより統合が容易になります。
2. 疎結合 – サービスは相互の依存関係を最小限に抑えます。
3. ナレッジカーテン/サービスの抽象化 – サービスは、ロジックをカプセル化し外部から隠します。
4. リソース管理/サービスの再利用性 – ロジックは、再利用を最大化するようにいくつかのサービスに分割されます。
5. サービスディスカバリ – サービスは見つけられるべきであり、見つけることができます（通常はサービスレジストリにおいて）。
6. 構造的独立性/サービスの自律性 – サービスは、サービス自身が消費するあるいは依存するリソースを制御する必要があります。
7. サービス構成/構成可能性 – サービスは、大きなタスクを小さなタスクに「分割」します。
8. 粒度/サービスステートレス性 – 理想的には、サービスはステートレスでなければなりません。
9. サービス品質 – サービスは、サービスプロバイダとクライアント間のSLAを順守します。
10. 高い凝集性 – サービスは、理想的には単一のタスクに対応するか、同じモジュールの一部として類似のタスクをグループ化する必要があります。

[1] https://www.oracle.com/assets/implement-soa-governance-1970613.pdf
[2] Erl quoted 7 principles, while others today mention 10 as they are listed above as we believe all of those principles should be mentioned.

### SOA の制約

サービス指向は人によって異なることを意味することから、SOAの課題は、ビジネス、技術/エンジニアリング、運用の3つのドメイン領域で見ることができます。サービスの実装、属性、説明、およびデータ型には劇的な変化があります。したがって、サービスを効果的に管理することは問題であり続けます。サービス指向アーキテクチャシステムに関連する課題領域の初期分類を検討するために、以下の表に、3つのドメイン[3] のそれぞれに分類されたSOA実装の課題を示します：[Lund University, 2015] [Beydoun et al, 2013]

| ドメイン | 課題 |
|---|---|
| ビジネス | 標準の定義 |
| ビジネス | 要件の理解 |
| ビジネス | サービス構成 |
| 技術/エンジニアリング | セキュリティと完全性 |
| 技術/エンジニアリング | サービスの互換性 |
| 技術/エンジニアリング | レガシーソフトウェアの移行 |
| 技術/エンジニアリング | リアルタイムコミュニケーション |
| 技術/エンジニアリング | 信頼性とテスト |
| 運用 | サービスの連携 |
| 運用 | SOAの複雑性のガバナンス |
| 運用 | 監視サービス |

*表 1: SOA 課題のドメイン領域*

[3] In SOA Adoption Challenges, [Beydoun at al.], Challenges fall into two domains: Technical and Organizational-Business.

さらに、SOA にはセキュリティ上の課題があり、一般的に次のシナリオで使われることはありません：

1. **レガシーアプリケーションセキュリティ**：レガシー機能は、SOA-ベースのアダプタの助けを借りて外出しできます。それは可能ですが、設計者は機能の既存のセキュリティモデルの制約を考慮する必要があります。ここで提案されているSOAアダプタは、モデル固有の性質が理解できていない場合があります。

2. **サービスとアプリケーションの疎結合**：SOAセキュリティは、全体的なソフトウェア設計原理（例えばソリューションコンポーネントの疎結合）に違反してはなりません。サービスは、機能に対するインターフェースの提供を持続できることを目的とし、サービスの利用者に設計と実装の詳細を見えなくします。

3. **組織の境界を越えて稼働するサービス**：従来の境界セキュリティ（企業のセキュリティ境界）は、組織間のやり取りによって生じるリスクを軽減するには不十分な場合があります。エンタープライズセキュリティポリシーへのコンプライアンスを維持するには、一連の代替コントロールが必要になる場合があります。

4. **動的な信頼関係**：SOA参加者は、おそらくSOAレジストリのホスティングとメンテナンスを担当する当事者を含め、相互信頼を確立する必要があります。これらの関係は動的な性質であるため、信頼も動的な性質を持つ可能性があります。

5. **複合サービス**：サービスの構成と集約は、サービスの関連付けの2つの形式です。ただし、サービスの構成は、SOA独立性原則に反する可能性があります。したがって、サービスオーケストレーションとコレオグラフィーは、望ましいサービス関連付け戦略です。

6. **増え続ける一連の標準に準拠する必要**：SOAの標準化にとって、増え続けるセキュリティの標準と規制は懸念事項です。ただし、このコンプライアンスの必要性は、他の機能統合領域と変わりません。

7. **SOAの柔軟性**：SOAソリューションは、柔軟でカスタマイズ可能です。IT がサポートするプロセスとビジネスソリューションの市場投入までの時間を短縮できます。ポートフォリオギャップ分析、移行計画、およびアーキテクチャガバナンスは、エンタープライズアーキテクチャに変更の機会を提供し、戦略的なビジネス目標とIT目標を統合することができます。サービス指向のポートフォリオギャップ分析を活用することにより、企業の計画サイクル戦略を特定の変更イニシアチブのロードマップに変換し、その結果のロードマップの実行を管理する機会を提供できます。次に、SOAライフサイクルは、ロードマップ内の１つ以上の特定のプロジェクトのコンテキストでソリューション配信を推進します。したがって、SOAソリューションはビジネスプロセスを、重要で、パーソナライズされて、反応がはやくするために適切にカスタマイズされ、拡張される必要があります。[Simplicable、2011] [The SOA Source Book]

**SOA の欠陥：**

SOA は現代的な抽象化アプローチと見なされていますが、依然として集中管理された抽象化を前提としています。潜在的な欠陥の一部は、以下のような制約やマイクロサービスが提供するものと結び付けられます：

8. サービスバスを使用してSOAの信頼性、スケーラビリティ、通信の不一致を実現するには、マイクロサービスとは異なる特定の制限（密結合、依存プロセスなど）が伴います。

9. SOAはパフォーマンスに影響を与えると認識されています。ただし、これは適切に設計されたサービス自体ではなく、プロトコルに一部起因しています。ただし、オーケストレーション/コレオグラフィーが不十分なサービスは、SLA/QSAに違反する可能性があります。

10. 確立されたイミュータブルサービスインターフェイスは、サービスの使いやすさにとって不可欠です。したがって、サービスライフサイクルが適切なガバナンスプロセスによって制御されることが期待されます。

11. SOAのテストは、SLA/QSAの対象となる利用者の目標が、OLA を維持するサービスコンポーネントの能力に大きく依存する場合、他の分散ソリューションと同様に複雑になる可能性があります。

ある意味では、SOA は、アーキテクトが DevSecOpsプロセスと関連する自由度を設計しているときに、モノリシックからマイクロサービスアーキテクチャ（MSA）までの他の一般的なアーキテクチャパターン（ストラングラー ファサード（strangler facades）[4]、破損対策レイヤ（anti-corruption layer）[5] など）の実装を支援する足掛かりと見なすことができます。このように、サービスの境界はすでに明確に定義されており、サービスは独立してマイクロサービスアーキテクチャに展開できます。

[4] Fowler M., (2004): StranglerFigApplication. https://martinfowler.com/bliki/StranglerFigApplication.html

[5] Kalske (2017), University of Helsinki: Transforming monolithic architecture towards microservice ar-chitecture. https://core.ac.uk/download/pdf/157587910.pdf

### 1.1.2 マイクロサービスは、どのように SOAを拡張するか

マイクロサービスは、SOAのプラクティスを拡張し、従来のSOAの欠陥を克服する新しいアーキテクチャのアプローチです。 [Haddad, CIOReview]

コンピュータサイエンスの観点からではなく、実装する開発者の観点から、マイクロサービスアーキテクチャ（MSA）にはまったく新しいものは導入されていませんが、マイクロサービスアーキテクチャへの移行は非常に急進的なものになる可能性があります。マイクロサービスアーキテクチャは、SOAを論理的に進化させたものであり、最新のビジネスユースケースをサポートします。

SOAと同様に、マイクロサービスアプローチは、個々のアプリケーションを複数の個別のサービスに分解します。単一責任の原則（SRP ： Single Responsibility Principle）[6] を使用したアトミックなマイクロサービスによって実装される機能ユニットには、有用なビジネスプロセスアクティビティを形成し、統合フローを作成し、ユーザエクスペリエンスを提供することが含まれます。MSA は、コンテキスト境界、バルクヘッド、サーキットブレーカパターンで、SOAパターンの集まりを強化します。

追加の原則とパターンは、チームが自律的、疎結合、回復力、および疎結合のソリューションを提供するのに役立ちます。これを説明する用語は、クラウドネイティブソフトウェア開発[7]と呼ばれます（セクション2を参照）。

マイクロサービスアプローチは、上記の追加の原則とプラクティスによりSOAを制約します。これには、コンテキストマッピング、疎結合/高凝集、シェアード・ナッシング・アーキテクチャ、動的展開、並列開発が含まれます。

[6] Robert C. Martin (2005): https://drive.google.com/file/d/0ByOwmqah_nuGNHEtcU5OekdDMkk/view

[7] These solutions are engineered to benefit from a cloud-native architecture. One of the four underlying pillars within each cloud application is: Microservices. (Along with Containers, Dynamic Orchestration, and Continuous Delivery).

以下は、SOAとマイクロサービスの主な違いで、マイクロサービスがどのようにSOA機能を拡張しているかを示しています[BMC Software, 2017] [Cerny et al., 2017]：

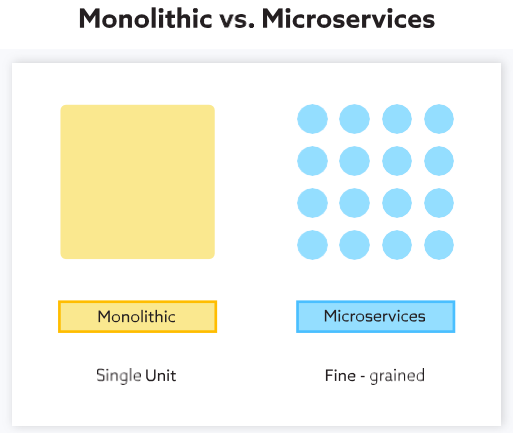
| SOA | マイクロサービス |
|---|---|
| 「可能な限り共有する」 アーキテクチャアプローチ | 「最小限の共有」アーキテクチャアプローチ |
| ビジネス機能の再利用に重きを置きます | 「コンテキスト境界」 の概念に重きを置きます |
| 共通のガバナンスと標準を提供 | コラボレーションと選択の自由に重点を置き、より緩和されたガバナンスを実現 |
| 通信にエンタープライズサービスバス（ESB）の使用 | 複雑でなくシンプルなメッセージングシステムを使用 |
| 複数のメッセージプロトコル | HTTP/REST & AMQPまたはJMSなどの軽量プロトコル |
| I/Oを処理するためのオーバーヘッドの多いマルチスレッド | 非ロックI/O処理用のシングルスレッド（前提条件ではない[8]）イベントループ機能 |
| コンテナの使用はあまりポピュラーではない | コンテナはマイクロサービスのパッケージングに便利 |
| システム変更にモノリスを変更することが必要 | システム変更に新しいサービスを作成 |
| アプリケーションサービスの再利用性を最大化 | 他のサービスから分離するために独自のデータベースを持つ疎結合自己完結型サービス[9] |
| DevOps / 継続的デリバリの主流ではない | DevOps /継続的デリバリに重点を置く |

*表２：SOA 対 マイクロサービス*

[8] https://www.infoq.com/articles/engstrand-microservice-threading/
[9] https://medium.com/@raycad.seedotech/monolithic-vs-microservice-architecture-e74bd951fc14

# 1.2 モノリシックアーキテクチャとマイクロサービスアーキテクチャの比較

アーキテクチャとして、モノリシックアプリケーションは単一のコンピュートノードとして設計されています。したがって、ノードはシステム全体またはアプリケーションの状態を認識しています。マイクロサービス環境では、アプリケーションは、それぞれがサービスを提供する複数のノードのセットとして設計されています。




## 1.2.1 論理的な違い

概念的には、モノリシックアーキテクチャが、全体としてデプロイされるアプリケーションを生成することを意味するのに対し、マイクロサービスベースのアプリケーションは、サービスまたはマイクロサービスと呼ばれる、自己完結型で疎結合な複数の実行ファイルであることを意味します。個々のサービスは独立してデプロイすることができます。アプリケーション全体の特定の機能に変更が必要な場合、再デプロイされる前に少なくともアプリケーション全体の再リンクと、場合によっては全体の再テストが行われます(どのような変更も、アプリケーションの重要度に応じて、いくつかのテストレベルを必要とするため)。しかし、マイクロサービスの場合、サービスの独立した性質により、あるサービスの変更が他のサービスの機能に論理的に影響を与えないことが保証されているため、関連するサービスのみが変更され、再デプロイされます。モノリシックアプリケーションの場合、（ユーザ数の増加やアプリケーションの使用頻度の増加による）ワークロードの増加に対応したスケーリングは、通常、負荷をより均等に分散させるために、アプリケーション全体を新しいインスタンスで実行することに限られます。

しかし、マイクロサービスでは、これまでとは異なるアプローチでのスケーリングが可能になります。リソースの増加分は、パフォーマンスが望んでいるレベルに達していないサービスに選択的に適用することができるため、スケーラビリティの取り組みに柔軟性を与えることができます。

モノリシックなアプリケーションの中には、モジュール式に構築されているものもありますが、意味的なモジュール性や論理的なモジュール性を持たないものもあります。モジュール式に構築されたアプリケーションは、異なるベンダーから提供された多数のコンポーネントやライブラリから構築され、いくつかのコンポーネント（データベースなど）はネットワーク上に分散されているかもしれないということです。

このようなモノリシックアプリケーションでは、API（アプリケーション・プログラミング・インターフェース）の設計と仕様は、マイクロサービスアーキテクチャのそれと似ているかもしれません。しかし、このようなモジュラー設計のモノリシックアプリケーション（古典的なモジュラー設計と呼ばれることもある）とマイクロサービスベースのアプリケーションの違いは、後者の場合、個々のAPIはネットワークに公開されており、それゆえに独立して呼び出し可能で再利用可能であるということです。

もう一つの重要な違いは、モノリスでモジュールを使用することは、分離が弱い形態だということです。というのは、モジュールは（一般的には）すべて同じプロセスで実行され、開発者はモジュール間の依存関係をより簡単に作ることができるため、凝集度が低く（モジュールの独立性が低く）なります。それに比べて、マイクロサービスはネットワークの境界を介して強力な分離を確立しますが、これは通常、マイクロサービスが別々の開発チームによってメンテナンスされていることによっても強化されており、マイクロサービス間の依存関係をより難しくし、結果として、よりクリーンな責任の分離をもたらします。モノリスなモジュールの変更には、まだモノリスの成果物全体の再生成/再テスト/再デプロイが必要となるため、このようなクリーンな分離こそがマイクロサービスを独立して進化させることを可能にしています。[Newman, 2015]

上述したモノリシックアプリケーションとマイクロサービスベースのアプリケーションの違いは、以下の表3のようにまとめられます。

| モノリシックアプリケーション | マイクロサービスベースのアプリケーション |
|---|---|
| まとめてデプロイしなければならない | 単独もしくは選択されたサービスのデプロイ |
| アプリケーションのごく一部に変更があった場合、アプリケーション全体の再配備が必要になる可能性があります | 変更されたサービスだけを再配備する必要があります。 |
| スケーラビリティとは、ロードバランサーがアプリケーションの必要に応じてリソースを増減させ、複数のポイントでバランシングできるようにすることで、必ずしもアプリ全体でバランシングする必要はありません。 | 個々のサービスは、より多くのリソースを割り当てることで、選択的にスケールアップすることができます。 |
| コンポーネントの呼び出しはローカル | ネットワークに公開されたサービス呼び出しは、より広範な配備と独立性を可能にします[10] |
| モジュールはすべて同じプロセスで実行されるため、分離が弱い形態 | ネットワークの境界を介した強力な分離 |

*表3：モノリシックアプリケーションとマイクロサービスベースのアプリケーションの論理的な違い*

[10] ここで明確にしておかなければならないのは、これはメリットとデメリットが同時にあるということです。なぜなら、マイクロサービスアーキテクチャに移行するということは、本質的には分散型のネットワーク化されたシステム・オブ・システムズ・アーキテクチャに移行することであり、柔軟性の向上に伴って複雑さが大幅に増大するからです。MSAのベストプラクティスの多くは、複雑さを管理し、またはその制限内で受け入れて作業するための取り組みです。

### 1.2.1.1 論理的な違いを説明するためのアプリケーション例

上で説明した論理的な違いを説明するために、小さなウェブショップやオンライン小売アプリケーションの例を見てみましょう。このアプリケーションの主な機能は以下の通りです。

- ウェブショップが提供する商品のカタログを、商品画像、型番、商品名、単価で表示するモジュール;
- その取引で注文されたすべてのアイテムを含む送り状を作成するように、顧客に関する情報（名前、住所など）と注文の詳細（カタログの商品名、数量、単価、合計価格など）を収集し、顧客の注文を処理するためのモジュール；
- 船荷証券（様々な品目とその品量を含む出荷パッケージ数）、発送の種類（即日集荷、翌日出荷など）、発送先住所などを指定して注文品を出荷準備するためのモジュール；
- クレジットカードや銀行口座での支払い機能を内蔵した請求書発行モジュール。

上記のウェブショップアプリケーションを、モノリスとして設計した場合とマイクロサービスベースとして設計した場合の違いを以下の表4に示します。

| アプリケーション構成 | モノリス | マイクロサービスベース |
|---|---|---|
| 機能モジュール間通信 | すべての通信は、プロシージャコールや内部データ構造(ソケットなど)の形式で行われます。– 注文処理を扱うモジュールは、配送機能を処理するモジュールにプロシージャコールを行い、正常に完了するのを待ちます。 | 配送機能と注文処理機能はそれぞれ独立したサービスとして設計されています。通信は、Web プロトコルを使用してネットワークを介して API コールとして行われます。注文処理サービスは、イベントをサブスクライブしている配送アプリケーションが、非同期にピックアップすることができるように、出荷する注文の詳細をメッセージキューに入れます。 |
| 変更・機能強化への対応 | 請求書発行モジュールは、デビットカードによる顧客の支払いを可能にするための機能強化が必要です - 必要な変更を行った後、アプリケーション全体を再コンパイルして再デプロイする必要があります。 | 請求書発行機能は別のマイクロサービスとして設計されているため、そのサービスのみを再コンパイルし再デプロイする必要があります。 |

| | | |
|---|---|---|
| アプリケーションのスケーリング – 追加リソースの割り当て | 注文処理機能は、出荷機能や請求書発行機能に比べてトランザクション時間が長くなるため、より多くのメモリ/CPUを搭載したサーバを使用する垂直方向のスケーリングをアプリケーション全体に導入する必要があります。 | 注文処理のマイクロサービスが展開されているハードウェア（ハードウェアの限界を考慮して）に増加したリソースを割り当てれば十分です。また、より良い負荷分散のために、受注処理マイクロサービスのインスタンス数を増やすことも可能です。 |
| 開発・デプロイメント戦略 | 開発は開発チームが担当し、（必要なテストを行った後）デプロイのための適切なリソースの割り当てを処理するインフラストラクチャチームにデプロイタスクを引き継ぎます。 | 開発からデプロイまでの完全なライフサイクルは、マイクロサービスごとに1つのDevSecOpsチームが担当します。マイクロサービスは、比較的小さなモジュールであり、単一の機能を持ち、その機能に最適なプラットフォーム（OS、言語ライブラリ、IDE）で構築されているためです。 |

*表4：アプリケーションのモノリスとマイクロサービスのアーキテクチャ設計*

### 1.2.2 アーキテクチャの違い

このドキュメントでは、SOAとマイクロサービスの違いは、[Zimmerman 2016]で主張されているように、アーキテクチャのスタイルには関係ないという見解を取ります。したがって、MSAとSOAの基本的なアーキテクチャの原則は同一であり、次のようなものが含まれます: アプリケーションをサービスと呼ばれる技術的に不可知な自己完結型モジュールに分解します。サービスは、相互運用性（リンクライブラリを介さずにサービスが直接相互に呼び出す）、高凝集度(サービスは単一または性質が似ているいくつかのタスクのみに対応する)と疎結合(依存性を持たない-あるサービスの変更が他のサービスの変更を必要としない状況)のような技術的な概念を具現化しています。このように、以下でマイクロサービスについて議論されているアーキテクチャの概念は、SOAから直接継承されています。

各モジュールは他のノードとの連携なしに動作するため、システム全体の状態は個々のノードにはわかりません。さらに、グローバル情報やグローバル変数の値がない場合、個々のノードはローカルで利用可能な情報に基づいて決定を行います。[Tanenbaum at et., 2007]

モノリシックアプリケーションとマイクロサービスアプリケーションのアーキテクチャの違いとその意味合いを以下の表5にまとめました：

| モノリシックアプリケーション | マイクロサービスベースのアプリケーション |
|---|---|
| 単一の計算ノードとして設計されています。そのため、全体的な状態情報が完全に把握されています。 | 各ノードがサービスを提供する複数のノードのセットとして設計されています。システム全体の状態は個々のノードにはわかりません。 |
| グローバル情報やグローバル変数の値を利用するように設計されています。 | 個々のノードは、ローカルで利用可能な情報に基づいて決定を行います。 |
| ノードの失敗は、アプリケーションのクラッシュを意味します。 | 1つのノードが故障しても他のノードに影響を与えません。 |

*表5：モノリシックアプリケーションとマイクロサービスベースのアプリケーションのアーキテクチャの違い*

**モノリシックアアーキテクチャおよびマイクロサービスアーキテクチャ下でのアプリケーション例**[11]

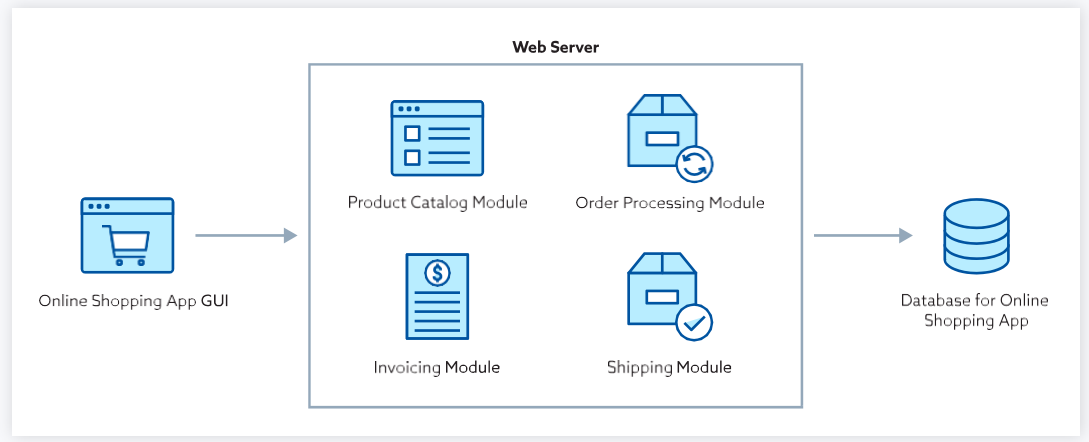



*図2：オンライン・ショッピング・アプリケーション - モノリシック・アーキテクチャ*

[11]この文書のようにセキュリティに関連したベストプラクティスを提示する際には、マイクロサービスアーキテクチャのベストプラクティスについても言及しなければなりません。そのようなベストプラクティスの1つは、各マイクロサービスが他のすべてのものから独立して独自のDBを管理するというものです。したがって、2...N個のマイクロサービスが1つのDBを共有することは、DB層でのサービス間カップリングを導入することになります。[Newman, 2015]

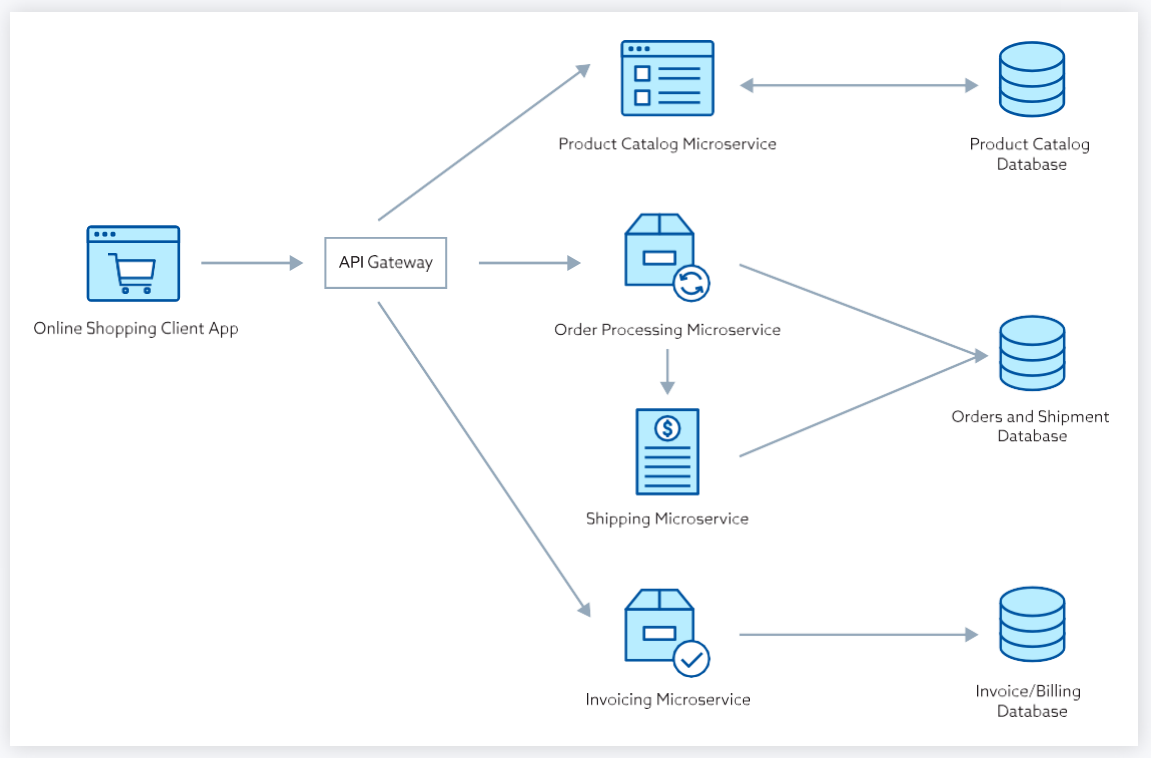



*図3：オンラインショッピングアプリケーション – マイクロサービスアーキテクチャ［Yarygina et al., 2018］*

# 1.3 マイクロサービスの利点と課題

最も重要な課題は、プロダクション環境に多数のマイクロサービスを配置した結果としてのオペレーション環境の複雑性です。データの一貫性（結果一貫性）、モニタリング、障害対応、セキュリティ強化、コンテナのセキュア化を含みます。

