AWS Well-Architected フレームワーク

SaaS レンズ

SaaS レンズ: AWS Well-Architected フレームワーク

Table of Contents

SaaS レンズ

公開日: 2020 年 12 月 3 日 (改訂履歴)

要約

本ホワイトペーパーでは AWS Well-Architected フレームワークの SaaS レンズについて説明しています。これにより、お客様はクラウドベースのアーキテクチャを評価および改善し、設計上の決定がビジネスに及ぼす影響をより理解できるようになります。ここでは Well-Architected フレームワークの柱とされる 5 つの概念領域における一般的な設計の原則と、特定のベストプラクティスおよびガイダンスを紹介します。

はじめに

AWS Well-Architected フレームワークは、AWS でシステムを構築する際に行う意思決定の#所と短所を理解するために役立ちます。効率が良く、費用対効果が#く、安全で信頼のおけるクラウド対応システムを設計して運#するために、アーキテクチャに関するベストプラクティスをこのフレームワークに従って学ぶことができます。これは、アーキテクチャをベストプラクティスに照らし合わせて一貫的に測定し、改善すべき点を特定する手段を提供します。システムを適切に設計することによってビジネスの成功の可能性が大いに高まると当社は確信しています。

この「レンズ」では、AWS クラウドでマルチテナント SaaS アプリケーションのワークロードを設計、デプロイ、設計する方法に焦点を当てます。ここでは簡潔に、SaaS ワークロードに固有の Well-Architected フレームワークの詳細のみを取り上げてみました。アーキテクチャの設計時には、このドキュメントに含まれていないベストプラクティスと質問を検討する必要があります。詳細は AWS Well-Architected フレームワークホワイトペーパーをご覧ください。

このドキュメントは、最高技術責任者 (CTO)、設計者、デベロッパー、運用チームメンバーなどの技術担当者のみなさまを対象に書かれています。このドキュメントを一通り読んでいただくことで、SaaS アプリケーションのアーキテクチャを設計するための AWS のベストプラクティスと戦略を理解することができます。

定義

AWS Well-Architected フレームワークは、運用上の優秀性、セキュリティ、信頼性、パフォーマンス効率、コスト最適化という 5 本の柱を基本としています。SaaS によって、このそれぞれの柱に新しい検討事項が追加されます。SaaS アプリケーションのマルチテナントという性質上、アーキテクトは、SaaS 環境のセキュリティ、オペレーション効率、安定性、俊敏性を確保する方法をよく検討する必要があります。このセクションでは、このドキュメント全体で扱う SaaS の概念の概要を説明します。SaaS アーキテクチャを構築する際は、検討すべき 10 の領域があります。

トピック

テナント

テナントとは、SaaS 環境における最も基本的な構造です。SaaS プロバイダーがアプリケーションを構築すると、このアプリケーションは顧客が利用できるように公開されます。この環境を利用するために登録している顧客は、システムのテナントとして表されます。例えば、あなたの組織が会計サービスを作成し、別の企業がそのサービスを使ってビジネスを管理できるようにするとします。このような企業はすべて、提供するシステムにとってのテナントと見なされます。

テナントは通常、登録時にテナント管理者のユーザー情報を提供します。このテナント管理者は、システムにログインしたうえで、自社のニーズに基づいた設定を行います。これには、テナント環境にユーザーを追加する機能などが含まれます。

このようなモデルで提供されるソフトウェアは、マルチテナント SaaS システムと呼ばれます。これは、このサービスの各テナントが、統合機能を通じてテナントのニーズをサポートする 1 つの共有

システムを使用していることによります。例えば、このようなシステムのアップデートは、システムを使うすべてのテナントに適用されます。

サイロモデル、プールモデル、ブリッジモデル

SaaS アプリケーションを構築するアーキテクチャモデルには、さまざまな種類があります。規制、競争力、戦略、コスト効率、マーケットごとの検討事項など、すべてが SaaS アーキテクチャの形になんらかの影響を与えます。これと同時に、SaaS アプリケーションの規模の定義には、戦略やパターンがあります。このパターンには「サイロ、ブリッジ、プール」の 3 つのカテゴリがあります。

まずサイロモデルとは、テナントに専用リソースが提供されるアーキテクチャです。SaaS 環境で、システムの各テナントが完全に独立したインフラストラクチャスタックを所有している状況を想像してください。または、システムの各テナントがそれぞれ別のデータベースを持っているとしましょう。テナントのリソースの一部または全部が、このように専用のリソースとしてデプロイされている場合、これをサイロモデルと呼びます。サイロ環境は専用のリソースを有しますが、アイデンティティ、オンボーディング、オペレーションが共有に依存することは同じであり、すべてのテナントは共有構造で管理およびデプロイされるという点に注意する必要があります。これが SaaS がマネージド型サービスモデルとは異なる点です。後者では、顧客は個別のオンボーディング、管理、オペレーション機能を使い、独自バージョンのシステムを実行する場合もあります。

これと反対に、SaaS のプールモデルとは、テナントがリソースを共有しているシナリオを指します。これはマルチテナントの古い概念で、テナントはスケールメリット、管理のしやすさ、俊敏性などの理由で、共有のスケーラブルなインフラストラクチャに依存しています。このような共有リソースは、コンピューティング、ストレージ、メッセージングなど、SaaS アーキテクチャの一部または全部の要素に適用できます。

最後のパターンはブリッジモデルです。ブリッジとは、SaaS ビジネスは必ずしもサイロまたはプールのどちらかに分類されるわけではないということを表しています。むしろ多くのシステムでは、混合モードにして、サイロモデルとプールモデルのシステムを併用しています。例えば、アーキテクチャにおける一部のマイクロサービスの実装にサイロを使用し、その他のサービスにプールを使用するような場合があります。サービスデータの規制のプロファイルと他のテナントによる影響の可能性は、マイクロサービスにサイロモデルを採用する理由になる場合があります。その一方で、俊敏性、アクセスパターン、コストプロファイルの課題などは、プールモデルを選択する理由になるでしょう。

SaaS アイデンティティ

ほとんどのシステムは、すでにアイデンティティプロバイダーを利用して認証を行っています。SaaS の世界では、アイデンティティの概念を拡大し、アイデンティティの定義にテナントの概念を組み込む必要があります。これは、ユーザーの認証後に、そのユーザーが誰なのか、どのテナントに関連付けられているのかを把握する必要があるということです。このユーザーアイデンティティをテナントのアイデンティティと結合したものを、SaaS アイデンティティと呼んでいます。この概念は SaaS アーキテクチャの基本要素であり、SaaS アプリケーションの一部であり基盤でもあるマルチテナントポリシーと戦略を適用する際に使用されるテナントコンテキストを提供します。

テナントの分離

テナントの分離は、すべての SaaS プロバイダーが取り組む必要のある基本的なトピックの 1 つです。独立系ソフトウェアベンダー (ISV) が、SaaS にシフトし、共有インフラストラクチャモデルを採用してコスト効率および運用効率を達成する際には、マルチテナント環境において、テナントが他のテナントのリソースにアクセスできないようにする方法を決定するという課題にも取り組む必要があります。SaaS ビジネスにとって、どのような形でもテナントの境界線を超えることは、重大かつ回復不能に至る可能性のあるイベントに相当します。

テナントの分離は SaaS プロバイダーにとって不可欠と考えられますが、この分離を達成するための戦略およびアプローチは一様ではありません。SaaS 環境でテナントの分離を実現する方法には、非常に多くの要素が影響を与えます。ドメイン、コンプライアンス、デプロイモデル、AWS サービスの選択など、すべてがテナント分離の方法にそれぞれの検討事項として関わってきます。

分離を実行する方法にかかわらず、各 SaaS アーキテクチャはテナントのリソースを効率的に分離するために必要な構造を必ず備えている必要があります。

データのパーティション化

データの構成方法は、マルチテナントデータを表すさまざまなアーキテクチャパターンを見て決定する必要があります。データは個別のデータベースに保管するのがよいでしょうか。それとも、共有構造に一緒に保存するのがよいでしょうか。 このようなマルチテナントのデータ保管のメカニズムおよびパターンは、一般的にデータのパーティション化といいます。

他のテナントによる影響

「他のテナントによる影響」というのは基本的なアーキテクチャのパターンおよび戦略においてよく使われる用語です。この背景には、システムのユーザーのワークロードがシステムのリソースに負荷

を与え、同じシステムの他のユーザーに悪影響を与える可能性があるという考え方があります。これによって、1 人のユーザーが他のユーザーのパフォーマンスを低下させてしまう結果となることもあります。

テナントが共有リソースを使い切ってしまう可能性のあるマルチテナント環境では、この考え方の重要性は増します。これをさらに複雑化させるのは、マルチテナント環境におけるワークロードは予測が難しいという点です。これによって、SaaS アーキテクチャにおいて「うるさいテナント」による影響の可能性を管理およびできるだけ軽減できる独自の戦略を実行する必要性はさらに高まります。

テナントのオンボーディング

SaaS アプリケーションでは、SaaS 環境に新しいテナントを迎え入れる際に、フリクションレスモデルを利用します。このときには、新しいテナントの作成に必要なすべての要素を正常にプロビジョニングして構成するために、数多くのコンポーネントを調整して配置する必要があります。SaaS アーキテクチャでは、このプロセスをテナントのオンボーディングと呼びます。テナントのオンボーディングは、テナントが直接開始する場合と、プロバイダーが管理するプロセスの一部として開始される場合がある点に注意してください。

テナントの階層

SaaS アプリケーションは広い範囲のマーケットセグメントをサポートするように設計されていることが多く、顧客のプロファイルのタイプに合わせた価格体系や機能を提供しています。このようなプロファイルは、よく階層と呼ばれます。さまざまな階層の多様なニーズをサポートするということは、それぞれに合わせて機能をカスタマイズできるような設計構造を導入するということです。これらの階層モデルは、SaaS ソリューションのコスト、オペレーション、管理、信頼性の規模に影響する可能性があります。

テナントのアクティビティと消費

SaaS のマルチテナント環境では、テナントによるアプリケーションの使用状況と、それがシステムのアーキテクチャにかけている負荷の状況を可視化することが重要です。テナントレベルでこのような情報をトラッキングすることにより、システムの効率的にスケーリングを行う能力と、システム環境に加わる、常に進化を続けるワークロードをサポートする能力を評価できます。SaaS システムから取得されるこのメトリクスおよびインサイトは「テナントのアクティビティと消費」と呼ばれています。

計測と課金

SaaS 製品は従量課金制で提供されることがほとんどで、製品のコストは顧客の消費プロファイルに基づき決定されます。これにより、価値と SaaS システムにかかる負荷に密接に結びついた料金体系モデルを顧客に提供できます。このモードでは、消費量の計測メカニズムの定義および導入を SaaS プロバイダーが行います。この計測データは通常、課金情報を集約して請求書を生成する請求システムに送信されます。消費量ベースの料金体系は、追加料金 (サブスクリプションなど) と組み合わせる課金モデルとして提示されます。

テナント対応オペレーション

SaaS 環境におけるオペレーション体制では、テナント対応オペレーションのオプショナルビューを表示できるオペレーションビューを作成するためのメカニズムおよびツールが必要になります。これは、SaaS プロバイダーがシステムアクティビティと健全性を、個別テナントおよびテナントの階層のレベルで表示する必要があるという考えに基づきます。このような仕組みは、個別テナントのアクティビティおよび消費のトレンドやパターンを診断したり評価したりするために必須となります。

一般的な設計の原則

Well-Architected フレームワークにより、SaaS アプリケーションのクラウド上における適切な設計をサポートする全体の設計原則を特定できます。

- SaaS アーキテクチャに万能の解は存在しない: SaaS を使ったビジネスのニーズ、各ドメインの性質、コンプライアンス要件、マーケットセグメント、ソリューションの特性など、すべての要素が SaaS 環境のアーキテクチャに明確な影響を与えます。すべての SaaS アーキテクチャは、SaaS 製品として成功するために必要な、俊敏性と SaaS の基本原則を実現する運用および顧客エクスペリエンスを提供するものである必要があります。システムはその設計方法にかかわらず、1 つの画面でオンボーディング、管理、オペレーションを実施できる機能をテナントに提供し、SaaS ビジネスを構築するための基盤となる俊敏性およびスケールメリットを、SaaS 組織が達成できるようにする必要があります。
- マルチテナントのワークロードと分離の特性に基づいてサービスを分解する: システムをサービスに分解する場合は、その分解方法の判断にあたって、マルチテナントのワークロード、テナントの階層、分離の要件が、システムの一部であるサービスにどのように影響するかを検討する必要があります。このようなケースでは、各サービスを独立させて考えなければなりません。例えば、あるサービスではデータの保存にプールを使用できたとしても、別のサービスではコンプライアンスや「他のテナントによる影響」の課題を考慮してデータの保存にサイロ化が必要である場合もあります。ほかにも、テナントの階層構造を実現させるため、サイロモデルでデプロイすべきサービスもあるかもしれません。例えば、プレミアム顧客であるテナントに対しては、プレミアム階層への価値提供の一部として、一部のサービスをサイロモデルで提供する場合があります。
- すべてのテナントリソースを分離する: SaaS システムの成功は、テナントリソースをクロステナントアクセスから保護するセキュリティモデルにかかっています。堅牢な SaaS アーキテクチャは、すべてのレイヤーにわたって分離戦略を導入し、テナントリソースにアクセスするときには、現在のテナントコンテキストのみがアクセス可能な特別な構造を持っています。
- 成長のためのデザイン: SaaS モデルへの移行は、多くの場合 SaaS 組織の成長を意味します。SaaS 製品のアーキテクチャおよびオペレーションの範囲を定義するときには、必ずその環境が、加速するテナント数の増加に対応できるかどうかを考える必要があります。SaaS のアーキテクトは、テナントのオンボーディングの急増をオペレーション費用の追加なしで処理できる、俊敏な対応力のあるスムーズな環境を構築する必要があります。このような考え方によって、顧客ベースが急成長しても、SaaS 環境のオペレーションやインフラストラクチャの規模を拡大する必要性はなくなります。
- ツールを配備してテナントのメトリクスの取得と分析を行う: 複数のテナントをサービス環境に配置するとき、特にこれが共有環境である場合は、テナントによるシステムの使用状況を明確に可視

化することが課題となります。SaaS チームは、テナントが使用している機能、システムに存在するワークロード、テナントが直面するボトルネック、アクティビティのコスト内容などに関するインサイトを画面で表示できるメトリクスツールに投資する必要があります。このデータは、SaaS 企業のビジネス・アーキテクチャ・オペレーションの健全性に直接的な影響を与え、企業戦略に情報をフィードバックするトレンド分析の中核となります。

- テナントを再現可能な自動一括プロセスでオンボーディングする: SaaS とは、俊敏性を実現することに他なりません。この俊敏性の点で重要なポイントは、テナントのオンボーディングプロセスです。堅牢な SaaS システムには、システムに新しいテナントをオンボーディングするためのスムーズで再現可能なプロセスが必要です。これはスケーリングを容易にすると同時に、成長を促進するために不可欠です。また、新規顧客に迅速に価値を提供することにもつながります。
- 複数テナントの機能のサポートを計画する: SaaS のマーケットおよび顧客には、さまざまなタイプがあります。SaaS 企業はほとんどの場合、アーキテクチャやオペレーションにさまざまな要望を出す多様なテナントをサポートする必要があります。SaaS プロバイダーおよびアーキテクトとしては、このようなテナントのペルソナをモデル化することによって、各ニーズに合わせた個々のバージョンの SaaS 製品を用意する必要がなく、幅広いテナントのユースケースをカバーできる構造とメカニズムを備えた 1 つの総合的な環境を構築することが重要です。また、ビジネスにおいて複数のセグメントにリーチできる製品階層を用意し、その階層によって顧客のレベルアップを推進するような、システムにおける提供価値の境界線を特定することも重要です。
- グローバルなカスタマイズで 1 回限りの要件に対応: SaaS の俊敏性とイノベーションは、すべての顧客に 1 つの環境を提供することによって達成されます。アップデート、管理、オペレーションをすべての顧客に対してまとめて実行できることが SaaS の基本です。ただし現実には、一部の顧客からカスタマイズを要望されることもあります。このようなカスタマイズは、他の顧客にも提供できる設定オプションとして導入すべきです。このような機能を製品の核とすれば、SaaS 企業はビジネスの俊敏性、オペレーション効率、イノベーションの目標を犠牲にすることなく、1 回限りのニーズに対応できるようになります。
- ユーザーアイデンティティをテナントアイデンティティと関連付ける: データロギング、メトリクスの記録、データへのアクセスなどを実行するテナントのコンテキストの概念は、程度は異なるもののアーキテクチャのすべてのレイヤーで必要になります。つまり、他のサービスを呼び出すことなく、アプリケーションのレイヤーによって解決および簡単にアクセスできる構造として、テナントコンテキストを最優先する必要があるということです。ソリューションの認証および承認プロセスは、テナントアイデンティティ (およびその他テナント属性も必要なケースも) と承認済みユーザーのアイデンティティを関連付ける必要があります。これによってシステムのすべてのレイヤーで利用できる SaaS アイデンティティが生成され、テナントコンテキストへのアクセスが可能になります。

- インフラ消費とテナントアクティビティを一致させる: SaaS 環境におけるテナントアクティビティは、ほとんど予測不可能です。テナントが消費するリソースの種類は、その消費方法や消費するタイミングなどを含めて、大きく変化する可能性があります。システムにおけるテナントの数も、日常的に変化します。このような要素はスケーリングの課題となる一方、堅牢な SaaS アーキテクチャでは、オーバープロビジョニングを制限するポリシーを実装し、アプリケーションのインフラ消費をテナントアクティビティのリアルタイムトレンドに合わせます。こうすることで、テナントのワークロードと SaaS インフラストラクチャ全体のコストプロファイルの乖離を防ぐことができます。
- デベロッパーがマルチテナントの概念を意識する範囲を制限する: テナンシーはアーキテクチャのすべてのレイヤーに通じるものですが、目標はデベロッパーがテナントを意識する範囲を制限することです。経験則として、デベロッパーがマルチテナントサービスを構築するときにすることは、テナントの概念がないサービスを構築することとそれほど変わりません。もしデベロッパーがコード全体にテナントの情報を追加する必要があったとしたら、アプリケーションのマルチテナントポリシーおよびメカニズムにおけるコンプライアンスのコントロールと適用は困難になるでしょう。つまり、テナントの詳細が表示されないようにしながら、デベロッパーにライブラリや再利用可能な構造を提供するということです。
- SaaS は技術的な実装方法ではなくビジネス戦略である: SaaS 環境およびその基盤となるテクノロジーの選択肢は、ビジネスの俊敏性、イノベーション、競争的なニーズによって直接決定されます。このポイントとマインドセットは、ダウンタイムなし、定期的なアップデート、顧客との密接なコネクションといった、顧客のためのサービス体験を生み出すための軸として展開します。つまり、継続的な進化とマーケットデマンドへの迅速な対応を推進できるようなアーキテクチャおよびオペレーションの規模を設計するということです。技術的には堅牢でも、俊敏性、イノベーション、オペレーション効率を考慮していないアーキテクチャは、特に他に競合の SaaS プロバイダーがいれば、競争の激しいマーケットで戦うことができないでしょう。
- テナント対応オペレーションビューを作成する: オペレーションチームは、マルチテナント環境において新しい課題に直面しています。システムの健全性およびアクティビティのグローバルビューを提供することは SaaS 環境においても重要である一方、堅牢な SaaS のオペレーションでは、これに特定のテナントやテナント階層によるシステムの利用状況に関するインサイトも追加されます。SaaS のオペレーションチームは、ダッシュボードおよび個別テナントのアクティビティとワークロードを分析してプロファイリングできるビューを構築する必要があります。個別テナントのレベルで利用状況を確認してトラブルシューティングできる機能は、プロアクティブで効率的なマルチテナントオペレーションに欠かせないものです。
- 個別テナントによるコストへの影響を測定する: SaaS 企業の経営チーム、アーキテクトチーム、オペレーションチームは、テナントが SaaS 環境のコストにもたらす影響について明確な共通認識を持つ必要があります。例えば、ベーシック階層に属するテナントには、プレミアム階層のテナントよりも高いコストがかかっていますか? テナントの消費パターンや特徴は、SaaS 環境のコス

ト内容に影響を与えていますか? これらは、テナントのコスト内容を明確に把握していればしっかりと応えられる質問の例です。これはテナントリソースが複数のテナントによって共有されているケースで特に重要となる内容です。このようなデータを収集して表示すれば、SaaS 企業のアーキテクチャやビジネスモデルの形成に役立つ貴重なインサイトを提供できるようになります。

シナリオ

このセクションでは、AWS で SaaS ソリューションを設計および構築する際に使用される、一般的なパターンと戦略を表した一連のシナリオを取り扱います。これらのシナリオごとの前提条件、設計の一般的なドライバー、および AWS アーキテクチャの構造を使用したこれらのシナリオの実現方法のリファレンスアーキテクチャについて説明します。

トピック

サーバーレス SaaS

SaaS 提供モデルへの移行には、コストと運用の効率を最大限にしたいという動機が伴っています。テナントのアクティビティの予測が難しい、マルチテナントの環境では、これは特に困難となります。テナントのアクティビティをリソースの実際の消費量に合わせるようスケーリング戦略のバランスを取ることは困難です。今は機能している戦略も、将来は機能しなくなるかもしれません。

これらの要因により、サーバーレスモデルにおいて SaaS が説得力のある選択となります。SaaS アーキテクチャからサーバーの概念を取り除くことで、組織はアプリケーションが消費する正確な量のリソースをスケーリングして提供するための、マネージドサービスを活用できます。これにより、アプリケーションのアーキテクチャと運用上のフットプリントが簡素化され、スケーリングのポリシーに継続的に追われ管理する必要性がなくなります。また、運用上のオーバーヘッドと複雑さも低減され、運用上の責務をマネージドサービスに任せることができます。

AWS は、サーバーレスの SaaS ソリューションを実装するために使用できる幅広いサービスを提供しています。図 1 はサーバーレスのアーキテクチャの例を示しています。

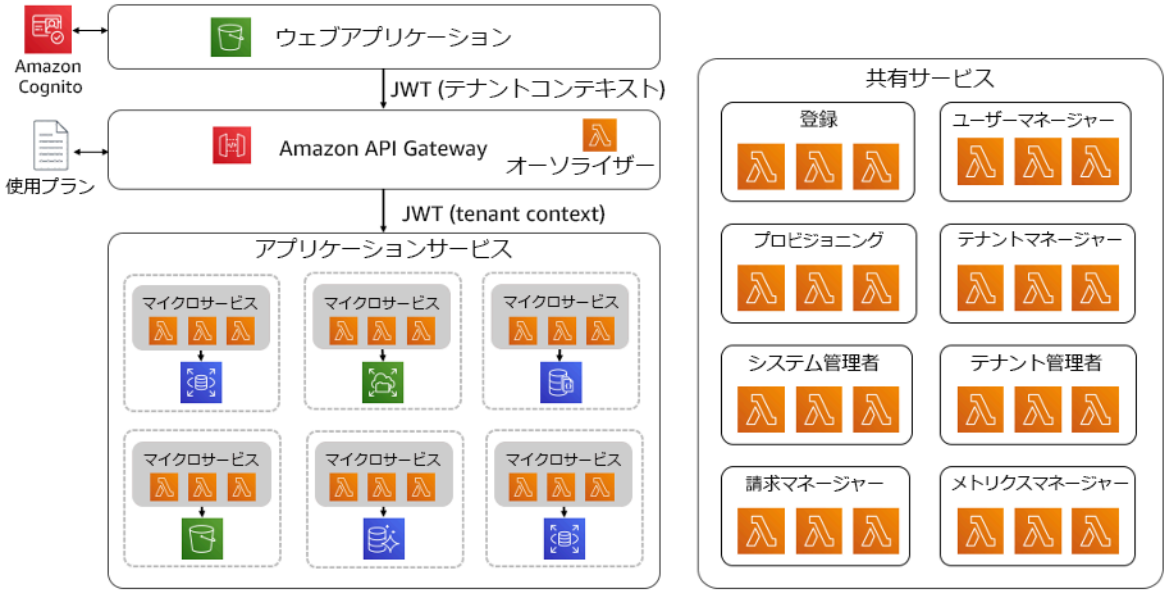

図 1: サーバーレス SaaS アーキテクチャ

サーバーレス SaaS アーキテクチャの稼動部は、従来のサーバーレスのウェブアプリケーションアーキテクチャとさほど変わらないことがわかります。この図の左側では、Amazon S3 バケットからホストおよび配信される、ウェブアプリケーションがあることがわかります (おそらく、React や Angular などの最新のクライアントフレームワークのいずれかを使用しています)。

このアーキテクチャでは、アプリケーションは Amazon Cognito を SaaS ID プロバイダーとして活用しています。認証サービスは、JSON Web Token (JWT) を介して提供される SaaS のコンテキストを含むトークンを生成します。このトークンはその後、すべてのダウンストリームのアプリケーションサービスとのやり取りに挿入されます。

サーバーレスアプリケーションのマイクロサービスとのやり取りは、Amazon API Gateway によって調整されます。ここで、ゲートウェイは複数の役割を果たします。受信するテナントのトークンを (Lambda オーソライザーを介して) 検証し、各テナントのリクエストをマイクロサービスにマッピングするほか、(使用プランを介して) 異なるテナント層の SLA の管理に使用できます。

SaaS アプリケーションのマルチテナントの IP を実装するさまざまなサービス用のプレースホルダーである、一連のマイクロサービスも示されています。各マイクロサービスは、マイクロサービスの契約を実装する 1 つ以上の Lambda 機能から構成されます。マイクロサービスのベストプラクティスに合わせて、これらのサービスは自身が管理するデータのカプセル化も行います。これらのサービスは、必要な場所にテナントのコンテキストを取得して適用するために、受信された JWT トークンを使用します。

また、各マイクロサービスのストレージも示されています。マイクロサービスのベストプラクティスに従うために、各マイクロサービスは自身が管理するリソースを有しています。例えば、データベースは 2 つのマイクロサービスが共有することはできません。マルチテナントのデータの提示はサービスごとに変化する性質のため、SaaS はここでも力を発揮します。あるサービスでは各テナントに対する個別のデータベースがあり (サイト)、一方他のサービスでは同じテーブル内のデータを混在させる場合があります (プール)。ここで行われるストレージの選択は、コンプライアンス、他のテナントによる影響、隔離、パフォーマンスの考慮事項によって決定します。

最後に、この図の右側には一連の共有サービスがあります。これらのサービスは、図の左側で稼動するすべてのテナントによって共有された、すべての機能を提供します。これらのサービスは、通常はテナントのオンボード、管理、運用が必要な、独立したマイクロサービスとして構築される一般的なサービスを表しています。

一般的なサーバーレス Well-Architected のベストプラクティスの詳細は、サーバーレスアプリケーションレンズのホワイトペーパーでご確認いただけます。

クロステナントのアクセスの防止

各 SaaS アーキテクチャでは、テナントが他のテナントのリソースにアクセスするのを防止する方法についても考慮する必要があります。設計上は、特定のタイミンで Lambda 機能を実行できるのは 1 つのテナントのみであるため、AWS Lambda を使用してテナントを隔離する必要性について疑問に思われるかもしれません。確かにそのとおりですが、隔離において、機能を実行しているテナントが、他のテナントに属する可能性のある他のリソースにアクセスしないようにする必要もあります。

SaaS プロバイダーにおいて、サーバーレス SaaS 環境に隔離を実装するための基本的なアプローチが 2 つあります。この最初のアプローチでは、サイロのパターンに従い、各テナントの Lambda 機能の個別のセットをデプロイします。このモデルでは、各テナントの実行ロールを定義し、各テナントの個別の機能をその実行ロールとともにデプロイします。この実行ロールは、特定のテナントでどのリソースがアクセス可能なのかを定義します。このモデルで一連のプレミアム層のテナントをデプロイするとします。しかし、これは管理が困難で、アカウントの制限に達する可能性があります (システムがサポートするテナント数による)。

もう 1 つのアプローチは、プールモデルにより合わせられています。ここでは、すべてのテナントからの呼び出しを受け入れるために十分な範囲を持つ実行ロールとともに、機能がデプロイされます。このモデルでは、マルチテナントの機能の実装でのランタイムで、隔離を適用する必要があります。図 2 は、これがどのようにして行われるかを示しています。

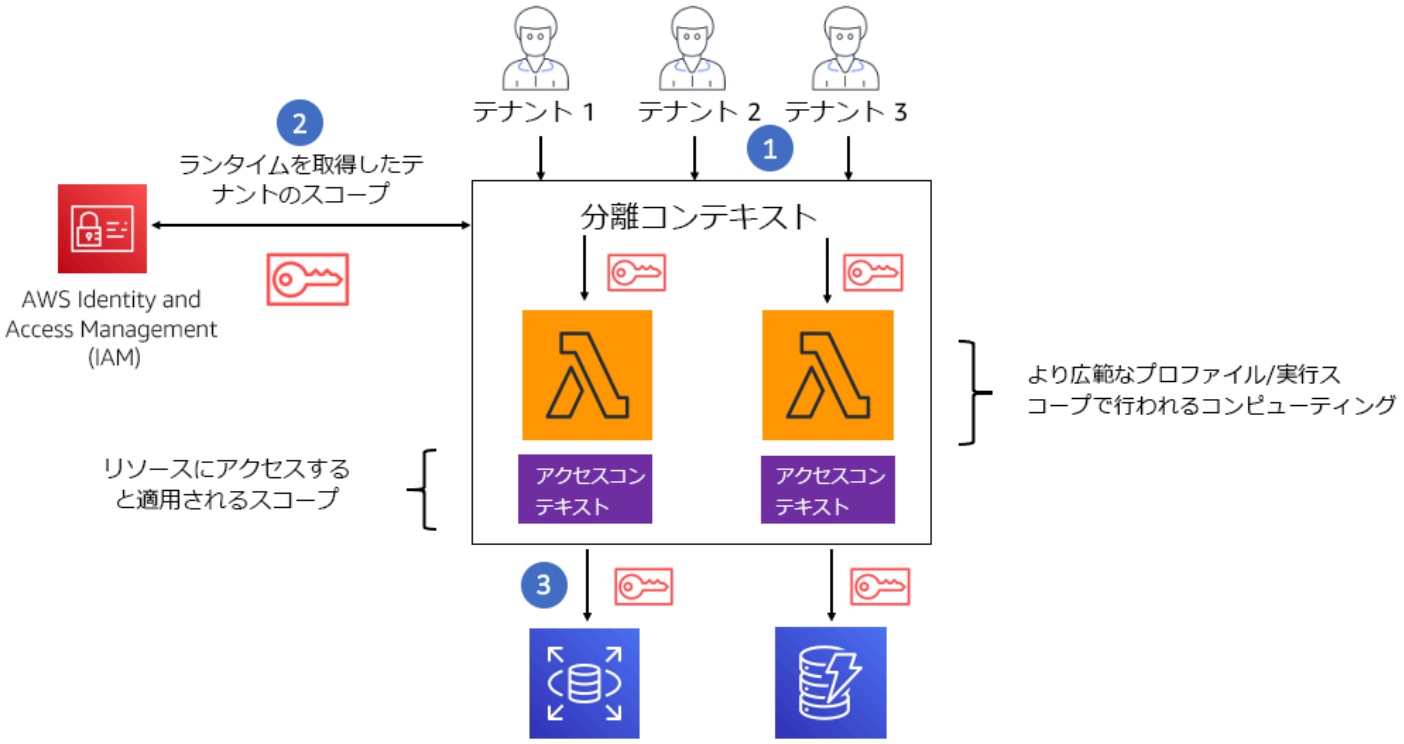

図 2: サーバーレス環境での隔離

この例では、一連の Lambda 機能にアクセスする 3 つのテナントがあることがわかります。これらの機能は共有されているため、すべてのテナントをカバーする実行ロールとともにデプロイされます。これらの機能の実装の中で、AWS Security Token Service (AWS STS) からの新たな一連のテナント範囲の認証情報を取得するために、現在のテナントのコンテキスト (JWT を介して提供される) が使用されます。これらの認証情報があれば、テナントのコンテキストでリソースにアクセスするために使用できます。この例では、ストレージにアクセスするためにこれらの範囲が指定された認証情報を使用します。

このモデルでは、隔離モデルの一部が Lambda 機能のコードにプッシュされるという点に注意してください。デベロッパーの視野外でこの概念を導入できる、機能ラッパーなどのテクニックがあります。これらの認証情報の取得の詳細を Lambda レイヤーに移動させ、さらにシームレスで中央管理された構造にすることもできます。

レイヤーがテナントの詳細を隠す

あらゆる SaaS アーキテクチャにおける目標の 1 つとして、デベロッパーにテナントの詳細を意識させないことがあります。サーバーレス SaaS 環境では、デベロッパーの視野外でマルチテナントのポリシーを実装できる、共有コードを作り出すための方法として、Lambda レイヤーを使用できます。

図 3 は、これらのマルチテナントの概念を実現するために Lambda レイヤーをどのように使用できるのかの例を示しています。ここで、ログと指標のデータを発行する必要がある、2 つの異なるマイクロサービス (生産と注文) があります。重要な点として、両方のサービスが、テナントのコンテキストをログメッセージと指標イベントに挿入する必要があります。しかし、各サービスにこれらのポリシーをそれぞれ実装することは理想的ではありません。代わりに、このデータの発行を管理するコードを含むレイヤーを導入しました。

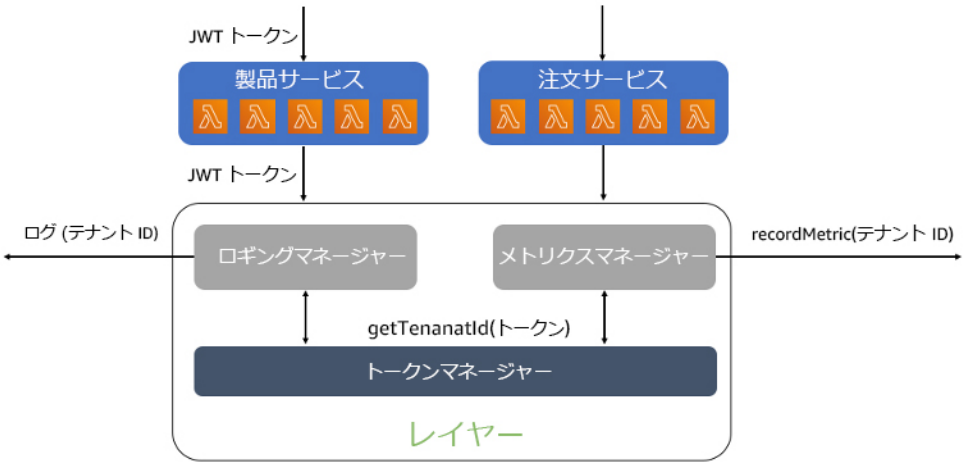

図 3: Lambda レイヤーがテナントの詳細を隠す

レイヤーに、JWT トークンを受け入れるログと指標のヘルパーが含まれていることがわかります。これにより、マイクロサービスがどのテナントを取り扱っているのかを心配することなく、各マイクロサービスがシンプルにこれらの機能を呼び出し、トークンを供給できるようになります。その後、レイヤー内で、コードが JWT トークンを使用してテナントのコンテキストを解決し、ログメッセージと指標イベントに挿入します。

これは、テナントのコンテキストを隠すためにレイヤーを適用する方法の一例にすぎません。テナントのコンテキストを挿入するためのポリシーとメカニズムを、デベロッパーは一切意識しなくてよいことに真の価値があります。これらはまた個別に更新、バージョン管理、デプロイされます。遅延とボトルネックの原因となる恐れがある個別のマイクロサービスを導入することなく、環境内でこれらのポリシーをさらに中央的に管理できるようになります。

Amazon EKS SaaS

多くの SaaS プロバイダーにおいて、Amazon Elastic Kubernetes Service のプロファイル (Amazon EKS) は、マイクロサービスの開発とアーキテクチャ上の目標への適合性を表します。開発のツールと思考を完全に変更する必要なく、俊敏性、規模、コスト、運用上の目標の実現に役立つマルチテナントのマイクロサービスを構築およびデプロイするための方法を提供します。Kubernetes のツール

とソリューションの豊富なコミュニティもまた、SaaS デベロッパーに SaaS 環境の構築、管理、保護、運用のためのさまざまな選択肢を提供します。

コンテナベースの環境では、多くのアーキテクチャは、クロステナントのアクセスをしっかりと防止する方法に焦点が当てられています。テナントにコンテナを共有させるようにしたいという誘惑があるかもしれませんが、これはテナントがソフトなマルチテナントにおいて有効であることを前提としています。しかし、ほとんどの SaaS 環境の隔離の要件では、隔離の実装がより堅牢である必要があります。

これらの隔離の要因は、Amazon EKS を使用して構築されるアーキテクチャ上のモデルに大きな影響を及ぼす可能性があります。Amazon EKS を使用した SaaS アーキテクチャの構築の一般的な指標は、テナントをまたいだコンテナの共有を防止することです。これによりアーキテクチャのフットプリントの複雑性が高まるものの、マルチテナントの顧客のドメイン、コンプライアンス、規制の必要性に対応した隔離モデルを確実に作成するための、基盤となるニーズに対応できます。

アーキテクチャのサンプルで、SaaS Amazon EKS 環境の基本的な要素を確認しましょう。この隔離には多くの可動部がありますが、まずは、すべてのテナントにまたがる、核となる水平の概念をサポートするために使用される、共有サービスについて見てみましょう (図 4)。

まず、可用性とスケーラビリティが高いすべての AWS アーキテクチャの一部である、基盤の要素があることがわかります。環境には、3 つのアベイラビリティーゾーンから構成される VPC が含まれます。テナントの受信トラフィックのルーティングは Amazon Route 53 によって管理されます。これは、アプリケーションの受信リクエストを、NGINX イングレスコントローラーによって定義されたエンドポイントへ誘導するように構成されています。以下に示されている、マルチテナントのルーティングに欠かせない Amazon EKS クラスタ内での選択したルーティングが、コントローラーによって可能となります。

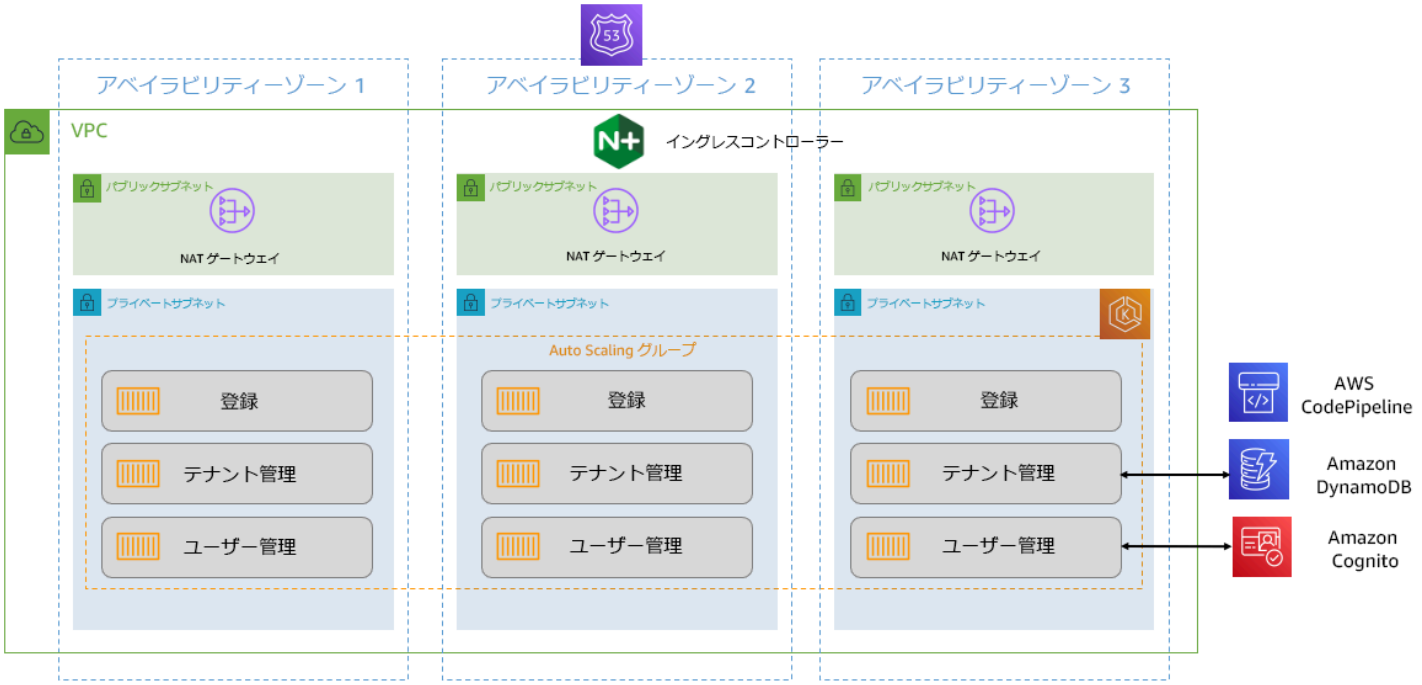

図 4: Amazon EKS SaaS 共有サービスのアーキテクチャ

Amazon EKS クラスタ内で実行中のサービスは、通常は SaaS 環境の一部である一般的なサービスのいくつかのサンプリングを示します。新しいテナントのオンボードの調整のために、登録が使用されます。テナント管理では、システム内のすべてのテナントの状態と属性を管理し、このデータを Amazon DynamoDB テーブルに保存します。ユーザー管理により、テナントの追加、削除、有効化、無効化、更新の基本操作が実現します。管理される ID は Amazon Cognito に保存されます。システムにオンボードされた新しい各テナントのプロビジョニングに使用されるツールを表すために、AWS CodePipeline も含まれています。

このアーキテクチャは、SaaS 環境の基本的な要素のみを表します。テナントをこの環境に導入することの意味について考える必要があります。前述の隔離の考慮事項を考えると、Amazon EKS 環境は各テナントの独立した名前空間を作成し、堅牢なテナント隔離モデルを確実にするために、これらの名前空間を保護します。

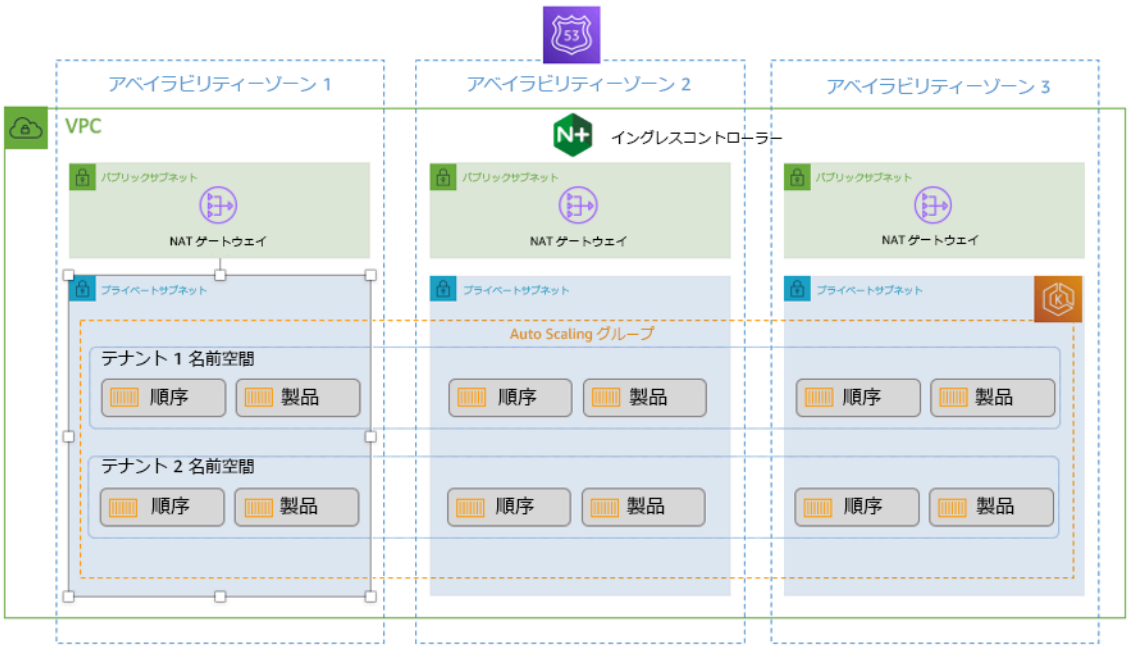

図 5: Amazon EKS でテナント環境をデプロイする

図 5 は、SaaS アーキテクチャ内のこれらの名前空間のビューを示しています。一見すると、このアーキテクチャは先ほどのベースラインの図と非常に似ています。主な違いとして、アプリケーションの一部であるサービスを、個別の名前空間にデプロイしたことです。この例では、固有の名前空間を持つ 2 つのテナントがあります。それぞれの中で、サンプルのサービス (注文と生産) をいくつかデプロイしました。