## 1.3.1 MSA と SOA のユースケースと比較

次の表［SOA Source Book］はマイクロサービスアーキテクチャ（MSA）とサービス志向アーキテクチャ(SOA)のユースケースとの比較です。次の二つのセクションが記述するマイクロサービスの利点と課題についてのコンテキストを設定します。

| ユースケース | MSAの提案 | SOAの提案 |
| --- | --- | --- |
| *ビジネス推進要因* | | |
| ソリューションレジリエンス（可用性、回復性など）はクリティカルな成功要因 | MSAはレジリエンスを提供するための安価な手段を提供することができます。なぜなら、障害が発生したマイクロサービスは、他のシステムに依存関係の無い並列化された（同じマクロサービスの）インスタンスによって引き継ぐことができるからです。 | SOAのレジリエンスは、より高価かも知れません。SOAアプリケーションを提供する場合は、サービスのコレオグラフィーによって、依存関係と相互作用を必要としている可能性があるためです。 |
| ソリューションの範囲が明確です。 | アーキテクチャを再構築するコストは価値がないかも知れません。 | ＳＯＡは既存のインフラストラクチャをサポートします。 |
| ソリューションは、探索的なものか、既に確立されたものかどうか。 | MSAは迅速な開発とデプロイメントサイクルを有効にします。それは迅速なフェイルオーバーと自己リカバリを可能にします。これらの特徴の結果、高可用性とシームレスなユーザとの関係を実現します。 | 比較すると、SOAは定められている要件を維持するビジネスドメインにより適しています。 |
| *技術的推進要因* | | |
| ソリューションは個別かつ独立したサービスコンポーネントに分割されるべきです | MSAのデザインは独立したサービスのコンセプトを中心としています。 | もしサービスを独立させられないならば、SOAはよい選択肢です。 |
| ソリューションはレジリエンスと柔軟なスケーラビリティを必須とします。 | MSAはレジリエンス（素早い回復）とスケーラビリティを並列処理によって提供します。 | SOAはこれらの目的を達成するために垂直方向にスケールしたマシンを使ったアプローチを提供し、これはより費用対効果が高いかも知れません。 |
| ソリューションはサービス間のオーケストレーションを必要とします。 | MSAはスマートエンドポイントを信頼します。一方で ダムパイプ[12] の考え方は低レベルのコレオグラフィーを提供します。 | SOAは BPEL や WS-Choreography のようなプロトコルをオーケストレーションのために利用します。 |

| *運用的推進要因* | | |
|---|---|---|
| 個別のサービスについて高頻度の解約や更新が想定されます | APIの定義、サービス構成、オペレーショナルな要件が変化する場合、MSAはよい選択肢です。 | 高頻度のサービス更新は、伝統的な SOA インフラストラクチャをサポートする場合には、より複雑になるでしょう。 |
| アプリケーションの開発やデプロイメントサイクルが短い | MSAはオーバーヘッドを低くし、迅速な並列開発を提供します。 | より複雑で完全なSOAは開発とデプロイのためにより長い時間が掛かるでしょう |
| 新しいアプリケーション（または、新しいビジネスエリア）とレガシーなアプリケーションの上に構築する場合の比較 | 新しいアプリケーションまたはビジネスエリア | レガシーなアプリケーションの上に構築 |

*図6：MSAとSOAのユースケースの比較*

12 https://martinfowler.com/articles/microservices.html

### 1.3.2 利点

セクション 1.1.2 で示唆したように、マイクロサービスアーキテクチャと SOA の特徴の違いは、以下のように、SOA が提供するメリットを上回るものです：

- ビジネス上の利点
  - ビジネスとの整合性
    - 一つのマイクロサービスは、一つのビジネスユニットが持つ分割不可能なビジネス能力を実現するアトミックなビジネス機能に基づいています。従って、ビジネスユニットの変更と一緒に更新することが可能です。
  - アジャイルかつビジネスニーズへの迅速な対応が可能
    - コンポーネントを更新したりスケールするためだけにアプリケーション全体を停止する必要はありません。各サービスは独立して更新またはスケールすることができます。これはビジネスに素早い対応の能力を提供します。
  - チーム化
    - 継続的インテグレーション（CI）と継続的デプロイメント（CD）のパラダイムを利用する、分散し自律的な多くのチームは、統合されたコードの自動化されたデプロイメントによって、ほとんど持続したソフトウェインテグレーションを行い、独立したサービスに取り組むことができます。
  - 高可用性 / ディザスタリカバリ
    - マイクロサービスの独立性による並列性によってレジリエンスが提供されます。それは迅速なフェイルオーバーとセルフリカバリーを可能とします。これら特徴の結果は高い可用性とシームレスなユーザとの関係です。
  - 本質的に相互運用可能なサービス
    - サービスは典型的には、定義されたサービス境界に従って、必ず明確に定義され公開されたインターフェースを使用して実装されます。従って、アプリケーションを構成するための時間とコストは低減され、市場変化への迅速な対応や参入を可能にします。
- 技術的な利点
  - 多角的なアプローチは最適な技術、ツール、プラットフォームを選び、採用することを可能にします。
    - 言語やテクノロジーのロックインはありません。それぞれのサービスは独立して動作するため、開発者は開発のためにどのような言語・技術も選ぶことができ、そのAPIエンドポイントは期待したアウトプットを返すことを保証します。
  - マイクロサービス内の各サービスは独立してデプロイすることが可能です。
  - サービスのコードが一度書かれれば、同じ機能が必要な他のプロジェクトで利用することが可能です。
  - マイクロサービスアーキテクチャは継続的デリバリと継続的インテグレーションを可能にします。
- オペレーションの利点
  - 一つのサービスに障害が発生しても、その障害が全体に波及することはありません。これはデバッグも助けてくれるでしょう（もちろん、マイクロサービスアーキテクチャが障害に強いという意味ではありません。さらに、デバッグは複雑さが増加するためにより難しくなるでしょう。カスケード効果については、1.3.3節で欠点として示します）。サーキットブレーキングパターンは、マイクロサービスの環境で一度サーキットブレーカーが実行された場合に、障害が発生したサービスに追加のコールが行われないようにすることができます。
  - コンポーネントは複数のサーバや複数のデータセンターに配置することができます。
  - マイクロサービスは Kubernetes, DC/OS や Docker Swarm のようなコンテナオーケストレ

ーションツールと相性良く動作します。

### 1.3.3 欠点と課題

多数の大規模分散サービスを管理することは、次の理由から難しいものです：

- ビジネス上の課題
  - プロジェクトチームは、潜在的に再利用可能なサービスを容易に見つける必要があります。これらのサービスは、ドキュメントやコンソールテスト等を提供するべきで、再利用がスクラッチ開発よりも格段に容易になります。
- 技術的な課題
  - 各サービスの相互依存関係はよく監視されていなければなりません。サービスのダウンタイム、サービスの障害、サービスの更新等は連鎖的にその先の工程に影響を及ぼす可能性があり、そのようなインパクトは事前に分析されるべきです。そうでない場合、上述したような強みとなり得るレジリエンスは獲得できずにマイクロサービスでのフォールトトレランスはモノリシックアーキテクチャよりも複雑になり欠点になってしまいます。
  - 適切なサービスサイズの選択
    - モノリスアプリケーションをやめ、マイクロサービスをスクラッチから作成する一方で、サービスに適した機能を選ぶことがとても重要です。例えば、もしビジネスがモノリスの各機能をマイクロサービスとして作成した場合、開発者は多くの小さなサービスを作成し、不要な複雑性をもたらすことになるでしょう。
  - テスト
    - 多数のサービスとそれらに関連する相互依存性のために、マイクロサービスのエンドツーエンドのテストを実行することが困難になる場合があります。
  - サービス間通信
    - サービス間通信は正しく実装していない場合に高いコストになり得ます。選択肢としてはメッセージパッシング、RPC等があり、そのような、最小限のオーバーヘッドで要件に合うものを選択するべきです。
  - データベースの管理
    - データベースはマイクロサービスごとに実装される必要があります。しかし、ビジネスの流れを閉じてしまうため、パーティション化されたデータベースなどのようにこの課題に対応することができる他のデータベースに変更することが必要となるかもしれません。
  - サービスディスカバリと文書化
  - フォールトトレランス
  - QoS（クオリティ・オブ・サービス）
  - セキュリティ
  - リクエストのトレーサビリティ
  - 障害のトリアージ
- 運用上／セキュリティ上の課題
  - サービスのソフトウェア開発サイクル（SDLC）は高度に自動化されているべきです。これはデプロイメント自動化技術や CI - CD フレームワークのようなテクノロジを必要としますが、より重要なのは開発者やオペレーターチームの規律です。開発者やオペレータだけの肩にかかっている責任ではなく、マネジメントが同様に関与し、トップダウンで指示し、チームをサポートすることは同様に重要です。経営層がセキュリティの利点を支持することはさらに重要です。
- デプロイメント
  - マイクロサービスをデプロイするために、Kubernetes のような分散環境が利用されるべき

です。一つの仮想マシンごとに一つのマイクロサービスをデプロイすることは可能ですが、このアプローチは重大かつ不要な技術的なオーバーヘッドをもたらすでしょう。Kubernetes は対照的に、モニタリングやセキュリティなどの多くの付加価値サービスをバンドルできる大きなオーケストレーションプラットフォームの一部として利用しなければ非常に複雑になります。

- モニタリング
  - マイクロサービス環境における個々のサービスのモニタリングは課題となり得ます。この課題は解決されつつあり、特定されモニターしデバッグするためのツールが開発されています。

上記の課題や欠点があったとしても、アプリケーションが複雑で継続的に進化している場合には、マイクロサービスの開発は理にかなっています。

# 2.0 クラウドネイティブアプリケーションのためのマイクロサービスセキュリティアーキテクチャ

開発者は、スケーラブルで、移植性があり、障害に強く、簡単に更新できる新しいアプリケーションの開発に関心を寄せています。以降の章ではクラウドネイティブアプリケーションのためのマイクロサービスアーキテクチャにおける考慮事項を記述します。

## 2.1 全体的な脅威モデルと関連するベストプラクティス

### 2.1.1 脅威とセキュリティの懸念

多くの組織がモノリスから分散マイクロサービスベースのアーキテクチャに移行するにつれて、セキュリティの懸念が高まっています。マイクロサービスアーキテクチャにおいては、全てのコンポーネントとの通信経路を確立するための多様なネットワーク接続とAPIの利用、新しい攻撃方法の出現、通信データの傍受や保管データの操作のために攻撃対象は増加し、セキュリティの課題が増加しています。マイクロサービスベースのアーキテクチャはより多くのシステムの機能をネットワークに直接公開するため、攻撃対象面を増やします。複数の小さなコンテナは多くの違うシステムやホストに展開することがあり、それらが凝集して機能しなければならないということは、脅威が著しく増加する状況であることを意味します。これはまた、それぞれのコンテナは正しく保守され、管理され、セキュアにされなければなりません。それらの全ては適切なツールが無ければ非常に時間がかかることになります。さらに、ソフトウェアとサービス開発者によって実装された全てのコントロールは、セキュリティコントロールとの不一致やギャップを引き起こすことがあります。セキュリティコントロールの一貫した執行と、コンテナを伴う開発プロセスへの強制を達成するには、特別なツールとテクニックが必要とされます。
もう一つの課題は、コンテナの標準化された複製可能な性質は、あるマイクロサービスに含まれた脆弱性がソースコードの再利用によって素早く複数回にわたって複製されることを意味することです。

従来のアプリケーション／サービスモノリスの場合、そのアプリケーション／サービスコンポーネントは、典型的に一つのネットワークにある一つ以上のサーバにホストされているため、公開されたポートとAPIに着目したり、IPアドレスを特定したり、その周辺の境界を設定することが容易になります。マイクロサービスでは、多くのポートやAPIゲートウェイが公開されており、APIの経済性が高まるにつれて攻撃対象が広くなり、認証も非常に分散されているため、より複雑になります。これは基本的に、分散されたサービスの集合を運用するには、セキュリティコントロールが分散的に実行されることを必要としており、コンテナとマイクロサービスのためのセキュリティエコシステムを成功させるには、ステークホルダーはそれぞれの役割を担当し参加しなければならないことを意味しています

特にアプリケーションコンポーネントとサービスが分離される点については、セキュリティの立場からはコンテナとマイクロサービスには明確な利点があります。特定のサービスやAPI、ネットワーク通信経路に適用するコントロールによって、より詳細なレベルでマイクロサービスのセキュリティを実装することができます。

マイクロサービスは多層防御を実装するための能力を提供します。しかし、そのセキュリティコントロールを実装する方法は従来の方法から大きく変わります。マイクロサービスアーキテクチャでは、複数のトランザクションとインタラクションが存在します。従って、アプリケーション／サービスコンポーネントのセキュリティはコンテナのインタラクションと、それらが何で、何を行うのかという知識に依存します。サービス間通信の可視性は課題となり、少なくとも問題の規模は大きくなります。

マイクロサービスをデプロイするパターンは幾つかあります

1. 仮想マシン（VM）でのマイクロサービス
2. コンテナでのマイクロサービス
3. サーバレスプラットフォームへのマイクロサービスの配備

#### 2.1.1.1 脅威モデル：仮想マシン（ＶＭ）でのマイクロサービス

仮想マシンでのマイクロサービスのための脅威モデルはよく知られており、幾つかは以下のように特定されています：

- プロセス分離の侵害 - VMエスケープ
  あらゆるハイパーバイザへの主要な脅威は不正な仮想マシンからやってきます。不正な仮想マシンは、VMM/ハイパーバイザによって行われるメモリページやストレージデバイスのようなハードウェアリソース分離機能を破壊します。これはハイパーバイザ設計の脆弱性、悪意のあるまたは脆弱なデバイスドライバなどの理由によって起こることがあります。不正なVMの影響は、ハイパーバイザのコントロールを奪取されることがあり、これには rootkit のインストールや同じ仮想化ホスト上の他のVMに対する攻撃が含まれます。
- ネットワーク分離の侵害
  - 仮想環境の分離に対する脅威は、不正VMによるIP または MAC アドレスのスプーフィングやスヌーピング、または同じ仮想ネットワークセグメントセグメント上のVMに対する仮想ネットワークトラフィックの傍受などがあります。
- サービス拒否(DoS)
  - 不適切に設定された、または悪意のあるVMは、ホストのリソースを過剰に消費して、結果として同じハイパーバイザにある他のVMをサービス拒否状態にさせる場合があります。

VM環境において、他にも多くの脅威があります。より詳細は NIST 800-125 Rev 1 で確認できます。

### 2.1.1.2 脅威モデル/ベストプラクティス：コンテナで実行されるマイクロサービス

コンテナで実行されるマイクロサービスの脅威モデルと関連するベストプラクティスを次の表に示します：

| # | 識別子 | 弱点 | 脅威の影響 | 考慮すべき緩和策 |
|---|---|---|---|---|
| 1 | **ホストの物理セキュリティ** | 基盤となるハードウェア、チップセットおよびBIOSの脆弱性 | コンテナを実行しているホストOSが危険にさらされる可能性があります. | 脆弱性が特定されたパッチのBIOSを更新し、セキュアブートが実装された信頼できるプラットフォームを使用する。<br>HOST OSの信頼性は、vTPM（仮想Trusted Platform Module）を使用しても実現できます。<br>このアプローチは、またコンテナが新しいモジュールをローディングする事やカーネルの脆弱性を利用する事でホストのカーネルを変更する事を確実に出来なくします。<br>コンテナは、vTPMが提供するハッシュ拡張機能を使用して、自身の状態を証明することもできます。<br>スタック全体で信頼のルートが維持されるように、各コンテナにvTPMを追加する必要があります。<br>完全性測定アーキテクチャ（IMA）も検討できます。なぜなら、測定リストは、検出されない限り攻撃によって危険にさらされることはないからです。<br>その他の緩和策としては、定期的に一定の間隔でコンテナを再設置することが挙げられます。 |
| 2 | **ホストOSの防御** | OS の脆弱性は、複数のコンテナが同じ OS カーネルを共有しているため、コンテナを多数の脅威にさらす可能性があります。 | 1つのコンテナがパフォーマンスに影響を与えたり、他のコンテナに対してサービス拒否を引き起こす可能性があります。 | 一般的なOSディストリビューションではなく、コンテナ固有のOSを使用し,OSバージョンを最新の状態に保ち、パッチを適用します。<br>使用されていないサービス、インターフェースをすべて無効にし、管理者アカウントグループを最小化します。また実行時に、自動パッチ、自動デプロイを使用し、手動での変更を許可しないようにしてください。コンフィグファイルには、バージョンなどを識別しやすいように、OSの種類やディストリビューションなどを識別できる属性が必要です。 |

| | | | | |
|---|---|---|---|---|
| 3 | HOST OS**の保護-コンテナエスケープ** | コンテナは共有OS環境とホストカーネルで動作します。 | 共有環境で実行しているときにアプリケーションを分離しないと、悪意のあるアプリが他のアプリコンテナとの接点となる可能性があるため、データが意図せず公開されたり、アプリコンポーネントの整合性に影響を与えたりする可能性があります。 | SELINUXのようなカーネルローダブルモジュールを使用します。<br>SELINUXは、ユーザ、プロセス、アプリにアクセス制御を強制することで、コンテナをホストや他のユーザから隔離するのに役立ちます。<br>AppArmorは、強制アクセスコントロール（MAC）システムで、限られたリソースにプログラムを制限するためのカーネル（LSM（Linux Security Module））の強化であり、基盤となるOSのセキュリティを強化するために使用することができます。<br>AppArmorのセキュリティモデルは、アクセス制御属性をユーザではなくプログラムにバインドするというものです。<br>AppArmorの制限は、カーネルにロードされたプロファイルを介して提供され、通常はブート時に行われます。<br>Cgroupは、リソースの割り当てと使用量を制御します。<br>コンテナに関連付けられたSecomp プロファイルは、コンテナが実行できるシステムコールを制限することができます。Linuxの機能を使用してコンテナ内にRoot をロックダウンすることができます。<br>軽量OS（Lightweight OS）は、コンテナ環境により適している可能性があるストリップダウン機能を使用することができます。 SELINUXが有効あるいは無効になっていることを識別する属性を持つことが不可欠です。<br>セキュリティをさらに強化するために、ユーザランド/カーネルコピーの境界チェックに対処するカーネルパッチを実装し、UDEREFとフォワードエッジCPIを利用することも推奨します。 |

| | | | | |
|---|---|---|---|---|
| 4 | **ホストOS保護 – ランタイム設定** | コンテナは共有OS環境で動作し、ホストカーネルはランタイム環境で動作します。<br><br>このように、クロスコンテナの脆弱性や設定ミスは、共有 OS 上で実行されているコンテナに影響を与える可能性があります。 | あるコンテナの設定を変更すると、他のコンテナに影響を与え、危うくなる可能性があります。 | コンテナを強化し、Seccomp、AppArmorなどのツールを適切に使用する必要があります。<br><br>ランタイムコンテナの構成/パラメータは、コンテナランタイムによって提供されるさまざまなAPIによって制御されます。 |
| 5 | **ホストとカーネルのセキュリティ** | 攻撃者がホストシステムに侵入した場合、コンテナの分離やその他のセキュリティ制御は無効にされます。 | 侵害されたホストで実行されているサービスおよびアプリコンポーネントの完全な侵害 | ホストおよびコンテナエンジンの構成は、認証された制限付きアクセスで強化する必要があります（Docker Bench Dockscan、Falco、CoreOS Clair、Anchore Engineなどのツールを使用）。<br><br>ベースシステムを更新し、サードパーティのリポジトリも更新する必要があります。<br>最小限のコンテナ中心のホストシステムを使用すると、攻撃面が減少します。<br><br>信頼されたアプリのコンテナを別のマシンにマップします。信頼されていないアプリを root 権限で実行しないでください。 |

| | | | | |
|---|---|---|---|---|
| 6 | **コンテナブレイクアウト** | コンテナブレイクアウトの悪用は、実際に利用可能であり、その環境内で顕在化します。 | ホストの完全な侵害 | 分離された名前空間を作成し、最大権限を制限します。特権付きコンテナを実行するには、信頼されたリポジトリからの信頼された特権の検証を確認すべきです。<br>カーネルのセキュリティの強化とその他のOSのセキュリティ強化技術を確保して、コンテナのブレイクアウトの保護、分離、および検出を確実にします。<br><br>Ref: http://gzs715.github.io/pubs/SECNAMESPACE_SEC18.pdf |
| 7 | **コンテナリソースの悪用** | アプリケーションコンポーネントは通信を目的として、REST APIの代わりにIPCリソースを使用する可能性があり、個別のサービスがHOST IPCオブジェクトを操作し、同一ホスト上のアプリケーションコンテナに影響を与え、サービス運用妨害状態や多数のメッセージによるバックキューを発生させる可能性がある。 | ホスト上の他のコンテナで使用されているリソースに影響を与えます。 | サービスにIPCや共有メモリセグメントの利用を許可しない。HOST名に関連付けられた名前空間を共有するオプションをコンテナが参照できないようにします。 |
| 8 | **コンテナリソースの悪用** | コンテナが読み込み/書き込みモードでマウントされている場合、ホストの機微ディレクトリやファイルシステムは、コンテにファイルの変権限を与える可性があります。 | これはホストをセキュリティ上の危険にさらす可能性があります。 | コンテナにコンテナのボリュームとして、読み込みまたは書き込みモードでホストの機微ディレクトリをマウントさせない。 |

| | | | | |
|---|---|---|---|---|
| 9 | **コンテナリソース悪用** | デフォルトオプションによるホストへのコンテナポートの公開は、攻撃面を増やします。 | コンテナからホストへの通信を広く許容することはサービス運用妨害の発生につながります。 | ポートがバインドするべき正確なインターフェースを明示的に指定します。コンテナへの送受信トラフィックインターフェースに限定します。 |
| 10 | **コンテナリソース悪用** | 設計上の計算ミスや意図的な攻撃は、リソースの侵害によりサービス運用妨害（DoS）を引き起こす可能性があります。保護すべきリソースはCPU、I/O、Swap、カーネルリソースなどです。 | 特定のサービスのDoSを引き起こす可能性があります。 | Linux Kernelにバンドルしてリソース制限を設定します（Cgroupを使用）。<br>スケーラビリティとパフォーマンスを判断するために、本番環境移行前にサービスの負荷テストを実施します。<br>リソースが制約される前に、監視とアラートを行うために閾値を設定する必要があります。 |
| 11 | **ベースイメージの脆弱性** | コンテナのパフォーマンスが良好であっても、古いバージョンのJavaなどのライブラリを利用している可能性があります。 | 古いバージョンは侵害を引き起こす可能性があります。 | 必要最低限のベースイメージにより、攻撃面を減らします。<br>更新、特にプログレッシブローリングアップデートを保証するディストリビューターが提供するソフトウェアを使用します。データはイメージから分離します。<br>イメージ中のソフトウェアを最小限にすることは、脆弱性のパッチの管理をシンプルにします。<br>ランタイムスキャンツール（Aquasec、Twistlock、StackRoxなど）を使用し、これらのツールを最新の脆弱性情報（Quay、Clairなど）で更新します。 |

| コンテナ・コンテンツに関連した脅威 | | | | |
|---|---|---|---|---|
| 12 | **オープンソースまたはサードパーティプロバイダー（Apache Webサーバ、node.jsなど）のコンテナ・コンテンツ** | 悪意のあるコンポーネントがサードパーティーコンテナの一部に含まれている可能性があります。 | 悪意のあるコンポーネントは攻撃者にバックドアを提供し、ホストで実行されるアプリケーションの完全な制御権を許容する可能性があります。 | コンテナで有効になっているパッケージと、コンテナを作成したユーザの特定が重要です。そのためには、有効なパッケージをCI/CDパイプラインで事前に強化し、新しいパッチがパッケージをデフォルト構成に戻さないようにする必要があります。オペレータは、CI/CDパイプラインに定期的に更新されるパッケージを含めるために、ソフトウェアサプライチェーンを管理する準備を整える必要があります。また、組織はパッケージの調達とコンテナイメージの構築を自身で行うことが望ましい。<br>これらのコンテナはツールを使用してスキャンできるステージング領域に配置され、コンテナイメージにパッケージ化されて内部レポジトリに昇格する必要があります。ステージング領域に新しいファイルがダウンロードされた際に自動的にスキャンを行うツールと、APIベースでの統合が可能です。<br><br>開発者は未検証のイメージを使用してはなりません。承認されたDockerの内部レジストリのイメージのみを使用します。<br><br>イメージの電子署名の検証は重要ですが、脆弱性評価の実行の代替手段とはなりません。 |

| | | | | |
|---|---|---|---|---|
| **コンテナのレジストリとイメージへのアクセス** | | | | |
| 13 | **コンテナのレジストリとイメージへのアクセス** | 内部で構築されたソフトウェアとコンテナの構築では、適切なアクセス制御が行われていなければ、セキュリティが侵害される可能性があります。 | これは、内部の脅威につながり、誤ってコードをリポジトリに昇格させ、本番環境にデプロイされてサービスの可用性に影響を与える可能性があります。 | 役割ごとに職務を適切に分離して、開発、テスト、本番のコード用に別々のブランチを持つプライベートレジストリを実装します。<br><br>コードを 本番用ブランチに昇格させる前のビルドのスキャンを自動化します。<br><br>ベースイメージに新しい更新が適用されると次のビルドサイクルで、本番環境で実行中のコンテナに更新が確実に適用されるようにします。 |
| **ビルドプロセスのセキュリティ** | | | | |
| 14 | **コンテナ内のランタイムライブラリとアプリコードの統合** | アプリコードに加えられた変更は、ソフトウェアスタックの整合性に影響を与える可能性があります。<br><br>開発者は、ランタイム環境への変更を許可されるべきではありません。 | 複数のアプリコード構成とインフラストラクチャへの依存性により、環境内のセキュリティ制御に一貫性がなくなる可能性があります。 | 一度ビルドして、イミュータブル・コンテナをデプロイします。<br><br>CI/CDパイプラインを介してランタイムにデプロイします。ビルドとデプロイを自動化することで、構成の統一性が保証され、リスクが大幅に軽減されます。新しいデプロイメントがドリフトしたり、デフォルトのコントロールでデプロイされたりしないことを保証するために、CI/CDパイプラインのパッケージに設定を適用する必要があります。 |

| | | | | |
|---|---|---|---|---|
| **配備** | | | | |
| 15 | **クラスタへの配備** | 配備のポリシーが制御されておらず、**配備**がポリシーに基づいていない場合、古いイメージが**配備**される可能性があります。 | 関連するサービスの統合だけでなく、可用性も影響を受ける可能性があります。 | アプリ/サービスがどのようなコンテキストで利用できるかについて、強力なセキュリティポリシーを定義する必要があります。（例： CIS ベンチマーク、Kubernetes の POD セキュリティポリシーでは、管理者が特権コンテナの実行、コンテナが実行時に要求できる機能、ホストディレクトリやボリュームの使用などを制御することができます）。 |
| **コンテナプラットフォームのセキュリティ** | | | | |
| 16 | **ホストへのコンテナ展開、ホストの容量、どのコンテナが相互に作用するのか、どのようにしてお互いを発見し、識別するのか** | コンテナ管理環境が操作され、アプリケーションのリソースに影響を与える可能性があります。 | 可用性は、ランタイムプラットフォーム全体のセキュリティと同様に影響を受ける可能性があります。 | APIは、自動化されたデプロイとコンテナ管理に使用されます。したがって、APIのセキュリティは、IDとアクセス管理と同様に鍵となります。<br><br>X509またはトークンを使用します。アプリケーションの分離は、漏洩したトークンによる被害を制限します。<br><br>すべての通信は、TLS 1.2/TLS 1.3を使用して実施する必要があります。<br><br>マスターデーモン（例： etcd）をサービスに直接公開しないでください。<br>ランタイムコンテナの健全性とセキュリティ検出は、重要なセキュリティコントロールです。<br><br>Linux の名前空間を使用し、IMAのようなフレームワークを検討してください。 |

| ネットワーク分離 | | | | |
|---|---|---|---|---|
| 17 | **複数のテナント間の脅威** | 同じアプリのコンテナは複数のクラスタにまたがる可能性があり、このマルチテナント環境では、1つのアプリが他のアプリのセキュリティに影響を与える可能性があります（例えば、侵害されたトークンを使用したラテラルムーブメントなど）。 | これにより、複数のアプリやサービスが危殆化し、データが漏洩する可能性があります。 | コンテナのコレクション（PODS）を分離するためにネットワーク名前空間を使用し、固有のIPとポート範囲を提供します。<br>他の PODS からトラフィックを受信するためには、明示的なネットワークポリシーを確立する必要があり、デフォルトでは PODS はあるネットワークネームスペースから別のネットワークネームスペースにトラフィックを送ることができない。<br><br>SDN オーバーレイはマスターノードとスレーブノード間の分離と同様にコンテナ間のネットワークポリシーをより細かくコントロールすることができます。<br><br>例えばデータベースへのアクセスなど、出口のプラットフォームコントロールの使用が確立される必要があります。<br><br>ネットワークポリシー違反検出のために、ネットワークスキャナーを導入する必要があります。 |