テナントの名前空間それぞれは、前述の登録サービスによってプロビジョニングされます。これは、AWS CodePipeline などの継続的配信サービスを使用して、名前空間を作成し、サービスをデプロイし、テナントのリソース (データベースなど) を作成し、ルーティングを構成する、パイプラインを開始します。ここで、イングレスコントローラーが使用されます。プロビジョニングされた各名前空間は、その名前空間内の各マイクロサービス向けの個別のイングレスリソースを作成します。これにより、テナントのトラフィックが適切なテナントの名前空間にルーティングされるようになります。

名前空間により、Amazon EKS クラスタ内のテナントのリソース間で、はっきりとした境界線を持たせることができるようになるものの、これらの名前空間はどちらかというとグループ化構造です。名前空間だけでは、テナントの負荷がクロステナントのアクセスから確実に保護されるようにできません。

Amazon EKS 環境の隔離のエクスペリエンスを強化するためには、特定の名前空間内で実行中の任意のテナントのアクセスを制限できる、別のセキュリティ構造を導入する必要があります。図 6

は、各テナントのエクスペリエンスを制御するために取ることができるアプローチの、概要を図示したものです。

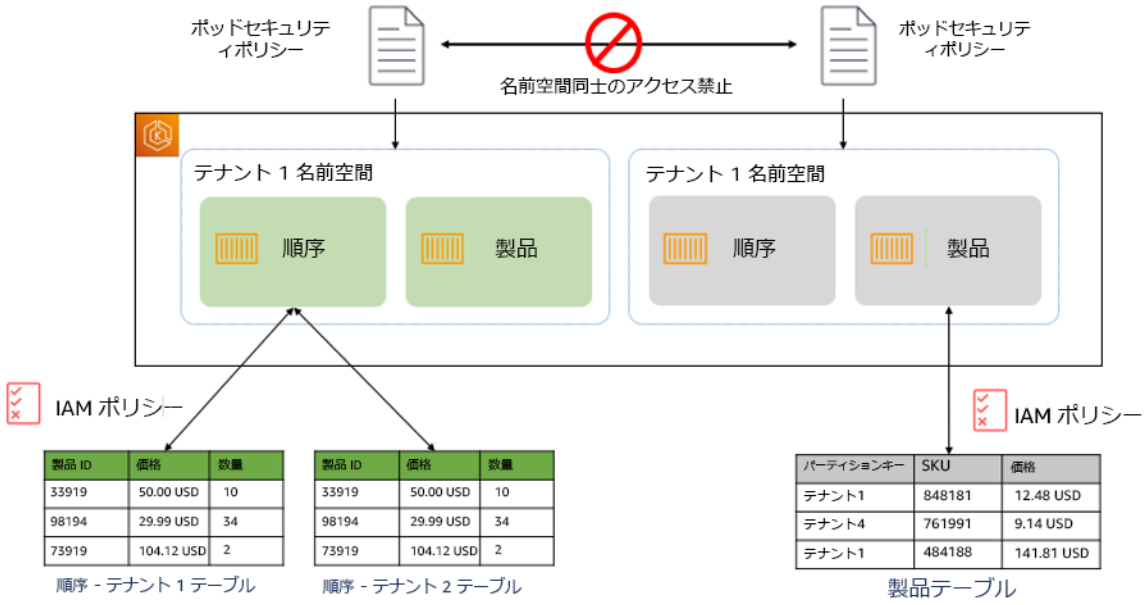

図 6: テナントのリソースの隔離

ここでは 2 つの特定の構造が導入されています。名前空間レベルでは、別個のポッドセキュリティポリシーを作成したことがわかります。これらはネイティブの Kubernetes ネットワークセキュリティポリシーであり、ポリシーにアタッチできます。この例では、これらのポリシーを使用して、テナントの名前空間同士のネットワークトラフィックを制限できます。これは、あるテナントが別のテナントのコンピューティングリソースにアクセスするのを防ぐための、少々荒い方法を表しています。

名前空間の保護に加えて、名前空間内で実行中のサービスによってアクセスされるリソースも制限されるようにする必要があります。この例では、隔離の 2 つの例があります。注文マイクロサービスはテナントあたりのテーブルモデル (サイロ) を使用し、特定のテナントへのアクセスを制限する IAM ポリシーがあります。生産マイクロサービスはプール化されたモデルを使用し、テナントデータが混在し、テナントアクセスを制限するために各アイテムに適用される IAM ポリシーを使用します。

フルスタック隔離

SaaS 組織はテナント間の共有インフラストラクチャが伴う規模の経済によって動かされています。同時に、環境内のテナントの一部またはすべてが専用の (サイロ化された) インフラストラクチャを必要とする、SaaS アーキテクチャが結果として生まれる可能性のある現実的な要因があります。コ

ンプライアンス、他のテナントによる影響、階層戦略、従来の技術などは、テナントに独自のインフラストラクチャを提供する結果となる多くの考慮事項の一部です。これは、サイロ化またはフルスタックの隔離 SaaS と呼ばれます。

このアプローチは、個人の顧客に 1 回限りの方法で企業の製品がインストールおよび管理される、マネージドサービスプロバイダー (MSP) とよく間違われます。このアプローチは有効なものの、SaaS の主義とは合致しません。SaaS モデルへの鍵は、単一の統合されたエクスペリエンスを通して、すべてのテナントが管理および運用される、価値システムを採用することです。これは、すべてのテナントがソフトウェアの同じバージョンを実行し、同時にデプロイされ、同じプロセスを通してオンボードされ、すべてのテナントに適用される単一の運用エクスペリエンスを通してすべて管理されることを意味します。

このアプローチでは、共有インフラストラクチャモデル (プール) のコスト効率性は実現されません。その分散された性質によっても、運用とデプロイがより複雑になります。それでも、正しく行えば、フルスタックモデルで、SaaS の考え方の中心である俊敏性、革新性、運用の効率性の目標を達成することができます。

図 7 は、フルスタック隔離 (サイロ) モデルのわかりやすい全体像です。このモデルの各テナント環境は明快です。各テナントに独立した VPC を提供したことがわかります。これらの VPC は、各テナントによって使用される完全に隔離されたスタックを持ちます。コンピューティングとストレージの構造は、AWS のサービスのあらゆる組み合わせで構成できます。各テナント環境が、同じインフラストラクチャ構成と、同じ製品のバージョンを持つようになっていることが鍵です。そのため、新しいテナントを追加することは、各テナントの同じインフラストラクチャのフットプリントを持つ別の VPC を追加するのと同じくらいシンプルです。

また、このモジュールは Amazon Route 53 を使用して受信トラフィックを適切な VPC にルーティングしていることがわかります。ルーティングは、サブドメインが各テナントに割り当てられているモデルに依存しています。これは SaaS 環境で非常に一般的なパターンです。このルーティングは、テナント JWT トークンの内容を調べ、テナントのコンテキストを決定し、挿入されたテナントのヘッダーを通してルーティングのルールをトリガーすることでも実現できます。ただし、このアプローチは、アカウントの制限に達し、テナントの ID のランタイム解決の遅延に対応するための調整が必要となる場合があります。それでも、これは一部の SaaS 環境では有効な選択肢です。

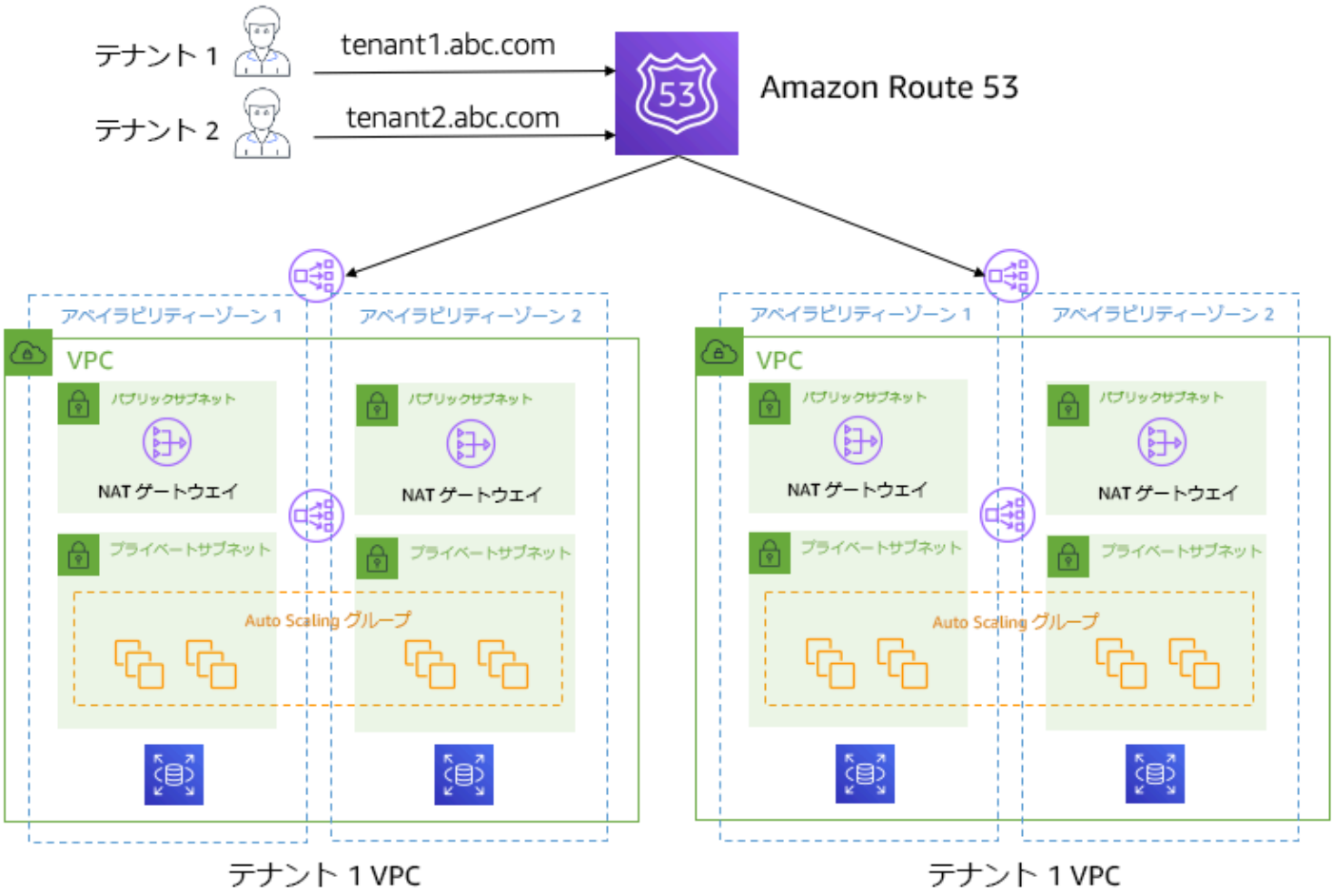

図 7: フルスタック (サイロ) 隔離

この例はテナントごとの VPCモデルを図示しているものの、フルスタックモデルの実装に使用できる他のアーキテクチャがあります。一部の組織では、各テナント環境が個別のプロビジョニングされたアカウントにデプロイされる、テナントごとのアカウントモデルの使用を選択する場合があります。選択するオプションは、サポートする必要があるテナント数と、目標とする全体的な管理エクスペリエンスによって変わります。サポートする必要のあるテナント数が、AWS アカウントで作成可能な VPC 数を超える場合があります。

統合されたオンボード、管理、運用

フルスタック隔離 (サイロ) モデルの実際の SaaS 要素は、このエクスペリエンスのオンボード、管理および運用を考えるときに現れます。SaaS の完全な価値提案を実現するために、このモデルはテナント環境すべてを管理および運用するために使用できる、一連のサービスを導入する必要があります。

図 8 は、これらの追加のサービスレイヤーを表しています。ここで紹介するのは、SaaS プロバイダーの管理エクスペリエンスのニーズの周囲に構築された、完全に独立した一連のサービスです。

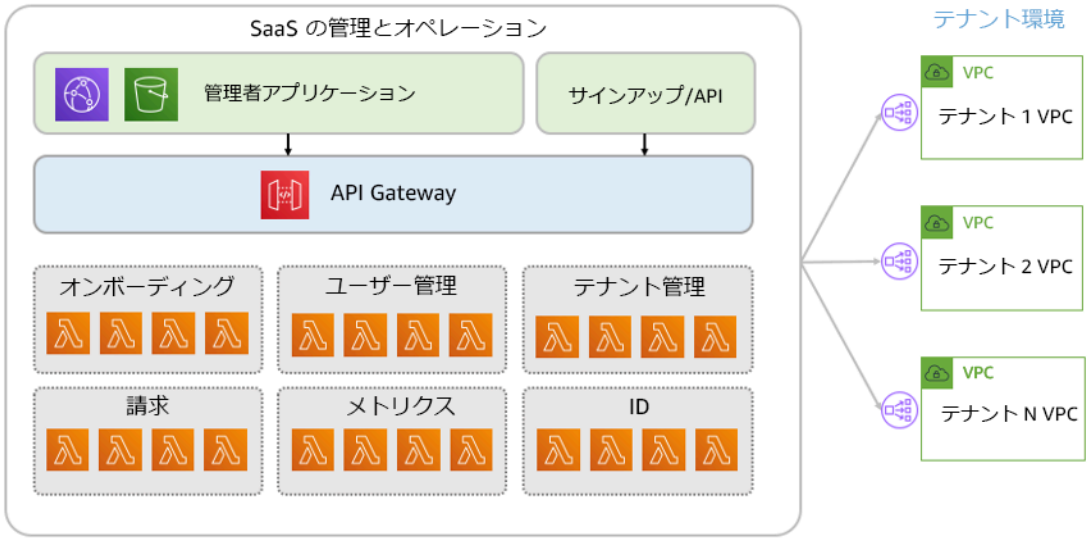

図 8: フルスタックの隔離環境の管理

テナント環境のスタックは、ドメイン、レガシー、およびその他の要件により形作られますが、管理エクスペリエンスのスタックは、テナントに適用されるものとはまったく異なる目標と使用パターンによって決まる場合があります。

この管理は、従来のサーバーレスの SaaS アプリケーションモデルと非常によく合致します。管理環境に 2 つのエントリポイントがあることがわかります。1 つは、Amazon CloudFront と Amazon S3 を使用する管理アプリケーションで、テナント環境を管理および構成するために SaaS 管理者によって使用されるアプリケーションのエクスペリエンスを提供するためのものです。また、サインアップと API のエクスペリエンスがハイライトされていることがわかります。これは、パブリック API リクエストを介してテナントのオンボードをトリガーできることを表します。この API は、SaaS プロバイダーのデベロッパーの API エクスペリエンスの一部にもなります。

最後に、オンボード、管理、運用のエクスペリエンスを統制するために使用される共有サービスである一連のマイクロサービスが示されています。管理、規模、コスト効率のモデルに基づいて、マイクロサービスのモデルとして Lambda を選択しました。これらのサービスは他の AWS コンピューティングサービスを使用して実装できます。

図の右側には、各テナント環境を表す個々の VPC があります。このテナントあたりのVPCモデルでは、管理プレーンがこれらのテナント環境のリソースを構成し、それにアクセスできるようにするために、有効なパスを開く必要があります。このために、AWS は AWS PrivateLink、VPC ピアリン

グ、Amazon EventBridge などの多くのオプションを提供しています。選択するオプションは、構築する管理エクスペリエンスの性質により変わります。

ハイブリッド SaaS デプロイ

SaaS 環境の顧客のニーズは、ビジネスごとに大幅に異なります。すべての顧客を共有インフラストラクチャモデル (プール) で稼動させるSaaS ソリューションを構築することには、大きなコストと運用のメリットがあるものの、顧客が他のテナントとインフラストラクチャリソースを共有したがらないこともあります。

1 度限りの顧客に独自の環境を用意する必要性は、SaaS プロバイダーにとって困難となります。SaaS プロバイダーはこのモデルをサポートするためのビジネス上のプレッシャーを感じているものの、この性質のバリエーションをサポートすることは、ビジネスの SaaS の全体的な目標をむしばむことも知っています。新しい 1 度限りの各顧客によって、運用のオーバーヘッドと複雑性が高まります。新しい各顧客がこのモデルに追加されるにつれて、共有インフラストラクチャモデルから得られる革新、俊敏性、コスト効率のメリットから遠ざかってしまうことがすぐに実感できます。

ただし、SaaS のビジョンを完全に損なうことなく、このモデルを利用できる、アーキテクチャおよび運用上の戦略があります。図 9 は、この課題にどのように対応すればよいかの概念ビューを示しています。

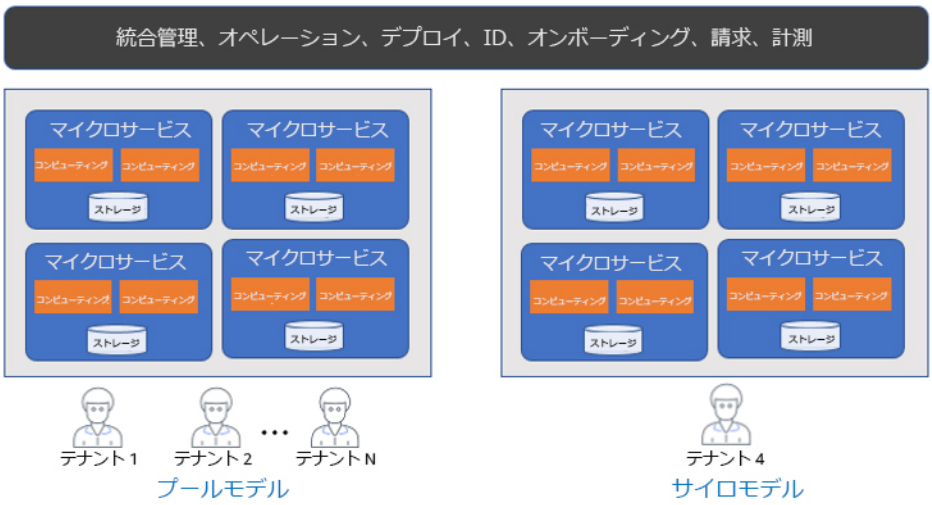

図 9: ハイブリッドデプロイモデル

この図では、SaaS 環境の 2 つの独立したデプロイがあることがわかります。左側は、共有インフラストラクチャを使用してプール化されたモデルでデプロイされています。テナントの大多数はこの環境で実行しています。一方、右側では、1 つのテナント (テナント 4) をホストする、マルチテナント環境の独立したコピーをデプロイしています。

この戦略の鍵は、左右両方の環境が、SaaS アプリケーションの同じバージョンを実行している点です。すべての新しい変更や更新は、両方の環境に同時に適用されます。さらに重要なのは、これら両方の環境にまたがる、単一の統合された管理エクスペリエンスが用意されていることです。オンボード、ID、計測、課金が両方の環境で共有され、共通のものとなっています。

この共有エクスペリエンスによって、SaaS 組織はこれらの環境を一元化して管理および運用できます。これらの 1 度限りのテナントをこのモデルに当てはめることで、SaaS 環境の全体的な俊敏性と運用効率性への影響を最低限に抑えることができます。このモデルはコスト効率性に影響を及ぼし、運用の複雑性は高まります。しかし、多くの SaaS 企業にとって妥当な妥協案となります。

マルチテナントのマイクロサービス

マルチテナント環境で実行中のマイクロサービスは、さらなる考慮事項に対処する必要があります。これらのマイクロサービスは、各サービス内でテナントのコンテキストを参照し適用できるようにする必要があります。同時に、デベロッパーがテナントへの認識をコードに導入する必要性の度合いを制限することも目標です。

この目標を達成するために、SaaS マイクロサービスは、コンピューティングモデルに関わらず、テナント固有の処理をコードにプッシュできる、ライブラリ、モジュール共有構造を導入する必要があります。この構造は、テナントのコンテキストを解決および適用するために必要なポリシーとメカニズムを隠します。

図 10 は、マイクロサービスのマルチテナントの開発をどのように効率化できるかを示しています。ここにおける鍵となるアイディアは、テナントのコンテキストに依存するすべてを取り、ライブラリを使用してマルチテナントのポリシーを適用するということです。

この例は、SaaS マイクロサービスがテナントのコンテキストにアクセスする必要があるシナリオをいくつか示しています。フローは、製品のリストを取得するリクエストを持つ、マイクロサービスへの呼び出しから始まります。サービスのコードのどこかで、サービスがメッセージを記録します。マイクロサービスのデベロッパーは単に、ロギングラッパーにメッセージを記録します。このラッパーはトークンマネージャーからテナントのコンテキストを取得し、そのコンテキストを挿入し、この例では Amazon S3 に発行されます。ログポリシーとテナントデータがどのようにログに到達するかは、ロギングヘルパーに含めることにしたいずれかのポリシーによって管理されます。

このテーマが、ここでのエクスペリエンスの残りの部分で継続されます。getProducts() への呼び出しは、まずトークンマネージャーからテナント ID を取得します。次に、このコンテキストを使用して、DynamoDB から製品データを取得するためにこの認証情報を使用する前に、テナント範囲の認証情報を隔離マネージャーから取得します。

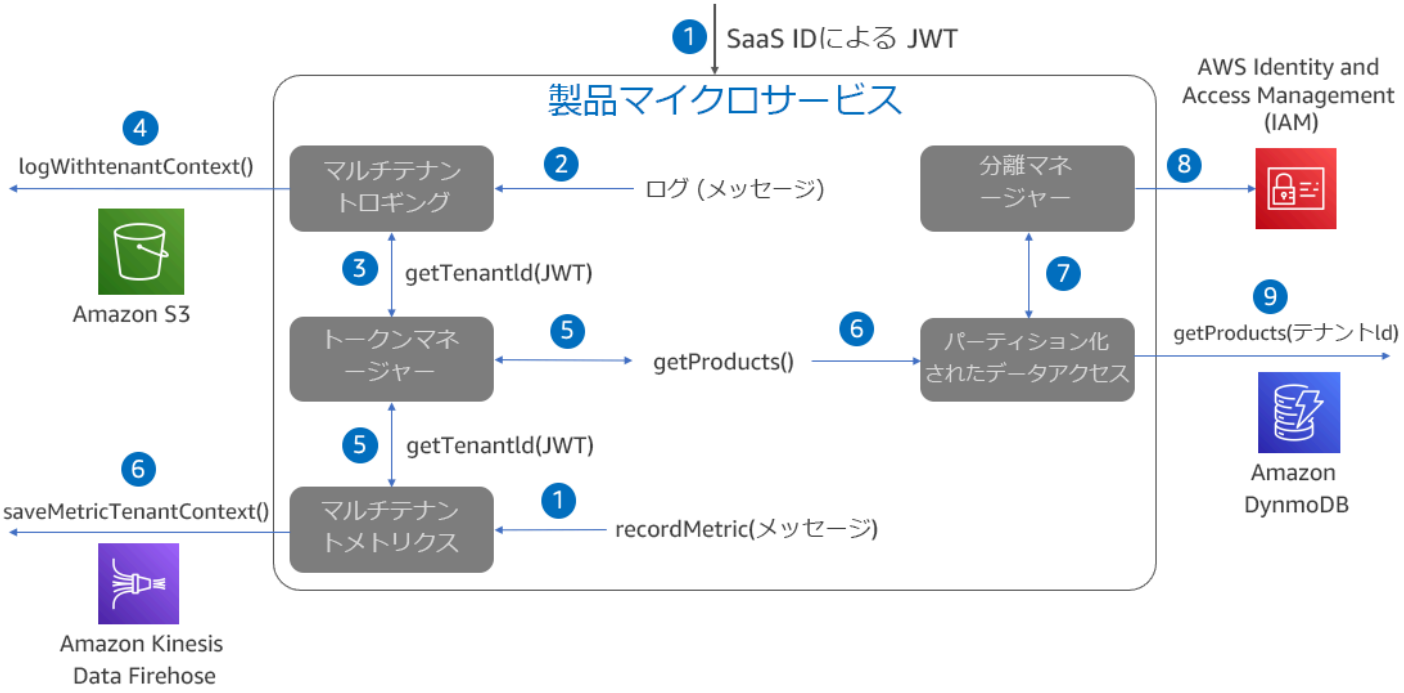

図 10: マルチテナントのマイクロサービスの開発

最後に、テナント ID を解決し、テナントのコンテキストを指標メッセージに挿入するのと同じ仕組みを使用して、サービスは指標 (おそらく実行時間) を記録します。

ここで説明した実践は、マイクロサービスに大規模な構造を導入することを意図したものではありません。このテナンシーの抽象化は、共通の概念を共有コードにプッシュするという一般的な設計プラクティスに従っています。この図で表されたすべての概念が、単一マイクロサービスのコンテキスト内で実行される点は重要です。これらの各ブロックは単純に、マイクロサービスによって再利用されるコードを表します。これらの概念を個別のサービスに分割すると、遅延と複雑性が高まり、通常は理想的ではありません。

テナントのインサイト

企業がマルチテナントのモデルに移行するにつれ、テナントがどのようにアプリケーションを使用するかのインサイトを取得し表面化させることは、ますます難しくなります。これは、テナントによって共有される SaaS 環境のリソースについて、特に当てはまります。

同時に、すべてのテナントが潜在的に 1 つの共有環境で実行されている状態で、テナントのアクティビティと消費をはっきりと把握できることは、堅牢な SaaS サービスの構築に不可欠です。製品のテナントが使用している機能を知り、テナントが環境のアーキテクチャをどのようにプッシュ

しているかを知り、テナントの異なる階層がどのようにスケーリングしているかを把握することは、SaaS 企業が健全な SaaS ビジネスを運用できるようにするために必要なインサイトの一部です。

SaaS 指標の範囲は意図的に広くなっています。これは、指標が、SaaS 組織内の異なる役割によって複数のコンテキストで消費されるためです。つまり、インフラストラクチャの健全性を超えて考える必要があります。SaaS 指標には、ビジネスの俊敏性、機能の消費、マイクロサービスのアクティビティ、スケーリングの傾向などがあります。テナントと階層の使用傾向を取得し、アプリケーションと消費アクティビティと相互に関連させます。

このデータを消費するさまざまな役割の、さまざまなダッシュボードがあることを想像してください。例えば、製品マネージャーが機能指向指標に対するインサイトを必要としています。一方、運用スタッフはこのデータを使用して個々のテナントと階層の健全性と消費傾向を評価します。

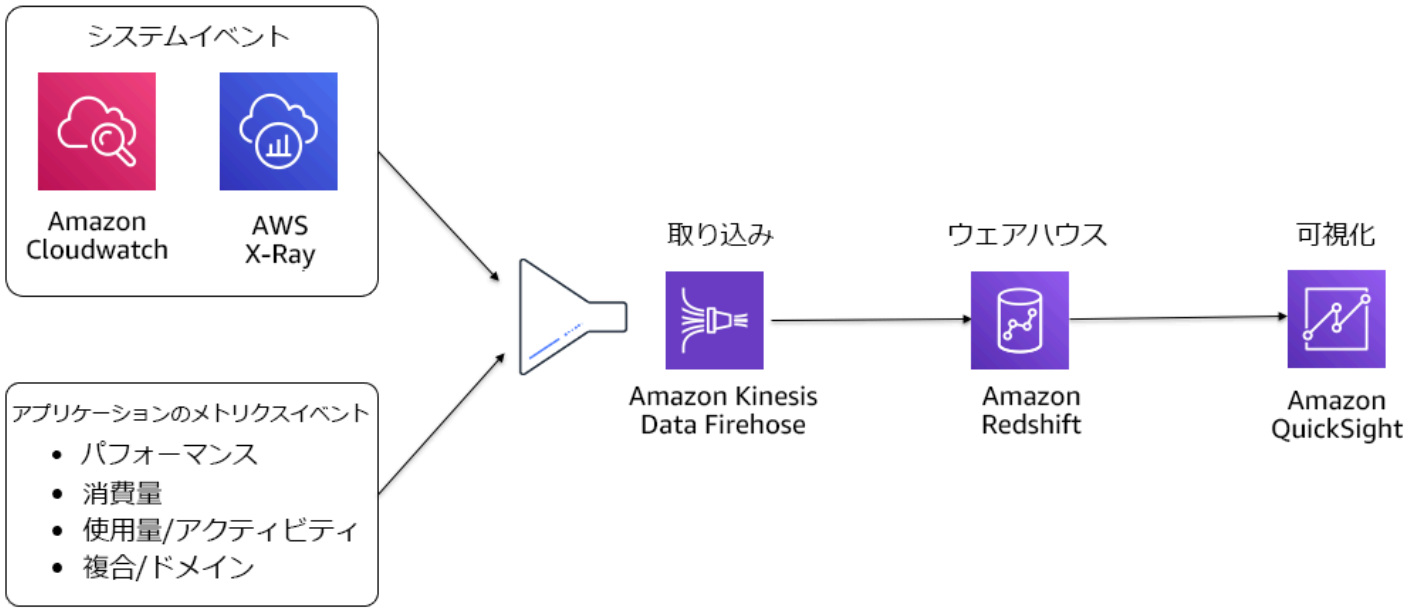

図 11: SaaS 指標の取り込みと視覚化

このデータを取得し表面化させるアーキテクチャは、比較的単純です。これらの SaaS 指標を伝え、発行し、取り込み、集約し、視覚化するために必要なメカニズムすべてが含まれています。図 11 は、SaaS 指標環境を AWS で構築するために使用できる、アーキテクチャ図を示しています。

取得する指標データのソースは多様となります。図が示すように、ソリューションはネイティブの AWS ソース (例えば CloudWatch) から指標の一部を取得する必要がある場合があります。ただし、このデータの大部分は、SaaS アプリケーションに実装したコードからのものです。ドメインのデータのビジネスおよび運用上の価値を最大限にできるインサイトを的確に取得する指標の発行に投資します。

ソースからのデータは、通常はアプリケーションのさまざまな種類の消費 (カウント、期間、機能など) を表すことができる、一般的なスキーマとともに表されます。この例では、このデータは Amazon Data Firehose を使用して取り込まれ、データを Amazon Redshift に発行します。最後に、Amazon QuickSight が、テナントと階層の傾向を確認するために組織全体で使用される、さまざまなダッシュボードの構築に使用されます。

Well-Architected フレームワークの柱

このセクションでは、各柱について説明し、マルチテナント SaaS アプリケーションのソリューションを設計する際に関連する定義、ベストプラクティス、質問、考慮事項、主要な AWS のサービスについて説明します。

説明を簡潔にするため、ここでは SaaS ワークロードに特化した質問のみを扱います。この文書に含まれていない質問は、アーキテクチャの設計時に考慮する必要があります。詳細は AWS Well-Architected フレームワークホワイトペーパーをご覧ください。

トピック

運用上の優秀性の柱

運用上の優秀性の柱には、ビジネス価値を提供し、サポートプロセスと手順を継続的に改善していくために、システムを実行してモニタリングする能力が含まれています。

運用上の優秀性の柱では、設計原則、ベストプラクティス、質問の概要について説明します。実装に関する規範的なガイダンスについては、運用上の優秀性の柱のホワイトペーパーを参照してください。

設計の原則

クラウドでは、運用上の優秀性をもたらす多様な原則があります。

定義

クラウド内での運用上の優秀性について 3 つの領域のベストプラクティスがあります。

- 準備
- 運用

- 進化

運用チームは、ビジネスのニーズと顧客のニーズを理解し、ビジネスの成果を達成できるように効果的かつ効率的に支援する必要があります。運用では、運用上のイベントに応答するための手順を作成および使用し、その効果を検証してビジネスのニーズに対応します。運用では、目的とするビジネスの成果の達成度を測定するために使用するメトリクスを収集します。ビジネスの状況、ビジネス上の優先事項、顧客のニーズなど、すべては変化し続けます。そのため、時間の経過に伴う変化に対応した進化を促進する運用を設計し、運用のパフォーマンスから学んだ教訓を取り入れることが重要です。

ベストプラクティス

トピック

準備

SaaS アプリケーションに固有の運用方法はありません。

運用

SaaS OPS 1: マルチテナント環境の運用状態をどのように効果的にモニタリングおよび管理していますか?

組織の顧客すべてが共有インフラストラクチャモデルにデプロイされる可能性がある共有マルチテナント環境では、より詳細な正常性の運用ビューを必要とすることが SaaS ビジネスの成功と成長に不可欠です。停止や正常性の問題はシステムの全テナントに連鎖し、全顧客の SaaS サービスがダウンする可能性があります。つまり、SaaS 組織は、運用チームが SaaS 環境の絶えず変化するワークロードを効果的に分析して対応できるよう、運用経験を積むことに重点を置く必要があります。

堅牢なマルチテナントの運用経験を積むために、SaaS 企業は SaaS 環境で必要とされる正常性とアクティビティのきめ細かなビューを導入するツールを作成またはカスタマイズする必要があります。これは、多くの場合、既存のツールとカスタムソリューションの組み合わせを使用し、運用経験で最

優先される概念としてテナンシー層とテナント層に対応するエクスペリエンスを作成することを意味します。

例えば、SaaS オペレーションダッシュボードにはテナントおよびテナント層のレンズを通して表示される正常性のビューを含める必要があります。正常性のグローバルビューの表示もこのエクスペリエンスの一部ですが、SaaS 運用チームはテナントのアクティビティの正常性も確認できる必要があります。図 12 に示す概念ビューは、SaaS プロバイダーが最優先構造としてテナントアクティビティを特定する方法の 1 つです。

この図の簡易化されたビューでは、マルチテナント環境の価値を高める運用ビューのサンプルがいくつか示されています。ページの左上には、最もアクティブなテナントの正常性が表示されます。表示されているカラーインジケータにより、正常性のグローバルビューを確認する際に、特定されていない問題が発生している可能性のあるテナントに注意を向けることができます。これにより、テナントに対して完全には明示されない可能性のある問題に対して、オペレーションで事前に対応できるようになります。

図 12: テナント対応オペレーションビュー

このビューの右上に、テナントのリソース消費に関するデータが表示されます。ここでは、テナントごとに AWS リソースの消費を表示し、特定のテナントがどのように特定のサービスを実行しているのかを確認することが目的です。下部には、テナントがアプリケーションのさまざまなマイクロサービスをどのように消費しているのかを表す消費のビューが表示されます。このデータにより、オペレーションで、システムのさまざまなサービスをテナントがどのように消費しているのかを表示し、特定のテナントまたは層が特定のマイクロサービスが飽和する可能性を判断できます。

テナントアクティビティのこの汎用ビューを提供するだけでなく、運用経験には、個々のテナントと層の運用データをドリルダウンする機能も含める必要があります。特定のテナントまたはテナント層で問題が発生しているサポートシナリオを想定してみます。この場合、運用データにアクセスして、その特定のテナントまたは層のレンズを通してデータを確認できる必要があります。これは、マルチテナント設定で問題をトラブルシューティングして診断するために不可欠です。

これらの運用ビューを作成するには、テナント別またはテナント層別に洞察を分析できる運用ビューの作成に必要なテナントと階層化コンテキストを含む運用データにアクセスする必要があります。これを行うために、正常性イベントとアクティビティイベントの記録に使用されるさまざまなメカニズムにテナントコンテキストを挿入する方法と場所を、SaaS アーキテクトが慎重に検討する必要があります。例えば、テナントコンテキスト (テナント識別子や層など) がシステムのログデータに必ず挿入されるようにするメカニズムをログに含める必要があります。

ただし、作成するビューは、アプリケーションの設計とアーキテクチャの性質によって異なります。一般に、チームは、運用チームがテナントの傾向を効果的にモニタリングし、最初の段階から計装を環境に組み込める運用ビューについて検討する必要があります。これらの概念を開発プロセスの後半でアプリケーションに追加するのは難易度が高く、開発経験と運用経験の両方が損なわれる可能性があります。

SaaS OPS 2: 新しいテナントはどのようにシステムにオンボードされますか?

SaaS ソリューションは俊敏性とイノベーションの最大化に重点を置いています。テナントのオンボーディングはこの俊敏性の事例において重要な役割を担います。スムーズで再現性のあるオンボーディングプロセスを促進するアーキテクチャを作成することにより、SaaS 組織は新しいテナントを SaaS 環境に導入する機能を合理化、最適化、拡張できます。これにより、SaaS 企業は急速な成長に対応して、価値実現までの全体的な時間を短縮するエクスペリエンスを顧客に提供できます。

自動化されたオンボーディングは B2C 環境と B2B SaaS 環境の両方に適用されることに注意してください。一部の SaaS 製品にはセルフサービスのオンボーディング手続きが含まれていない場合がありますが、これによってスムーズなオンボーディング手続きの必要性が減少することはありません。オンボーディングが社内で実行されるプロセスでも、テナント作成のすべての要素を自動化する必要があります。摩擦を低減する必要性は、SaaS ベストプラクティスに沿ったソリューション作成の基礎となります。

一般に、SaaS アプリケーションのオンボーディングプロセスには、新しいテナントの導入に必要なリソースすべてを構成してプロビジョニングすることのできる一連の共有サービスの調整が必要です。図 13 は、一連のマイクロサービスを介してこのオンボーディングを実装する方法の概要です。

この図に示されているオンボーディング手続きのフローは、テナントを導入して SaaS システムの使用を開始するのに必要なすべてのステップが網羅されています。このプロセスの前に、テナントは登録サービスに問い合わせて、新規テナントの作成を依頼します。それ以降、登録サービスはオンボーディングプロセスの中間に位置し、新しいテナント環境の作成に必要なすべてのサービスを調整します。

このプロセスの次のステップで、新規ユーザーを作成します。この新規ユーザーは、この新規テナントの管理者を表します。このプロセスをサポートするために、ユーザー管理サービスが追加されました。このサービスではユーザーに関するデータが保存されませんが、ID プロバイダー (この場合、Amazon Cognito) でユーザーが作成されます。また、このテナントの分離要件に対応するために必要な任意の IAM ポリシーが作成されます。

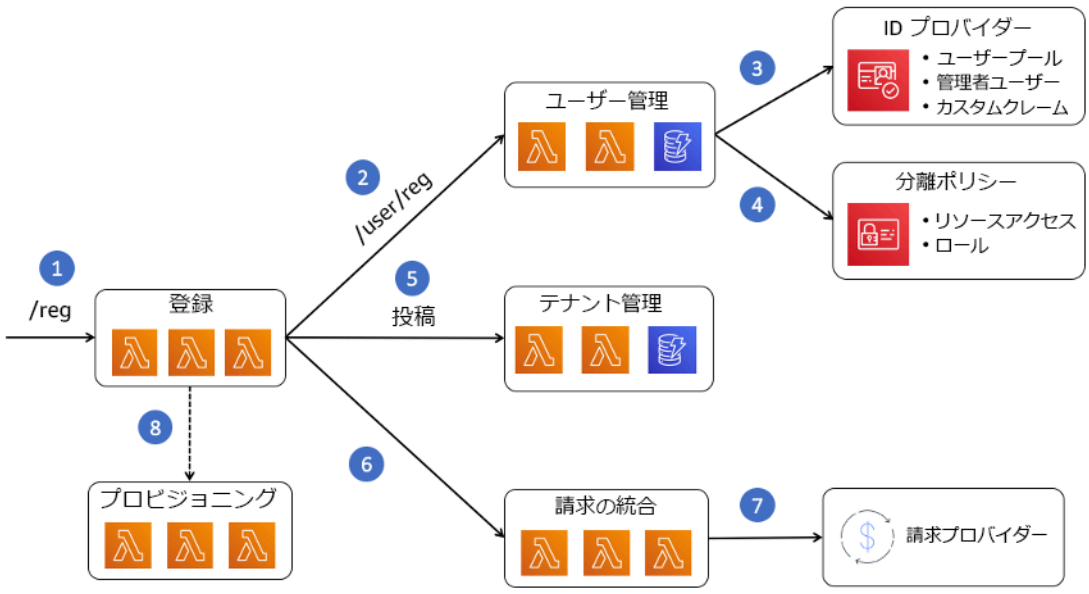

図 13: スムーズなテナントオンボーディング

このモデルでは、テナントごとに個別の Amazon Cognito ユーザープールを使用する戦略も利用しました。これにより、システムの各テナントを識別する手続き (パスワードポリシー、多要素認証など) をカスタマイズできます。このアプローチを選択する場合、各ユーザー ID を対応するユーザープールにマッピングする必要があります。このマッピングはユーザー管理サービスで管理されます。

ユーザーが作成されると、プロセスでテナントが作成されます。個別のサービスとしてのテナント隔離は SaaS 環境に不可欠です。そのテナントに関連付けられているユーザーから完全に分離されたテナントの状態と属性を一元的に管理できます。例えば、テナントの層やステータスはテナント管理サービスによって制御されます。

オンボーディングの一環として、SaaS システムは請求システムにフットプリントを確立する必要があります。この例では、新しいテナントを作成していることについて、請求統合サービスに知らせていることがわかります。これにより、このサービスでは、請求プロバイダーで新しいアカウントを作成する責任が想定されます。これには、テナントのプランまたは層の構成が含まれます (無料、ブロンズ、プラチナなど)。

図に示すプロセスの最後のステップは、テナントインフラストラクチャのプロビジョニングに関連します。一部の SaaS アーキテクチャには専用のテナントリソースが含まれます。これらのシナリオでは、テナントをアクティブ化する前に、オンボーディングプロセスでこれらの新しいリソースをプロビジョニングする必要があります。

ここに示すフローは環境の特性によって異なる可能性がありますが、ここに示す概念はオンボーディングプロセスで共通しています。テナントリソースの作成、構成、プロビジョニングの自動化は、機能豊富でスケーラブルなマルチテナントエクスペリエンスを作成する基礎となります。

SaaS OPS 3: テナント固有のカスタマイズの必要性にどのように対応しますか?