*テーブル７：　コンテナ環境におけるマイクロサービスの脅威モデル*

### 2.1.1.3 脅威モデル/サーバレス環境で実行されるマイクロサービスのベストプラクティス

コンテナPaaSで実行されるマイクロサービスの脅威は基本的にコンテナの抱える脅威と同じです。とはいえ、コンテナPaaS環境は難読化されており、且つサービスプロバイダによって管理されていることから、マイクロサービスの開発者はコンテナPaaS環境内の潜在的な脅威について恐れる必要はありません。基本的に開発者は基盤となるコンテナ環境をどうすることもできないからです。しかし、プロバイダが安全なプラットフォームを提供しなかった結果として脅威が存在している場合には、悪意のあるエンティティがその脅威を活用し、サービスを攻撃することが可能です。

多くのクラウドプロバイダはサーバレスコンピューティング（例・AWS Lambda）サービスを提供しています。以下はサーバレス環境において実行されるマイクロサービス向けのベストプラクティスです。

- アプリケーションのマップを定義し文書化します
  - 数百もの機能で構成されたサーバレスアプリがあるため、潜在的なリスクを把握するための全体像を持つことが重要です。コードは機微なものも含むビジネスプロセスを公開する可能性もあります。このため、マップは以下の点に留意する必要があります。
    - アプリケーションにおいてどのようなデータが処理され保存されるか。
    - データにはどのような価値があるか。
    - データにアクセスできるサービスにはどのようなものがあるか。
- 機能レベルでの境界セキュリティ

  呼び出し可能で多様なソースからのトリガー（ストレージ、メッセージキュー、データベースなど）に関連づけられた、小さなコンポーネントに分割されたアプリケーションの分割は、攻撃者にとってより好都合な標的であり、より多くの攻撃経路があることを意味します。WAFやAPIゲートウェイのような緩和策を利用することはよいことですが、もう一つのサーバレスのベストプラクティスは、機能レベルの境界セキュリティの適用です。機能レベル境界セキュリティの実装は、一般的に統制が取れたセキュアな開発プラクティスが前提となります。OWASP Security Champions は、脅威モデリングと機能レベルの緩和策の適用によりOWASP Top 10を緩和するために有用であり、明確に定義されたDevSecOpsプラクティスのニーズが増すでしょう。
- 各機能に対する適切かつ最小限の役割の実装
  - サーバレスは、動作するまたは動作可能なリソースの数を大幅に増やします。開発者は多数の双方のパーミッションを含む多数のリソースの相互作用をガバナンスするポリシーを検討すべきです。
  - 各機能に対して適切かつ最小限の役割の実装に投資することは重要です。加えて、いかなる問題からも被る被害を最小限に止めるため、それぞれの機能は最小限の実行権限によって実行されることを保証しなければなりません。役割と機能はなるべく頻繁に見直しを行うべきです。サーバレスにおいては、いったん適切に設定されたものが、アプリケーションの他の部分を脆弱にするような役割やポリシーや機能の変更が行われた結果、適切ではなくなることが起こりえます。開発者はこれらのポリシーの生成と、いかなる場合においても変更が行われたことを通知するアラートの発行を行うために、新しい技術を考慮するべきです。
- セキュアなアプリケーション依存関係
  - 関数は、例えばnpm(Node.js)、PyPI(Python)、Maven(Java)などのレポジトリから取得した依存関係を含むことがあります。アプリケーションの依存関係は一般的であり、定期的に新しい脆弱性が公開されていることからわかるように脆弱であることが多いです。サーバレスの性質は、サードパーティの依存関係を手動で管理することを困難にします。さらに相互依存コードはこの問題を生み出します（例：コードベースにある依存関係Aが依存関係Bをインポートし、Bが脆弱性が存在する依存関係Cをインポートするような場合）。アプリケーションの依存関係を安全にするには、良質なデータベースへのアクセスと自動化ツールを必要とします。それは、脆弱なパッケージを使用しないようにし、新規に公開された問題の通知を継続的に受けられるようにするためです。さらに、脆弱なライブラリのインジェクションを相殺するラッピングコントロールとして、影響の最小化のためにアプリケーションのセグメントを適切にサービス単位に分割し、最小限の権限を厳密に適用することは極めて重要です。セグメンテーションと最小限の権限は、問題発生時の影響範囲を限定的にすることが可能です。

- 質の悪いコードの検知と、それに対する行動の重要性
  - 多様なトリガーと無限のスケーリングを持つサーバレスの展開は、小さなコードエラーがアプリケーション内部での自分自身に対するDoS攻撃に繋がることを意味しています。より公開された攻撃対象面では、バグはより容易にセキュリティ上の問題になります。開発者が定期的かつ適切にトレーニングを受けることは極めて重要です。コードレビューの実施に加えて、コードや設定の監視と設定をテストするためのツールの利用も役立つでしょう。
- CI/CDと本番環境へのサービス設定テストの組み込み
  - サーバレスの展開は企業向けアプリケーションが生む価値のスピードや、製品展開の速さの増加を招きますが、同時にセキュリティの問題を招く機会も増やします。設定の意図と、本番環境で実際の動作との継続テストと検証を必要とします。
- 情報のフローの監視
  - – データフローの影響はアプリケーション内に極めて多数のコンポーネントが含まれることからサーバレス独特のものです。情報とデータのフローの監視は正しい情報が正しい場所へ向かうこと、意図したサービス間でのみ使われていることを確認するために推奨されます。
- DoS（Denial-of-Service）とDoW（Denial-of-Wallet）の緩和
  - -サーバレスアプリケーションのスケールが容易である一方、スケーリングは無限ではない。アプリケーションは攻撃者がリソースの制限を飽和させることでDoSの影響を受ける可能性があります。現在の制限が問題回避には充分なだけ高い場合には、攻撃者がコストに大打撃を与えるタイプの攻撃であるDenial-of-Wallet 攻撃を警戒すべきでしょう。サーバレス環境によるサービスの大規模なオートスケーリングの実現はDoS攻撃を緩和する一方でDoW攻撃を引き起こす可能性があります。APIゲートウェイのようなDoS攻撃への適切な緩和策も有効ですが、同時にKinesisやS3などのトリガーを介した攻撃についても考慮する必要があります。機能による自己保護は、これらの攻撃を検出し、その影響を最小限に抑え、スケーリングの選択肢を動的に調整して脅威の軽減に役立てることができます。
- FaaS （Function as a Service)コンテナのリフレッシュ
  - 開発者は関数インスタンスのライフタイムに関する戦略について考慮する必要があります（プラットフォームによってはインスタンスの更新を行うAPIがあります）。関数インスタンス、例えば追加のコードや隠されたネイティブプロセスといった関数インスタンスを探索しようとすることを検知し、コンテナから消すことができるようなセキュリティソリューションの採用を検討すると良いでしょう。こういったセキュリティソリューションは追加の複雑性やサービスとして実行するエージェントの追加を必要とする場合があります。

サイバーセキュリティと同様にサーバレスアプリケーションの安全性の確保はアプリケーションの開発ライフサイクルとサプライチェーンを通じて、多様な戦術を必要とします。ベストプラクティスを常に追い続ける姿勢がセキュリティ対策全体を改善するでしょう。しかし、適切な開発だけでは十分ではありません。理想的な保護のためには継続的なセキュリティの保証、攻撃の検知と防御とディセプションをサポートし、提供ツールを最大限生かすことが重要です。

現在、単一のFaaS機能を展開するためのツールはありますが、複数の機能を同時に調整、展開するために必要なツールが欠けています。例としては、特定のフローまたはすべてを一度に展開することが必要となるサーバレスアプリケーションを構成する複数の機能の場合です。このようなユースケースの管理に役立つツールは多くありません。サーバレスアプリケーションのゼロダウンタイムを保証するツールもまた堅牢ではありません。サーバレスアプリケーションの統合テストは困難です。FaaS環境同様、開発者は状態を維持するために外部リソースに依存しており、実際の本番環境でテストを行う必要があるケースもあります。

## 2.1.3 認証

モノリシック環境において認証と認可は、必要とされる認証認可情報のための入力要求を検証するモジュールを利用して、アプリケーション内で処理されます（次図）。このモジュールは、権限を与えられたユーザに役割とパーミッションを定義することを許容し、安全なリソースへ他のユーザがアクセスできるよう、これらの定義を割り当てます。

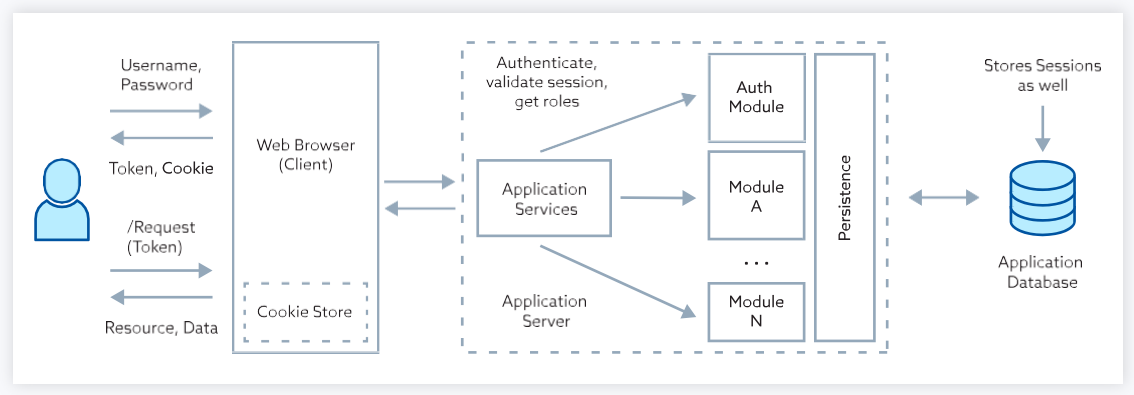



*図 4: Clojure, Packt Publishingによるマイクロサービス*

マイクロサービス上のアプリケーションでは、マイクロサービスベースのアプリケーションはそれぞれ隔離された状態で配備され、独立しているユーザデータベースやセッションデータベースの維持に責任を持つべきではありません。
認証には次の2種類がなければなりません；一つはサービスに対するユーザ認証、もう一つは、通常JWTトークンやmTLSを使用して実行されるアプリケーション間の認証です。さらに、マイクロサービス間でのユーザの認証と認可の標準的な手法が必要です。オープンポリシーエージェント（OPA）がその手法です。OPAはポリシーの実行からポリシーの定義を分離します。
マイクロサービスがポリシーの評価を行う必要がある場合、マイクロサービスは構造型データ（例JSON）を入力値としてOPAにクエリを投げます。OPAは入力値として任意の構造データを受け入れます。OPAはクエリの入力値をポリシーとデータに対して検証し、ポリシーの評価を行います。OPAとRegoはドメイン依存ではないため、マイクロサービスはポリシー内のいかなる不変の値をも記述します（例：どのユーザがどのリソースにアクセスできるか、どのサブネットに下りトラフィックが許可されているか、どのクラスタにワークロードをデプロイする必要があるか、どのレジストリバイナリがからダウンロード可能か コンテナが実行できるOSの機能はどれか、およびシステムにアクセスできる時間帯はいつかなど）。

ポリシーの評価は単純なイエス/ノーや許可/拒否といったものに限定されません。クエリインプットのように、ポリシーは任意の構造化データを出力値として生成することができます。理想的には、認証と認可を、ユーザの認証認可のためのユーザデータベースを持つAuthサービスとして分離することが不可欠です。エンタープライズ環境においては、サービスを跨いだユーザ認証はADFSを利用したエンタープライズ版のシングルサインオン（SSO）あるいはSAML2.0を用いておこなうことができます。これは、認証サービスを介してユーザ認証を行ったのち、関連サービスや当該サービス内でのリソースアクセスの認可を行うことができます。認証情報やトークンの窃取を目的とした行為のターゲットとなる可能性があることから、認証サービスの保護のための適切なコントロールが行われなければなりません。適切な保護と検知も、侵害行為の発見のために行われるべきです。

認証認可のために異なる形式のトークンを使用することができます（例：JWT,JSON Web Signature（JWS)、JSON WebEncryption（JWE））。暗号化されたトークンは、暗号化した際の鍵によって復号されたのちにのみパースできます。マイクロサービス間で鍵を共有する代わりに、トークンの復号とサービスに代行してリクエストを認可するためにAuthサービスへトークンを送信することは有益です。トークンの送信によるパフォーマンスへの影響は、トークンの有効期限に基づいて設定された最小限の期限内で、各マイクロサービスレベルで事前検証されたトークンをキャッシュすることで軽減可能です。有効期限は認証トークンにとって重要な基準です。長い有効期限を持つ認証トークンは窃取されたトークンの利用による攻撃を避けるためにも回避すべきです。認証トークンのフォーマットについてはJWT RFC-7519、JWS RFC-7515、JWE RFC-7516に例があります。サービスがホストされているコンテナの身元を保証する必要があります。2つのコンテナ間の通信には両方のエンドポイントを認証するmTLSを使用すべきです。このアプローチは、トークンを窃取しサービスの悪用を引き起こす、いかなる中間者攻撃をも緩和することに貢献します。

## 2.2 APIをセキュアにする

APIは、定義されたアプリケーション間のインターフェース上でアプリケーションが機能を共有することを許容するものです。適切に設計されていれば、APIは制御された方法でアプリケーションの内部を公開します。APIセキュリティはソフトウェアの機能的・非機能的要件により、入出力の機密性、完全性、可用性（CIA）を保証するものです。「忠実性」、「正確性」、「完全性」はCIAを言い換えたAPIセキュリティの用語です。

インターフェースの入力と出力は、アプリケーションのさまざまなモジュールの内部で隠ぺいするか、アプリケーションの境界の外に公開して、他の外部に公開されたインターフェースとの間でデータを送受信できるインターフェースとして機能させることができます。

全ての機能的又は非機能的要件（NFR）は設計の指針となる一般的なソフトウェアアーキテクチャの原則によりサポートされている必要があります。要件およびAPIの構成を示す原則との間でのトレーサビリティはそれ自体が機能的要件です。

セキュリティ指針の基礎としてのAPIのベストプラクティスは以下を含みます：

- トークンベースの認証やセッションベースのアクセスとの組み合わせで、粗い粒度と細かい粒度の両方のアクセス制御インターフェースとデータアクセスの利用が可能であること。
- 基本的に、開発者は、サービスとインターフェースのセグメンテーションを実践すること。
- 開発者は、高度にパーティショニングされ、モジュール化された設計で、分離されたサービスで構成されたソリューションを志向すること。
- 通信において、標準ベースのメッセージングと暗号化プロトコルが利用できること。
- インターフェースとその基礎となるロジックの実行は疎結合であること。
- 開発者用インターフェースエクスペリエンスはシンプルで、アプリケーションの操作が容易であること。
- 開発者は、確実なエラー・ハンドリングとサービスの見つけやすさを提供し、可視的で検証可能な機能を可能にすること。
- 開発者は、メソッドの使用方法（例：GETよりもPOSTが望ましい）、クエリパラメータ、データパス、ヘッダコンテンツ、許容・非許容のステータスコードの宣言など、コーディング標準を規定することで、バックエンドAPI設計の決定を標準化すること。

- 開発者は、期待されるSLA、サービス契約、APIメディエーションのユースケース（例：ダイレクトコネクトやインターメディエーション）、および許容されるスタイル（例：RESTを好む、条件付きSOAPを受け入れる、条件付きHATEOSを受け入れる、RPCを推奨しない、特別な用途のためのストリーミングを受け入れる）を規定することによって、フロントエンドAPIの設計の決定を標準化すること。
- バージョニングスキームと命名規則が適用されること。
- 開発者は、APIの仕様を合意された非機能要件とテスト可能な受入れ基準のセットにバインドすること。

### 2.2.1 サービスディスカバリ

マイクロサービスアーキテクチャでは、ビジネス機能はコンテナ内のサービスとしてパッケージ化されてデプロイされ、APIコールを使用して相互に通信します。このため、異なるインタラクションスタイルを実装できる軽量メッセージバスを実装することが推奨されます。サービスディスカバリのサービスが実装される他の方法には、(a) クライアントがサービスレジストリのサービスにアクセスする方法と、(b) 集中型サービスレジストリと分散型サービスレジストリがあります。

企業は、集中型サービスレジストリで全てのサービスが1カ所で発行・登録されるかどうか選択します。ただし、この方法は機密性、完全性、可用性に影響を及ぼす単一障害点となりえます。レジストリの侵害は、他のサービスには正規のサービスに見えるが、不正なサービスの展開につながる可能性があります。サービスレジストリは適切なアクセス制御を介して認証されていないユーザから保護される必要があります。サービスレジストリへの認可されていない変更によって、悪意のあるエンティティが正規ではない又は承認されていないサービスの公開を行う可能性が潜在的に存在し、企業のプラットフォームの完全性に影響を及ぼす可能性があることから、適切な検出制御を実施する必要があります。

### 2.2.2 APIゲートウェイを利用する場合

マイクロサービスには同期または非同期通信を行うため、各種の通信プロコルをサポートし、サービスを仲介するAPIゲートウェイの存在が重要になります。APIゲートウェイはID管理サービスと統合されている必要があります。適切な認証・認可は、サービスへのいかなる接続やアクセスを許容する前に行われる必要があります。APIゲートウェイはセキュアであり、かつ、攻撃検知のために集中監視システムあるいは検出システムに全てのトラフィック情報を送る必要があります。

# 2.3 マイクロサービスの認可とアクセスコントロール

CIA / IAAA機能の継続的に維持するために、マイクロサービスの一つのアプリケーションは複数のプロセスに分割されます。 各マイクロサービスは、オリジナルの単一のアプリケーションに1つのモジュールのビジネスロジックを実装しています。そのため、アプリケーションが分割された後、各マイクロサービスのアクセス要求を認証して認可する必要があります。

しかし、マイクロサービスアーキテクチャにおける認証と認可には、より複雑なシナリオが含まれます。これには、ユーザからサービスへの通信だけでなく、マシンからサービスへの通信（アプリからサービスへ、サービスからサービスへ）も含まれます。 [Medium, 2018].

以下は、認証および認可の問題を解決するためのいくつかのマイクロサービスアーキテクチャソリューションです:

- 認可処理
- サービス間の認可
- API アクセスコントロール
- ファイアウォール
- シークレット情報管理

## 2.3.1 認可処理

「認可」という用語は、ある人が何をできるかを指します。たとえば、ある文書へのアクセス、編集、削除などで、認証の後に行われます。認可は、その行為を行っている人の身元を確認する認証とは別のステップであることに注意してください。

認可は一般に以下で管理されます：

- **ルール**：一連のオブジェクトで許可される動詞の集まり
- **ロール**：ルールの集まり。ユーザとグループは、同時に複数のロールに関連付けられたり、バインドされたりします。ロールは、保護されたリソースまたは操作にアクセスする権限を表します。
- **バインディング**：ユーザまたはグループ間のロールの関連付け
- RBAC：ロールベースアクセスコントロール（役割ベースのアクセス制御）。 [Red Hat OpenShift]

上記のようなマイクロサービスアーキテクチャにおける認可の問題を処理するために、従来の認可モデルは「エンタイトルメント」の概念を追加することで強化することができます。

エンタイトルメントは、現在、POCとしてDockerに実装されていますが、これらのメカニズムは本番で使用する準備ができておらず、まだ未完成の状態です。 [LWN.net、2018]

**ラベルベースのID**
ラベルは、どのコンテナがどのリソースにアクセスできるかを定義し、コンテナがアクセスするシークレット

情報のラベルを持つ必要があります。ラベルを使用すると、ユーザは事前に定義されたポリシーを参照して、コンテナがシークレット情報を受信することを許可することができます。コンテナとそのラベルは、特定のコンテナハッシュが有効であることを確認するために、コード完全性ツール[Wei et al., 2018]を使用して一緒にバインドされるべきです。暗号技術を使用することで、コンテナごと、ポッドごと、またはサービスごとのレベルで、強力なアイデンティティと認証を作成するのに役立ちます。 [Unbound, 2018]

**APIゲートウェイを使用したクライアントトークン**

ここでは、トークン認証の基本的な処理とは逆に、外部からのリクエストの入り口としてAPIゲートウェイが追加されています。これは、すべてのリクエストがAPIゲートウェイを通過することを意味し、マイクロサービスを効果的に隠しています。リクエストに応じて、API ゲートウェイは、元のユーザトークンを、それ自身のみで解決できるopaque トークンに変更します。API ゲートウェイは、ユーザがログアウトしたときにユーザのトークンを削除することができます。また、クライアントがログオフした際に、クライアントからトークンを隠すことで、authトークンが復号化されるのを防ぐこともできます。

## 2.3.2 サービス間の認可

すべてのマイクロサービスで認可を実行することにより、きめ細かなオブジェクト権限が可能になるほか、マイクロサービスごとに異なるユーザ認証メカニズムが可能になります。

**サードパーティアプリケーションのアクセス**
これは3つの方法で行われます：

- API トークン
- OAuth
- フェデレーション

**APIトークン**：（APIにアクセスするために直接ユーザ名/パスワードを使用する代わりに）APIトークンを提示する利点は、ユーザのパスワードが公開されるリスクを減らし、パスワードを変更する必要なくいつでもトークンの利用許可を再申請することができます。ここでは、サードパーティはアプリケーションが発行したAPIトークンを使用してアプリケーションのデータにアクセスします。そして、トークンはアプリケーション内のユーザによって生成され、サードパーティのアプリケーションが使用するために提供されます。

**OAuth**：最良のアプローチの1つは、JSONトークン(JWT - JSON Web Token)を使用したOAuth代理アクセスを認可するプロトコルです。JWTはオープンスタンダード(RFC 7519)で、JSONオブジェクトとして当事者間で情報を安全に送信するためのコンパクトで自己完結型の方法を定義しています。この情報はデジタル署名されているため、検証し信頼することができます。JWT は、共有鍵（HMAC アルゴリズムを使用）、あるいは、 RSA または ECDSA を使用した公開/秘密鍵ペアを使用して署名することができます。

マイクロサービスでのOAuth認証は、良く知られた認可プロバイダ（訳注： 原文はauthentication providerとなっていますが、内容的にAuthorization Providerの間違いと思われる）によって提供され、面倒な登録操作を簡素化します。 ここで、マイクロサービスはOAuthアーキテクチャでクライアントの役割を果たします。このようにして、クライアント/マイクロサービスは、アクセストークンを申請するときに認可サーバと直接対話します。認可サーバは、クライアントのトークンリクエストを処理するときにクライアントを認可し、他のユーザが認証コードのクライアントIDを偽造できないようにします。[Medium, 2018]

トークンは、「秘密鍵/公開鍵」方式で署名できます。 このように、他のマイクロサービスには、署名をチェックして公開鍵を知るためのコードを含めるだけで済みます。トークンは認可ヘッダーとともに持参人トークン(bearer トークン)として送信されるため、マイクロサービスによって検証できます。この署名は、URLに関する制限なしに提供されます。したがって、クロスサイト認可も可能とします。これは、シングルサインオン（SSO）をサポートすることになり、ユーザにとって非常に便利です。 [LeanIX、2017]

**OAUTH2 と OIDC トークン**：大まかで機能的なアクセス制御に適しており、OIDCは一般的にフロントエンドサービスで使用され、IDトークンはOIDCクライアントに送信されます。OIDC クライアントのみが署名を検証でき、ID トークンを他のパーティに渡すことはできません。OAUTH2はスコープを使用して権限を決定しますが、スコープはユーザが誰であるかを示すのではなく、既存の利用権限のみを示します。

**フェデレーション**：フェデレーションセキュリティは、管理者とユーザだけでなく組織にも役立つ多目的システムです。 フェデレーションは、複数のサービスで同じユーザ資格情報を使用して実行するか、異なるセキュリティドメインでシングルサインオン（SSO）を可能にします。 [Nordic APIS, 2017]

### 2.3.3 API アクセスコントロール

ここでのAPIゲートウェイのベストプラクティスは、ユーザが今現在の職務を実行するために必要となるリソースにアクセスを限定する「最小権限」です。

OAuth / OAuth2サーバは、通常、開発者と管理者がAPIを使用した認証用のトークンを取得するために使用されます。セキュリティ上の理由から、すべてのクライアントサーバの通信中はトランスポートレイヤーセキュリティ（TLS）で暗号化することをお勧めします。