SaaS アーキテクトが直面する重要課題の 1 つは、すべてのテナントが同じバージョンの製品を実行していることを確認する必要があることです。これは特に、SaaS に移行しており、1 回限りの製品バージョンを通じた独自の顧客要件への対応に慣れている企業に当てはまります。

これは効果的に思われますが、顧客管理、運用、サポート、デプロイの統合されたエクスペリエンスから離れた動作が行われると、SaaS 組織の全体的な俊敏性が直接損なわれます。新しいカスタム環境が導入されるたびに、SaaS 組織はゆっくりと従来のソフトウェアモデルへの移行を進めます。最終的には、これが SaaS ビジネスモデルの中核となるコスト、運用効率、一般的なイノベーションの目標を損なうことになります。

次の課題は、製品のフォークバージョンを作成せずに、不定期に発生する 1 回限りの必要性を満たすことができる戦略を見つけることです。ここでは、製品全体に追加されるカスタマイズオプションの導入によって妥協点を調整できます。そのため、個別のバージョンをスピンオフするのではなく、すべての顧客が利用できるようにこれらの新機能を追加する方法を見つける時間と労力をさらに費やす必要があります。次に、テナント設定でこれらの新機能を有効にするテナントを決定できます。

この問題の一般的なアプローチでは機能フラグを使用します。機能フラグは、実行時にさまざまな機能を有効/無効にするフラグを用いた共通のコードベースで複数の実行パスを使用する方法として、アプリケーションデベロッパーによって一般的に使用されます。この手法は、一般的な開発戦略としてよく使用され、SaaS 環境にカスタマイズを導入する効果的な方法です。各機能フラグは、テナント設定オプションに関連付けられます。この設定は実行時に評価され、各テナントで有効にする機能に影響します。

図 14 に示す概念ビューは、これらのフラグがどのように適用されるのかを示しています。個々のテナントに対して一連のフラグがオンまたはオフになり、テナントで有効にする機能が決定されます。これらの設定オプションは、テナントが SaaS サービスの新機能にサインアップすると変更されます。これらのフラグは (個々のテナントではなく) 層に関連付けることもできます。

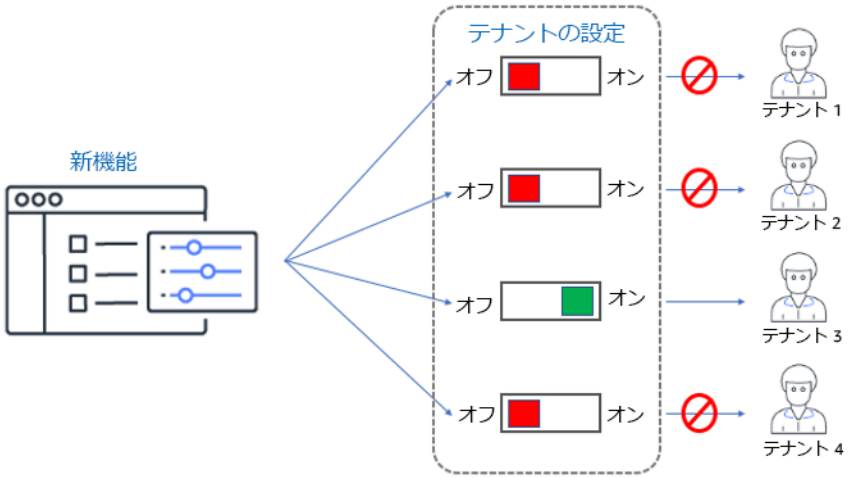

図 14: テナントの管理には機能フラグが必要

機能フラグはこれを達成する 1 つの方法にすぎません。ここで重要なのは、1 つのテナントに固有の機能が必要な場合でも、その機能をコアプラットフォームのカスタマイズとして導入する必要がある点です。そのカスタマイズの適用方法はシステムのスタックと設計によって異なる場合があります。目的は、製品の 1 つのバージョンを確実にデプロイして管理できるようにすることです。

これは効果的な構造ですが、適用する場合は注意する必要があります。機能フラグを導入すると、最終的にすべてのテナントが独自のエクスペリエンスを持つ複雑なオプションの迷路を作成することになり、すぐにシステムで管理できなくなります。これらのフラグの導入方法とタイミングは慎重に選択してください。経験則として、このフラグを組織が新規顧客に 1 回限りのカスタマイズを提供可能にする販売ツールと見なすべきではありません。

進化

SaaS OPS 4: 個々のテナントの使用状況と消費傾向の分析に使用できるメトリクスデータをどのように取得し、特定していますか?

SaaS 運用経験を継続的に進化させるには、マルチテナントの SaaS 環境分析に使用できる運用データの豊富なコレクションにアクセスできる必要があります。多くの場合、これは、環境を使用しているテナントのアクティビティと消費パターンを正確に取得できるアプリケーションから、より充実したテナントメトリクスの収集を計装して公開することを意味します。

SaaS メトリクスは、インフラストラクチャの消費 (CPU、メモリなど) の基本を超えて、環境内のマルチテナント負荷の運用を理解するうえで基本となる特定の運用上の使用状況パターンを識別します。このデータの分析により、SaaS 組織はシステム全体で発生している傾向を評価し、SaaS サービスの信頼性、スケーラビリティ、コスト効率、全体的な俊敏性を継続的に改善するポリシーまたは基盤となるアーキテクチャへの変更を導入する機会を特定できます。

メトリクスデータの取得と特定には、2 つの異なる要素があります。まず、アプリケーションで、有用な運用上の洞察を SaaS 環境に提供できるメトリクスデータを公開する必要があります。これらのメトリクスを取得して公開するアプリケーションの要点を特定する必要があります。

もう 1 つのパズルのピースは、このデータの取り込み、集計、特定です。ここでは、AWS またはいずれかの AWS パートナーネットワーク (APN) パートナーから選択できるさまざまなツールがあります。最終的には、この取り込みコンポーネントがデータウェアハウスとビジネスインテリジェンスに近いものになります。このドキュメントで前述した SaaS アーキテクチャのシナリオには、テナントの洞察シナリオが含まれます。このシナリオでは、メトリクスデータを取り込むアーキテクチャの概要を説明します。このアーキテクチャは、このニーズに対応するよう適用できるパターンの 1 つを表しています。

ここでは、これらのテナントメトリクスの運用ビューに焦点を当てていますが、これらのメトリクスは SaaS ビジネス全体にわたり複数のコンテキストで使用されることが想像できます。運用上の要件は、組織がテナントのアクティビティと消費傾向を積極的に収集および分析して SaaS システムに進

化をもたらす機会を特定することをアーキテクチャで確実にすることです。これは、絶えず変化するテナントとテナントワークロードの組み合わせを評価するために使用できる基本的なツールの構築という、幅広いテーマに適しています。

リソース

運用上の優秀性に関する AWS のベストプラクティスの詳細については、以下のリソースを参照してください。

ドキュメントとブログ

- GPSTEC309-SaaS Monitoring Creating a Unified View of Multi-tenant Health featuring New Relic Slides
- SaaS Quick Start Highlights Identity and Isolation with Amazon Cognito
- Feature Toggles (aka Feature Flags)

動画

- AWS re:Invent 2016: The Secret to SaaS (Hint: It's Identity) (GPSSI404)
- AWS re:Invent 2017: GPS: SaaS Monitoring - Creating a Unified View of Multi-tenant Health featuring New Relic (GPSTEC309)

セキュリティの柱

セキュリティの柱には、リスクの評価とその軽減戦略を通してビジネス価値を提供しながら、情報、システム、資産を保護する能力が含まれます。

設計の原則

クラウドには、システムセキュリティの強化に役立つ多くの原則があります。

定義

クラウド内でのセキュリティには、5 つのベストプラクティス領域があります。

- アイデンティティ管理とアクセス管理
- 発見的統制

- インフラストラクチャ保護
- データ保護
- インシデント対応

マルチテナンシーにより、SaaS アーキテクチャにその他の検討事項というレイヤーが追加されます。SaaS では、特定のテナントのコンテキストで共有環境にアクセスしているユーザーがいます。このコンテキストは、アプリケーションのアーキテクチャのレイヤーすべてにわたり取得され、伝達される必要があり、環境のフットプリント全体を保護するうえで基本的な役割を果たします。

セキュリティの観点から、テナンシーが環境にどのように導入され、テナントリソースを保護するためにどのように使用されるのかを確認する必要があります。全体で、各テナントがその他すべてのテナントリソースにアクセスできないよう、慎重に制限されたエクスペリエンスがあることを確認する必要があります。

ベストプラクティス

トピック

アイデンティティ管理とアクセス管理

SaaS SEC 1: テナントコンテキストをどのようにユーザーに関連付け、そのコンテキストを SaaS アーキテクチャ内で適用していますか?

多くの場合、マルチテナントアーキテクチャへの移行はアイデンティティから開始されます。アプリケーションにアクセスする各ユーザーはテナントに接続する必要があります。このユーザー ID からテナントへのバインディングは、非公式に SaaS アイデンティティと呼ばれます。SaaS アイデンティティのキー属性は、テナントコンテキストを最優先される構造に引き上げ、SaaS アプリケーションの認証と承認モデル全体に直接接続します。

このアプローチにより、テナントコンテキストはユーザー ID の伝達とアクセスに使用されるアーキテクチャ構造と同じ構造を使用して、アーキテクチャのすべてのレイヤーを経由できます。例えば、アプリケーションに 100 のマイクロサービスがある場合、これらの各サービスが別のサービスにラウンドトリップしないで、テナントコンテキストを取得して適用できる必要があります。別のサービスを介してこのコンテキストを管理するとレイテンシーが増加し、多くの場合、アーキテクチャにボトルネックが生じます。

テナントコンテキストは、複数のパターンを使用してアイデンティティに挿入できます。アプリケーションに選択した ID プロバイダーとテクノロジーによって、このコンテキストをエクスペリエンスに組み込むために最終的に適用するアプローチと戦略が直接形成されます。このツールは変わる場合もありますが、基本的に必要なのは、ユーザーがアプリケーションに入る際にテナンシーが挿入される環境の全体的な認証エクスペリエンスにテナンシーを導入することです。

図 15 は、AWS のサービスを使用してこれを実現する一般的な方法の例を示しています。この例には SaaS 環境へのテナントコンテキストの挿入に使用される一般的なコンポーネントとテクノロジーが含まれています。これについては図の左側で示します。ここでは、テナントがサインアップフォームに入力し、アプリケーションの登録サービスに対する呼び出しをトリガーします。

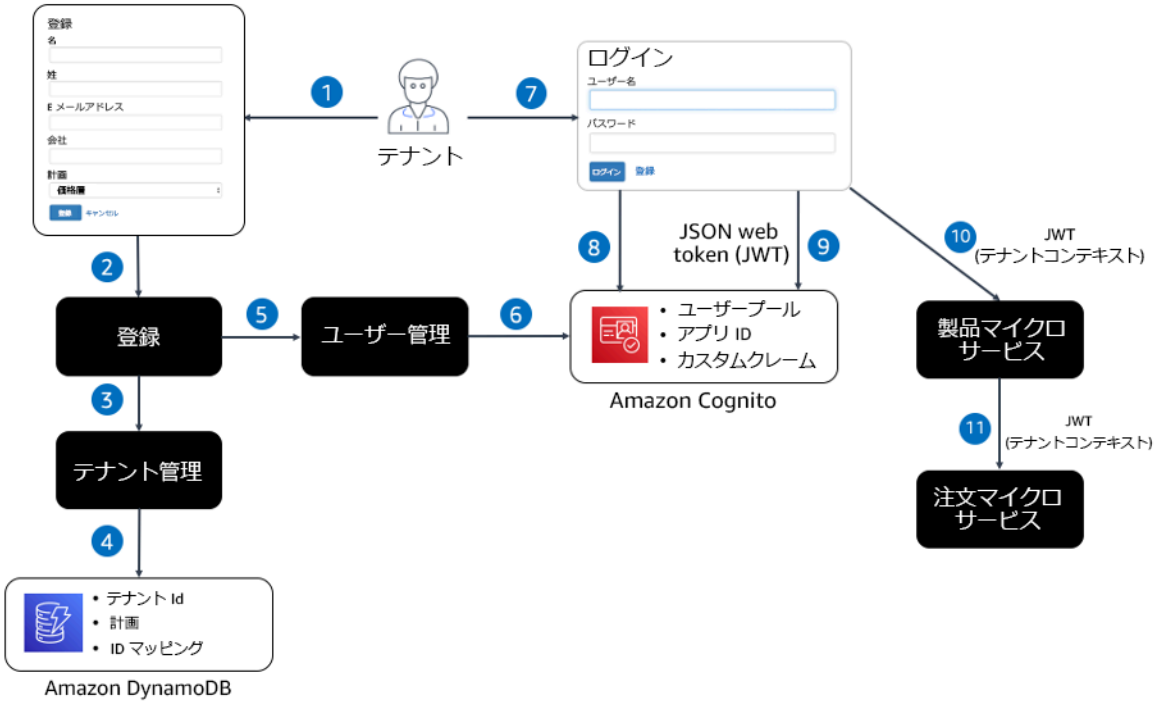

図 15: テナントコンテンツの挿入

この登録サービスはテナントを作成してから Amazon Cognito でユーザーを作成します。このプロセスの一環として、ユーザーのテナントとの関係に関する情報が保持されるカスタムクレームをユー

ザーの属性に導入します。これらのカスタムクレームは、ユーザー ID 署名の一部となり、最優先される構造としてテナントに直接接続されます。

オンボーディングが完了した後、ユーザーのログイン時にこれらのユーザー属性とテナント属性が適用される方法を確認できます (図の右側)。ここでは、ユーザーが Amazon Cognito に対して認証し、そのプロセスの一部として、オンボーディングプロセス中に作成されたカスタムクレームを含む JSON Web トークン (JWT) を返すことがわかります。

これで、テナントコンテキストをマルチテナントアプリケーションサービスとの対話に挿入する際に必要なすべての情報を含むトークンを使用できるようになります。この例では、Product マイクロサービスに対する各リクエストのヘッダーでベアラートークンとして渡される JWT を示します。このサービスは、別のサービスを呼び出さなくても、このトークンからコンテキストを取得して適用できるようになりました。

最後に、この Product マイクロサービスは Order マイクロサービスを呼び出し、リクエストのヘッダーで JWT を渡します。これは、ルックアップやレイテンシーを増やさずに、すべてのマイクロサービスの呼び出しをテナントコンテキストがどのように経由するのかを示しています。

この例では、Amazon Cognito を使用してユーザー ID とテナント ID を接続しています。ただし、これと同じモデルは他の ID プロバイダーや代替認証スキームでも実装できます。ここで重要なのは、テナントとユーザーを接続する表現を生成できる認証エクスペリエンスを構築していることです。この表現は、ソリューションのすべてのレイヤーで使用できる必要があります。

発見的統制

SaaS アプリケーションに固有のセキュリティプラクティスというものはありません。

インフラストラクチャ保護

SaaS SEC 2: テナントリソースがクロステナントアクセスから保護されることをどのように確認していますか?

テナントの分離は、すべての SaaS プロバイダーが取り組む必要のある基本的なトピックの 1 つです。独立系ソフトウェアベンダー (ISV) は、SaaS にシフトし、共有インフラストラクチャモデルを採用してコスト効率および運用効率を達成する際に、マルチテナント環境において、テナントが他のテナントのリソースにアクセスできないようにする方法を決定するという課題にも取り組みます。SaaS ビジネスにとって、どのような形でもテナントの境界線を超えることは、重大かつ回復不能に至る可能性のあるイベントに相当します。

テナントの分離は SaaS プロバイダーにとって不可欠と考えられますが、この分離を達成するための戦略およびアプローチは一様ではありません。SaaS 環境でテナントの分離を実現する方法には、非常に多くの要素が影響を与えます。ドメイン、コンプライアンス、デプロイモデル、AWS サービスの選択など、すべてがテナント分離の方法にそれぞれの検討事項として関わってきます。

トピック

- 分離の考え方
- 基本的な分離の概念

分離の考え方

テナントリソースを保護して分離する重要性と価値について、概念レベルでは多くの SaaS プロバイダーの意見が一致します。しかし、分離戦略の実装を詳しく掘り下げていくと、多くの場合、SaaS ISV ごとに十分な分離について独自の定義があることがわかります。

さまざまな考え方があるという点を踏まえて、テナントの分離の価値システム全体を理解しやすくする基本的な概念をいくつか説明します。すべての SaaS プロバイダーは、チームが SaaS 環境の分離フットプリントを定義する際に、チームを導く明確な一連の分離要件を確立する必要があります。以下は、一般的に SaaS テナントの分離モデル全体を作成するいくつかの重要な基本的概念です。

分離は任意ではない – 分離は SaaS の基本要素であり、マルチテナントモデルでソリューションを提供するすべてのシステムで、テナントリソースが確実に分離される手段を講じる必要があります。

認証と承認は分離と同じではない – 認証と承認により SaaS 環境へのアクセスを制御できますが、ログイン画面または API のエントリポイントを越えたからといって、分離が実現されたことにはなりません。これは分離パズルの 1 つにすぎず、それだけでは十分ではありません。

分離の実施をサービスデベロッパーに任せるべきではない – デベロッパーが分離に違反する可能性のあるコードを組み込むことはありませんが、テナントの境界を無意識に越えることはないと想定するのは非現実的です。これを軽減するために、分離ルールを適用する共有メカニズムを介してリソースへのアクセス範囲を (デベロッパーの観点外で) 制御する必要があります。

すぐに利用できる分離ソリューションがない場合は自分で構築する必要がある - AWS Identity and Access Management (IAM) など、テナントを分離するパスの簡素化に役立つセキュリティメカニズムがいくつかあります。これらのツールをより広範なセキュリティスキームと組み合わせて分離プロセスを簡素化できます。ただし、対応するツールまたはテクノロジーによって分離モデルが直接処理されないシナリオが存在する場合もあります。独自の何かを構築する場合であっても、明確なソリューションのないことが分離要件を引き下げる機会を表すべきではありません。

分離はリソースレベルの構造ではない – マルチテナンシーと分離の世界では、分離を、明確なインフラストラクチャリソース間に厳密な境界線を引く手法と考える人もいます。これは、多くの場合、テナントごとに個別のデータベース、コンピューティングインスタンス、アカウント、または Virtual Private Cloud (VPC) がある分離モデルという形になります。これらは一般的な分離の形式ですが、テナントを分離する唯一の方法ではありません。リソースが共有されるシナリオでも、特にリソースが共有される環境では分離を実現する方法があります。この共有リソースモデルにおいて、分離は、実行時に適用されるポリシーによって実施される論理構造にすることができます。ここで重要なのは、分離はリソースのサイロ化と同一にすべきではない点です。

ドメインは特定の分離要件を課す場合がある – テナントの分離を実現するアプローチは多数ありますが、ドメインによっては特定の種類の分離が必要になる制限が課される場合があります。例えば、コンプライアンスの厳しい業界ではテナントごとに独自のデータベースが必要になる場合があります。このような場合、共有ポリシーベースのアプローチでは、分離に不十分である可能性があります。

基本的な分離の概念

分離には、テナントの分離に複数の定義が存在するという課題があります。一部の人にとって、分離とは概してビジネスの構造であり、独自の環境を必要とする顧客全体を想定します。一方、分離とは、マルチテナント環境のサービスと構造をオーバーレイするアーキテクチャ構造のようなものと考える人もいます。後続のセクションでは、さまざまなタイプの分離について説明し、特定の用語を各分離構造に関連付けます。

トピック

- サイロ分離
- プール分離
- ブリッジモデル
- 階層ベースの分離
- 的を絞った分離

サイロ分離

多くの場合、SaaS プロバイダーはリソース共有の価値を重視しますが、完全にサイロ化されたリソースのスタックを実行しているモデルに一部 (またはすべて) のテナントをデプロイすることを SaaS プロバイダーが選択するシナリオもあります。このフルスタックモデルは SaaS 環境を表すものではないと考える人もいます。ただし、これらの個々のスタックを共有のアイデンティティ、オンボーディング、計測、メトリクス、デプロイ、分析、運用に含めている場合、これは、スケールのメ

リットと運用効率をコンプライアンス、ビジネス、またはドメインとトレードする SaaS の有効なバリアントになります。このアプローチでは、分離は顧客スタック全体にまたがるエンドツーエンドの構造です。図 16 に示す概念ビューは、この分離のビューの概念ビューを示しています。

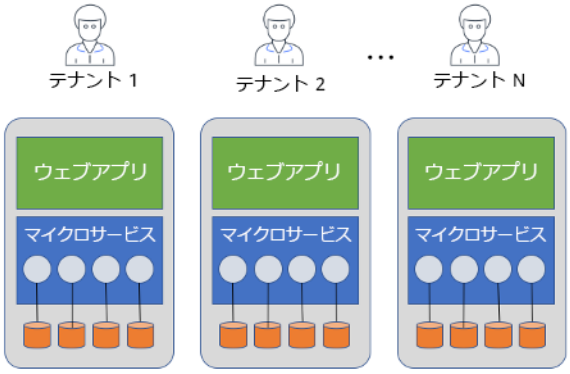

図 16: サイロ分離モデル

この図は、サイロ化されたデプロイモデルの基本的なフットプリントを示しています。これらのスタックの実行に使用されるテクノロジーは、ここではほとんど関係ありません。これは、モノリス、サーバーレスである場合や、さまざまなアプリケーションアーキテクチャモデルの任意の組み合わせである場合もあります。重要な概念は、テナントが持つスタックすべてを一部の構造に含めて、そのスタックの可動部すべてをカプセル化することです。これが分離の境界になります。完全にカプセル化された環境からテナントがエスケープするのを防止できる場合、分離が実現されたことになります。

一般に、この分離モデルは実施がはるかに簡単です。多くの場合、堅牢な分離モデルを実装できるようにする構造は明確に定義されています。このモデルは SaaS 環境のコストと俊敏性の目標達成に関して難しい課題を抱えているものの、非常に厳格な分離要件のあるユーザーにとっては、魅力的なものとなり得ます。

サイロモデルのメリットとデメリット

各 SaaS 環境とビジネスドメインが持つ独自の要件一式によっては、サイロが適している場合があります。しかし、この方向性に傾いている場合には、当然サイロモデルに関連する課題と経費について、細かく確認しておきたいでしょう。SaaS ソリューションにサイロモデルを検討している場合に考慮しておくべきメリットとデメリットがいくつかあります。

メリット

- 困難なコンプライアンスモデルをサポート – 一部の SaaS プロバイダーは厳格な分離要件が課せられている規制のある市場で販売しています。サイロモデルによって、これらの ISV はテナントの一部またはすべてに対して、専用モデルにデプロイするというオプションを提供できています。

- 他のテナントによる影響の懸念がない – すべての SaaS プロバイダーは他のユーザーの状況によって生じる影響を抑制しようとしていますが、それでも一部の顧客はシステムを使用している他のテナントのアクティビティによって、自分のワークロードが影響を受ける可能性に難色を示します。サイロモデルではこの懸念に対応し、他のテナントによる影響を受ける可能性がない、専用の環境を提供します。
- テナントコストの追跡 – SaaS プロバイダーは大抵の場合、各テナントがどれほどインフラストラクチャコストに影響を与えているかを把握することに重点を置いています。テナントあたりのコストの計算は一部の SaaS モデルでは困難です。しかし、サイロモデルでは複数に分かれているという特性によって、簡単な方法でインフラストラクチャコストを把握し、各テナントに紐付けられます。
- 影響範囲を軽減 – サイロモデルは通常、SaaS ソリューションで停止状態やイベントが発生した場合、その影響を軽減します。各 SaaS プロバイダーは独自の環境で実行しているため、特定のテナントの環境内で発生した障害は、大抵の場合その環境内に限定されます。1 つのテナントは停止に陥るものの、システムを使用している残りのテナントで、次々とエラーが発生することはありません。

デメリット

- スケーリングの問題 – プロビジョンできるアカウント数に制限があります。この制限のため、アカウントベースモデルを選択できない場合があります。また、急激にアカウント数が増加することによって、SaaS 環境の管理およびオペレーション上の操作性を低下させるという一般的な懸念もあります。例えば、テナントにつき 20 のサイロ化されたアカウントは管理できますが、千ものテナントがいる場合、その数はオペレーションの効率性と俊敏性に影響を与えるようになる場合があります。
- コスト – 独自の環境で実行しているすべてのテナントに関して、従来の SaaS ソリューションに伴うコスト効率性のほとんどは実現しません。これらの環境を動的にスケーリングしたとしても、おそらく使われない無駄なリソースが発生する期間が発生します。このモデルは完全に許容範囲のモデルであるものの、SaaS モデルに欠かせないスケールメリットと増益を実現できる可能性は低くなります。
- 俊敏性 – SaaS への移行は、迅速なペースでのイノベーション実行という願望が直接の動機となっていることがほとんどです。これは組織が迅速に市場動向に応え、対応できるようにするモデルの採用を意味します。この観点で重要となるのは、カスタマーエクスペリエンスを統一し、迅速に新しい機能や性能をデプロイできることです。サイロモデルによる俊敏性への影響を抑えるためにできる対策はあるものの、細かく分散化されているというサイロモデルの特性によって、テナントの管理、運営、サポートを容易に行えるという機能に影響を及ぼす複雑さが付加されます。

- オンボーディング自動化 – SaaS 環境は新規テナント追加の自動化に対してプレミアムが発生します。これらのテナントがセルフサービスモデルまたは内部管理のプロビジョニングプロセスのどちらでオンボーディングを行う場合でも、オンボーディングを自動化する必要があります。各テナントに個別のサイロがある場合、この作業はさらに負担の大きいプロセスとなることがしばしばあります。新規テナントのプロビジョニングは、新規インフラストラクチャのプロビジョニングを必要とするほか、新規アカウント制限の設定も必要となる場合があります。このような可動部の追加は、全体的なオンボーディングの自動化に追加の側面での複雑さをもたらす経費につながり、顧客に注力できる時間が少なくなります。
- 分散管理およびモニタリング – SaaS での目標は、1 つの画面ですべてのテナントアクティビティの管理およびモニタリングができるようになることです。この要件は特に、サイロ化したテナント環境の場合に重要となります。この場合の課題は、より分散化したテナントフットプリントからデータを蓄積しなければならないことです。テナントの集計ビューを作成するメカニズムはあるものの、このメカニズムを構築し、管理するために必要な手間と労力は、サイロ化モデルにおいてはより複雑になります。

プール分離

分離サイロモデルは多くの SaaS 企業で非常にうまく位置づけられていることがわかります。SaaS に移行している企業の多くは、テナントに基盤インフラストラクチャの一部またはすべてを共有できる、効率性、俊敏性、コスト面での利点を求めています。この共有インフラストラクチャアプローチは、プールモデルと呼ばれ、分離の議論に一定の複雑さを付加しています。図 17 はプールモデルでの分離の実装に関連する課題を表しています。

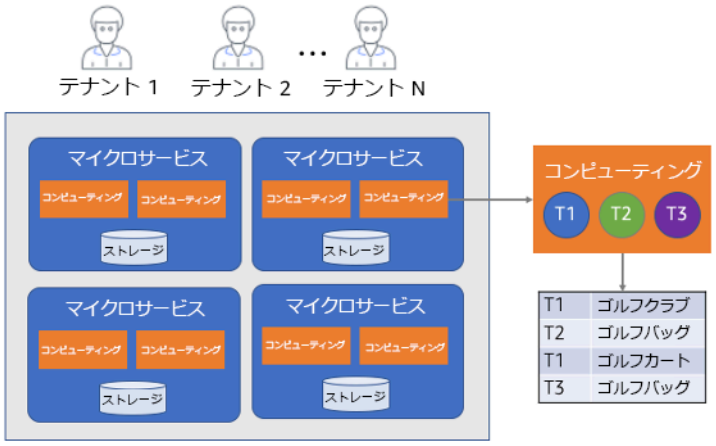

図 17: プール分離モデル

このモデルでは、すべてのテナントで共有しているインフラストラクチャを、テナントが利用していることがわかります。これにより、リソースをテナントによる実際の負荷に正比例してスケーリング

できます。図の右側は 1 つのサービスをコンピューティングの観点から詳しく表したものです。テナント 1 から N までがすべて、特定の時間に共有コンピューティング内で隣り合って実行する可能性があることを示しています。この例のストレージもまた共有されており、個別のテナント識別子でインデックス付けされたテーブルとして示しています。

このモデルは SaaS プロバイダーにとっては適切に機能しますが、分離の議論全体を複雑にする可能性があります。共有されたリソースで分離を実装することは、それほどわかりやすく典型的なネットワーキングではなく、テナント間の境界作成は IAM 構造に依存できません。

この重要な点は、分離がより難しい環境であるものの、それを理由にして環境の分離要件の緩和はできないということです。共有モデルではクロステナントアクセスの可能性が増加するため、リソースが分離されていることの確認を徹底する必要がある領域といえます。

プール分離モデルを掘り下げると、このアーキテクチャ上のフットプリントは課題が独自に交じり合っており、テナントのリソースを適切に分離するには、それぞれに独自のタイプの分離構造が必要となることがわかります。

プールモデルのメリットとデメリット

すべてを共有することで高い効率性と最適化を実現できますが、SaaS プロバイダーはこのモデルの採用に伴ういくつかのトレードオフを強調する必要があります。多くの場合、プールモデルのメリットとデメリットは、サイロモデルで取り上げたメリットとデメリットの逆となります。以下は一般的にプール分離モデルに関わる主なメリットとデメリットです。

メリット

- 俊敏性 – テナントを共有インフラストラクチャモデルに移行すると、その特性である効率性とシンプルさを利用して、SaaS サービスの俊敏性を整備できます。基本的にプールモデルは、SaaS プロバイダーが統一された単一の操作性でテナントすべての管理、スケーリング、運営ができるようにするためのものです。操作性の一元化と標準化は、SaaS プロバイダーがより簡単に管理を行い、テナントごとに 1 回限りのタスクを実行する必要なく、すべてのテナントに変更を適用できるようにする基盤となります。このオペレーション上の効率性は、SaaS 環境の全体的な俊敏性のフットプリントの鍵となります。
- コスト効率性 – 多くの企業は SaaS の高いコスト効率性に魅力を感じています。このコスト効率性の大部分は、一般的に分離のプールモデルによるものです。プールされた環境では、システムはすべてのテナントによる実際の負荷とアクティビティに基づきスケーリングします。すべてのテナントがオフラインであれば、インフラストラクチャコストは最小限になります。この主要な概念としては、プールされた環境はテナント負荷に動的に適応でき、テナントアクティビティをリソース使用量により最適に合わせられることにあります。

- 管理と運営の簡素化 – 分離のプールモデルでは、システム内のすべてのテナントを一括で確認できます。すべてのテナントの管理、更新、デプロイが、システム内のすべてのテナントに対応する単一の操作で可能です。これにより、管理と運営に関わるフットプリントのほとんどの部分が簡素化されます。
- イノベーション – プールされた分離モデルによって実現する俊敏性は、SaaS プロバイダーが迅速なペースでイノベーションを実行するための中核となり得ます。分散型の管理、サイロモデルの複雑さから離れるほど、製品の性能と機能により重点を置けるようになります。

デメリット

- 他のテナントによる影響 – リソースが共有されるほど、あるテナントが他のテナントの操作性に影響を及ぼす可能性は高くなります。例えば、あるテナントのアクティビティによってシステムに大きな負荷がかかると、他のテナントに影響を生じさせる場合があります。適切なマルチテナントアーキテクチャと設計ではこれらの影響を抑制しようとしますが、プールされた分離モデルでは、他のテナントの状況によって 1 つ以上のテナントが影響を受ける可能性が常にあります。
- テナントコストの追跡 – サイロモデルでは、非常に簡単にリソースの消費を特定のテナントに紐付けられます。一方、プールされたモデルでは、リソースの消費を紐付けることはより難しくなります。各 SaaS プロバイダーはシステムを測定して、効果的に消費量を個々のテナントに紐付けるために必要な詳細データを取得する方法を見つける必要があります。
- 影響範囲を軽減 – すべての共有リソースを共有することは、オペレーションリスクにもつながります。サイロモデルでは、あるテナントに障害が発生すると、大抵の場合その障害の影響はそのテナントのみに限定されます。一方、プールされたモデルでは、停止はシステム内のすべてのテナントに影響する可能性が高く、ビジネスに大きな影響を及ぼす場合があります。通常、障害を特定して明らかにし、スムーズに障害から復旧できるよう、耐障害性の高い環境の構築に対して、より高いコミットメントを必要とします。
- コンプライアンスによる抵抗 – プールモデルのテナントを分離できる手段はあるものの、インフラストラクチャの共有という概念によって、このモデルで実行することを躊躇させる状況になる場合があります。特に、ドメインのコンプライアンスまたは規制ルールで、リソースのアクセシビリティと分離に厳格な制限がある環境が該当します。そのような場合、システムの一部分については、サイロ化する必要性が生じる場合があります。

ブリッジモデル

サイロモデルとプールモデルは分離に対してまったく異なるアプローチをしますが、多くの SaaS プロバイダーの分離の状況はそれほど絶対的なものではありません。実際のアプリケーションの問題を確認し、システムをより小さいサービスに分解すると、大抵の場合、ソリューションにはサイロモデ

ルとプールモデルを混合したものが必要だとわかります。この混合モデルを分離のブリッジモデルと呼んでいます。図 18 は SaaS ソリューションでのブリッジモデルの実装方法の例を示しています。

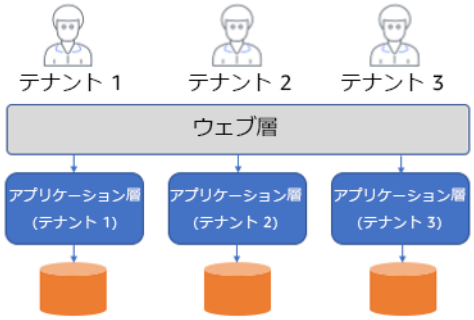

図 18: ブリッジ分離モデル

この図は、ブリッジモデルによってどのようにサイロモデルとプールモデルを組み合わせられるかを示しています。こちらでは、従来型のウェブとアプリケーション層のモノリシックアーキテクチャがあります。ウェブ層はこのソリューションの場合、すべてのテナントが共有するプールモデルにデプロイされています。ウェブ層は共有されていますが、アプリケーションの基盤となるビジネスロジックとストレージは実はサイロモデルにデプロイされており、各テナントは独自のアプリケーション層とストレージを持っています。

モノリスをマイクロサービスに分解すると、システム内のさまざまなマイクロサービスそれぞれで、サイロモデルとプールモデルを組み合わせて利用できます。このアプローチの詳細については、さまざまな AWS 構成へのサイロモデルとプールモデルの適用に関する詳細説明で取り上げます。こちらでの重要なポイントは、サイロモデルとプールモデルに対する見方は、異なる分離要件を持つサービスの集合体に分解される環境において、より粒度が細かくなることです。

階層ベースの分離

分離に関する議論のほとんどがクロステナントアクセスを回避するメカニズムに特化していますが、サービスの階層化が分離戦略に影響を与える可能性がある場合もあります。そのような場合、テナントをどのように分離するかよりも、異なるプロファイルを持つ異なるテナントに対して、どのように異なるタイプの分離をパッケージ化し、提供するかが重要となります。また、対象とする幅広い顧客に対応するために、どの分離モデルが必要かを判断するうえで、別の考慮事項もあります。図 19 は階層による分離の違いを示した例です。

以下の例では、サイロ分離モデルとプール分離モデルの組み合わせを使って、階層としてテナントに提供しています。シルバー階層のテナントはプールされた環境で実行しています。これらのテナントは共有のインフラストラクチャモデルで実行しているものの、自分のリソースがクロステナントアクセスから保護されることを完全に期待できます。右側のテナントには、完全に専用の (サイロ) 環境

を提供する必要があります。それを実現するため、SaaS プロバイダーはプレミアム階層モデルを作り、テナントがこの専用モデルで実行できるようにしています。これは通常非常に高価となります。

SaaS プロバイダーは一般的に顧客にサイロモデルを提供することを制限しようとし、多くの SaaS ビジネスではこのモデルにデプロイするにはテナントがプレミアムを支払うという、個別料金の概念があります。実際に、SaaS 企業はこのオプションを選択する顧客の数を制限するため、これをオプションとして公表することもなければ、階層として認識もしていません。あまりにも多くのテナントがこのモデルに該当する場合には、完全にサイロ化したモデルを再検討し、先ほど説明した課題の多くを受け入れることになります。

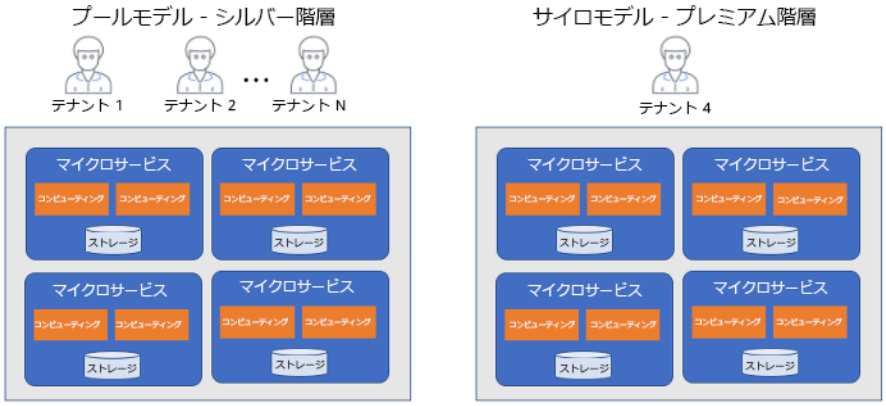

図 19: 階層ベースの分離

こういった単発の環境の影響を制限するために、SaaS プロバイダーは大抵の場合、これらのプレミアム顧客に対して、プールされた環境にデプロイされた製品の同じバージョンを実行することを求めます。これにより ISV は単一の画面から両方の環境の管理および運営が引き続きできます。基本的に、サイロ環境はたまたま 1 つのテナントをサポートしているプールされた環境のクローンとなります。

的を絞った分離

システムでの分離オプションは、非常に細分化されることを理解しておくことが重要です。システムの各マイクロサービス、およびそのサービスが使用する各リソースには、異なる分離モデルを設定できるオプションがあります。マイクロサービスの例をいくつか確認し、異なるマイクロサービス間で分離モデルを変える方法について理解を深めましょう。図 20 は、分離のサイロモデルとプールモデル両方を使用するマイクロサービスを表しています。

この図から、システムは 3 つの異なるマイクロサービスである、製品、注文、アカウントを実装していることがわかります。これらのマイクロサービスそれぞれのデプロイおよびストレージモデルは、どのように SaaS 環境で分離 (セキュリティまたは他のテナントによる影響の分離) が行われているかを示しています。

これらサービスそれぞれの分離モデルを確認していきましょう。右上にある製品のマイクロサービスは、完全なプールモデルにデプロイされており、コンピューティングとストレージの両方がすべてのテナントに共有されています。すべてのテナントが同じテーブルでインデックス付けされた個別のアイテムとなっていることを、こちらのテーブルでは反映しています。前提として、このテーブルのテナントアイテムへのアクセスを制限できるポリシーでもって、データは分離されます。注文のマイクロサービスは、テナント 1 から 3 専用で、プールモデルに実装されています。こちらの唯一の違いは、テナントのサブセットをサポートしている点です。基本的には、注文のマイクロサービスの専用 (サイロ) デプロイがされていないテナントは、このプールされたデプロイで実行します (サイロマイクロサービスとして除外された一部のテナント以外のテナント 1 から N とします)。

この点を説明するため、専用の注文のマイクロサービス (右上) とアカウントのマイクロサービス (下) で表されているサイロ化したサービスに注目しましょう。テナント 4 と 5 に対して、注文のマイクロサービスのスタンドアロンインスタンスをデプロイしていることがわかります。この理由は、これらのテナントは注文処理に関する要件 (コンプライアンス、他のテナントによる影響など) があり、このサービスをサイロモデルでデプロイする必要があったためです。こちらでは、コンピューティングとストレージは両方とも、完全に各テナント専用となっています。

最後に、下にあるアカウントのマイクロサービスはサイロモデルを表していますが、ただしストレージレベルのみです。マイクロサービスのコンピューティングはすべてのテナントに共有されていますが、各テナントにはアカウントデータを保持するための専用データベースがあります。このシナリオでは、分離に関わる懸念事項は、データの分離のみに絞られます。コンピューティングは共有することが可能です。

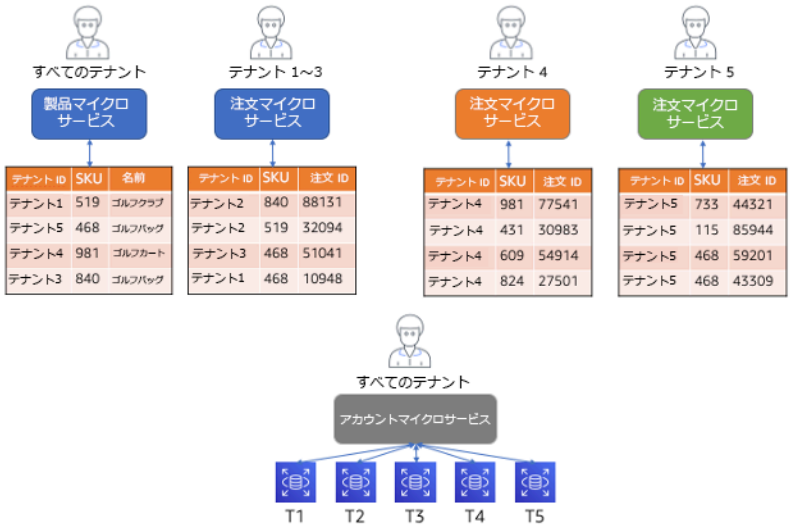

図 20: 的を絞った分離

このモデルはサイロに関する議論がより細分化することを示しています。セキュリティ、他のテナントによる影響、さまざまな要因によって、どのようにかつどういう場合にサイロ分離モデルをサービスに採用するかが決まります。こちらでの重要なポイントは、サイロモデルは使うか、使わないかのみの判断に限らないということです。サイロモデルをアプリケーションの特定のコンポーネントで使用することを検討できます。また、見込み顧客がこのモデルの使用を求めた場合など、必要な場合にのみこのモデルの課題を受け入れます。この場合、顧客とのより詳細な議論で、ストレージと処理における特定の領域にのみ懸念があることがわかります。そうすることにより、サイロ分離を必要としないシステムのその部分のプールモデルの効率性が得られると同時に、階層ストラクチャを提供できる柔軟性が確保でき、個々のサービスにおいてサイロモデルとプールモデル両方の組み合わせをサポートできます。

データ保護

SaaS アプリケーションに固有のセキュリティプラクティスというものはありません。

インシデント対応

SaaS アプリケーションに固有のセキュリティプラクティスというものはありません。

リソース

セキュリティのベストプラクティスの詳細については、以下のリソースを参照してください。

ドキュメントとブログ

- Isolating SaaS Tenants with Dynamically Generated IAM Policies
- Partitioning Pooled Multi-Tenant SaaS Data with Amazon DynamoDB
- Multi-tenant data isolation with PostgreSQL Row Level Security
- Identity Federation and SSO for SaaS on AWS
- Managing SaaS Identity Through Custom Attributes and Amazon Cognito
- SaaS Quick Start Highlights Identity and Isolation with Amazon Cognito
- Onboarding and Managing Agents in a SaaS Solution
- Amazon Cognito API リファレンス
- Building Serverless SaaS with Lambda layers
- Modeling SaaS Tenant Profiles on AWS

ホワイトペーパー

- マルチテナント環境でリソースを分離する SaaS テナント分離戦略に関するホワイトペーパー
- SaaS ストレージ戦略に関するホワイトペーパー

動画

- AWS re:Invent 2017: SaaS and OpenID Connect: The Secret Sauce of Multitenant Identity and Isolation
- AWS re:Invent 2016: The Secret to SaaS (Hint: It's Identity)
- AWS re:Invent 2019: SaaS tenant isolation patterns

信頼性の柱

信頼性の柱には、インフラストラクチャまたはサービスの障害からの復旧、必要に応じた動的なコンピューティングリソースの獲得、設定ミスや一時的なネットワークの問題などによる障害の軽減などのシステムの能力が含まれます。

設計の原則

クラウドには、信頼性の向上に役立ついくつかの原則があります。

定義

クラウドにおける信頼性に関するベストプラクティスには 3 つの領域があります。

- 基盤
- 変更管理
- 障害の管理

高い信頼性を実現するため、システムの基盤について十分に計画し、モニタリングを実施する必要があります。需要や要件の変更に対応するためのメカニズムも必要です。障害を検出し、自動的に修復できるシステムを設計することが必要です。

ベストプラクティス

トピック

- 基盤

- 変更管理
- 障害の管理

基盤

SaaS 信頼性 1: 個々のテナントがシステム内の他のテナントの可用性に影響を与える可能性がある負荷をかけることをどのように制限しますか?