**双方向TLSクライアント証明書**Bound Access Token
クライアントがトークンエンドポイントへの接続で双方向TLSを使用する場合、認証サーバは発行されたアクセストークンをクライアント証明書にバインドすることができます。このようなバインディングは、証明書ハッシュを発行されたアクセストークンに直接埋め込んだり、トークンイントロスペクションを通じて証明書をトークンに関連付けたりすることで、保護されたリソースがアクセスできる方法で証明書をトークンに関連付けることで達成されます。このような方法でアクセストークンをクライアント証明書にバインドすると、認証サーバとのクライアント認証からそのバインドを切り離すことができるという利点があります。これにより、保護されたリソースへのアクセス中の双方向TLSは、純粋に所有証明のメカニズムとして機能することが可能になります。

**ロードバランシングに組み合わせたAPI ブローカーセキュリティ**
マイクロサービスのアドレス指定能力とセキュリティを同時に確保するためのベストプラクティスの1つは、セキュリティを提供するためにAPIマネージャまたはブローカーに依拠することです。APIマネージャ/ブローカーは、セキュリティとロードバランサーを組み合わせて、作業を分割して管理しやすくし、サービス停止を減少します。ブローカーが呼び出しアプリケーションにセキュリティトークンを提供する限り、すべてのAPIブローカーとマイクロサービスを共通のサブネットに配置することが重要です。ブローカーがマイクロサービス自体を呼び出す場合、マイクロサービスは独自のサブネットワークに配置する必要があり、許可されるトラフィックは、マイクロサービスにアクセスするAPIブローカーからのトラフィックのみです。 [TechTarget, 2018]

**SPIFFE/SPIRE**
SPIFFE（発音はスピッフィー）は、Secure Production Identity Framework For Everyoneの略です。これは、オンプレミス、プライベートクラウド、およびパブリッククラウド環境に展開された分散システムをサポートするオープンソースのワークロードIDフレームワークです。SPIFFEは、最新の本番環境のすべてのワークロードに、特別に加工されたX.509証明書の形式で安全なIDを提供します。SPIFFEは、アプリケーションレベルの認証と複雑なネットワークレベルのACL設定の必要性を取り除きます。

SPIRE（SPIFFE Runtime Environment)は、実行中のソフトウエアシステム（ワークロード）に、証明書（SVID）を他の実行中のワークロードに証明するだけでなく、IDを引き出すことができるようにAPI(SPIFFE Workload API)を公開するソフトウェアシステムです。SPIREはオープンソースであり、組織が混在した本番インフラストラクチャ全体でSPIFFE IDをプロビジョニング、展開、および管理できるようにします。

ここでのワークロードAPI（他のシステムのAPIと同様）は、APIを呼び出すときに、呼び出しワークロードが自身のIDの事前情報や認証トークンを持っていることを必要としません。これにより、認証シークレットをワークロードと一緒に展開する必要がなくなります。

しかしながら、他のAPIと比較して、SPIRE APIは複数のプラットフォームでさえも実行され、プロセスレベルおよびカーネルレベルで実行中のサービスも識別できます。これにより、Kubernetesなどのコンテナスケジューラでの使用に適しています。

SPIREでは、すべての秘密鍵（および対応する証明書）は有効期間が短く、頻繁かつ自動的にローテーションされて、鍵の漏えいや侵害の可能性を最小限に抑えます。ワークロードは、対応する鍵の有効期限が切れる前に、Workload APIから新しい鍵と信頼バンドルをリクエストできます。

具体的には、ワークロードAPIにより、ワークロードは次のものを取得できます:

> - SPIFFE IDとして記述されているアイデンティティ
> - ワークロードの代わりにデータに署名するために使用できる ID に紐付けられた秘密鍵
> - 対応する X.509-SVID という X.509 証明書も作成されます。これは、mTLS の確立、JWT トークンの署名、または他のワークロードへの認証に使用できます。
> - ワークロードが他のワークロードの身元を確認するために使用することができる証明書のセット（信頼バンドル）[SPIRE]

**Firewall**

攻撃面をはるかに小さくするために、ファイアウォールでの出口トラフィックの制御を可能にするプラットフォームが検討されています。[Apriorit, 2018]

ユーザが各マイクロサービスをよりきめ細かく制御できるよう集中管理された分散型ファイアウォール（サービスメッシュ）もまた重要なコンポーネントです。このようにして、開発者はどの接続が許可され、どの接続が許可されないかを細かく定義することができます。これは事実上、各サービスが独自のマイクロファイアウォールを持っていることを意味します。したがって、あるサービスが破られても、残りのサービスは安全なままです。[Project Calico]

コンテナとサービスのきめ細かなセキュリティを可能にするセキュリティポリシーは、サービスを相互に保護し、オーケストレータ自身を保護することで、サービスの安全性を維持するための良いアプローチです。オペレータは、高レベルのセキュリティポリシー定義をコンテナごとにファイアウォールルールのセットに変換し、それらのコンテナごとにファイアウォールルールを検証して維持することができます。

最後に、コンテナの起動、停止、セキュリティポリシーの更新に合わせて、すべてのファイアウォールルールの変更を即座に管理し、調整することができます。 [Weaveworks]

### 2.3.4 シークレット管理

シークレットとは、API トークン、SSH キー、パスワードなど、サービスが他のサービスを認証して通信するために必要な資格情報のことです。

API の認可は分散した方法で実装する必要がありますが、これは困難である可能性があります。また、分散アプリケーションのための秘密管理と鍵の配布は、開発者にとってもう一つの頭痛の種となります。シークレットをコンテナイメージに入れると、多くのユーザやプロセスに公開され、悪用される危険性があります。

自動化されたクレデンシャル管理システムは、マイクロサービス環境を作成するためのSecurity by Designアプローチに傾いています。これにより、開発者は、重要な職務の分離を維持しながら、秘密管理を即座に自動化する強力なツール（例：DevOps Audit Toolkit[13]）を手に入れることができます。

開発者/オペレータのシークレット管理のベストプラクティスには、以下のようなものがあります:

- シークレットの集中管理、シークレットへのコンテナのアクセス、どのコンテナがそれらのシークレットをリアルタイムで使用しているかの表示を検討します。
- 実行時にコンテナにシークレットを注入し、シークレットがメモリに格納され、指定されたコンテナのみがアクセスできるようにします。
- シークレットが必要なコンテナに暗号化されて配信されることを確実にします. [Aquablog, 2019]

シークレットを一元管理するために、シークレット管理、encryption as a service、特権アクセス管理のためのツールを利用することができます。

シークレット管理は、シークレットを安全に保管するようにガイドします。それは、以下のようなものがあります:

- **Static Secrets** は、誰がアクセスできるかを制御します。シークレットは永続的なストレージに書き込む前に暗号化されなければならないので、ローストレージにアクセスするだけではアクセスできません。
- **Secret Engines**は、データを保存、生成、または暗号化します。シークレットエンジンの中には、単にデータを保存して読み取るだけのものもあります。
  他のシークレットエンジンは他のサービスに接続して、必要に応じて動的なクレデンシャルを生成します。データは保存せずに暗号化して復号化する必要があります。これにより、セキュリティチームは暗号化パラメータを定義することができ、開発者は独自の暗号化方法を設計することなく、暗号化されたデータをSQLなどの場所に保存することができます。
- **Secret as a Service**は、Dynamic Secretsは、各アプリケーションやシステムが独自のクレデンシャルを取得できるように、オンデマンドでデータベースのクレデンシャルを生成し、そのパーミッションを厳密に制御することができます。Dynamic Secretsを作成した後、リースが終了すると自動的に取り消されます。
- **Database Root Credential Rotation**は、システムで管理されているデータベースルート認証情報のローテーションを可能にします。シークレットが長く存在すればするほど、それが危殆化する可能性が高くなります。頻繁にローテーションを行うことで、シークレットの寿命を短くし、暴露のリスクを減らすことができます。

[13] Separation of Duties in the DevOps Audit Toolkit: https://www.oreilly.com/library/view/devops-sec/9781491971413/ch05.html

- **Cubbyhole Response Wrapping**は、期待するクライアントだけがラッピングを解くことができる場所に、ラッピングしたシークレットを配布する安全な方法です。ルートがkey/valueのシークレットエンジンとは逆の使い方をしていても、他のトークンのCubbyholeに通信を試みることはできません。
- **One-Time SSH Password**は、 SSH のシークレットエンジンを利用して、クライアントがリモートホストに SSH しようとするたびにワンタイムパスワード (OTP) を生成します。
- **Build Your Own Certificate Authority** は、PKI シークレットエンジンを使用して独自の認証局を構築し、動的なX509証明書を生成します。
- **Direct Application Integration** は、アプリケーションへのコード変更なし、または最小限のコード変更で、システムからシークレットを取得するための Consul Templateと Envconsul ツールの使用が含まれています。

## 2.4 マイクロサービスアーキテクチャにおける安全な配備のスタイルと戦略

マイクロサービスには、クラウドサービスプロバイダー (CSP)モデルに合わせた4 つの主要な配備スタイルがあります。これらのスタイルとサービスは、PaaS、サーバレス (FaaS)、コンテナ (CaaS)、および IaaSモデルによって提供される仮想マシンです。以下の図は、プラットフォームのスペクトルと実例を示しています：

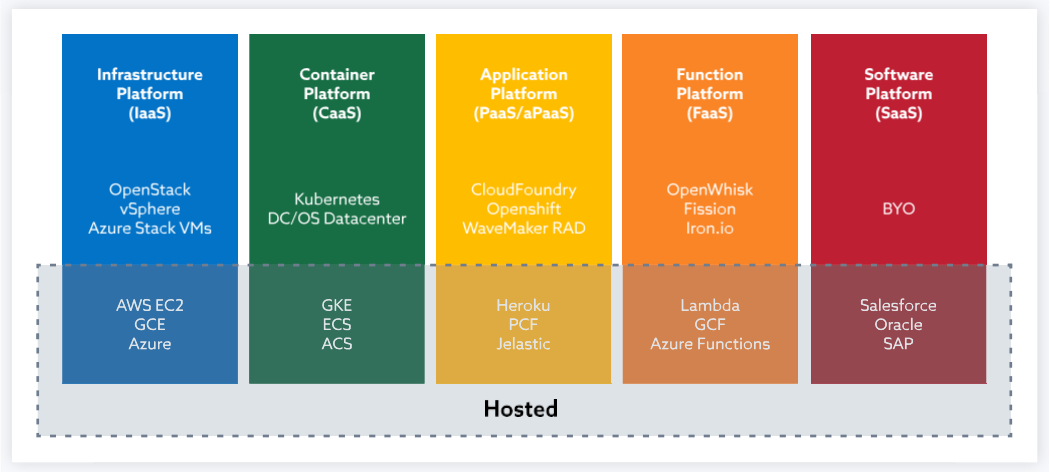



*図6: マイクロサービス配備スタイルのプラットフォームスペクトラム*
[Picture Reference：http:www.Mesosphere.com]

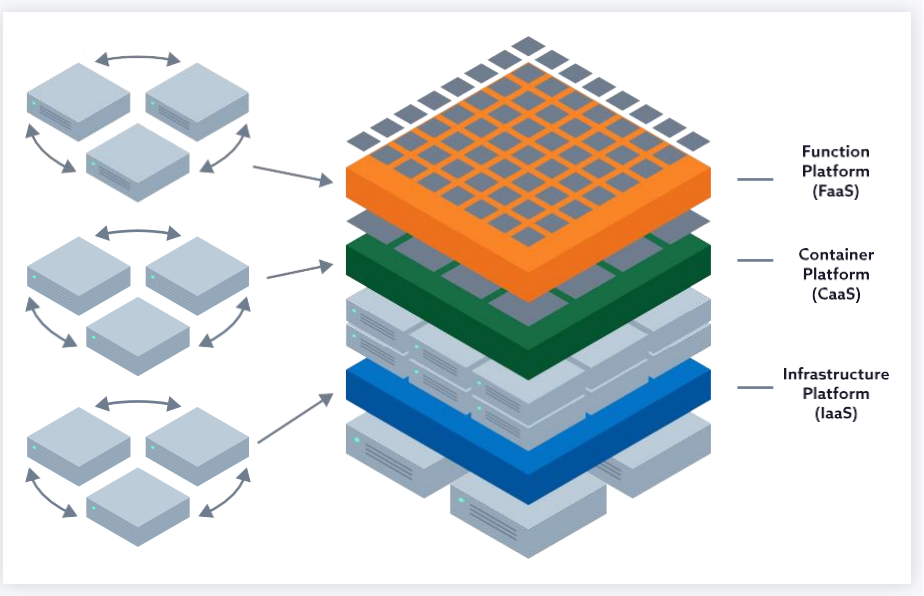



*図7: マイクロサービス配備スタイルとサービス*
[Picture Reference：http:www.Mesosphere.com]

クラウドサービスモデルと同様に、マイクロサービスは、選択した配備スタイルによって、利用者とCSP 間の責任共有モデルが決まります。以下に、マイクロサービス責任モデルを図示します。ただし、利用者の責任とクラウドの場所だけがマイクロサービス配備のスタイルと戦略に関わる決定事項ではありません。配備のその他の考慮事項は、速度、移植性、および管理です。これらの要素のバランスをとることが、配備スタイルの鍵となります。インテグレーションのためにAPIを公開しているパブリックまたはプライベートクラウドに移行する組織にとって、配備戦略は、マイクロサービスのワークロードとその適合性に基づくことになります。

その他のマイクロサービス配備パターン：

- ホストごとのマイクロサービスの複数インスタンス
- ホストごとのマイクロサービスの単一インスタンス
- 仮想マシンごとのマイクロサービスの単一インスタンス
- コンテナごとのマイクロサービスの単一インスタンス
- マイクロサービス配備プラットフォーム
- サーバレス配備

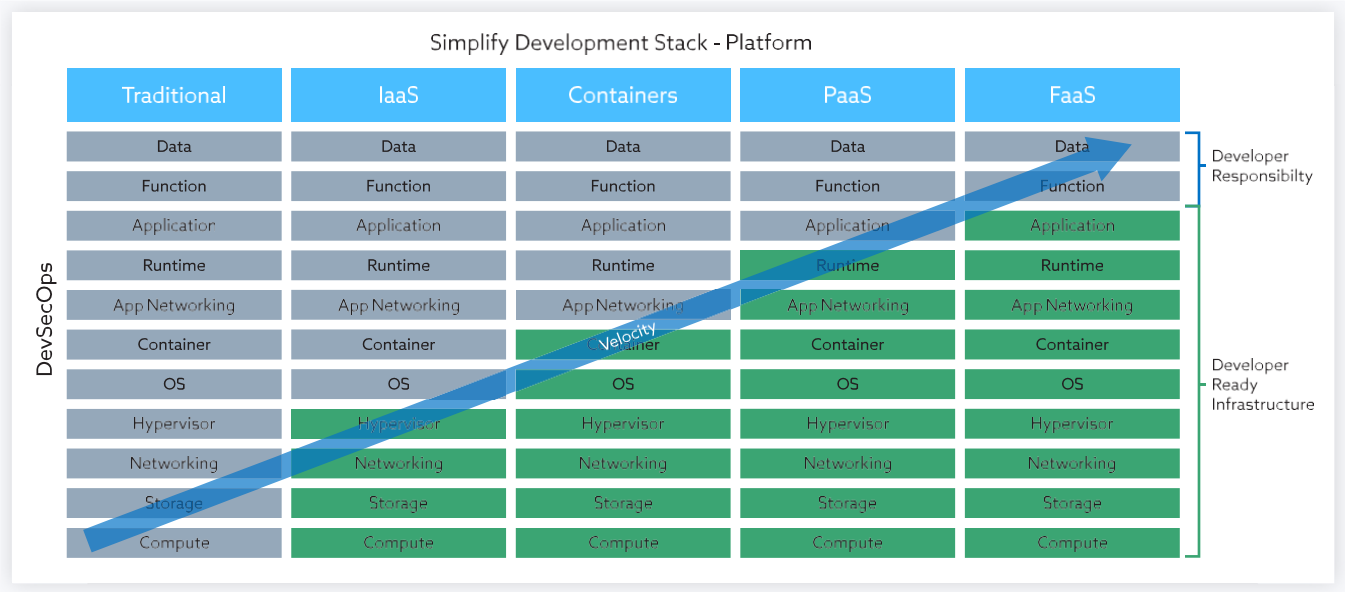



*図8：マイクロサービス責任モデル*
(Reference: https://blogs.vmware.com/vov/files/2018/08/development-stack.png)

# 2.5 ステートフルおよびステートレスのマイクロサービスセキュリティ

**ステートフル マイクロサービス**:マイクロサービスで構成されるアプリケーションは、ステートレスとステートフルの両方のサービスを含むことができます。ステートフルサービスを実装する際の制約事項と制限事項を理解することが重要です。各マイクロサービスインスタンスが、システム全体で考慮しなければならない独自のステータスを持つ「スノーフレーク（訳注：ユニークな存在をネガティブに表現するスラング）」になるため、迅速なスケーリングと環境の再構築をかなり困難にします。動的コンテナ環境のセキュリティは、既知の正しいイメージからコンテナを迅速に破棄し再作成する能力に大きく依存します。これは、マイクロサービスが設計においてステートレスであるべきであることを意味します。したがって、ステートレス設計が重要なマイクロサービスのベストプラクティスです。開発者は、ステートフルマイクロサービスの実装を選択する前に注意を払う必要があります。サービスがステートに依存している場合は、容易にアクセスできる専用のストレージレイヤーに分離する必要があります。

マイクロサービスベースのアプリケーションには、RDBMS、NoSQLデータベース、およびファイルシステムの形式でステートフルサービスを含むことができます。これらのマイクロサービスは、固有の属性を持つコンテナとしてパッケージ化されます。ホストに依存しない個別の永続レイヤの作成をサポートするテクノロジを使用する必要があります。これにより、ホストの移植性の問題に対処し、整合性に必要となる遅延を最小限に抑え、可用性を最大化できます。

一般的に、ステートフルサービスは、ホストに対する永続性をオフロードするか、永続性レイヤを提供するため、可用性の高いクラウドデータストアを使用します。このどちらのアプローチも複雑になります:ホストへのオフロードは、あるホストから別のホストにコンテナを移植することが困難になり、データストアの高い可用性は、一貫性との引き換えです。つまり開発者はデータモデルの最終的な一貫性を設計する必要があります。

**ステートレスマイクロサービス**：マイクロサービスの主な利点の 1 つは、迅速に拡張可能であることです。他の分散コンピューティングアーキテクチャと同様に、マイクロサービスはステートレスな場合に上手く拡張できます。拡張の容易さは、サービス全体の可用性をサポートします。

ステートレスシナリオでは、マイクロサービスはステートの記録を保持せず、セッションレベルのセキュリティの管理が困難になります。

有効期限はトークンに埋め込まれた情報であるため、アクティブなセッションの有効期限を取り消すことはできません。この問題に対処するベストプラクティスの 1 つは、トークンの機密性の管理と共に、各トークンに短い有効期限を定義することです。

マイクロサービスアーキテクチャのセキュリティモジュールは、主に要求ごとのトークンを更新する役割を担うため、これはサービス利用者にとって問題を引き起こすものではありません。

## 2.6 コンテナストレージインターフェース

データベースは、基盤となるストレージプラットフォームにデータを保存します。アプリケーション開発者は、永続的なストレージシステムをデータベースそのものと見なすことがあります。 しかし、その裏側では、データベースはブロック、ファイル、またはオブジェクトストレージのいずれかに書き込みを行っています。コンテナオーケストレーションシステムは、アプリケーションコンテナを管理します。 コンテナストレージインターフェース（CSI）は、コンテナオーケストレーターとバックエンド ストレージシステムの間の共通インターフェースレイヤーです。コンテナストレージインターフェースグインを実装するコンテナオーケストレーターストレーターはCSIをサポートするだけでよいという考え方です。

上記を実現するために、コンテナを外部のストレージに接続し、データを保存するのが一般的です。これは、コンテナのためのストレージとボリュームを追加することで実現できます。 ホストにストレージ領域を追加しても、そのホスト上で実行しているコンテナで使用可能なストレージが自動的に増えません。しかしながら、コンテナ内でホストストレージボリュームをマウントして使用する方法があります。ある1つのコンテナにマウントされたボリュームは、他のコンテナでも使用できます。これらはデータボリュームコンテナと呼ばれ、 サービスは、1つ以上のターゲットコンテナと、ソースコンテナからのボリュームを共有できるようにします。この手法には、複数のコンテナ間で永続的なストレージを共有する、バインドマウントの抽象化レイヤを提供するといったいくつかの利点があります。

ストレージボリュームをコンテナにマウントする機能により、開発者はコンテナをシンプルかつポータブルに保ち、さらに、各コンテナの外部でのデータアクセスを可能にできます。イメージとコンテナ自体がホスト システム上の領域を消費するため、ホストのディスク領域を拡張できることは有益です。

## 2.7 ランタイムセキュリティ

マイクロサービス環境では、ランタイムのセキュリティとポリシーの適用が重要です。ランタイムプロテクションは、コンテナ自身で実行できる一連のコードを制限するという別の概念です。攻撃者がコンテナにアクセスできても、そのコンテナ内で彼ら独自のコードを実行できない場合、考えられる損害は限定的です。

従来のVMツールはコンテナをサポートせず、コンテナの検出を提供できません。ランタイムの検出とポリシーの適用に役立つ新しい種類のコンテナセキュリティツールが利用可能です。これらのツールは、コンテナブレイクアウト、ネットワークポリシーの適用、ホストOSおよびカーネルの設定変更を検出し、ランタイムの脆弱性スキャンなどを実行することができます。その他の考慮すべきポリシーがあります。それは、通常のアプリケーションの挙動を確立し、それに基づいてセキュリティポリシーを制定し(例えば、コンテナがルートアクセスを要求したら、配備を防止する)、すべてのコンテナの異常と影響を受けたイメージを自動的に置き換えるイメージレジストリを監視し、実行中のコンテナとホストのライブスキャンを実施し、何かが通常時から大きく外れたときにアクティビティをブロックし、根本原因の特定と今後の攻撃を防ぐためにすべてのセキュリティイベントのデータを保存し分析する、ということを実施します。これらのツールはまた、ランタイムイメージスキャンやサードパーティのイメージスキャンと同様に、コンテナレジストリのスキャン機能も提供します。

つまり、ランタイムにおいて効果的なセキュリティを実現するには、以下のベストプラクティスに従うことが重要です：

- システムは、信頼されたイメージリポジトリからのみ取得できるようにすべきです。
- 署名されていないイメージの取得は拒否されるべきです。
- リソースの使用状況を監視し、異常な使用状況を調査すべきです。
- コンテナは、直接のリモートアクセス(たとえば、SSH) を許可せず、そのかわりに、変更を追跡するためのバージョン管理されたアーティファクトストアとソース管理を頼りにすべきです。
- コンテナによるシステムコールを監視し、異常な呼び出しを調査すべきです。
- コンテナは、最大有効時間を持ち、定期的にローテーションさせるべきです。
- 可能であれば、読み取り専用のコンテナを使用すべきです。
- コンテナ内の 特権ユーザは、ホスト上の非特権ユーザに割り当てるべきです。
- 強制アクセス制御(たとえば、SELinux/AppArmor)ポリシーを有効にして設定すべきです。
- マイクロサービスとコンテナの動作を分析すべきです。
- 分散された脅威インジケータを関連付け、単一のコンテナが感染しているのか、あるいは、多くのコンテナに広がっているのかを見極めます。
- 承認されていないコンテナエンジンコマンドを傍受してブロックします。
- 応答のアクションは自動化します。

# 3.0 マイクロサービスによる安全な開発とガバナンス

マイクロサービスとセキュリティを共存させるには、マイクロサービスの開発、ガバナンス、および管理のフレームワークと計画を開発することが必要になります。マイクロサービスの開発にあたっては、サービスとそれに関連するインテグレーション、機能、コンポーネントの計画を練る必要があります。マイクロサービスのガバナンスと管理においては、すべての機能およびセキュリティポリシーが、開発時だけでなく実行時にも確実に実施されるようにすることが不可欠です。

## 3.1 マイクロサービスにおけるコンテナセキュリティのベストプラクティス

### 3.1.1 コンテナの安全な構築と配備

次のベストプラクティスは、読者がコンテナを活用したマイクロサービスの配備モデルを選択し、この文書の第2章で示されているベストプラクティスに従っていることを前提としています。

**サーバをステートレスにします**
サーバは、特に利用者向けのコードを実行するメンバーの場合、互いに代替可能なグループのメンバーとして取り扱われなければなりません。それらはすべて同じ機能を実行するため、サイトリライアビリティエンジニアリング（SRE）チームはその機能について個別に考慮する必要はありません。唯一の問題は、そのワークロードをサポートするためのスケーラビリティ（つまり、サーバが十分にあるか）です。自動スケーリング、自己修復、およびフォールトトレラントインフラストラクチャを採用して、1つのサーバが動作を停止した場合に別のサーバに自動的に置き換えることができるようにすることが推奨されます。機能別にサーバを使用することは避けてください。

**ビルドとランタイムにおけるイメージのセキュリティ**
コンテナ固有のオペレーティングシステム（OS）を使用し、最小限必要なサービスとソフトウェアを採用してコンテナを実行することで、OSの攻撃対象面を減らすことが重要です。オペレータは、可能な限り読み取り専用のファイルシステムを使用するべきです。コンテナの新しい脆弱性の対策のための継続的なOSの更新が、環境のセキュリティを保護するために重要です。このようなOSの例は、Weave Igniteを使用したオープンソースのWeave Firekubeです。

開発者とオペレータは、類似の機能と同等の機微性を有するマイクロサービスを一緒にホストすることで、異なる機微性のアプリから分離することを確実にする必要があります。同じホスト上のコンテナで実行されているアプリケーションの機微性に基づいて、一貫したポリシーとコントロールを適用する必要があります。

ほとんどの脆弱性スキャンツールはVM用に構築されています。コンテナの場合、コンテナのスキャンをサポートしているツールを使用することが重要です。脆弱性に積極的に対処し、イメージやコンテナのデプロイメントで同じ脆弱性を複製してしまうことを防止する必要があります。

コンテナのビルドプロセスでは、アプリケーションのコンポーネントがビルドされ、イメージとして配置されます。イメージは、コンテナの実行に必要なすべてのファイルを含むパッケージです。アプリイメージは、下位のOSイメージで提供するすべてのOS機能とともに、アプリが必要とする実行可能ファイルとライブラリのみを含めるようにする必要があります。アプリ開発者は、可能な限り読み取り専用モードで階層化イメージとマスターイメージを使用することで、マスターイメージを変更して脆弱性を含めてしまわないようにする必要があります。マスターイメージに変更が必要な場合は、CI / CDパイプラインとセキュリティ検証プロセスを通す必要があります。目指すところは、アプリケーション開発者が下位のインフラストラクチャやパッチ適用について心配する必要がないように、ビルドプロセスを簡略化することです。