マルチテナント環境におけるテナントのワークロードは、常に変化します。テナントが異なるタイプの負荷をシステムにかける場合があります。新しいワークロードプロファイルを持つ新規のテナントが、継続的にシステムに追加されることもあります。これらの要因により、SaaS 企業にとって、こういった拡大するニーズに応え、対応できる十分な耐障害性を持ったアーキテクチャを構築することは非常に難しくなっています。

このような負荷の変動は、テナントの操作性に悪影響を及ぼす可能性があります。1 つのテナントがシステムの一部に極度な負荷をかけた場合を想像してみましょう。API 経由でシステムを利用できる場合は、これは特に顕著となります。この場合、このテナントの負荷はシステムのリソースを過剰に消費することになり、さらにシステム全体の信頼性または他のテナントの操作性に影響を与えます。

あるテナントが他のテナントに影響を与える可能性があることは、階層サービスのあるシステムではより複雑な問題になります。例えば、システムにベーシック、アドバンスト、プレミアム階層があるとします。これらの各階層がシステムにどの程度の負荷もかけられる場合、ベーシック階層がプレミアム階層のテナントの信頼性に影響を及ぼす場合があります。

マルチテナント環境では、システムの信頼性に影響を与えうるワークロードと利用パターンの特定に、特に積極的に取り組む必要があります。マルチテナント環境における信頼性の問題は、システム全体に広がりやすく、さらにはすべての顧客の操作性に影響を与える可能性もあります。

これらのワークロードの問題に対処するため、ワークロードの問題がアプリケーションの信頼性に影響を及ぼす前に、それを検知し、解決するメカニズムをシステムに導入する必要があります。

アプリケーションとアプリケーションスタックのアーキテクチャは、テナントがシステムの信頼性と他のテナントの操作性に影響を及ぼさないために採用する戦略に影響があります。一般的なアプローチの 1 つとしては、スロットリングを使用して、テナントによる過度なリソースの消費を防ぐ方法があります。図 21 はそれを Amazon API Gateway を使って行う例を示しています。

この例では、API Gateway で使用量プランを活用して、システム内の各テナントに対して別々の使用量を定義していることがわかります。このアプローチでは、異なる API キーをシステムのベーシックおよびアドバンスト階層に対して使用します。これらのキーは異なる SLA の使用量プランにつながっています。このアプローチにより、2 つの異なるレベルでの制御が可能になります。まず、使用量プランの設定を通じたリクエストによって、すべてのテナントがシステムを飽和状態にしないようにします。もう 1 つのメリットは、使用量設定を使って、低い階層のテナントが高い階層のテナントの操作性に影響を及ぼすことがないようにできます。

このモデルは API Gateway を基盤としてスロットリングポリシーを実装しますが、この中核となる概念は他のインフラストラクチャ構造に適用できます。このアプローチの基本的な目的は、エントリポイントでアプリケーションへのアクセスをモニタリングして管理し、環境の全体的な信頼性に影響を与える可能性がある負荷をかけているテナントを検知し、スロットリングすることです。

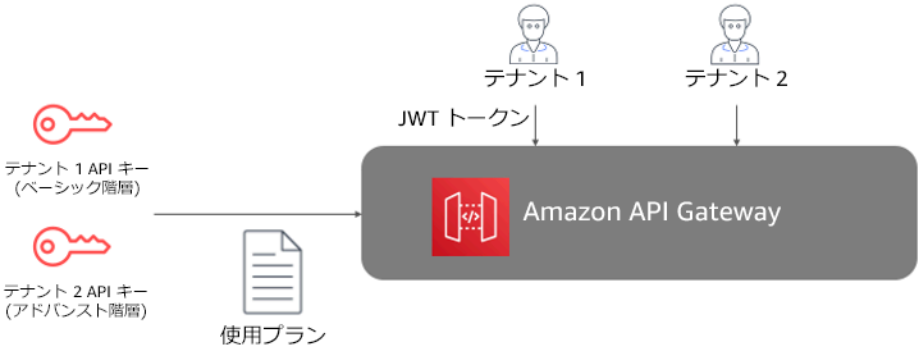

図 21: 階層ごとにテナントをスロットリングする

SaaS 信頼性 2: テナントの健全性をどのように積極的に検知および維持しますか?

SaaS 環境の信頼性は、問題がテナントに影響を及ぼす前に、前もって特定し解決できるプロバイダーの能力に大きく左右される傾向にあります。マルチテナント環境において、健全性を積極的に把握するためには、テナントのワークロードの健全性の状況について、より詳細でテナントに対応したインサイトが得られる追加の信頼性データを取得する必要があります。

これらのテナントに対応したインサイトは、テナント固有の傾向、アクティビティ、インサイトの特定に使用され、そのテナントの信頼性またはシステム全体の信頼性に影響を及ぼす状況を効果的に把握できるようになります。このデータを保持し、積極的に確認することで、アラーム、ポリシー、停止状態に陥ることなくシステムの修復を試みる自動化を構築できます。

これを可能にするには、アプリケーションにコードを導入して、テナントのコンテキストとともに健全性のインサイトを作成する必要があります。これはまず、有益な健全性データである、アーキテクチャのワークフローとイベントを特定することから始めます。消費データ、スケーリングインサイト、レイテンシーメトリクスなどが該当します。これらのインサイトはログファイル経由で、またはデータを蓄積するデータウェアハウスに作成されます。

リポジトリにこの健全性データが蓄積された後、蓄積した健全性データの分析に基づき、アラームを出すツール、またはポリシーをトリガーするツールを導入します。

SaaS 信頼性 3: SaaS アプリケーションのマルチテナント機能をどのようにテストしていますか?

マルチテナント環境のテストモデルは、単にアプリケーションの機能が想定どおりに機能することを確認するのみにとどまりません。SaaS プロバイダーは、システムが効果的にマルチテナントソリューションに関連する一般的な信頼性の問題に対応できることを検証するテストを構築する必要もあります。

マルチテナントのテスト戦略の一部として含められるさまざまな観点があります。多くの場合、これらのテストは現在ある構造を検証して、SaaS 製品のスケーリング、運営、信頼性のフットプリントに対応することを目的としています。

SaaS テストは大抵の場合、アプリケーションが経験する極端な状態をシミュレーションします。システムが想定内と想定外の状況にどのように対応するかを効果的にモデル化し、評価できる一連のテストの構築に注力する必要があります。顧客が満足できる操作性を確保することに加え、どの程度のコスト効率性でスケーリングを実現できるかも、テストでは考慮する必要があります。アクティビティに対して過剰なリソースを割り当てている場合、ビジネスの収益に影響を及ぼしていることがあります。

SaaS 環境における負荷とパフォーマンスのテスト戦略を向上する特定の領域には以下のものがあります。

- クロステナントインパクトテスト – テナントのサブセットがシステムに過度な負荷をかけるシナリオをシミュレーションするテストを作成します。これにより、負荷がテナント間で均等に分散していない場合のシステムの対応方法を決定でき、それが全体的なテナントの操作性にどのような影響を及ぼすかを評価できます。システムが個別にスケーラブルなサービスに分割されている場合、各サービスのスケーリングポリシーを検証するテストを作成して、適切な条件に基づきスケーリングしていることを確認することもできます。

- テナントの消費量テスト – 幅の広い負荷プロファイル (変化が少ない、急増、ランダムなど) を作成して、リソースとテナントアクティビティメトリクスの両方を追跡し、消費量とテナントアクティビティの差分を特定します。最終的にはこの差分をモニタリングポリシーの一部として使用し、最適でないリソースの消費を特定できます。また、このデータを他のテストデータとともに使用して、インスタンスが適切なサイズになっているか、IOPS が適切に設定されているか、AWS のフットプリントが最適化されているかを判断できます。
- テナントワークフローテスト – これらのテストを使用して、SaaS アプリケーションの異なるワークフローがどのようにマルチテナントコンテキストの負荷に対応しているかを評価します。ソリューションの中でよく知られたワークフローを選び、これらのワークフローに複数のテナントによる負荷を集中させ、マルチテナント設定でボトルネックまたはリソースの過剰な割り当てを引き起こしたかを確認します。
- テナントのオンボーディングテスト - テナントによるシステムへのサインアップに際しては、テナントが肯定的な体験をできるようにし、オンボーディングフローに回復性、拡張性、効率性を備えさせておく必要があります。これは特に、SaaS ソリューションによってオンボーディングプロセス時にインフラストラクチャをプロビジョニングする場合に当てはまります。アクティビティの急増がオンボーディングプロセスに悪影響を与えないように確認する必要があります。また、サードパーティの統合 (課金など) を利用するような場合には、これらの統合が SLA をサポートしているかどうかを検証する必要があります。場合によっては、これらの統合で起こり得る停止に対処するためのフォールバック戦略を実装します。これらのケースでは、これらのフォールトトレランスメカニズムが期待どおりに動作するかどうかを検証するテストを導入する必要があります。
- API スロットリングテスト - API スロットリングの考え方は、SaaS ソリューションに固有のものではありません。一般的に、API を公開する場合には、スロットリングの概念を含める必要があります。また、SaaS では、異なる階層のテナントが API を介してどのように負荷をかけることができるかを考慮する必要があります。例えば、無料利用枠のテナントは、ゴールド階層のテナントと同じ負荷をかけることができないようにすることもできます。テストを行えば、各階層に関連付けられたスロットリングポリシーが正常に適用され、実施されていることを確認することができます。
- データ配信テスト - ほとんどの場合、SaaS のテナントデータが一律に配信されることはありません。テナントのデータプロファイルの変化は、データフットプリント全体のバランスを崩す可能性があり、ソリューションのパフォーマンスとコストの両方に影響を与える可能性があります。この動的な要素を相殺するために、SaaS チームは通常、これらのバリエーションを考慮して管理するシャーディングポリシーを導入します。シャーディングポリシーは、ソリューションのパフォーマンスとコストプロファイルに欠かせないものであり、優先順位の高いテスト候補となります。データ配信テストを行うと、システムが遭遇する可能性のあるテナントデータのさまざまなパターンについて、採用したシャーディングポリシーがうまく配信できるかどうかを検証することができま

す。これらのテストを早期に実施すれば、大量の顧客データを保存した後に新しいパーティショニングモデルに移行する際にかかる高額なコストを避けることができます。

- テナントの分離テスト - SaaS の顧客は、自分たちの環境が保護され他のテナントからアクセスできないように、あらゆる対策が講じられることを期待しています。この要件をサポートするために、SaaS プロバイダーは、各テナントのデータとインフラストラクチャを保護するための多数のポリシーとメカニズムを構築しています。これらのポリシーの実施を継続的に検証するテストの導入は、どの SaaS プロバイダーにとっても不可欠です。

ご覧のように、このテストリストでは、SaaS ソリューションがマルチテナントのコンテキストで負荷を処理できるようにすることに重点を置いています。SaaS の負荷は予測できないことが多いため、これらのテストは、テナントの 1 つまたはすべてが影響を受ける前に、主要な負荷やパフォーマンスの問題を発見する絶好の機会となります。場合によっては、これらのテストで新たな変曲点が明らかになることもあるため、システムの運用ビューに含めてもよいかもしれません。

変更管理

SaaS アプリケーションに固有の信頼性に関する運用方法はありません。

障害の管理

SaaS アプリケーションに固有の信頼性に関する運用方法はありません。

リソース

信頼性に関するベストプラクティスの詳細については、以下のリソースを参照してください。

ドキュメントとブログ

- Monolith to serverless SaaS: Migrating to multi-tenant architecture
- Testing SaaS Solutions on AWS
- Importance of Service Level Agreement for SaaS Providers
- Using Amazon SQS in a Multi-Tenant SaaS Solution
- Partitioning Pooled Multi-Tenant SaaS Data with Amazon DynamoDB
- Architecting Successful SaaS: Interacting with Your SaaS Customer's Cloud Accounts
- AWS Auto Scaling
- Creating and using usage plans with API keys
- Managing Concurrency for a Lambda Function

- Amazon API Gateway: Throttle API requests for better throughput
- AWS とオンプレミスにおける AWS のリソースとアプリケーションの Amazon CloudWatch オブザーバビリティ
- Amazon CloudWatch Publishing Custom Metrics

動画

- AWS re:Invent 2017: SaaS Monitoring - Creating a Unified View of Multi-tenant Health featuring New Relic
- AWS re:Invent 2019: Building serverless SaaS on AWS
- AWS re:Invent 2019: Serverless SaaS deep dive: Building serverless SaaS on AWS (ARC410-R)

パフォーマンス効率の柱

パフォーマンス効率の柱では、コンピューティングリソースを効率的に使ってシステム要件を満たし、需要の変化と技術の進化に合わせて効率性を維持することに重点を置きます。

定義

クラウドのパフォーマンス効率を向上させるには、次の 4 つのベストプラクティス領域があります。

- 選択 (コンピューティング、ストレージ、データベース、ネットワーク)
- レビュー
- モニタリング
- トレードオフ

高性能なアーキテクチャを選択する際は、データ駆動型のアプローチを選択します。ハイレベルな設計から、リソースタイプの選択と設定に至るまで、アーキテクチャのあらゆる側面に関するデータを収集してください。選択した内容を定期的にレビューすることで、絶えず進化し続けている AWS クラウドを活用できているかを確認できます。

モニタリングを実施すれば、予想したパフォーマンスからのずれを把握し、その対策を講じることができます。さらに、圧縮やキャッシュを使用したり、整合性に関する要件を緩和したりするなど、アーキテクチャにおけるトレードオフを行ってパフォーマンスを向上させることができます。

ベストプラクティス

トピック

選択

SaaS PERF 1: あるテナントが別のテナントの体験に悪影響を与えることを防ぐにはどうすればいいのでしょうか?

マルチテナント環境では、システムにかかる負荷が大きく異なるプロファイルやユースケースをテナントが持つ場合があります。新しいワークロードプロファイルを持つ新規のテナントが、継続的にシステムに追加されることもあります。このような要素があると、SaaS 企業にとって、急速に進化するパフォーマンス要件を満たすアーキテクチャを構築することが非常に困難になる可能性があります。

テナント負荷のこのような変動を処理して管理することが、SaaS 環境のパフォーマンスプロファイルの鍵となります。SaaS アーキテクチャは、これらのテナントの消費傾向を正しく検知することが求められるほか、テナントの需要を満たしたり、個々のテナントのアクティビティを制限したりするために、効果的に拡張できる戦略を適用できなければなりません。

テナントによってシステムに不釣り合いな負荷がかかっている可能性があるこれらのシナリオを管理するために、さまざまな戦略を使用できます。これは、需要の高いリソースの分離、スケーリング戦略の導入、スロットリングポリシーの適用によって実現されます。

最もシンプルで極端なケースでは、アプリケーションの一部にテナント固有のデプロイを作成することもできます。図 22 は、システムを分解することによってパフォーマンスの課題に対処する方法を示しています。

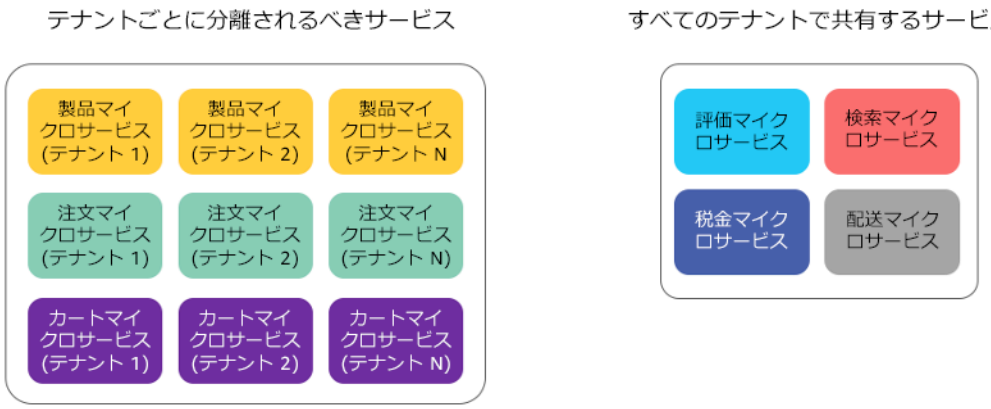

図 22: サイロ化されたサービスによるパフォーマンスの対処

この例では、2 つの異なるデプロイフットプリントがあります (一部はサイロモデルで、一部はプールモデルです)。図の左側では、各テナントに対して製品、注文、カートの各マイクロサービスの個別インスタンスがデプロイされていることがわかります。一方、図の右側には、すべてのテナントで共有されているマイクロサービスの集合体が表示されています。

このアプローチは、アプリケーションのパフォーマンスプロファイルにとって重要と見なされる特定のサービスを切り分けることを基本的な考え方としています。これらを分離することで、システムにおいて、(この一連のサービスについて) あるテナントの負荷が他のテナントのパフォーマンスに影響を与えないようにすることができます。この戦略では、コストが増加し、環境の運用上の俊敏性を低下することになりますが、パフォーマンス領域をターゲットにするための有効な手段になります。また、同様のアプローチは、コンプライアンスと分離の要件にも適用することができます。

例えば、各テナントに注文管理マイクロサービスを導入すれば、あるテナントが他のテナントの注文処理経験に悪影響を与えることを制限することができます。これにより、運用が複雑になり、コスト効率が低下しますが、アプリケーションの重要な領域に対してクロステナントのパフォーマンスの問題を選択的にターゲットにするブルートフォースの手段として使用することができます。

理想としては、個別にデプロイされたサービスのオーバーヘッドやコストを吸収することなく、テナントの負荷の問題に対応できる構成を通じて、これらのパフォーマンス要件に対応してください。ここでは、共有インフラストラクチャがテナントの負荷やアクティビティの変化に効果的に対応できるようにするためのスケーリングプロファイルの作成に焦点を当てます。

Amazon EKS や Amazon ECS のようなコンテナベースのアーキテクチャは、リソースの大幅なオーバープロビジョニングを必要とせずに、テナントの需要に基づいてサービスをスケーリングするように構成することができます。コンテナが迅速にスケーリングできることで、テナントの負荷が急増した場合でもシステムが効果的に対応できるようになります。コンテナのスケーリング速度と AWS

Fargate のコストプロファイルを組み合わせることによって、伸縮性、運用の俊敏性、コスト効率がしっかりと融合し、環境にオーバープロビジョニングすることなくテナントの負荷の急増に対応できるようになります。

また、AWS Lambda を使用してて構築されたサーバーレス SaaS アーキテクチャは、テナント負荷の急増への対応にも適しています。AWS Lambda のマネージド型の特質により、テナント負荷の急増に対処するために、アプリケーションのサービスを迅速に拡張することができます。このアプローチには、同時実行性とコールドスタートの要素が必要になる場合がありますが、それは、クロステナントのパフォーマンスの影響を制限するための効果的な戦略にもなります。

応答性のあるスケーリング戦略はこの問題を解決するのに役立ちますが、単にテナントがクロステナントに影響を与えるような負荷をかけることを防ぐためだけであれば、他の対策を講じることもできます。これらのシナリオでは、システム上の負荷のレベルを制御するためにスロットリングを適用する制限 (階層別の制限も可能) を設定することによって、テナントのアクティビティを検出して制限することもできます。これは、テナントの負荷を調査し、制限を超えるアクティビティを特定し、その経験をスロットリングするポリシーを導入することで実現できます。

SaaS PERF 2: インフラストラクチャリソースの消費とテナントのアクティビティやワークロードを調整するにはどうしたらよいですか?