ビルドを作成するプロセスは通常、開発者によって管理され、開発者がテスト検証への引き渡しのため、アプリをパッケージ化する責任を負います。CI / CDパイプラインでは、ビルドを次のステージに進める前に、セキュリティ・バグバーとクオリティゲートを採用して実施する必要があります。DevSecOpsとして知られているこのパラダイムでは、アプリケーションの完了後にセキュリティを追加するのではなく、最初からセキュリティを組み込みます。「シフトレフト」セキュリティは、開発者がパイプラインの早い段階でセキュリティと品質の問題に対処し、最も右側の本番環境に安全で高品質のコードがデプロイされることを確実にします。このパイプラインのすべての段階でフィードバックループを設定し、前のステップの自動化と修正を採用することで、セキュリティの欠陥が本番環境に到達する可能性を減らします。

ソフトウェア開発ライフサイクル（SDLC）パイプラインでは、次の方法を組み合わせることで実現できます：

- 単体テストを拡張して、アプリケーションが期待どおりに動作することを確認します。
- サプライチェーンのセキュリティを検証し、サードパーティコンポーネントの評価と素性を確認します。
- システムが適切にセキュリティ強化され、構成の誤りや安全でないデフォルト構成についてテストされていることを確認します。
- 動的テストと静的テストをパイプラインに追加し、最も重要なアプリケーションリスクを評価し、見える化し、開発者をトレーニングします。
- OWASP Foundationチェックリストをパイプラインに適用、追加します。

イメージの作成後、イメージは、最終的なアプリケーションの機能を検証するためにテスト自動化ツールを使用してテストされ、認可される必要があります。セキュリティチームは、静的および動的なコードスキャンで見つかったすべての対処策が適用されていることと最終的なイメージがデジタル署名されていることを確認して、イメージを認証する必要があります。セキュリティ・バグバーに整合しない脆弱性がアプリケーションサービスで識別された場合、最終のビルドは失敗します。これは理想的には、セキュリティ・バグバーが適用された自動CI/CDパイプラインを使って実施される必要があります。ツールの利用は推奨される方法ですが、ツールは完全ではなく、コンテキストを理解できないため、人間による検証は不可欠です。人間が組織に対するリスクをトリアージして正確に計算するために、コードレビューでセキュリティの欠陥を正しく評価するにはコンテキストが最も重要です。そのため、コードレビューフレームワークを使用すると、コラボレーションを改善し、セキュリティの欠陥を減らし、標準化を強化できます。コードレビューにはいくつかのタイプがあり、一般的に正式なコードレビュー（例：Fagan検査または軽量コードレビュー、同期、非同期、またはインスタントコードレビュー）として分類されます。

**イメージストレージ**
イメージの保存とIAM権限に基づいたアクセスを一元管理すると、イメージの管理が容易になり、必要に応じて複数の環境に展開することもできます。イメージレジストリを使用すると、開発者はイメージを作成時に、公開して再利用しやすいように識別とバージョン管理のためのタグを付けてカタログ化して保存することが簡単にでき、他の開発者が作成したイメージも同じように見つけてダウンロードすることができるようになります。レジストリは、プライベートの場合も、サードパーティが提供するパブリックレジストリの場合もあります。
レジストリは、自動化のためのAPIを提供します（たとえば、テストされ、セキュリティと品質のバグバーを満たしているイメージのプッシュを自動化します）。

サードパーティのパブリックリポジトリを使用する場合には、サプライチェーンの管理を外部の組織に委託することになる点に注意する必要があります。これは許容できるでしょうが、組織が引き受けているリスクを理解して対処できるように、リスクについて徹底的に評価する必要があります。

**サードパーティイメージのセキュリティ**
コンテナイメージレジストリは、イメージの共有に使用される集中型のリポジトリです。Docker Hubなどのコミュニティベースのパブリックレジストリ、またはQuay、Docker Trusted Registry（DTR）、AWS Elastic Container Registry（ECR）、Google Container Registry、Microsoft Azure Container Registryなどのエンタープライズレジストリを使用することで、コンテナイメージが利用できるようになります。

ほとんどのエンタープライズレジストリでは、保存されているイメージのセキュリティは、ソフトウェアスイートの一部としてサービスプロバイダによって実施されます。たとえばDocker DTRには、アクティブなイメージスキャンを採用したDocker Security Scanningがあります。公開イメージを使用する場合には、たとえそのイメージが局所的にビルドされ利用されるものであっても、信頼できるユーザからコンテンツをダウンロードする、展開前にイメージをスキャンする、認定されたコンテンツを持つ厳選された公式イメージを使用するなどのベストプラクティスに従うことが推奨されます。エンタープライズイメージリポジトリの場合は、レジストリの監査プロセスを実装し、前のイメージと既知の依存関係にフラグが付けられ、その相互依存関係が記録されるようにすることが重要です。これらのイメージを、サードパーティのリポジトリからダウンロードする場合、良く知られている認証局（CA）が発行したTLS証明書を使用する必要があります。セキュリティ上のベストプラクティスとして、自己署名証明書を使用したり、暗号化されていないHTTP接続でレジストリを使用したりすることは、転送中のデータが改ざんされる可能性があるため推奨されません。さらに、イメージの真正性が検証されること、イメージをダウンロードする前には署名の検証が行われることが必要になります。

組織には通常、アプリケーションのすべてのコンポーネントに信頼できる唯一の情報源を提供する中央のアーティファクトリポジトリがあります。これにより、1つの場所でビルドされたアーティファクトとリリース候補の管理、コンポーネント品質とセキュリティの透明性、ライセンスの管理、ステージングやリリース機能など各段階のソフトウェア開発の最新化が可能になり、高可用性とクラスタリングによるDevSecOpsデリバリのスケーリングを支援します。

**CI / CDパイプラインのセキュリティ**
継続的インテグレーション/継続的デプロイメント（CI/CD）パイプラインは、安全なコードをリリースするサイクルを増やす一方で、インテグレーションとデプロイメント中のエラーを減らすことを目的としています。実装はそれぞれ異なりますが、ベストプラクティスを着実に実行することで、組織のセキュリティ体制が強化されます。開発の異なる段階でコードベースと資格情報に完全にアクセスできる

ため、CI/CDシステムを分離することは最も重要です。分離はさまざまな手段を使用して、異なる実装レイヤで実施できます。たとえば、CI/CDシステムは、内部の公開されていないネットワークにデプロイする必要があります。これらのシステムへのアクセスは、所定の自動化された監視メカニズムによって厳しく制限する必要があります。システムの複雑さに応じて、自動監視メカニズムを使用したシステムのリポジトリの保護が実施されるべきです。

コードのセキュリティテストは、CI/CDパイプラインの不可欠な要素です。パイプラインには、あるステージから別のステージへのコードの安全なテストを可能にするためのゲートキーパープロセスが必要です。これとは別に、開発者は共有リポジトリにコミットする前に、ローカルで行ったのと同じくらい多くのテストを実行する必要があります。開発者がコードの変更とテストを並行して早く頻繁に実行できるようにすることで、早期検知が可能になります。

CI/CDシステム自体もコンテナ化する必要があります。一時的で短命な環境にはさまざまな利点があり、そこでホストシステムを抽象化し、標準APIを提供するコンテナでセキュリティテストを実行します。このアプローチにより、後続のテストケースの実行において、前のテストの影響が残ってしまうことがなくなります。ステートレスマイクロサービスを使用すると、テストがはるかに簡単になることに注目してください。これによって、デプロイメントのためにデリバリされたときに、より検証しやすくセキュアになります。

CI/CDパイプラインをネットワークの他の部分から分離することで、開発者がパイプラインを部分的にも全体的にも回避できないことを確実にすることが重要です。同時に、ビルドパイプラインのID/アクセス管理は、実装の観点で重要です。適切な認証とアクセス許可を、開発、セキュリティ、運用、QA/テストチームの役割に基づいて適切に設定する必要があります。

コンテナ環境では、ほとんどのアプリケーションコンテナがビルドのためにマスターイメージを使用するため、意図せずに脆弱性を複製してしまうことが起こりやすくなります。したがって、マスターイメージのセキュリティ検証は最も重要です。さらに、マスターイメージを更新できるユーザに対する適切なアクセス制御と、適切なバージョンコントロールを確実に行うことが重要です。いかなるイメージのダウンロードについても、承認されているレジストリへのアクセスを制限することが不可欠です。組織のシナリオでは、プライベートコンテナレジストリを作成するのが堅実であり、そのレジストリに追加されるものはすべて、適切なセキュリティテストと検証を受ける必要があります。これにより、開発者がコンテナイメージをインターネットからダウンロードして、誤って脆弱性を複製することを防ぎます。

## 3.2 マイクロサービスの検知コントロール

アクセス、認証、キーストレージ、暗号化などの保護コントロール（インシデントの発生を防止するように設計されたコントロール）をこれまで取り上げたので、このセクションでは、全部というわけではありませんが、主に検知コントロール（進行中のインシデントを、特定し、特徴付け、アラートを提供し、解決するように設計されたコントロール）を取り上げます。

コンポーネントが内部的に動作し通信し、単一障害点を持つモノリシックアプリケーションとは対照的に、マイクロサービスアプリケーションは内部と外部の両方で動作して通信します。このアプリケーションの分離または個別のサービスへの分解は、アプリケーションの回復力を高める一方で、監視および保護する必要がある攻撃面を増やします。

さらに、アプリケーションを個別のコンポーネントに分解すると、マイクロサービスにスピード、レ

ジリエンス、柔軟性、および拡張性の利点がもたらされる一方、オーケストレーションされた分散型システムに伴うセキュリティの課題も生じます。これらのセキュリティの課題に対処するには、セキュアな開発、実装、運用、および継続的な監視において、インシデント管理とプラットフォームと同様に、ポリシー、ロギング、監視、アラートを考慮する必要があります。

### 3.2.1 ポリシー

ポリシーは、ロギング、監視、アラート、およびインシデント解決のアクションのベースラインを確立するため、先行して必要になります。ポリシーは、監視システムが、ふるまい、イベントと状況を評価するために使用する基準です。アプリケーションは、コンテナに格納されているかどうかに関係なく、マイクロサービスとしてコンポーネントに分解されているため、この組み合わせまたは分離による特定の機能とふるまいは、十分に理解して予測できます。これにより、特定の運用ポリシーを設定し、悪意のあるふるまい（標準のふるまいからの逸脱）の監視が可能になります。いかなるセキュリティポリシーの実装も、実行用に生成された構成のアーティファクトの一部として定義する必要があります（構成ファイルには、ツールやそれを適用するためのポリシーが必要になります）。これらのポリシーには、マイクロセグメンテーション、通信、構成、インターフェース、ロギングおよびアラートのターゲットなどを含めることができます。

開発者とオペレータは、以下のポリシーのベストプラクティスを採用する必要があります：

1. OS、ネットワーク、ホスト、コンテナ、アプリケーションおよびマイクロサービスレベルでポリシーを設定します。
2. コンテナ、マイクロサービスおよびアプリケーションなどの数に関係なく、サービスレベルポリシーを使用してサービスを保護し、セキュリティを確保します。
3. マイクロサービスがプロパティファイルをロードできる開示された集中構成管理サーバを設置します。
4. OS、ネットワーキング、コンテナ、マイクロサービスコンポーネント、アプリケーションレベルでポリシー管理プラットフォームを採用します。使用するポリシー、それらのパラメータ、およびそれらを管理システムに適用および実装する方法を判断します。
5. 詳細なポリシーを使用して、特定の深刻度のCVEの影響を受けるすべてのイメージなど、セキュリティ要件を満たさないイメージを検知します。
6. イメージのスキャンをして、ランタイムの適用と修正機能を対応させます。

### 3.2.2 ロギング

マイクロサービス/コンテナ戦略を利用するシステムで何が起こったかを判断するために使用できる、一番とは言えないまでも最も重要なツールのうちの1つは、ロギングシステムです。ロギングは、マイクロサービスを維持させ、健全な状態に保つための重要な役割を担っています。ログには、スタックトレースやデータの送信元に関する情報などの重要な情報が含まれており、サーバがクラッシュしたときに再構築するために役立ちます。マシンにSSHで接続してSTDOUT（標準出力）を確認することは、単一で実行中のインスタンスでは機能するかもしれませんが、高可用性と冗長性のあるマイクロサービスの世界では、これらすべてのログを共通の場所に集約し、異なる基準に基づいてクエリを実行する必要があります。たとえば、１つのインスタンスに問題がある場合、SRE（サイトリライアビリティエンジニア）は、プロセスID（PID）、ホスト名、およびポートを確認する必要があります。組織内のさまざまなチームが所有するサービスから発生しているので、集約されたログは独自の洞察とコンテキストを提供します。

ログファイルは、何が起こったか（イベント、トランザクション、メッセージ）の記録です。ロギングソリューションは、これらのアクションの発生時に、その保存、記録、分析、およびレポートを提供します。マイクロサービスやコンテナのアクティビティをログに記録する際の主要な課題の1つは、この分散アーキテクチャの一時的で短命な性質です。端的に言うと、コンテナとそれに含まれるマイクロサービスは永続的に存在するわけではないので、ロギングシステムを適切に設計するためには、このことを考慮する必要があります。

開発者とオペレータは、以下のロギングのベストプラクティスを採用する必要があります。

1. すべてのマイクロサービスリクエストの呼び出しに一意のIDをタグ付けして、いかなるエラーでも、エラーが発生したサーバやコンテナとマイクロサービス（アプリケーション）の呼び出しや応答を、破棄された後であっても追跡できるようにします。
2. エラー応答を一意のIDでコードにし、それら（コンテナおよびマイクロサービスのエラー）をより簡単にグループ化および分析できるようにします。
3. コンテナとマイクロサービスのログデータは、標準形式（JSONなど）で構成します。
4. 選択されたすべての標準形式のログフィールドを検索可能にします。
5. 集計、分析、レポートの目的でメッセージのUTC（協定世界時）時間を記録します。
6. 重要な情報の欠落を避けるために、記録を少なくするのではなく、より多くのデータを記録するようにしますが、コンテナとマイクロサービスのデータ量が原因で過剰な要求が発生する場合、開発者とオペレータは、削減を図ったり、オフラインストレージと分析ツールで作業できるように準備をする必要があります。
7. 専用のログ配布コンテナを介して、ログを流れるデータのストリームとして調査します。
8. 専用のログ配布コンテナを介して、すべてのログを一元管理された場所に転送し、コンテナとマイクロサービスのアクティビティのレポートと監視を容易にします。
9. コンテナのマイクロサービスが利用できない場合（コンテナが破棄された場合など）や、ストレージスペースのリソースの可用性のため、ログをホストの外部に保存します。
10. 目的とするツール（高速、大量のデータの処理、可視化とAIの使用）のコンテナとマイクロサービスを採用して、ログアクティビティを保存、集約、およびレポートします。

### 3.2.3 監視システムとアラート

監視は、システムが設計どおりに動作し、データが安全であることを確実にします。繰り返しになりますが、マイクロサービスやコンテナのアクティビティを監視して警告することに関する主な課題は、コンテナとマイクロサービスが永続的に存在するわけではないという事実です。

さらに、分散システム全体で数百から数千のコンテナとマイクロサービスが、何時でも同時に実行される可能性があります。監視は、これまで述べてきたシステムポリシーとロギングの組み合わせによって実施され、イベントやステータスなどの情報が収集、集約、要約、解釈されて、ダッシュボードとレポートに表示されます。

もう1つの大きな課題は、VMの監視とアラートに使用されていた従来のツールは、コンテナにはあまり効果的ではないことです。したがって、コンテナに対応し、エコシステム全体を可視化し、マルウェアの感染を検知し、コンテナイメージのランタイム検証を実行し、ネットワークマイクロセグメンテーションポリシーを適用し、コンテナの大量発生を識別し、リソースポリシーを適用できるツールを実装することが適切です。

アラートは、監視システムによって生成されるメッセージや通知であり、実行責任のある関係者に送信されて、誰が、何を、どこで、いつ、なぜ、何が必要かを判断するために、レビューする必要があるイベントやステータスを通知します。

開発者とオペレータは、以下の監視システムとアラートのベストプラクティスを採用する必要があります。

1. ホスト、コンテナ、マイクロサービスレベルで、エンドツーエンドの継続的に可視性の高い環境を構築します。
2. create、run、kill、launch、syscall、seccompなどのコンテナやマイクロサービスコマンドを制限および監視します。
3. ホスト、インフラストラクチャ、マイクロサービスレベルで主要な指標を特定して監視します。
4. サービスレジストリを使用して、実行時に利用可能なマイクロサービスの場所を保持します。
5. クライアント側やサーバ側の開示情報を使用して、使用可能なマイクロサービスを見つけます。
6. 複数のコンテナやマイクロサービスが「動作している」という観点でクラスタを監視します。
7. （複数のホスト、インフラストラクチャ、マイクロサービスに）分散された脅威を関連付けます。
8. コンテナやマイクロサービス情報の膨大で分散された性質に対応した、大規模で高速な相関分析を実装します。
9. コンテナやマイクロサービスのふるまいをベースラインに比較して分析し、悪意のあるアクティビティを検知します。
10. AI（予測分析）を使用したコンテナやマイクロサービスの要求量の分析により、ふるまい、パターン、分類を関連付けます。
11. 解釈しやすく、すべての主要な指標を含むダッシュボードを実装します。
    a. ダッシュボードの指標からホスト、インフラストラクチャ、マイクロサービスのログにドリルダウンする機能を備えます。
    b. ホスト、インフラストラクチャ、マイクロサービスのログからダッシュボードの指標にドリルアップする機能を備えます。
12. DevSecOpsによってマイクロサービスチームメンバーに割り当てられたアラートに応じ、適切な実行責任者に直ちにアラートを送信します。

13. 解決の方向性を含めた明確で実行可能なアラートを作成します。

### 3.2.4 インシデント管理およびフォレンジックとプラットフォーム

情報セキュリティにこれまで対処したことのある人なら誰でも、セキュリティインシデントの蔓延が、関係者全員にとってどれほどひどいストレスをかけるかを知っています。このため、インシデントを特定するためのツールや手法の準備と同様に、インシデントを封じ込めて迅速に対応する能力も重要です。

インシデントレスポンスチームやデジタルフォレンジック調査員が直面する課題もテクノロジとともに進化します。その解決策は、コンピュータやサーバのハードディスクだけでなく、多くの場所で見つけることができます。クラウドストレージとサービスは、ビジネスの観点から見ると非常に便利で費用対効果に優れていますが、証拠保全の観点から見ると、形勢を一変させてしまうものでもあります。

インシデントレスポンスとデジタルフォレンジックの役割は明確に重複しています。この重複を認識し理解することは、インシデントレスポンスがデジタルフォレンジック調査に変わったとき、または逆に、調査からインシデントレスポンスをトリガーする必要が生じたときに、チームがそれぞれの組織に価値を提供するのに間違いなく役立ちます。情報セキュリティの他の側面におけるリスクのバランスをとることは重要であり、インシデントのフォローアップとして完全なデジタルフォレンジック調査をしないリスクについてもバランスをとる必要があります。

コンテナは一時的で短命になるように設計されており、実際に複数のコンテナセキュリティ製品がこの機能をセキュリティ機能として重視しています。コンテナが5分間しか実行されない場合、攻撃者がコンテナを侵害しても、一度にアクセスできるのは5分間だけです。コンテナのこの特性は、基本的にフォレンジックが必要とする、証拠を保存するために必要なことに反して実行されます。常に開始および停止するコンテナのイメージは、移動するターゲットということにとどまらず、頻繁に存在しなくなるターゲットです。ただし、大多数のコンテナプラットフォームではコピーオンライト方式のファイルシステムを使用しているため、フォレンジック調査員が実行中のコンテナを調査するときに非常に役立ちます。基になるコンテナイメージは1つの場所に格納され、このイメージには、コンテナイメージを形成する構成データとアプリケーションが含まれています。
コンテナの実行中に行われたいかなる変更も別のファイルに書き込まれ、実行中のコンテナに影響を与えることなく、実際にその新しいイメージに対してその場でコミットできます。コンテナが侵害されていると思われる場合、フォレンジック調査員はその新しくコミットされたイメージを実行することでその内容を調査できます。このようなアクションを実行すると、イメージからオリジナルのコミットされたコンテナと同じではない、新しいコンテナが作成されることに注意してください。ターゲットコンテナで実行中のプロセスはイメージに含まれないため、これはVMのスナップショットアプローチとは異なります。

マルウェアがコンテナイメージを回避してホストマシン上のリソースにアクセスすることを可能にする脆弱性の例がいくつかあります。コンテナ管理プラットフォームのセキュリティ更新は頻繁に行われ、バグが発見されるとすぐにパッチが適用されます。したがって、コンテナが何らかのマルウェアやその他の悪意のある攻撃者の影響を受けると考えられる場合、調査者はコンテナを単に「侵害された別のアプリケーション」となるように、ホストマシン全体のイメージを別に作成して対処することができます。

クラウド内のコンテナでホストされているかどうかに関係なく、アプリケーションは、異なるライフサイクルで運用され、1つまたは複数の他のマイクロサービスと相互作用する、数百ではないにせよ数十のマイクロサービスに分離される可能性があり、永続的に存在しないことは、インシデントの、識別、ロギング、分類、優先順位付け、調査、診断、エスカレーション、回避策や解決とクローズにおける課題

です。この課題には、監視と人工知能を組み合わせたプラットフォームが最も効率的かつ効果的に対処できます。

開発者とオペレータは、以下のインシデントとフォレンジックのベストプラクティスを採用する必要があります。

1. すべてのマイクロサービスインフラストラクチャアクティビティを記録して、基準となるベースラインを作成します。
2. すべてのイメージ、オーケストレータ、ホスト、コンテナ、イベント、トランザクション、およびメッセージを記録します。
3. すべてのイメージをスキャンし、このランタイムポリシーの実施と修復を調整します。
4. 後のフォレンジックレビューのため、すべての情報をオフラインで保存します。
5. ダッシュボードの指標からホスト、インフラストラクチャ、マイクロサービスのログにドリルアップまたはドリルダウンします。
6. 上記のベストプラクティスを組み合わせ、AI（ベイズ分類器）を適用して、発生前、発生中、発生後のインシデントを検知します。
7. インシデントにAIを適用して、重大性、適切なルーティング、および可能な修復シナリオの適用を判定します。
8. アラート監視を採用し、インシデントへの対応の実行責任を負う人々に通知します。チームはマイクロサービスをミラーリングする必要があります。
9. DevSecOpsによってインシデント処理を可能な限り自動化することを確実にし、マイクロサービスチームによるアラート、指標、ダッシュボードが分離できるよう、監視プラットフォームの利用を可能にします。
10. インシデント管理戦略を確立します。サイトリライアビリティエンジニアリング（SRE）は、信頼性を基礎としてインシデント対応を行うことで、ストレスの多いイベントに対応するサポートチームに体制と親しみやすさを提供すると同時に復旧時間を短縮できます。
11. インシデント解決後、ポリシーの変更を含め、修正が適切な担当者によって実際に実行されていることを必要に応じて確実にするため、DevSecOpsや他の関係者に報告します。

### 3.2.5 サードパーティとの情報交換

適切なセキュリティ対策の一環として、ホワイトリストまたはブラックリストに基づくセグメンテーションが広く使用されており、マイクロサービスとそのアーキテクチャの違いによってオペレータはこの方法をより有効に利用することができます。マイクロサービスはパブリックエンドポイントを公開しているので、高度な標的型攻撃やその他の悪意のある攻撃者がアクセスを試みることに対して脆弱になる可能性があります。このタイプのデータの課題は、データの検証と標準化です。

いくつかの企業（例えば、IBM X-Force、Facebook ThreatExchange、AlienVault）は、これらのプラットフォームへのアクセスを提供しています。Open Threat Exchange（OTX）は、脅威インテリジェンスへの無料アクセスも提供しています。
マイクロサービスでは、サードパーティのコミュニティを利用した脅威データを利用して、さまざまなサービスをマイクロセグメント化し、協調的な防御を効果的に実現できます。

### 3.2.6 レスポンスへの一意のIDの付与

マイクロサービスのユーザがエラーに直面する場合があります。そのエラーの原因を知ることが重要です。したがって開発者は、クライアントが受け取る応答には、一意のIDとエラーに関するその他の有用な情報が含まれるようなコードにする必要があります。この一意のIDは、要求を関連付けるために使用されたものと同じかもしれません。要求の応答のペイロードに一意のIDを含めると、そのIDを使っているサービスにおける問題をより迅速に特定するのに役立ちます。リクエストの日付、時間、およびその他の詳細のパラメータは、インシデント対応者が問題をよりよく理解するのに役立ちます。また、オープンソースの検索および分析エンジンを介してサポートエンジニアにデータへのアクセスを提供することを推奨します。

### 3.2.7 ログデータの構造化

一部のログは他よりも多くのフィールドを必要とする場合があり、これらの過剰なフィールドをすべて必要としないログは記憶容量を浪費してしまうので、ログデータのフォーマットを定義することはほとんど不可能です。マイクロサービスアーキテクチャは、さまざまなテクノロジの積み重ねを利用してこの問題に対処し、各サービスのログ形式に影響を与えます。1つのサービスはコンマを使用してフィールドを区切り、他のサービスはパイプまたはスペースを使用します。

これらすべてが複雑さをもたらしています。したがって、ログデータをJavaScript Object Notation（JSON）のような標準形式に構造化することにより、ログの解析を簡素化することが重要です。JSONを使用すると、分析者が1つのログイベントで必要に応じてより多くのセマンティック情報を取得できるように、複数のレベルのデータを保持できます。または、スキーマオンリードを使用して、半構造化データをその場でフォーマットすることもできます。

解析は、特定のログ形式を処理するよりも簡単です。構造化データを使用することで、ログに異なるフィールドが含まれていても、ログの形式を標準化することができます。

**ログのコンテキスト化**

開発者とオペレータは、以下のログのベストプラクティスを採用し、ログに確実に含まれるようにする

必要があります。

- 日時;
- スタックエラー。例外オブジェクトをパラメータとしてロギングライブラリに渡せるようにします。
- どのログがどのマイクロサービスのものかを区別するための、サービス名やコード。
- サービスアナリストが問題の発生場所を推測する必要がないようにするための、エラーが発生した関数、クラス、またはファイル名。
- 開発者がDBへのどの呼び出しが問題のある呼び出しであるかを認識するための、外部サービスの相互作用名。
- サーバおよびクライアント要求のIPアドレス。この情報により、異常なサーバを特定したり、DOSまたはDDOS攻撃を特定したりすることが容易になります。
- 問題が発生しているブラウザーまたはユーザを特定するための、アプリケーションのユーザエージェント。
- エラーのセマンティクスをさらに取得するためのHTTPコード。これらのコードはアラートを作成するのに役立ちます。
- ログによりリクエストをコンテキスト化すると、システムの問題へのトラブルシューティングの時間を節約できます。

## 3.3 マイクロサービスメッセージングパターン

マイクロサービスアーキテクチャでは、サービスは自律的にネットワークを介して通信し、ビジネスのニーズに対応します。そのようなサービスの集合体がシステムを形成し、利用者はしばしばそれらのシステムとやり取りします。したがって、マイクロサービスベースのアプリケーションは、異なるネットワーク上で複数のサービスを実行している分散システムと考えることができます。与えられたサービスは、それ自身のプロセス上で動作します。したがって、マイクロサービスは、プロセス間またはサービス間の通信スタイルを使用して相互接続します。

マイクロサービスの通信スタイルは、主に1つのサービスから別のサービスへデータを送受信する方法です。マイクロサービスで使用される通信スタイルの最も一般的なタイプは、同期型と非同期型です。

**同期通信とプロトコル**

同期通信では、クライアントはリクエストを送信すると同時に、サービスからのレスポンスを待ちます。クライアントがレスポンスを受信するまで、両方の接続を開いたままにする必要があります。クライアントの実行ロジックは、レスポンスなしでは続けられません。たとえば、HTTPは同期プロトコルです。クライアントは、リクエストを送信し、サービスからのレスポンスが届くまで一時停止します。リクエストは、同期（スレッドがブロックされている）または非同期（スレッドはブロックされず、レスポンスはコールバックに最終的に到達する）であるクライアントコードの実行の主導権を有します。ここで重要な点は、プロトコル（HTTP / HTTPS）が同期を維持し、利用者コードがHTTPサーバのレスポンスを受信した場合にのみタスクを継続できることです。

RESTはどの実装プロトコルにも依存しませんが、最も一般的な実装はHTTPアプリケーションプロトコルです。ユーザがHTTPプロトコルでRESTfulリソースにアクセスすると、リソースURLはリソース識別子として機能し、GET、PUT、DELETE、POST、およびHEADは標準のHTTP操作として、そのリソースで実行されます。 RESTアーキテクチャスタイルは、もともと同期メッセージングに基づいています。

**非同期通信とプロトコル**

マイクロサービスアーキテクチャの初期の実装では、事実上のサービス間通信スタイルとして、同期通信が採用されていました。ただし、マイクロサービス間の非同期通信はサービスをより自律的にするため、一般的になっています。

非同期通信では、クライアントはレスポンスをタイムリーには待ちません。クライアントがレスポンスをまったく受信しない、もしくはレスポンスは別のチャネルを介して非同期に受信します。マイクロサービス間のこのメッセージングは、軽量で単純なメッセージブローカーを使用して実装されます。ブローカーにはビジネスロジックはなく、高可用性を備えた集中型エンティティです。非同期メッセージングスタイルには、主に単一受信者と複数受信者の2つのタイプがあります。

a. 単一受信者。 各リクエストは、1つの受信者またはサービスのみによって処理される必要があります。 このステートメントの例として、Commandパターンがあります。
b. 複数受信者。 各リクエストは、ゼロから複数の受信者で処理できます。 このタイプの通信は非同期である必要があります。例えば、イベントドリブンアーキテクチャなどのパターンで使用されるパブリッシュ/サブスクライブメカニズムがあります。 これは、イベントを介して複数のマイクロサービス間でデータ変更を伝播するときのイベントバスインターフェイスまたはメッセージブローカーに基づいています。これは通常、トピックやサブスクリプションを使用して、Microsoft Azureサービスバスのような類似のアーティファクト、もしくはサービスバスを介して実装されます。

マイクロサービスベースのアプリケーションは、多くの場合、これらの通信スタイルを組み合わせて使います。 最も一般的なタイプは、通常のWeb API HTTPサービスを呼び出す際にHTTP/HTTPSなどの同期プロトコルを使用した単一受信者通信です。 マイクロサービスは、HTTP/2を利用して、ワークフローのセキュリティと速度を向上させることもできます。多重化されているため、単一の接続で並列処理を行えるようにします。

多くのオペレーティングシステムやクラウド環境でサポートされているプロトコルであるAdvanced Message Queing Protocol（AMQP）などの他のプロトコルは、非同期メッセージを使用します。RabbitMQキューまたは他のメッセージエージェントにメッセージを送信するときと同じようにメッセージを送信します。

ActiveMQやRabbitMQなど、他にもいくつかのブローカーがあります。これらのブローカーは、中規模から大規模の非同期メッセージングシナリオ向けのpub-subメッセージング機能を提供できます。 完全に分散されたスケーラブルな非同期メッセージングインフラストラクチャのため、クラウドプロバイダはAWS Kinesis、Azure Event Hub、Google Pub / Subなどのシステムを提供します。 一部のクラウドプロバイダもKafkaを管理しています。

# 3.4 マイクロサービスガバナンス

開発者がマイクロサービスアーキテクチャに移行すると、従来のガバナンス概念のほとんどが通用しなくなります。マイクロサービスでのガバナンス概念は、個々の開発チームに独自ドメインを管理する自由を与える分散プロセスとして解釈されます。 分散型ガバナンスは、サービス開発、デプロイメント、および実行プロセスに適用できますが、考慮すべき他の側面があります。

マイクロサービスガバナンスは、実世界のソリューションを実装するために組み合わされる人、プロセス、およびテクノロジで構成されます。これらの概念のほとんどは新しいものではありませんが、SOAガバナンスですでに正常に使用されているものです。マイクロサービスアーキテクチャでも同様に適用できます。

**サービス定義**
開発されるマイクロサービスやマイクロサービス自体の機能、そして利用者がマイクロサービスをどのように利用するか、を一意に識別するのに十分な情報が必要です。サービス定義を明示するメカニズムを備え、サービス利用者がすぐに利用できる必要があります。

OpenAPI、gRPC-Web、プロトコルバッファなど、サービスインターフェイスの定義に役立ついくつかのテクノロジがあります。これらのテクノロジにより、開発者はサービス識別子、サービスインターフェイス、およびサービスメッセージモデル（サービスリクエストやレスポンスなど）を定義できます。サービス所有権、SLA、およびそれに付随するSLO（サービスレベル目標）およびSLI（サービスレベル指標またはターゲット）などの他のサービスメタデータも、サービス定義の一部にあたります。通常、サービス定義はセントラルリポジトリに格納され、利用者はそれにアクセスでき、サービス所有者はそれを公開できます。

**サービスライフサイクル管理**
マイクロサービスには、異なるライフサイクルステージ（計画、設計、実装、デプロイ、維持、廃止など）があります。マイクロサービスの非中心型と分散型の性質を考えると、これらのタスクの所有権は各マイクロサービスを所有するチームにあります。実用的な目的のためには、ビジネススコープや開発に使用する技術に関係なく、マイクロサービスのライフサイクルステージを統一するのが一般的です。サービスライフサイクル管理の技術は、マイクロサービスアーキテクチャに集中的に適用されます。これには、デプロイのライフサイクル管理、構成管理の方法、バージョンサービスなどが含まれます。これらの機能のほとんどは、API管理やサービスメッシュのコントロールプレーンの一部として実装されています。

**Quality of Service (QoS)**
サービスを利用者に公開する前に、考慮されるべきQuality of Service（QoS）の側面が多数あります。サービスは、セキュリティで保護されたサービスとして公開され、さまざまなセキュリティのプロトコルと標準（TLS、アクセストークンなどをはじめとする）を活用します。

帯域制限とスロットリングを使用して、サービスへのアクセスを制御する必要があります。キャッシングや各種のフックをモニタリングと計測に組み込むことは、QoSの重要な機能の一つです。 これらの要件のほとんどは、マイクロサービスのガバナンスに直接関連しており、集中管理されています。

**開発ライフサイクル管理**

サービスのライフサイクル管理は、多くの場合、サービスのデプロイメントレベルで実装されます（つまり、特定のサービスを複数の環境にデプロイする必要がある場合）。デプロイメントプロセスは、これらの環境全体で同じサービスコードを複製または移動する要件に対応します。これには、さまざまなDevSecOps関連のデプロイメント方法（blue-green、カナリア（canary）、ABテストなど）や、開発(dev)、テスト、品質保証(QA)、ステージング、本番前、本番(prod)などのさまざまな環境の管理方法も含まれます。

**API管理**

APIゲートウェイとAPI管理は、いくつかのマイクロサービスガバナンスを実現する上で重要な役割を果たします。 API管理の一部として、実行時にサービスにセキュリティ、サービスのバージョン管理、スロットル、キャッシング、収益化(monetization)などを適用することが重要です。これらの機能のほとんどは、サービスを呼び出すために集中的に適用する必要があることを理解することが重要です。したがって、APIゲートウェイは一元的に統治または管理する必要があり、これらの機能の適用は一元化または分散化できます。また、APIゲートウェイは、外部または内部の利用者のために使用できます。これらの機能は、マイクロサービスが内部APIゲートウェイを介して互いに通信する場合にも同様に適用できます。API管理ソリューションは多くの場合、サービスレジストリと連携してサービスを検出し、APIリポジトリとして使用します。これは、既存のサービスが使用され、それらから新しいAPIが作成される場合にも非常に役立ちます。その他の重要な側面の1つは、API管理ソリューションがAPIを検出して使用するための豊富な機能を提供することです。したがって、API管理を利用してすべてのマイクロサービスを管理することが可能です。

**サービスの可観測性**

アプリケーションが複数のマイクロサービスとやり取りする場合、すべてのサービスについてメトリック、追跡、ロギング、視覚化、および検出やアラートの機能を持つことが不可欠です。それにより、サポートやトラブルシューティングの目的で相互のやり取りの明確なイメージを得ることができます。
これらの要件はすべて、可観測性と呼ばれる1つの概念の下に統合されています。マイクロサービスアーキテクチャでは、数百または数千のサービスが相互に通信する可能性をもっています。サービスメトリックの取得、メッセージとサービスのやり取りの追跡、サービスログの取得、サービスの実行時依存関係の理解、障害発生時のトラブルシューティング、異常アラートの設定はすべて、可観測性の傘下で考慮されます。

**サービスのレジストリと検出**

個々のサービス定義、サービス識別子の定義、メッセージモデル、インターフェースなどは、集中管理なしで実行できるものです。ただし、これらのサービス定義は、一元化されたサービスレジストリに公開する必要があります。サービスレジストリは集中化されたコンポーネントであり、サービスを記述するための標準モデルも定義します。すべてのサービス所有者は、サービスの定義をその標準形式でサービスレジストリに公開する必要があります。サービスは大きく異なるテクノロジー（OpenAPIとgRPCなど）で実装されていますが、そのようなサービスの共通メタデータが存在する場合があり、それはサービスレジストリに追加する必要があります。また、セキュリティの観点から、すべてのマイクロサービスを一元化されたリポジトリに登録して、セキュリティツールで必要な追加のランタイムメタデータを取得することも重要です。サービスレジストリの完全性は最も重要です。サービスレジストリデータベースが破損すると、サービスリクエストが誤ったサービスにリダイレクトされ、サービスが拒否されたり、悪意のあるサービスにリダイレクトされたりして、アプリケーション全体が侵害される可能性があります。

# 4.0 モノリシックアプリケーションの分解

## 4.1 マイクロサービス： ユースケース

以下では、マイクロサービスのユースケースを概要レベルで説明します。

### 4.1.1 マイクロサービスへのアプリケーションの分解

モノリシックアプリケーションをより小さなパーツに分割するアプローチには、分解とリファクタリングが含まれます。分解は、コードベースを分析し、アプリケーションをコンポーネントパーツに編成するための戦略的な作業です。リファクタリングは、外部的振る舞いを保ちつつ、既存のコンピューターコードを再構築します。分解は静的な依存関係を解き明かすことを目指し、その影響を最小限に抑えますが、ソフトウェアのリファクタリングは、ソフトウェアの機能以外の属性の改善を対象としています。非機能要件の例には、レジリエンス、セキュリティ、修正可能性、保守性が含まれます。分解とリファクタリングを組み合わせることで、ソフトウェアコードを再編成し、内部の特性に望ましい変更を加えるためのパラダイムが作成されます。

短期的にはモノリスの方が良い場合もあります。モノリスは、製品を市場に投入する必要がある場合や、便宜的に機能上のギャップを埋める場合に適しているかもしれません。ソフトウェア開発の観点からは、モノリスで構築することはマイクロサービスのリファクタリングを念頭に置いて行うこともできます。これは「捨てるために構築する」と解釈されることもありますが、過渡的なアーキテクチャには価値があります。それは、リリースまでは知られていなかった要件を明らかにしたり、初期条件に敏感な特定の要件や機能を発見したりすることができます。モノリシックアプリケーションのNFR（非機能要件）に例示される初期のソフトウェアアーキテクチャの品質特性が、モジュール性を維持するために注意を払ってマイクロサービスのリファクタリングに持ち越される限り、ソフトウェアの構築はアーキテクチャのスタイルを選択することの問題であり、開発時だけの問題ではありません。

NFR は、しばしば "-ilities"と表示されます。より完全な扱い方は、カーネギーメロン大学ソフトウェア工学研究所から出版されたBass, Clements and Kazmanの「Software Architecture 3rd Edition」というタイトルの教科書に記載されています。ソフトウェアアーキテクトと開発のリーダーがアプリケーションの成功の鍵となるNFRを受け取ると、後のコード検査でNFRを明示したソフトウェアの構造が明らかになるように、開発のリーダーはそれらのNFRを使う必要があります。マイクロサービスアーキテクチャのスタイルは、スケールとモジュール性の問題に対応しています。開発者は、アプリケーションの現在の拡張性と期待される拡張性のニーズに基づいて、このスタイルの使用を検討すべきです。マイクロサービスアーキテクチャスタイルは、テスト可能性や適応性のようなモノリスコードベースのリファクタリングに、新しい追加のアーキテクチャ品質属性(NFR)を導入します。開発者は迷わずに、「最もシンプルなサービスリファクタリングでうまくいくものは何か？外部の動作に影響を与えずにモノリスから剥がせるものは何か？」を確認すべきです。モノリスは、技術的な理由だけでなく、業務遂行上の理由からも分解してリファクタリングする価値があるものでなければなりません。開発者は、マイクロサービスの開発、配信、および継続的なサポートの両方で経験を積むために、小さく始めるべきです。

アプリケーションをマイクロサービスに分解する際に遭遇するまたは認識される課題には、次の視点が含まれます:

開発者の視点：

1. アプリケーションの断片化が進むと、開発者はマイクロサービスの可視性が低下し、各コンポーネントが他のコンポーネントと問題なく動作することを保証するという課題が生じます。適切な機能がないと、データの流出、データの破壊、可用性の問題のリスクが高まります。
2. 他のいくつかのマイクロサービスとインターフェースを持つマイクロサービスを作成または維持する開発者は、相互運用性を確保するための課題に取り組んでいます。信頼できるテストケースが必要です。弱いテストケースは、ソフトウェア開発サイクルの後半で可用性とパフォーマンスの問題のリスクを高める可能性があります。
3. モノリスの分解の結果、内部ソフトウェアモジュールは、顧客の利用面に向いているもの、プロセス間通信に関連する内部ユーティリティまたは中間機能であるもの、記録システムまたは参照システムであるデータストアに向いているものに分類されます。分解はまた、インターフェースの割り当て、TCP/IP プロトコルの使用状況、データストアの位置、静的に宣言された変数とクレデンシャルの存在、および コンパイラやインタプリタに使用される特定の言語とは独立して特定のタスクを達成するために併用される複数のプログラミング言語の発見をもたらす。ここで見いだされたもののいずれかは、マイクロサービスの採用の潜在的な源となりうるもの、または採用を妨げる制約の源となりうるものです。
4. マイクロサービスアーキテクチャアプローチはRESTfulなメソッド駆動型のスタイルであり、すべての状況がアプローチに簡単に適合するわけではありません。SOAPプロトコルバックエンド接続はWS-SecurityまたはWS-Reliability SOAP拡張機能を使用している可能性があり、SOAPインターフェイスリファクタリングでは、RESTがそのようなセキュリティと信頼性のNFRを最終的なデータ整合性のみを提供する開発スタイルに引き継ぐ必要があります。開発者は、他のSECOPSチームと協力して、RESTfulスタイルによって提供される機能がSOAPによって提供される機能を複製できるようにする必要があります。さらに、レガシーバックエンドデータストアやその他のアプリケーションはRESTをサポートしていない可能性があり、開発者はSOAPエンドポイントをRESTでラップする必要があり、これは一般にパフォーマンスのオーバーヘッドになります。モノリスがRPC（リモートプロシージャコール）とNFS（ネットワークファイル共有）を使用する場合、使用法を問題解決するのが難しい場合があります。たとえば、開発者は、NFSを置き換えるためにファイル転送にHTTP/TLSを使用することを評価する必要があり、NFSで使用される認証およびアクセス方法の再評価と潜在的な置き換えを必要とします。別の例では、RPCはTLSトランスポートを簡単に使用できません。JSON-RPCを使用したい場合は、モジュールを完全に書き直す必要があります。RPC over SSL/TLSは、取り組むのが難しい問題です。標準では、RPC/TLSサポートを削除し、代わりに、信頼できないネットワークを介したリモートプロシージャコールを処理するためのOS固有のトンネリングメカニズムを採用します。歴史的に、RPCには、DoS、アクセス、および認可に関連した脆弱性があります。

5. マイクロサービスのスタイルでは、すべての開発者が「ネットワークプログラミング」の開発者となります。モノリスの制約の中では、コンテキストに境界を持たせることが知られており、モジュール間で信頼が構築されます。マイクロサービスは、アプリケーションのコンテキストを変更します。開発チームは、APIコールがネットワークを横断して変化するレイテンシと連続した往復時間のパフォーマンスを提供するためのコードを書かなければなりません。開発者はまた、スタイル自体が対応できないRESTfulの弱点を緩和する必要があります。それは、API 認証、暗号化されたアイデンティティ、ペイロードの暗号化、HTTPS の不正使用、API 鍵の強度、ロジックとセキュリティを間違った場所と間違った時間に適用する一般的な可能性などです。程度の差こそあれ、モノリスは独自の内部処理境界内でこれらのニーズを処理していました。モノリスを分解すると、開発者は機能を満たすためのコードを書くか、ニーズを満たすことができるネットワークエンジニアリングとセキュリティエンジニアリングの制御を継承しなければならなくなります。

6. マイクロサービスのコードを配信することで、コードプロモーションとテストプロセスが再構築されます。この変更の主な推進要因は、コンテナの要求と配送、ユニットテストとエンドツーエンドのテストの両方の観点から、開発者は一般的にシフトレフトを必要とします。コンテナへのソフトウェアのプロモートは、ローカルの静的セキュリティテストとコード品質評価のような簡単なものから、認証されたプロキシを介してローカルリポジトリから安全なFTPを実行するものまで、また、必要な安全なコードテストをすべて含むパイプラインデプロイジョブを使用して、次の最上位の開発環境でマイクロサービスとコンテナをブートストラップ（セルフスタート）するような複雑なものまで、さまざまです。
モノリシックなアーキテクチャの下では、これはovert testingの問題ではありません。他の複数のマイクロサービスと連動するマイクロサービスを書いたり維持したりする開発者は、相互運用性を確保することが難しいかもしれません。信頼性が高く再現性のあるテストケースのライブラリ、特にユニットテストに加えてエンドツーエンドのインターフェイステストを実行できるものが必要です。また、開発者は、サードパーティのコンテンツへの呼び出しを模倣し、データや値を返す合成スタブサービスを記述する必要があります（サービスの仮想化として知られています）。堅牢なマイクロサービステスト、合成データ、および SWAGGER のような API ドキュメンテーションフレームワークがないと、予期せぬセキュリティやパフォーマンスの欠陥につながるコードプロモーションのリスクが高まり、不注意な情報開示まで含めてリスクが高まります。

<u>オペレータの視点:</u>

1. アプリケーションが様々なコンポーネントに分解されていく中で、開発プロセスやステージング、本番環境で使用するための実績のあるコードプロモーションシステムを提供することは、継続的な DevSecOps の課題となっています。これがないと、コードプロモーションの遅れが増える可能性があります。
2. マイクロサービスは、レガシーアプリケーションに見られるものよりも、コンポーネント間のセキュアな通信に対する要求が高まります。オペレータは、選択されたマイクロサービス間の柔軟で安全なネットワーク転送を提供することが課題となる可能性があります。これがないと、情報の流出やシステムの危殆化を招く可能性があります。
3. オペレーションは、コンテナが実行されなくなったときに、マイクロサービスが状態を失わないようにするための設計責任を共有しています。サービスの状態が失われると、サービスの中断やデータの喪失につながります。信頼性と再現性の高いマイクロサービ

ス製品の提供は、3つの異なる構成要素によって支えられています。オペレーションはコンテナのライフサイクルを制御し、セキュリティはコンテナイメージの忠実性と完全性を制御し、開発者はコンテナのサービスレベルとオペレーションレベルの合意を守ります。レジリエンスやセキュリティなどの特定の非機能要件は、コードとインフラストラクチャの両方によって提供されます。マイクロサービスは、レガシーアプリケーションに見られるものよりも、コンポーネント間の安全な通信に対する要求が高くなります。オペレータは、選択されたマイクロサービス間の柔軟で安全なネットワーク転送を提供することが課題となる可能性があります。これがないと、意図しない情報の漏洩やシステムの危殆化につながります。

4. アプリケーションが断片化すると、運用チームはマイクロサービスアプリケーションの動作を把握できなくなります。可視性の課題により、各コンポーネントが他のマイクロサービスやサポートプラットフォームと連携してどのように機能するかに関して、稼働環境が不安定になります。モノリスは均一かつ一方的に監視できます。すべてのスライスされたサービスにとって、マイクロサービスのアーキテクチャスタイルは非常に重要です。リファクタリングされたサービス１つごとに、2つのアプリケーションモニタリングポイント、1つのマイクロアプリケーションログストリーム、1つ（または複数）のコンテナーモニタリングポイント、そして場合によっては1つのI / Oデータストア監視ポイントが生成されます。ランドスケープは、リファレンスまたはレコードのバックエンドシステムに関連付けられた内部API RESTfulマイクロサービスと、カスタマーエクスペリエンスの提供の一部である外部API RESTfulマイクロサービスにさらに分割できます。最新のアプリケーションパフォーマンスプラットフォームでは、複数のマイクロサービスとコンテナでテレメトリをサポートするためにサービスメッシュが必要です。サービスメッシュには、コンテナ固有のセキュリティ、ポリシーベースのネットワークセキュリティ、および従来のセキュリティと情報管理プラットフォームとの統合が可能なアプリケーションパフォーマンスロギングの集中化で構成される基本的なバックネットが必要です。拡張するマイクロサービスアーキテクチャでのイベント管理と機械が読み取り可能なロギングは、ビッグデータ分析の要求に進化していきます。スケーラブルなイベント管理プラットフォームがないと、長期にわたってログ情報を保持できないために、データの流出、破壊、および完全な喪失のリスクが高まります。

アーキテクトの視点

1. アーキテクトは、マイクロサービスアーキテクチャに再構築し、それらのマイクロサービスを調整することのコストとメリットのバランスを見つけることが求められます。バランスが取れていない場合、コストが超過するか、マイクロサービスアーキテクチャのメリットを十分に活用できないアプリケーションが発生する可能性があります。アーキテクチャとは、人によって異なることを意味します。一般的には、経験豊富で多様なスキルを持つソフトウェアプロジェクトマネージャー、上級開発者、またはビジネスプロセスアナリストは、一歩下がってビジネスステークホルダーと複数の技術チームの間のコミュニケーションギャップを埋めることができる人が、「アーキテクト」の役割を果たすことができます。しかし、すべてのアーキテクトは、コードの近くにいる必要があります。アーキテクトは、教育またはレジャーの目的でコードを書く必要がありますが、必ずしも職業としてソフトウェアを書く必要はありません。アプリケーションの分解は、ソフトウェア分野の経験に基づきます。その結果、「アーキテクト」は、アジャイル開発クルー、クループログラムマネージャーとスクラムマスター、技術チーム、およびビジネスステークホルダーの間で作業するソフトウェアアーキテクトとして実際に機能しています。ソリューション、インフラストラクチャ、セキュリティ、データ、ネットワークアーキテクトなど、すべての情報技術アーキテクトがこの役割で機能するわけではありません。アプリケーションの分解とリファクタリングは、グループ間の境界を越えて機能するソフトウェアアーキテクトを常に必要とするわけではありませんが、次のスプリント期間の前に作業をプロモートさせ、非機能要件に焦点を当て続け、開発リーダーと製品マネージャが作業と全体像に集中し続けるのに役立ちます。ソフトウェアテストおよびQAプロセスで役割を果たさないソフトウェアアーキテクトは、コードが合意されたアーキテクチャの品質属性（NFRなど）を満たしているかどうかを確認できません。開発者は機能要件の履行を制御しますが、アーキテクトは非機能要件の履行を制御して影響を与えます。この役割で権限を与えられていない場合は、セキュリティ、レジリエンス、変更可能性、モジュール性、可用性などのNFRがあるかどうかの評価は、アジャイルチームまたはテスト組織に委ねられますが、どちらもNFRを考慮に入れる必要はありません。

### 4.1.2 マイクロサービスアーキテクチャを使用した新しいアプリケーション開発

開発者はモノリスを分解する必要があるかもしれませんが、マイクロサービスアーキテクチャを使用して新しいアプリケーションを作成する必要があります。マイクロサービスアーキテクチャにより、複数のプレゼンテーションコンポーネントをビジネスロジックやデータストアと統合できると同時に、継続的インテグレーションおよび継続的デプロイメント（CICD）環境での高速な変更が可能になります。モノリスを分解する最初のステップは、マイクロサービスの構築に密接に関連する主要なソフトウェアアーキテクチャを順守することです。
DEVリーダーまたは上級開発者は、DEVチームと協力して、モノリスのエッジから引きはがして分離できるビジネス機能のコードベースを調査します。チームは、RESTfulマイクロサービスのエンドポイントとしてコードに落とし込めるビジネス機能、データニーズ、境界づけられたコンテキストを探しますが、統合された狭義のインターフェース、独立してデプロイ可能で、疎結合で、かつ言語にとらわれない状態を維持します。RESTエンドポイントと制御APIは、ビジネスロジックを実行するためのfacadeを提供します。マイクロサービスのコア原則には、次のものがあります：

1. 各サービスは独自のプロセスを実行し、サービスのデリバリは1コンテナ1サービスです。
2. 1サービス1ビジネス機能のみで、一つの完成されたサービスを最適化します。つまり、マイクロサービスには、変更する理由が1つだけあるはずです。
3. サービス間通信は、メッセージブローカー、分散疎結合データストア、およびRESTful APIを介して発生します。 ストアは疎結合を阻害するのでデータストアには注意を要します。
4. マイクロサービスは異なる速度で進化することが期待できます。システムは進化することができますが、ソフトウェアアーキテクチャの原則は、時間をかけて開発を導くものです。
5. マイクロサービスは、サービスごとの明確な高可用性とクラスタリングの決定に依存しています。すべてのサービスを拡張する必要はありません。自動スケーリングが必要なものもあり、共通のスケーリングポリシーがすべてのマイクロサービスプロファイルに適合する可能性は低いです。

モノリスの各部分は、ビジネスの観点からサービス機能を分割することにより、すべてスライスできます。内部の技術的な表現は、外部のビジネス行動を変えるべきではありません。開発者は、多くの相互依存性のない、明確なもしくは明確に定義されたインターフェースのコードベースを調べるか、特定の言語または特定の種類のデータストアを中心に編成されたコードベースの領域を探す必要があります。データストアを調べる場合、データがモノリスコードベースに密結合されているか、モノリスがデータ整合性とデータレイテンシの変化する状態において機能できるような疎結合であるかを判断する必要があります。直感に反して、開発者はボトルネックについてコードベースを分析しなければならない。それにより、リファクタリングが独立してデプロイ、スケーリング、および失敗できる限りリファクタリングから利益を得ることができます。最後に、将来の機能セットがモノリシックコードベースのアップグレードではなく、マイクロサービス拡張としてより適切に当てはまるかどうかについて評価を行う必要があります。モノリスの規模によっては、プログラムマネージャーと開発リーダーが、従来のモノリシックアプリケーションと新しいマイクロサービススタイルアプリケーションの共存を許可する必要があります。非常に多くのポイントのみがスプリントに適合し、複数のスプリントがエピックに適合します。残念ながら、範囲、コスト、時間、リスク、および資金の制約はすべて、新しいマイクロサービスアプリケーションへの強引なカットオーバーがリスクを大きくすることになります。並行して実装すると、ビジネスが中断する可能性が少なくなります。

マイクロサービスアーキテクチャを使用した新しいアプリケーション開発で直面した、もしくは認識した課題には以下の視点が含まれます：

開発者の視点：

1. マイクロサービスアーキテクチャでは、各マイクロサービスは自律的である必要があります。これは、緊密に結合されたモノリシックアプリケーションに慣れている開発者にとっては課題となります。各マイクロサービスは自律的である必要があります。つまり、そのプロセスドメインの障害または停止は、他のマイクロサービスの機能に影響を与えることはありません。開発者は、密結合されたモノリシックアプリケーション内での作業に慣れていると、未知の要素や、テストや機能の提供を他の関連するセキュリティチームや運用チームと調整する必要があるため、機能セットのリリースが遅れる可能性があります。遅延は、ストーリーのリファクタリングを含むアジャイルクルーの開発速度に影響します。多くの場合、これはアジャイルクルーがコードをプロモートすると同時に、インボックス/アウトボックスの直線的な処理を行うウォーターフォールインフラストラクチャおよびセキュリティプロセスのサポートと統合または実装する必要がある場合に発生します。このような予見可能な遅延を回避するための推奨されるプラクティスは、スプリント計画に「強固なスプリント 0」を追加することです。これにより、必要なサポートのウォーターフォールプロセスとサービスレベルアグリーメントをすべて事前に把握し、リスク評価、コードレビュー、API暗号鍵、コンテナ配送、テスト環境、サービスアカウント、および TLS 証明書の展開の予想されるサポートがスプリント計画プロセスの一部となるようにします。それ以外の場合、機能の実装は終了しますが、ウォーターフォールリクエストプロセスが完了するのを待つため、リリース時にブロックされます。
2. マイクロサービスを使用する場合、マイクロサービスのいくつかの要因に応じて、同期プロトコルまたは非同期プロトコルを使用できます。開発者にとって、これは特定のサービスがどのように動作するかを追跡する課題を提供する可能性があります。スケーリングと可用性の要件をサポートするために、マイクロサービスは複数のシステムにわたって複数実行されることがよくあります。開発者は、共存する複数のコピーを同時に実行できるような方法でサービスを作成するという課題に直面します。サービスの共存と並行性がないと、データ損失とデータ破壊のリスクが高まります。サービス利用の動作を追跡することは、プロダクションプロモーションとソフトウェアリリースに続く開発者の一般的な課題です。マイクロサービスコードをリリースする前に、チームは同期または非同期メッセージングのエンドツーエンドのサービスパフォーマンスプロファイルと、通信に使用されるプロトコル、およびコンテナーライフサイクルガバナンスがマイクロサービスにどのように影響するかを知っている必要があります。最も重要なこととして、組織は、エンドツーエンドのアプリケーションパフォーマンスモニタリングを評価する手段と、開発者が独自のダッシュボードを作成し、ログにアクセスして、アプリケーションの問題が表面化したときにデバッグできるようにする手段を必要とします。

オペレータの視点：

1. 好ましくは、マイクロサービスを本番環境に提供する前に、マイクロサービスを適切に検出するために必要なサイドカー機能を処理するために、堅牢なサービスメッシュが存在し、プロキシ、ロードバランシング、暗号化、認可とアクセス、サーキットブレーカーのサポートなどの他の必要な機能を提供します。サービスメッシュはコンピュータコンテナクラスタ内に統合され、マイクロサービスビジネスロジックから独立したサービス間通信を容易にします。そうでない場合、これらのIT機能は、アプリケーションランタイムの管理がはるかに困難で不透明な個別のアプライアンスにホストされることによって提供しなければならなくなります。

2. 成熟したネットワークポリシールーティング、鍵ライフサイクル管理、コンテナセキュリティ、およびイベントロギングのバックネットサービスメッシュ。開発者は、アプリケーションログにアクセスする必要があります。業界のプラクティスとしては、マイクロサービスをホストするコンテナまたはクラスタ全体が失われたり応答しなくなったりした場合に、開発者がマシンで生成されたデータを解析して不具合の根本原因を理解できるように、内部プライベートVLANに存在する隣接したロギングインフラストラクチャがプラットフォームとは別にログを取ります。これは開発者のニーズとして当てはめられますが、成熟したセキュリティ組織にとっては必須のプラクティスです。 壊滅的な障害が発生した場合、隣接するロギングインフラストラクチャは、旅客機のフライトデータレコーダーおよびブラックボックスと同じように機能します。

アーキテクトの視点:

1. ソフトウェアアーキテクチャの役割で作業を行うアーキテクトは、組織内で2つまたは3つの異なるレベルで作業する必要があります。ソフトウェアアーキテクトは、開発者リーダーとプログラムマネージャーに直接サポートを提供し、コードの近くにいる必要があります。また、ソフトウェアアーキテクトは、開発者のワークスペースに密接な関係はないが、マイクロサービスをデプロイする機能的なプラットフォームに不可欠な機能を含むソリューションの提供を調整するために、組織内のより高いレベルで機能する必要があります。マイクロサービスワークスペースのアーキテクトには「2つのギア」があります。1つは複数のアジャイル開発チームにサポートを提供し、それらのチームに先行して、将来の課題、プロトタイプ機能を整理し、UMLでコードとインフラストラクチャ間の統合をレンダリングします。もう1つのギアは、従来のビジネス技術の整合性の視点を使用して、上級管理職およびビジネス関係者とコミュニケーションを取ります。

2. マイクロサービスアーキテクチャが認証の手段を提供することを保証と、マイクロサービスへのアクセスの認可はさまざまなIAMソリューションとマイクロサービスのホスティングアーキテクチャとの互換性のため、困難な場合があります。マイクロサービスはそれぞれ、より頻繁にアクセスされ、より小さく、通常はより高速なサービスを提供するため、本質的に効率的な認証および認可機能が必要です。マイクロサービスの数が増えると、API呼び出し元を認証および認可するための時間が増加します。
3. 現在、コードに落とし込むことができるアーキテクチャのビューポイントをレンダリングしながら、技術的またはビジネスの整合性の視点に遡ることができる標準化された方法はほとんどありません。ほとんどの場合、結果として生じるアーティファクトの組み合わせは、UMLとビジネスの支持者のニーズと文化を満たすためにレンダリングされた独自の視点の組み合わせになります。シーケンス図、クラス図、およびコンポーネント図を伝えるために、アーキテクトが標準のUMLテンプレートの安定したセットを持つことは一般的です。そのような例の1つは、UMLを使用して、暗号とIDおよびアクセス管理（IAM）がどのように相互作用し、サービス認可を検証するかを理解することです。しかし、アーキテクトがコードをサポートするプラットフォーム間の技術的な相互作用を理解しようとすると、UMLはネットワーク中心の表現を優先してスケールアウトし始めます。ビジネス環境では、ビジュアル表現とナレッジドメインは、開発チームから遠く離れていても、基盤となるマイクロサービスの相互作用と完全に一致しています。アーキテクトを採用している教育機関は、アーキテクトがさまざまな操作モードを持ち、モードをいつ変更するかを理解し、各モードで成功することを期待しているため、マイクロサービスアーキテクチャスタイルに従事する2つのギアを持ったアーキテクトが必要です。

## 4.2 マイクロサービスの機能

マイクロサービスが持つ機能について、以下に概要を説明します。

### 4.2.1 マイクロサービスの完全性検証機能

APIがその完全性を確保するために必要なすべての制御を提供できないため、多くのアーキテクチャパターンが互いに重なって多層防御を提供します。APIの完全性は、IDをAPIのインターフェースに割り当てるために使用される暗号化機能だけに限りません。完全性はAPIを構成する他の基本要素にも適用され、受け渡されるデータが高い完全性、忠実性、正確性を備えていることを保証します。暗号化の他に、IDとアクセス制御、およびゲートウェイプロキシ、オフロード、集約、変換、ルーティングなどの従来のネットワークパターンの付属機能があります。基本的なセキュリティおよびネットワークポリシーの安全機能が存在しない場合、特定の課題が生じます。マイクロサービスの分散は、バックプレーンの制御の弱点を継承しています。なぜなら、防御的なソフトウェアプログラミングだけではこの弱点を補うことができないからです。

APIは、アプリケーションの下層にある実装を開示します。REST APIは標準で汎用性があります。REST APIは、様々な機能に使用できる予測可能なエントリポイントを提供します。APIを使用したアプリケーションの第1世代では、データのGET/フェッチは通常、1回の送信とそれに対応する要求/応答でした。APIベースのアプリケーションというコンテキストでの最新のREST API開発では、多くのフェッチが同時に発生し、対応する生データとパラメータがアプリケーションに返されます。クライアントはレンダリングエージェントであり、クライアントの動作の大部分とユーザ状態の維持を制御し、サーバはAPIに対応するプロキシ機能として機能します。ソフトウェア言語は、コードがサーバ側の設定と相互作用する特定のセキュリティモデルを提供します（Angular JS Frameworkセキュリティモデルは例です）が、一般的に今日のクライアントは独自のセキュリティモデル（ドキュメントオブジェクトモデル[DOM]およびコンテンツセキュリティポリシー[CSP]など）を用いてリクエストやデータレスポンスを処理します。

開発者は、高品質で、アンチパターンの使用がなく、欠陥のない安全なAPIコードを作成する責任があります。ただし、コードでは不可能なセキュリティおよびポリシー制御を提供しサポートしなければならないのは、エンジニアおよび運用スタッフです。開発者、オペレータ、およびアーキテクトは、協働して、マイクロサービスのコアソフトウェアアーキテクチャの非機能属性（自律性、遍在性、疎結合、再利用性、構成可能性、フォールトトレランス、発見可能性、およびビジネスプロセスの整合性）が損なわれないこと、 階層化されたセキュリティとネットワークポリシーの制御に制約されないことを確実にするようにしなければなりません。

マイクロサービスの完全性の検証において直面するまたは想定される課題には以下のものがあります：

<u>開発者の視点：</u>

1. 脆弱なコードはセキュリティの脆弱性につながります。一般に、開発プロセスの早い段階でコーディングの欠陥を解決する方が、マージまたはリリースした後よりも費用が抑えられます。プロセスの未熟さ、開発者の準備不足や認識不足が、安全でないソフトウェアの根本的な原因です。リリース後に脆弱性を開示することは、パッチ当てに最も費用と時間が掛かる段階でデータが公にさらされるリスクを生じます。準備ができていない開発者は、特有の課題にさらされます。何よりもまず、開発者は、安全なAPIベースのコードを構築するための統合開発環境（IDE）を整え

るために、ツール、防御的なプログラミングのスキル、予知知識、ジャストインタイムのトレーニングが必要です。IDEツールセットにおいてこの特定の課題を軽減するための主要な側面としては、次のような基本的な要素、プラグイン、および拡張機能があります。

a. Javascript、C、C++、C# （.NET）、Java、Kotlin（Android）、Swift（IOS）、R、Python、SQL、noSQLなどの一般的な言語での防衛ソフトウェアプログラミングのプラクティスに関する専門のトレーニング。
b. セキュアコーディングにおける10大トピックに関する専門のトレーニング。例えばエンコード、機密トランザクションの管理、スレッドとメモリリソースの管理、カプセル化、信頼できない応答のエスケープ、例外の処理、入力の検証、シリアル化およびシリアル化解除のタイミング、セッション状態、資格情報の使用およびストレージの管理、パラメータクエリの処理、保存中および転送中のデータの保護、暗号化。
c. Linter – コードをハイライトするソフトウェアのアタッチメントで、フォーマットエラーや、一般に受け入れられているコーディング標準や規約への違反を示してくれます。Linterは開発者がコードを作成するときに論理エラーを指摘してくれます。
d. Doc Generator – Doc Generatorは、コードが具体化されるとコーディングドキュメントを作成します。Doc Generatorは、バックグラウンドのIDEタスクとして動作し、開発者の手戻り作業が不要になります。コーディングドキュメントは、コードを保守する必要がある次の開発者にとって非常に貴重なアウトプットになります。
e. Open APIの設計/ドキュメンテーションのプラグインまたは拡張機能 – 開発者は、Open API（OAS）準拠の仕様を使用する必要があります。サービスが他の開発者によって広められ使用される場合、API仕様は絶対に必要です。たとえば、Swagger Hubは、API設計ツールとドキュメントツールセットの両方であり、開発者はAPI仕様の作成を、その構築やテストと並行して実施できます。代替手段にはYAMLおよびRAMLがあり、REST APIの記述、生成、可視化、および使用に利用できます。
f. 静的コードテストのプラグインまたは拡張機能 – シフトレフトの動きの一部として、開発者は、静的コード分析を実行できる高品質のプラグインまたは拡張機能にアクセスする必要があります。この分析は、リモートコードリポジトリに最終バージョンをコミットしてDEVリードによるコードテストのためにプルリクエストを発行する前に、または分岐のマージのためにCIパイプラインに入れる前に実施します。多くのオープンソースおよび商用製品が存在します。
g. リポジトリの見取り図またはビュー – 最新のIDEは、ソースコードリポジトリとの直接統合、およびSeleniumやJenkinsなどのジョブ自動化フレームワークとの統合を提供します。
h. 資格情報と暗号鍵の管理 – 開発者は、プロモートしたコードに固定のもしくはクリアテキストの資格情報を絶対に埋め込んではなりません。開発者はSSHとTLSをIDEに組み入れることで、リモートのコードリポジトリとの間で機密性を確保できます。より堅牢な方法であるキーリング、キーチェーン、暗号化アーカイブ、およびハードウェアセキュリティモジュールを組み入れるには、開発者がコントロールできないプラットフォームが関係してきます。各IDEには鍵を管理する独自の方法があり、それはさまざまなコマンドライン操作方法でも同様です。ただし、IDEごとに操作方法が構文的に異なっていても、適用される原則は同じです。
i. 単体テストとエンドツーエンドのテストフレームワーク– ソフトウェア言語によって、さまざまなテストフレームワークがあります。開発者は、テストと再テストの間の信頼性と再現性が良好で、使用する言語に合ったものを使用する必要があります。シフトレフトの開発者は、自らのユニットテストをビルド、実行、およびレポートする必要があります。マイクロサービスの場合、総合的な顧客体験は多くの分散した半独立の部分の組み合わせであるため、高レベルのテストにはエンドツーエンドのテストが

必要になります。一般的なルールとして、開発者は90～95%のユニットテストケースをカバーする必要があります。率が低いと信頼を得られず、率が高いとコストがかかります。カバレッジは、コードの品質ではなく、テスト済みのカバレッジを意味することに注意してください。以前に報告された他の手法は、コード品質をカバーしています。コード品質とテスト範囲は別々の指標です。

j. Service-to-Container Bootstrapフレームワーク – このフレームワークは、できれば、マイクロサービスとそのすべての必要なサポートインフラストラクチャを完全に自動化（本質的に自己起動）できるようなプラグイン、プラグイン可能なモジュール、インクルードライブラリ、または拡張機能として起動します。ブートストラップは重要で複雑な開発上のテーマですが、本書では取り上げません。

2. 認証とアクセス制御： 認証とアクセス制御のポリシーは、マイクロサービスが提示するAPIの種類によって異なる場合があります。あるものはパブリックAPIであり、あるものはプライベートAPIかもしれません。あるものはパートナーAPIで、ビジネスパートナーのみが利用できます。開発者は、LDAP、Kerberos、SAML、OAuth、TLS証明書管理といったプラットフォームのオーナーと協力して、APIがIDを証明し、割り当てられた認証認可の境界内で機能し、サービス通信メッシュ内の他のサービスのIDを検証できるようにする必要があります。このコントロールプレーンの機能がないと、データ漏えいのリスクが認識できなくなり、動作環境が信頼できなくなります。
3. エンドツーエンドトランザクションの機密性の確保：TLSを使用する場合、CPUを集中的に使用する処理は、ハンドシェイクフェーズで最も多く発生します。TLSハンドシェイクはHTTP接続の作成中にのみ発生するため、この負荷を削減する最も簡単な方法は、HTTP常時接続を有効にすることです。このメカニズムを使用すると、ハンドシェイクの負荷が複数の要求に分散されます。または、常時接続の最大リクエスト数またはタイムアウトに達した後でも、TLSセッションIDを使用することで、負荷のかかる新たなハンドシェイクを行わずに済みます。
4. 開発者は、利用可能な最も強力な暗号機能を使用する必要があります。一般的なプロファイルとしては、2048ビットの鍵長（規制のある業種向けに開発していない場合は1024ビット）、SHA512、メッセージ認証コードとしてHMAC-MD5、AES-256またはTDEA（3DES）、および低い値への再ネゴシエーションを防ぐためのハードストップを備えたTLS 1.3です。 開発者には3つの課題があります。暗号攻撃は常に時間とともに強力になりますが、めったに弱体化することはありません。TLSの使用以外の暗号化のより詳細な部分は知識が不足している可能性があります。成熟したセキュリティ組織では一般に、セキュリティエンジニアリングチームが暗号化ポリシー、ライフサイクル、標準を管理しており、開発者の手から暗号の使用を完全に取り除きます。暗号セキュリティチームが暗号管理の使用方法にどのようにアプローチするかを完全に理解するには、NISTのSP800-130「暗号化鍵管理の設計のためのフレームワーク」のセクション6「暗号化キーとメタデータ」を参照してください。

オペレータの視点:

1. マイクロサービスの完全性に関してオペレータができることは、再現性と信頼性の高いコンテナーイメージリポジトリとコンテナライフサイクル（作成、保存、使用、共有、アーカイブ、破棄）の提供です。ライフサイクルは開発者が使用するマイクロサービスブートストラップフレームワークの両方に適切に統合され、開発チームが使用するマイクロサービスのバージョン管理プロセスを一貫して補完し、CIツールチェーン（通常、それはJenkinsで、調整が必要な多数のブートストラップジョブを引き受けている）に強固に結び付けられています。開発者環境にこのレベルの緊密さがないと、2つの厄介な問題が生じます。1つは、ブートストラッププロセスでのコンテナの配信がイメージコンテンツの点で非常に変化しやすく、ランタイムバージョンまたはパッチ適用のバッティングにつながる可能性があること。また、オペレーティングシステム内で実行する実行環境の全体的な管理が完全に変更不可(immutable)ではなく、何らかの理由で常駐せざるを得ないコンテナを生成します。完全に変更不可(immutable)ではないコンテナ環境で特に課題になるのは、実行中のワークロードのbreak-fix（止めて直す）ためにワークロードへのリモートからの対話型ログインを許可する必要があることです。これにより、すべてのマイクロサービスコンテナのセキュリティプロファイルが低下し、コンテナイメージレベルのセキュリティプロファイルの変更になります。
2. オペレータは、開発者のブートストラップフレームワークにおけるプロビジョニング手順に結びつく、明確なコンテナ配信パイプラインを必要とします。複数のチームがイメージレイヤーを提供し、それらがならされてゴールデンイメージになることがあります。ソースコードとサーバの設定は、組み合わされてコンテナイメージを構成し、バージョン管理されたリポジトリにアーカイブされます。指定されたセキュリティチームがイメージレイヤーで作業し、候補となるマスターイメージのセキュリティユニットテストを実行しないと、コンテナの証明認定プロセスはエンジニアリングチームと運用チームに委ねられ、コンテナ配備のパイプラインにおける職務分離の管理が破綻します。オペレータとエンジニアには、パッチの適用、新しい攻撃ベクトル、従来型のガバナンス、リスク、コンプライアンス管理など、管理する多くのタスクがあります。実行する役割が多すぎると、マイクロサービスをサポートするプラットフォーム全体の完全性に影響を与える可能性があります。

アーキテクトの視点:

1 アーキテクチャの3つの異なる役割が関係しています。エンタープライズアーキテクトは、事業目的に関して技術適用の方向性を示しますが、プラットフォームまたはソリューションに関しては何も示しません。ソリューションアーキテクトは、運用、セキュリティ、ネットワーク、アプリケーション開発と直接連携して、コンテナ、マイクロサービス、および選択したブートストラップフレームワークをサポートできるプラットフォームを設計します。ソフトウェアアーキテクトは最もコードに近く、セキュアなコーディングとスクリプティングプラクティスの情報セキュリティの専門家としばしば協働します。これには開発者をリードすることも含まれます。これらすべては、安全なマイクロサービスを作成することが目標です。特に問題なのは、多くの組織がここまで揃えるほどの力を持たず、DevSecOpsチームまたはDevSecOps担当者にすべてのアーキテクチャ機能を任せてしまうことです。

2 一人で複数のスキルセットを身につけていることはまれで、それは一般的にアーキテクトの仕事には特有の課題です。アーキテクトの仕事は、高度に専門化されるか、さもなければ一般的な呼び方に帰結してしまいます。こうした弱点の対策として、組織は、外部のコンサルティングを雇うか、安全なコーディング、防御的なソフトウェアプログラミング、環境設定などの特定のニーズに合わせて調整された既製品の教育パッケージを採用する必要があります。このサポートなしでは、組織は、忠実性と完全性の低いマイクロサービスアーキテクチャを組織として用いるリスクに直面します。そのようなマイクロサービスは、コード自体の問題によってではなく、コントロールプレーンインフラストラクチャが貧弱なためにスケールできません。

# 4.3 モノリシックアプリケーションを分解する際のベストプラクティス

Mark Zuckerbergによる Facebookの指針「素早く動け、破壊せよ」によると、アプリケーションの分割はやるべきではありません。インフラストラクチャとコードに対する悪いアドバイスです。「安定したインフラストラクチャで素早く動け」の方が適しています。

システムを一連のサービスにどのように分割するかを決定するのは非常に難しいことですが、役立つ戦略がいくつかあります。1つのアプローチは、動詞またはユースケースによってサービスを分割することです。別の分割のアプローチは、名詞またはリソースによってシステムを分割することです。このアプローチは、特定のタイプのエンティティ/リソースで動作するすべての操作を担当するサービスを見つけ出します。第三の方法は、入力および出力パスを追うことで、データの作成、編集、更新、削除が発生する場所を発見することです。データソースから逆方向に作業することで、データを使用するサービスが明らかになります。理想的には、マイクロサービスの設計では、単一責任原則（Single Responsibility Principle SRP）を使用して、各サービスに小さな責任セットのみを割り当てる必要があります。 SRPは、クラスの責任として変更の理由を定義し、クラスには変更の理由が1つだけあるべきであるとしています。アプリケーションの分解では、開発者は既存のアプリケーションをマイクロサービスのコンポーネントに再分解する必要があります。SRPは、マイクロサービスの主要な属性を再定義するもう1つの方法です。SRPでは、構築されたすべてのサービスは、1つのビジネスルール、ロジックセット、またはプロセスの実行のみを提供します。Web Scalabilityの完全な解説は、Abbott and Fischerの「The Art of Scalability：The Scalable Web Architecture、Processs、and Organizations for the Modern Enterprise」第2版、Addison-Wesley Professional、2015に載っています。

開発者の視点：

マイクロサービスはAPIです。効果的で安全なAPI設計により、クライアントが人であろうと他のマシンであろうと、再現性と信頼性の高いクライアントの扱いが実現します。APIは、任意の数の繰り返しタスクに使用できる機械可読コンテンツを生成します。マイクロサービスはステートレスです。モノリスのアプリケーション境界内のステートフルインターフェイスであったものが、ステートレスRESTful APIになります。2014年、James Lewis と Martin Fowlerは、マイクロサービスの概念的解釈を「サービス指向、個別に配備可能、個別に管理可能、一時的、弾力的」と説明しました。それ以来、マイクロサービスはAPI通信およびコンテナ技術と切り離せなくなりました。マイクロサービスをセキュアにすることは、API、コードそのもの、コンテナプラットフォームの3つの面で行われます。3つのレイヤはすべて連携して、マイクロサービスの完全性を維持します。

開発者は以下のAPI設計上のベストプラクティスに従うべきです：

1. 基礎となるデータモデルを理解し、ステートの情報なしでデータを処理するための新しい機能を提供します。REST APIは結果としての一貫性しか提供できません。これは、データが複数のネットワークセグメントとセキュリティゾーンを通過する場合のプログラミングの課題です。
2. テンプレート、スキャフォールディング、およびスケルトンフレームワークからREST APIを構築して、生産性を向上させます。これにより、APIの構築が標準化されるだけでなく、「ツリー」を作成する機能を気にすることなく、API仕様に残りのソフトウェア機能を手動で追加する「ツリーアンドリーフ」アプローチが提供されます。

3. コードジェネレータを使用して、構築中のサービスのクライアントスタブを作成します。クライアントスタブはサービス仮想化の形式で、開発者が、合成データと適切なテストカバレッジに必要な最小限のサービスで、APIを単体テストできるようにします。クライアントスタブサービスは、将来のテストで使用するためにアーカイブしたりライブラリに取り入れたりできます。
4. 構築されたすべてのAPIについて、Swagger、YAML、またはRAMLを使用したAPI仕様ドキュメントを提供する必要があります。Swaggerは主要な仕様ですが、他のものもあります。このようなツールは、定義された構文、明快さ、構造をもたらします。
5. APIはビジネスロジックへのインターフェースであり、アクセスされるビジネス機能は1つのビジネス機能のみを取り扱いますが、APIは複数のストリームとデータタイプを取り扱いできます。APIがデータを収集、構造化、フォーマット、配信、および保護し、防御することを理解してください。
6. オープンスな標準と業界の規約を使用して、クライアントがAPIを使用できるようにします。
7. APIは、他のAPIと相互運用できるように構築する必要があります。処理と変換の負荷が生じるので、ラッピング手法と非標準プロトコルを避けるようにしてください。ラッピングは避けられない場合があります。サービスをラップする必要がある場合は、負荷がかかる場合に特定のパフォーマンステストとメモリチューニングをサポートすることも必要になります。
8. APIは自明で予測可能な形で構築する必要があります。API仕様とAPI定義は、これらの非機能要件の実現に大きく役立ちます。
9. APIの利用者（他のマシンおよび他の開発者）は、APIの機能を調べる能力と、発見できる能力を持つ必要があります。ただし、APIが一般に公開されている場合は、"need to know"原則を適用して、APIが意図しない情報開示を引き起こすことを防ぎます。
10. APIは、問題の解決を図るのに必要最低限のエラーメッセージを適切な関係者に配信するようにする必要があります。
11. APIは、再利用可能、バージョン管理可能、および以前のバージョンとの下位互換性が必要です。
12. すべてのAPI要求は、資格情報、アサーション（SAMLまたはOauthトークン）、暗黙的な信頼関係（Webトークンまたはキー）、これらの機能の組み合わせのいずれかを介して認証される必要があります。APIには独自の認証メカニズムはありません。なぜならば、まず、これはオープン標準ではありません。次に、信頼できるコンポーネントまたは仲介者（LDAP、PKI、HSM、キーリング、またはキーストア）に認証を委任しないということは、APIのアンチパターンです。
13. APIは、認可を別のサービスまたは仲介者に委任します。インターフェースとデータアクセスの認可は、別のプラットフォームで管理される粒度が粗いおよび粒度が細かいアクセス資格付与機能に任せたほうがよいでしょう。
14. インフラストラクチャに関しては、セキュリティゾーンとネットワークセグメンテーションのプレーンから受け継いだり派生したりするセキュリティコントロールでAPIを肥大化させないように、アーキテクチャ全体に絶対的な信頼部分を作成します。信頼は透過的で既知である必要があります。開発者は、セキュリティ管理策がどこから来たのかわからない場合、彼らは尋ねるべきです。誰も答えられない場合は、管理策の出所が明らかになるまで再度質問する必要があります。
15. 信頼関係を形成するためにLOCALHOSTに依存しないでください。テスト用には機能しますが、実稼働用には機能しません。リソースを見つけ、さらに検証するためのビルディングブロックを提供する手段として、HTTPのURI名前空間とDNS解決を用いてください。
16. アプリケーション間のセキュリティトークンのやり取りは、Oathなどのどこにも属さないオープン標準である必要があり、マイクロサービスのアーキテクチャは、トークンを配布し、トークンのライフサイクルを管理するために信頼できる第三者を必要とします。

17. 状況によっては、認証されていない利用者（Webページへの訪問者でAPIにアクセスする者など）としての最初のクライアント要求によって、トークン認証を提供する機能的必要が生じる場合があります。Oathの暗黙的な許可を使用するよりも、Oauth2とPKCE（Proof of Key for Code Exchange）を使用することをお勧めします。
18. SAML2のみが利用可能なフェデレーションIDプロバイダである場合、ペイロードにbase64 uuencodeを決して使用せず、常にAES-256でコンテンツを暗号化します。Base64は簡単にクリアテキストに変換されます。規制対象の業界は、電子メールが一般的に個人識別情報であり、機密性を付加することが必要であることを認めています。開発者はこの機能を制御できない場合がありますが、この状況では、フェデレーションIDプラットフォームにAES-256ペイロード暗号化を要求できます。
19. JSONベースのトークンと標準プロトコルの使用が、現状のプラクティスとされていることです。JSONベースのトークンは短命であり、制限された機能のみを扱います。信頼できない環境や、クライアントが信頼できない場合の脅威の可能性を限定するのに理想的です。
20. APIキーを使用する特定の実装があります。狙いを定めてくるハッカーは、そのキーを見つけ出して（メモリ内のみで保持されていてストレージには保存されていない場合でも）、キーを復元してAPIに対してのみ信頼できるクライアントになることができます。APIキーは信頼できるものではないため、既知の弱点をカバーするために、IAMコントロール設定のレイヤ内で使用する必要があります。APIキーはアプリケーションを識別するもので、アプリケーションのユーザは識別しません。
21. マイクロサービスの目標が、信頼できるクレームにJWT（JSON-webトークン）を使用することである場合、JWTをABAC（attribute based access control）とともに使用して、クレームをIDベースのクレーム（つまり、「私はすべてにアクセスできます」）からアクセス要求が属性セットのみ（つまり、「私はこのリソースにアクセスできます」）である属性ベースのクレームにシフトします。
22. APIの通信の機密性を保持するためにTLS1.3を使用します。
23. TLSネゴシエーションの最も要求の厳しい部分は、HTTP要求中のハンドシェイクです。一般的なルールとして、すべてのTLS終端をAPIゲートウェイやロードバランスアプライアンスなどの中継装置にオフロードします。そうすることで、パフォーマンス面の要求をオフロードするだけでなく、このクラスの専用アプライアンスが提供するDoS対策機能も利用できます。
24. 利用可能な中で最強の暗号化手法を使用します。ただし機能とパフォーマンスを犠牲にしないように。
25. コード署名と証明書の管理のための信頼できるソースを使用して、各マイクロサービスの完全性を強化します。開発者は、ソースコードと、場合によっては使用されるコンテナイメージを検証することにより、マイクロサービス環境全体の信頼性を確認します。その結果、マイクロサービスの信頼できるソースの検証が可能になります。
26. 再利用可能なコードを特定し、共通ライブラリが普及し開発コミュニティが利用できることを確認します。共通ライブラリには、メンテナンスプロセス、所有権、定期的なコードリファクタリング、確かなテストフレームワークの付属物としての抽象化が必要です。
27. 開発者は、シフトレフト、単体コードテスト、テスト駆動型セキュリティ（TDS）を追求する必要があります。TDSは、開発者が望ましい動作を表現するテストを作成してから、テストを実装するコードを作成することを推奨します。TDSは最初のインスタンスで失敗することが予想されますが、このアプローチはコードの重大なセキュリティギャップを識別し、次の実行で対処できるようにします。

28. すべての開発者は、お互いのコードのレビューに参加する必要があります。できれば、機能の確認に際して小グループのピアレビューを行ってください。
29. 開発チームは、テスト方法と結果を公開し、そのチームのサービスを使用する他の開発者がテスト方法の理解と信頼を構築し、そのチームのアプリケーションがマイクロサービスをどのように使用するかに影響を与えることができるようにする必要があります。

<u>オペレータの視点:</u>

現代の組織の状況では、開発（Dev）、セキュリティ（Sec）、および運用（Ops）は、これらすべてのハイブリッドであるDevSecOpsとなる必要があります。DevSecOpsは一人の人でもチームでも構いません。DevSecOpsは、サーバの仮想化がインフラストラクチャサービス管理を変更したときにストレージ、サーバ、ネットワーキングのチームが統合したのと同様に、効率の向上を目指しています。DevSecOpsの成立は、3つの別個のドメインをマージして、目指すところを一つにして効率を高めました。DevSecOpsは、基礎となるinfrastructure as code、その上のマイクロサービスコード、ブートストラップフレームワークを継承して、CI / CDツールチェーンに収容し、さらにQAとテストの要件や従来からのインフラストラクチャに関するタスクを取り込みます。Devの「オペレータ」は、新たなプラクティスを継続的に取り入れて、マイクロサービスとコンテナ化の進化をサポートします。

マイクロサービスをサポートするDevSecOpsエンジニアのための一般的ベストプラクティス：

1. 組織が従来のアプリケーションとサービスからマイクロサービスに移行するにつれて、組織が使用するCI/CDサービスは拡張される必要があります。これは、マイクロサービスではお互いの互換性を確保するために、より多くの単体テストが必要になるため、特に重要です。
2. 安全な通信チャネルを活用して通信する必要があります。モノリシックアプリケーション内で転送されていた機密データは、今やネットワークを行き交っています。 アプリケーション自体がトラフィックを暗号化/復号化できない場合、オペレータは安全なネットワーク通信を提供する必要があります。これは例えば、プロバイダのロードバランサーでSSL終端を提供することで実行できます。
3. ネットワークセグメンテーションを活用して、安全な通信エリアを形成する必要があります。露出を最小限に抑えるには、"need to know"モデルでマイクロサービスアーキテクチャを実装する必要があります。外部者は、公開されたサービスのみにアクセスでき、マイクロサービスとそれにぶら下がるものの間の通信にはアクセスできません。APIゲートウェイパターンを利用して、クライアントが単一エントリポイントにリクエストを送信するようにすると、セキュリティ対策を実装することにより、下層にあるマイクロサービスを保護できます。
4. マイクロサービスとコンテナは短命になる可能性があります。 オペレータとエンジニアは、アプリケーションの状態をいかにして確認するかを考慮する必要があります。その結果、より少ない障害点でより優れたプラットフォームが実現します。マイクロサービスを拡張し、障害が発生した場合にマイクロサービスを再起動できる信頼性の高いオーケストレーションを提供することにより、オペレータは、開発者が自分で開発し使用するマイクロサービスを実行するプラットフォームに対して信頼できるようにすることができます。
5. セキュリティ組織と協力して、オペレータは信頼できる署名と証明書の発行元を確保する必要があります。信頼できる署名と証明書の発行元を提供することで、通信をセキュアにしてマイクロサービス環境内の信頼を確保し、検証可能なコードとコンテナイメージのベースを提供できます。オペレータは、マイクロサービスベースの環境内で信頼性を確保するために、信頼できる署名の

発行元を用意する必要があります。 以前はアプリケーションパッケージとコード内にあったトラストが、ネットワーク上のマイクロサービスとコンテナの間のトラストに変わっています。

アーキテクトの視点：

1. アーキテクトは、バランスの取れたアーキテクチャに注力すべきです。マイクロサービス環境では、アーキテクチャ全体を理解して、再利用可能な要素による一貫した開発を保証する必要があります。マイクロサービスの各要素に分解された機能をベースにした強力なアプリケーションアーキテクチャ設計を打ち建てると、設計段階からセキュリティが組み込まれた疎結合の独立したコンポーネントの開発が可能になります。マイクロサービスが1つの機能を適切に実行することを確実にすることと、労力と機能の重複をなくすという要求との間には、常にバランスがあります。セキュリティオーケストレーターとして、セキュリティサービスをサイドカーサービスまたは再利用可能な独立したサービスにオフロードするのは価値があります。
2. アーキテクトは、規制要件を忠実に守って、設計段階でプライバシー要件を取り入れる必要があります。新しいプライバシー規制の広がりと従来型ネットワークの境界が侵害されている状態では、プライバシーバイデザインは今日のアプリケーションにとって不可欠な要件です。
3. 他のマイクロサービスがデータに直接アクセスすることはできません。各マイクロサービスには個別のデータベースIDが必要です。これにより、個別に分離したアクセスを与えてバリアを形成し、他のサービステーブルを使用できないようにすることができます。
4. サービスごとの共有データベースはお勧めできません。アプリケーションを分解する場合、共有データベースは、アプリケーションをより小さな論理部分に分割するのに適当な場所です。ただしこの戦術は、新しいマイクロサービスアプリケーションには適用すべきではありません。1つのデータベースを複数のマイクロサービスと連携させる場合がありますが、過渡的なソフトウェアアーキテクチャとして最大2～3に制限する必要があります。そうしないと、拡張性、自律性、独立性の実現が難しくなります。各マイクロサービスによる"need-to-know"アクセスを実現するには、きめ細かいアクセス制御を導入する必要があります。目標とするアーキテクチャの状態は、各マイクロサービスが独自のデータを制御することです。
5. 共有データベースからサービスごとのデータベースに移行する場合、データのルーティング、メッセージング、変換、キューイング中のデータのセキュリティとプライバシーを考慮することが不可欠です。これらのアクティビティは、異なるクラス分けのデータを混在させ、データ漏洩につながる可能性があります。アーキテクトは、オーケストレーションサービスを使用して、さまざまな分類のさまざまなデータを処理する必要があります。
   アーキテクトは、セキュリティとプライバシーを念頭に置いたデータモデルを開発し、秘密指定データと非秘密指定データの分離にフォーカスする必要があります。可能であれば、機密データの使用を減らすための仮名化、匿名化、およびトークン化の使用を検討する必要があります。必要に応じて難読化を使用するか、プレゼンテーション層におけるアクセスの分離を行います。
6. 機密データの使用と保存は制限する必要があります。アーキテクトは評価を実施して、機密データ要素が不必要に転送または保存されていないことを確認する必要があります。可能であれば、トークン化を使用してデータ漏洩リスクを軽減するべきです。

# Appendix A: Acronyms

Selected acronyms and abbreviations used in this paper are defined below.

| | |
|---|---|
| ACM | Application Container and Microservices |
| API | Application Interface |
| CI - CD | Continuous Integration - Continuous Delivery |
| CMP | Container Management Platform |
| CSP | Cloud Service Provider |
| ESB | Enterprise Service Bus |
| HSM | Hardware Secure Module |
| IAM/ IdAM | Identity and Access Management |
| IPC | Interprocess Communication |
| KMS | Key Management System |
| NFR | Non-functional Requirements |
| OS | Operating System |
| QSA | Quality Service Agreement |
| RBAC | Role-Based Access Control |
| SDLC | Software Development Lifecycle |
| SLA | Service-Level Agreement |
| SOA | Service Oriented Architecture |
| SOTA | State Of The Art |
| VM | Virtual Machines |

# Appendix B: Glossary

| | |
|---|---|
| Application container [NIST SP800-180] | An Application Container is a construct designed to package and run an application or its components running on a shared Operating System. Application Containers are isolated from other Application Containers and share the resources of the underlying Operating System, allowing for efficient restart, scale-up or scale-out of applications across clouds. Application Containers typically contain Microservices. |
| Container management platform | A Container Management Platform is an application designed to manage containers and their various operations including, but not limited to, deployment, configuration, scheduling and destruction. |
| Container lifecycle events | The main events in the life cycle of a container are Create container, Run docker container, Pause container, Unpause container, Start container, Stop container, Restart container, Kill container, Destroy container. |
| Developer [NIST SP 800-64 Rev 2/ ISO/IEC 17789:2014] | The cloud service Developer is a sub-role of cloud service partner which is responsible for designing, developing, testing and maintaining the implementation of a cloud service. This can involve composing the service implementation from existing service implementations.<br><br>The cloud service Developer's cloud computing activities include:<br><br>• design, create and maintain service components (clause 8.4.2.1);<br>• compose services (clause 8.4.2.2);<br>• test services (clause 8.4.2.3).<br><br>NOTE 1 – Cloud service integrator and cloud service component Developer describe sub-roles of cloud service Developer, where the cloud service integrator deals with the composition of a service from other services, and where cloud service component Developer deals with the design, creation, testing and maintenance of individual service components.<br><br>NOTE 2 – This includes service implementations and service components that involve interactions with peer cloud service providers. |
| Host | OS supporting the container environment |
| Lateral movement [LIGHTCYBER 2016] | Lateral action from a compromised internal host to strengthen the attacker foothold inside the organizational network, to control additional machines, and eventually control strategic assets. |

| | |
|---|---|
| Microservices [NIST SP800-180] | A microservice is a basic element that results from the architectural decomposition of an application's components into loosely coupled patterns consisting of self-contained services that communicate with each other using a standard communications protocol and a set of well-defined APIs, independent of any vendor, product or technology. Microservices are built around capabilities as opposed to services, builds on SOA and is implemented using Agile techniques. Microservices are typically deployed inside Application Containers. |
| Microservices Architecture | A microservices architecture usually refers to an application that has been structured to use basic elements called Microservices, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies. |
| Microservices Systems Software Development | This is the process of breaking down an application into components (microservices) basic elements to create through code extraction or rewrite capability (greenfield) microservice architecture of self-contained services that achieve a business objective. |
| Enterprise Operator | The individual or organization responsible for the set of processes to deploy and manage IT services. They ensure the smooth functioning of the infrastructure and operational environments that support application deployment to internal and external customers, including the network infrastructure, server and device management, computer operations, IT infrastructure library (ITIL) management, and help desk services.[14] |
| Enterprise Architect | The individual or organization responsible for strategic design recommendations. They determine, by applying their knowledge of cloud, container and microservices components to the problems of the business, the best architecture to meet the strategic needs of the business. Additionally, they develop and maintain solution roadmaps and oversee their adoption working with developers and operators to ensure an efficient and effective solution implementation. |
| Container resources | Three resources required for containers to operate are CPU, Memory (+swap) and Disk (space + speed) and Network. |

[14] https://en.wikipedia.org/wiki/Information_technology_operations

| | |
|---|---|
| Container resource requests | The amount of CPU, memory (+swap) and Disk (space + speed) that the system will allocate to the container its share considering the Resource Limit. |
| Container resource limit | The maximum amount of resources (CPU, memory (+swap) and Disk (space + speed)) that the system will allow a container to use. |
| Service Registry | The registry contains the locations of available instances of services. Service instances are registered with the service registry on startup and deregistered on shutdown. Client of the service and/or routers query the service registry to find the available instances of a service. |
| Client-Side Discovery | The client requests the network locations of available services from the service registry. |
| Server-Side Discovery | The Server requests the load balancer for the network locations of available services from the service registry. |

# Appendix C: References


| | |
|---|---|
| [NIST SP 800-180] | NIST Special Publication (SP) 800-180 (Draft), *NIST Definition of Microservices, Application Containers and System Virtual Machines*, National Institute of Standards and Technology, Gaithersburg, Maryland, February 2016, 12pp. http://csrc.nist.gov/publications/drafts/800-180/sp800-180_draft.pdf |
| [NIST SP 800-160] | NIST Special Publication (SP) 800-160, *Systems Security Engineering: Considerations for a Multidisciplinary Approach in the Engineering of Trustworthy Secure Systems*, National Institute of Standards and Technology, Gaithersburg, Maryland, November 2016, 257pp. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-160.pdf |
| [NIST SP 800-123] | NIST Special Publication (SP) 800-123, *Guide to General Server Security*, National Institute of Standards and Technology, Gaithersburg, Maryland, July 2008, 53pp. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-123.pdf nistspecialpublication800-123.pdf |
| [NIST SP 800-64 Rev 2] | NIST Special Publication (SP) 800-64 Rev 2, *Security Considerations in the System Development Life Cycle*, National Institute of Standards and Technology, Gaithersburg, Maryland, October 2008, 67pp. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-64r2.pdf |
| [NIST SP 800-190] | NIST Special Publication (SP) 800-190, *Application Container Security Guide*, National Institute of Standards and Technology, Gaithersburg, Maryland, September 2017, 63pp. https://doi.org/10.6028/NIST.SP.800-190 |
| [NIST SP 800-204] | NIST Special Publication (SP) 800-204 *Security Strategies for Microservices-based Application Systems*, National Institute of Standards and Technology, by Ramaswamy Chandramouli Computer Security Division Information Technology Laboratory, Gaithersburg, Maryland, August 2019, 33pp. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204.pdf [accessed 8/26/2019] |
| [Apriorit, 2018] | Apriorit. (2018). Microservice and Container Security: 10 Best Practices, by Bryk, A. https://www.apriorit.com/dev-blog/558-microservice-container-security-best-practiceshttps://www.apriorit.com/dev-blog/558-microservice-container-security-best-practices [accessed 9/10/2019] |

| | |
|---|---|
| [Addison-Wesley Professional, 2015] | Abbott, M., Fisher, M. (2015). The Art of Scalability: The Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise. 2nd Edition. Addison-Wesley Professional. |
| [API University Press, 2016] | Biehl, M. (2016). API-University Series volume 3: RESTful API Design First Edition. API University Press. |
| [Aquablog, 2017] | Aqua. (2017) Managing Secrets in Docker Containers. The Challenges of Docker Secrets Management.Jerbi, A. Aquablog https://blog.aquasec.com/managing-secrets-in-docker-containers [accessed 9/4/2019]<br><br>Microservices Governance : by Kasun Indrasiri, Prabath Siriwardena https://doi.org/10.1007/978-1-4842-3858-5<br><br>https://dzone.com/articles/microservices-logging-best-practices |
| [Beydoun et al., (2013)] | Beydoun, G., Xu, D., Sugumaran, V. (2013). Service Oriented Architectures (SOA) Adoption Challenges Service Oriented Architecture (SOA) Adoption Challenges. International Journal of Intelligent Technologies. 55. |
| [BMC Software, 2017] | BMC Blogs - *Microservices vs SOA: How Are They Different?* by Watts, S., 2017. https://blogs.bmc.com/microservices-vs-soa-whats-difference/?print=pdf [accessed 9/5/2019] |
| [Cerny et al., (2017)] | Cerny T., Donahoo J. M., Pechanec J. (2017). Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems. RACS '17 Proceedings of the International Conference on Research in Adaptive and Convergent Systems Pages 228-235 . |
| [Erl (2005, p. 54, p.263)] | Erl, T. (2005). Service-Oriented Architecture: Concepts, Technology, and Design: Prentice Hall |
| [Haddad, CIOReview] | CIOReview - *Moving from SOA to Microservices*. Haddad C. https://service-oriented-architecture.cioreview.com/cxoinsight/moving-from-soa-to-microservices-nid-6865-cid-95.html [accessed 4/9/2019] |
| [HashiCorp Vault] | Hashicorp Vault. Learn about secrets management and data protection with HashiCorp Vault. Operations and Development Tracks. https://www.vaultproject.io/guides/secret-mgmt/index.html [accessed 9/13/2019] |

| | |
|---|---|
| [IETF, 2018] | Internet Engineering Task Force (IETF). (2018). OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens. https://tools.ietf.org/html/draft-ietf-oauth-mtls-08#section-3 [accessed 9/10/2019] |
| [LeanIX, 2017] | LeanIX. Authorization and Authentication with Microservices. https://blog.leanix.net/en/authorization-authentication-with-microservices [accessed 9/10/2019] |
| [LIGHTCYBER 2016] | LightCyber. Cyber Weapons Report. (2016). Ramat Gan, Israel. 14pp. http://lightcyber.com/cyber-weapons-report-network-traffic-analytics-reveals-attacker-tools/ [accessed 5/11/17]. |
| [Lund University 2015] | Lund University. (2015). Knutsson, M., Glennow, T. Challenges of Service-Oriented Architecture (SOA)-From the public sector perspective. School of Economics and Management Department of Informatics. https://pdfs.semanticscholar.org/23ef/7b2abcfaed46f37ba7330c1f4409ca8aff6a.pdf [accessed 9/4/2019] |
| [LWN.net, 2018] | LWN.net. (2018). Easier Container Security with Entitlements, by Beaupré, A. https://lwn.net/Articles/755238/ [accessed 9/10/2019] |
| [Medium, 2018] | Medium Corporation. (2018). Tech Tajawal - *Microservices Authentication and Authorization Solutions*, by Ajoub, M. https://medium.com/tech-tajawal/microservice-authentication-and-authorization-solutions-e0e5e74b248a [accessed 9/5/2019] |
| [Newman, 2015] | Newman S. (2015). Building Microservices. Designing Fine-Grained Systems. O'Reilly Media. |
| [Nordic APIS, 2017] | NORDIC APIS. (2017). API Security: The 4 Defenses of The API Stronghold. Sandoval, K. https://nordicapis.com/api-security-the-4-defenses-of-the-api-stronghold/ [last accessed 11/11/2019] |
| [O'Reilly Safari Publications, 2018] | McLarty, M., Wilson, R., and Morission, S. (2018). Securing Microservice API's: Sustainable and Scalable Access Control. First Edition. O'Reilly Safari Publications. |
| [Project Calico] | Project Calico. Policy-driven Network Security. Tigera Inc. https://www.projectcalico.org [accessed 9/10/2019] |
| [Red Hat OpenShift] | Red Hat OpenShift - OpenShift Container Platform - Authorization. https://docs.openshift.com/container-platform/3.6/architecture/additional_concepts/authorization.html#rules-def [accessed 9/10/2019] |

| | |
|---|---|
| [Rosen at al., (2012)] | Rosen, M., Lublinsky, B., Smith, K. T., & Balcer, M. J. (2012). Applied SOA: Service-oriented Architecture and Design Strategies: John Wiley & Sons |
| [Simplicable 2011] | Simplicable Technology Guide. (2011). SOA Security Challenges by Spacey, J. https://arch.simplicable.com/arch/new/9-soa-security-challenges [accessed 9/4/2019] |
| [SPIRE] | SPIRE. The SPIFFE Runtime Environment. https://spiffe.io/spire/ [accessed 9/10/2019] |
| [Stojanovic et al., (2004)] | Stojanovic, Z., Dahanayake, A., Sol, H.G. (2004) Modeling and Design of Service-oriented Architecture. Conference: Proceedings of the IEEE International Conference on Systems, Man & Cybernetics: The Hague, Netherlands. |
| Tanenbaum at al., (2007)] | Tanenbaum, A. and Van Steen, M. (2007). Distributed Systems: Principles and Paradigms. Pearson Prentice Hall. |
| [TechTarget, 2018] | TechTarget. (2018). How Proper Networking Supports Microservices Security by Nolle, T., CIMI Corporation. https://searchapparchitecture.techtarget.com/tip/How-proper-networking-supports-microservices-security [accessed 9/10/2018] |
| [The SOA Source Book] | Microservices Architecture – SOA and MSA. https://www.opengroup.org/soa/source-book/msawp/p3.htm [accessed 8/30/19]<br><br>Governance Framework – SOA Governance Reference Model (SGRM) https://www.opengroup.org/soa/source-book/gov/p4.htm [accessed 9/4/2019] |
| [Unbound, 2018] | Unbound Tech. (2018). Introduction to Cloud-Native Secrets Management: Part III. https://www.unboundtech.com/importance-of-container-identity/ [accessed 9/10/2019] |
| [Weaveworks] | Weaveworks. Microservices Security: Built-In Best Practices. https://www.weave.works/use-cases/security/ [accessed 9/13/2019] |
| [Wei et al., (2018)] | Wei, F., Ramkumar, M., Mohanty, S.D. (2018). A Scalable, Trustworthy Infrastructure for Collaborative Container Repositories. *ArXiv, abs/1810.07315*. |

| | |
|---|---|
| [Zimmerman, 2016] | Zimmerman, O. (2016). Microservices tenets: an Agile Approach to Service Development and Deployment. Computer Science – Research and Development, pp. 1-10. |
| [Yarygina et al., 2018] | Yarygina, T. and Bagge, A.H. (2018). Overcoming Security Challenges in Microservice Architecture. Proceedings of 2018 IEEE Symposium on Service-Oriented System Engineering. |