SaaS 企業のビジネスモデルは、多くの場合、インフラストラクチャのコストをテナントの実際のアクティビティに合わせる戦略に大きく依存しています。SaaS システムのテナントの負荷や構成は常に変化しているため、SaaS 体験の一部であるリアルタイムで予測不可能な消費パターンを非常によくミラーリングしたパターンで、リソースの消費を効果的にスケーリングできるアーキテクチャ戦略が必要です。

図 23 のグラフは、インフラストラクチャの消費とテナントのアクティビティを調整した環境の仮想的な例を示しています。青の実線は、ある期間にまたがるテナントの実際のアクティビティ傾向を表し、赤色の破線は、テナントの負荷に対応するためにプロビジョニングされている実際のインフラストラクチャを表しています。

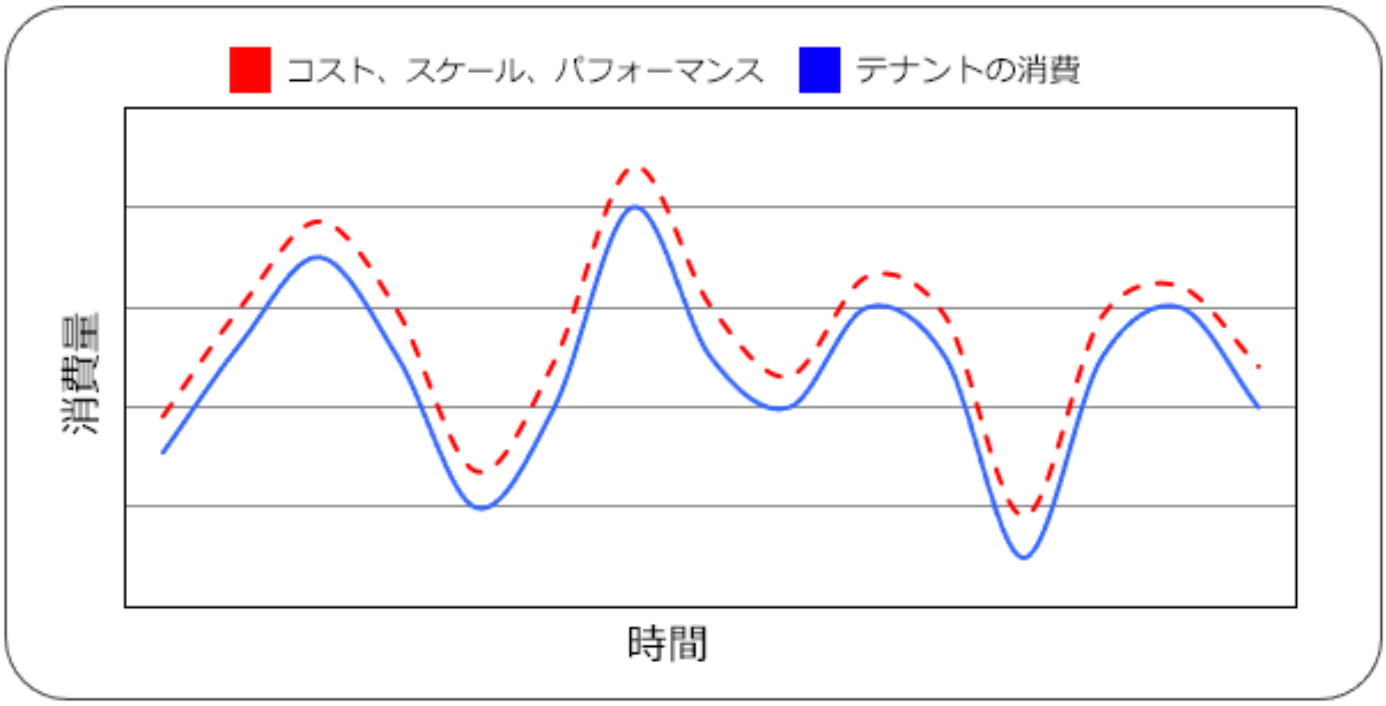

図 23: テナントのアクティビティと消費の調整

ここでの戦略は、理想的な環境において、赤色の線と青色の線とのギャップをできるだけ小さくすることです。ここでは、システムの可用性やパフォーマンスに影響を与えないようにするためのクッションがあるため、常に多少の誤差が生じます。同時に、テナントの現在のパフォーマンスニーズをサポートするのに十分なインフラストラクチャを提供できるようにする必要があります。

ここでの主な課題は、この図に示されている負荷が予測できないことが多いということです。一般的な傾向は考えられますが、アーキテクチャやスケーリングの戦略では、今日の負荷が明日も、あるいは 1 時間後も同じであると想定することはできません。

消費とアクティビティを調整するための最もシンプルなアプローチは、サーバーレスな体験を提供する AWS のサービスを利用することです。これの典型的な例としては、AWS Lambda があります。AWS Lambda を使用することで、サーバーやスケーリングポリシーが自分の責任ではないモデルを構築することができます。サーバーレスの場合、SaaS アプリケーションには、テナントの消費と直接関係のある料金しか発生しません。SaaS システムに負荷がなければ、AWS Lambda コストはかかりません。

また、AWS Fargate を使用すると、このサーバーレスマインドセットのコンテナベースのバージョンが可能になります。Fargate を Amazon EKS または Amazon ECS と合わせて使用すると、アプリケーションで実際に消費されるコンテナの計算コストのみを支払うことになります。

このサーバーレスモデルは、コンピューティング構造を超えて使用できます。例えば、ソリューションのストレージ部分にも、サーバーレステクノロジーを利用できます。Amazon Aurora Serverless では、データベースを実行しているインスタンスのサイズを調整することなく、リレーショナルデータを保存することができます。その代わり、Amazon Aurora Serverless では実際の負荷に基づいて環境のサイズが決定され、アプリケーションが消費する分だけ料金が請求されます。

スケーリングポリシーを作成する必要性のないモデルでは、運用とコストの経験が合理化されます。とらえどころのない完璧な自動スケーリング設定を求め続けるのではなく、アプリケーションの機能に時間と工ネルギーを集中させることができます。さらに、AWS の請求額の予期せぬ跳ね上がりを気にすることなく、ビジネスを成長させ、新たなテナントを受け入れることも可能になります。

サーバーレスが選択肢にない場合は、従来のスケーリング戦略に立ち戻る必要があります。これらのシナリオでは、テナントの消費メトリクスを取得して公開し、これらのメトリクスに基づいてスケーリングポリシーを定義する必要があります。

レビュー

SaaS アプリケーションに固有のパフォーマンに関する運用方法はありません。

モニタリング

SaaS PERF 3: テナントの階層やプランによって異なるパフォーマンスレベルを可能にするにはどうすればよいですか？

SaaS ソリューションは、多くの場合、テナントがさまざまな経験を利用できるように階層化されたモデルで提供されます。パフォーマンスは多くの場合、SaaS 環境の階層を差別化するために使用される領域であり、テナントに高レベルの階層への移動を促すための値の境界を作成する方法として使用されます。

このモデルでは、各階層の体験をモニタリングし、制御するための構造をアーキテクチャが導入されます。これは単にパフォーマンスを最大化するだけではなく、低階層のテナントの消費を制限することにもなります。たとえシステムがこれらのテナントの負荷に対応できたとしても、純粋にコストやビジネス上の配慮から、この負荷を制限することになる場合もあります。これは多くの場合、テナントのコストの規模と、テナントによるビジネスへの貢献がもたらす収益との相関性を確保するための手段の一部になります。

この問題にアプローチするための最も複雑さの少ない方法は、個々のテナント層に関連付けられたスロットリングポリシーの導入です。テナントが制限に達すると、スロットリングが適用され、消費が制限されます。

また、特定の AWS 構成を使用してテナント層の消費プロファイルを設定するシナリオもあります。例えば、AWS Lambda では、同時実行の予約を使用して、特定のテナント層の消費を制限することができます。図 24 は、これがどのようにして実現されるかを示しています。

図 24: 同時実行の予約によるテナントパフォーマンスの制御

この例では、3 つの異なるテナントの階層を作成し、それぞれの階層に SaaS アプリケーションのマイクロサービスを 3 つの別々のコレクションとしてデプロイしています。また、これらのコレクションは、そのグループの関数に対して実行可能な同時実行関数の呼び出し数を決定するために使用される、個別の同時実行の予約設定で構成されます。ベーシック階層には 100、アドバンスト階層には 300 の同時実行の予約があります。ここでは、プレミアム階層の残りの同時実行はすべて残され、下層の消費が制限されるという考え方になります。

このアプローチは、優先順位の高い層に最高の体験を提供し、リソースを過剰に消費して上位層のテナントのパフォーマンスに影響を与えないように下位層を制限するという当社の目標にうまく合致しています。

また、コンテナには、階層化を対処することによってパフォーマンスを実現するための独自の戦略があります。例えば、Amazon EKS では、個別の `ResourceQuotas` と `LimitRanges` を構成して、名前空間で利用可能なリソースの量を制御します。

これらの制約はテナントのパフォーマンスエクスペリエンスを構成するのには役立ちますが、SaaS アプリケーションの中には、アプリケーション設計と分解の戦略によって実際にパフォーマンスに対処するものもあります。これは、上位層のテナント向けにサイロ化されたマイクロサービスをデプロイすることで実現できますが、その場合は、これらの特定のサービスに対する他のテナントによる影響を考慮する必要がなくなります。実際、システムのマイクロサービスへの分解は、階層化とターゲットにするパフォーマンスプロファイルに直接影響されることがわかります。

場合によっては、SaaS アプリケーションに、より高いレベルのテナントのエクスペリエンスを最適化するアーキテクチャ構造を導入することもできます。例えば、プレミアム階層のテナントに重要なデータのキャッシングを提供すると仮定します。キャッシュをこれらのユーザーだけに限定すれば、すべてのユーザーをサポートするキャッシュにかかる費用を回避することができます。これらの最適化を導入するための労力は、投資を保証するための十分な価値を顧客とビジネスに提供することでオフセットする必要があります。

トレードオフ

SaaS アプリケーションに固有のパフォーマンに関する運用方法はありません。

リソース

当社のパフォーマンス効率性に関連するベストプラクティスの詳細については、以下のリソースを参照してください。

ドキュメントとブログ

- Optimizing SaaS Tenant Workflows and Costs Blog
- Using Amazon SQS in a Multi-Tenant SaaS Solution
- Importance of Service Level Agreement for SaaS Providers
- Amazon API Gateway: Throttle API requests for better throughput
- Creating and using usage plans with API keys
- Monitoring CloudWatch metrics for your Auto Scaling groups and instances
- Dynamic scaling for Amazon EC2 Auto Scaling

ホワイトペーパー

- SaaS Storage Strategies Building a Multi-tenant Storage Model on AWS
- Whitepaper: SaaS Solutions on AWS Tenant Isolation Architectures

動画

- AWS re:Invent 2018: SaaS Reference: Review of Real-World Patterns & Strategies
- AWS re:Invent 2016: Optimizing SaaS Solutions for AWS
- SaaS Metrics: The Ultimate View of Tenant Consumption
- Microservices decomposition for SaaS environments (ARC210)

- SaaS tenant isolation patterns

コスト最適化の柱

コスト最適化の柱には、システムを全ライフサイクルにわたり洗練および改善する継続的プロセスが含まれます。最初の PoC (実証支援) の初期設計から、製品ワークロードの継続運用まで、コスト意識の高いシステムを構築・運用する際に紙形式のプラクティスを採用すれば、ビジネス上の成果を上げながら、コストの最小化と投資利益率の最大化の両方を達成できるようになります。

定義

クラウド内でのコストの最適化に関する 4 つの分野のベストプラクティス:

- 費用対効果の高いリソース
- 需要と供給の一致
- 費用認識
- 長期的な最適化

他の柱と同様、検討する必要があるトレードオフも存在します。例えば、最適化したいのは市場投入までのスピードですか、それともコストですか。 市場投入のスピードを向上する、新しい機能を導入する、締め切りに間に合わせるといったケースでは、前払いコストの投資を最適化するよりも、スピードを重視して最適化することが最善です。

設計上の決定は、実際のデータに導かれずに、急いで行われる場合があります。なぜなら、コスト最適化のベンチマークを行う時間をかけず、過剰な投資を「念のため」行う傾向があるからです。

その結果、過剰なプロビジョニングと、あまり最適化されていないデプロイが生じることになります。以下のセクションでは、デプロイにおける初期段階でのコスト最適化と継続的なコスト最適化について、そのテクニックと戦略的ガイダンスを紹介します。

ベストプラクティス

トピック

- 費用対効果の高いリソース
- 需要と供給の一致
- 費用認識
- 長期的な最適化

費用対効果の高いリソース

SaaS アプリケーションに固有のコストに関する運用方法はありません。

需要と供給の一致

SaaS アプリケーションに固有のコストに関する運用方法はありません。

費用認識

SaaS COST 1: 個々のテナントのリソース消費はどのように測定されますか?

マルチテナント環境でのコストの測定と要因の判断は、消費の要因がテナントであることを明らかにするための確固たる戦略を持つことから始まります。そのためには、テナントがシステムのリソースをどのように消費しているかを明確に表す消費マッピングモデルの設計と開発をチームで行う必要があります。最終的な目標は、システムの各テナントに消費の割合を割り当てることができるようにするためのインサイトを集めることです。

テナントがシステムのリソースの一部または全部を共有する可能性があるマルチテナント環境では、このような消費に関する情報をまとめることは特に困難です。このより詳細な消費モデルは、AWS 環境で消費の要因を判断するためによく使われるオプションやツール戦略の多くを排除しています (タグ付けなど)。

SaaS アーキテクチャでテナントの消費をどのように捉えるかを定義するための単一のモデルはありませんが、アプリケーションの戦略を選択する際に考慮すべき共通の戦略がいくつかあります。まず、SaaS 環境の全体的なコストプロファイルを確認して、アプリケーションが AWS の請求書のコストにどのような影響を与えているかを判断するとします。環境によっては、アプリケーションのいくつかの領域にコストが集中している場合があります。これらのシナリオでは、請求書に最も影響している領域のみの消費データを収集することによって、ROI を改善できることがあります。例えば、Amazon S3 が請求書を占める割合が 1% である場合、テナントの Amazon S3 の消費を計算してもほとんど意味がないかもしれません。

ここで考慮が必要なもう 1 つの要素は、詳細度です。環境に適合すると考えられるテナントの消費を推測できる侵襲性の低いアプローチがあります。これは、結局のところ、求める消費の詳細のレベルと、消費を帰属させるために必要なデータを取り込んで取得することの複雑さのバランスを取るという問題になります。

まずは、テナントの消費を推測するための最も単純なモデルを見てみましょう。図 25 は、低侵襲モデルでテナントのアクティビティを取得する方法の概念図です。ここでの基本的なアプローチは、AWS Lambda オーソライザーとして使用される API への各呼び出しを検査することです。オーソライザーは、受信した JWT からテナントのコンテキストを抽出し、テナントのアクティビティを記録するイベントを発行します。これに代わるアプローチとして、Amazon API Gateway の代わりに AWS X-Ray を使用してこのデータを取得する方法があります。

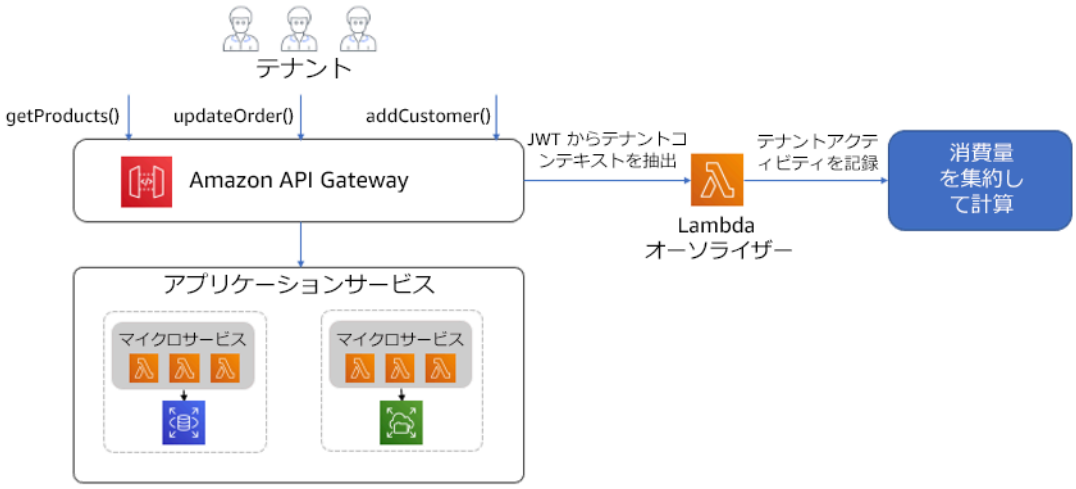

図 25: テナントの消費の低侵襲取得

図には、テナントの消費を集計して計算するためのプレースホルダーもあります。このギャップを埋めるために選択する戦略とツールは、データの性質、そのライフサイクル、より広範な SaaS のメトリクスと分析にどのように適合するかによって異なります。このデータを SaaS 環境の一般的なメトリクスフットプリントの一部として含めると、テナントへの消費配分に不可欠なインサイトを引き出すことができます。

この特定のアプローチは、各テナントの消費レベルを推測する方法として、各テナントの呼び出し頻度の追跡を利用します。サービスの呼び出し数と消費は正確には相関しないかもしれませんが、環境によってはこれが妥当な妥協点になることがあります。

アプリケーションの詳細を取得するより専門的な計測器を導入すれば、消費をより詳細に明らかにすることができます。図 26 は、SaaS アプリケーションのマイクロサービスにメトリクス計測を導入する方法を示しています。

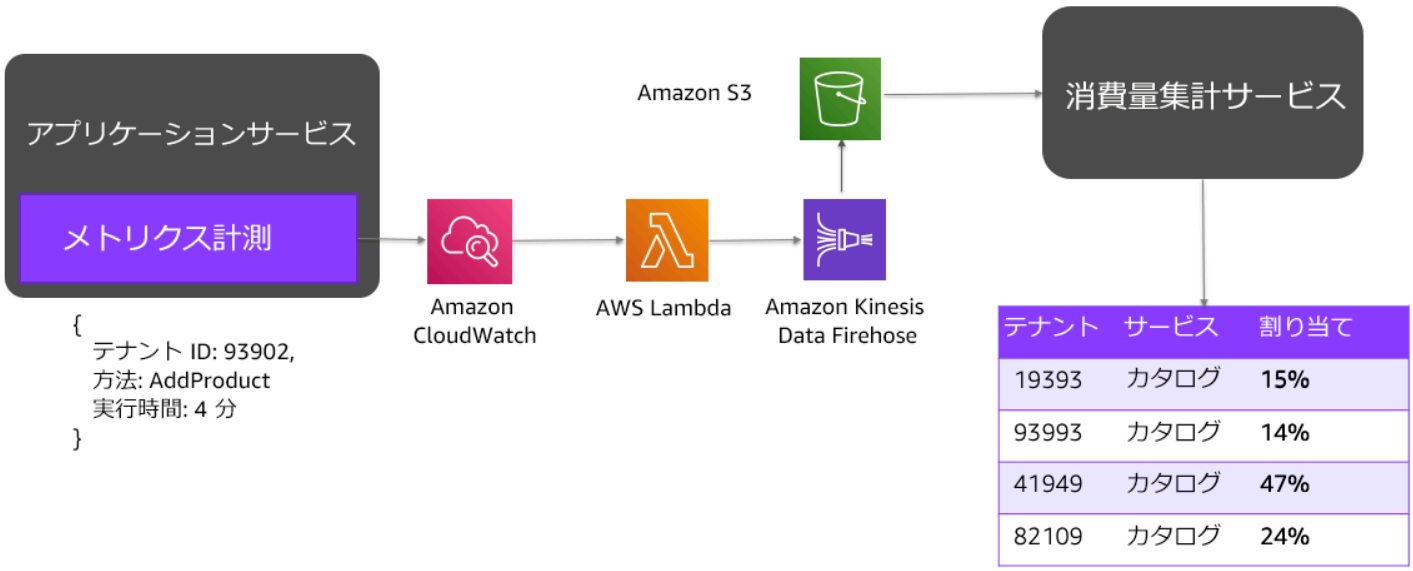

図 26: テナント消費イベントによるマイクロサービスの計測

この例では、アプリケーションの各マイクロサービスにメトリクス計測を導入しています。これらのマイクロサービスは、テナントによるこのサービスの消費状況、およびその関連リソースについてのより詳細なデータを取得します。この詳細データはイベントとして公開され、集計されます。ここでは、Amazon CloudWatch、AWS Lambda、Amazon Data Firehose、Amazon S3 がデータの公開と取り込みに使用されていることがわかります。その後、このデータが分析され、独自のモデリングに基づいて、テナント間の消費の分布が明らかになります。

アプリケーションのマイクロサービスを超えると、テナントの消費の要因を判断することが困難になります。サービスごとにターゲットを絞った具体的な戦略作りが必要になる場合もあります。例えば、ストレージサービスに、ストレージの消費をプロファイリングすることができる別のサービスが必要になるかもしれません。その場合、テナントの消費を分析するために、IOPS やデータフットプリントなどを調査する必要があるかもしれません。

SaaS COST 2: テナントの消費とインフラストラクチャのコストはどのように関連付けられますか?

SaaS 環境は動的な性質を持つことから、システムのインフラストラクチャのコストプロファイルがどのように変化しているかを理解することは困難です。システム内のニーズの変化やテナントの混在により、SaaS 環境の運用コストは大きく変動する可能性があります。同時に、SaaS ビジネスで

は、SaaS アプリケーションの構築、販売、運用方法を戦略的に決定するために、テナントがどのようにコストに影響を与えているかを明確に把握しておく必要があります。

テナントがビジネスのコストにどのように影響を与えているかを把握することのビジネス上の価値を理解するために、SaaS 環境へのコストデータの適用例を見てみましょう。図 27 のグラフは、SaaS 環境のコストがテナントからの収益とテナントが管理する e コマースカタログのサイズと相関関係にあるシナリオの一例を示しています。

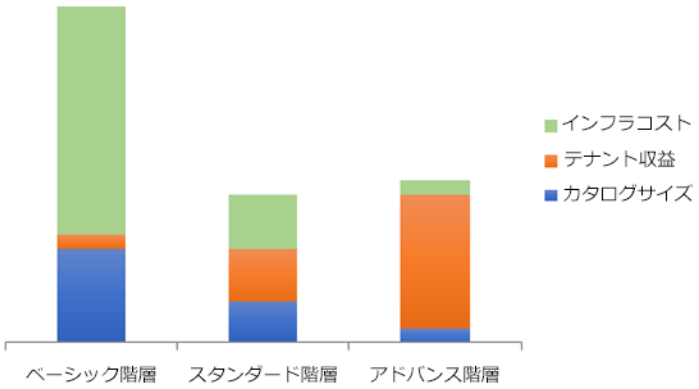

図 27: 階層ごとのテナントあたりのコスト

このグラフは、SaaS サービスの各層におけるコストの分布を示しています。ここでは、ベーシック階層とアドバンスト階層のテナントのインフラストラクチャコストに大きな差があることがわかります。収益の最も少ないベーシック階層が、システムのインフラストラクチャコストの最大の部分を担っているということを読み取ることが重要です。一方で、収益が最も多いアドバンスト階層は、コストフットプリントがはるかに小さくなっています。このバランスの悪さは、おそらくこのモデルに何か問題があるということを意味しています。

これは、SaaS プロバイダーにとって、テナントや階層別にコストを分類してアクセスできることが不可欠であることを示すほんの一例です。このテナントごとのコストデータにアクセスできれば、SaaS 組織は、環境のコストプロファイルに影響を与える可能性のある幅広いアーキテクチャ上の考慮事項を評価することができます。また、料金や階層化戦略のガイドとしても役立ちます。

このコストをテナントデータごとにまとめることには、2 つの基本的な側面があります。まず、テナントの消費の要因を判断して計算し、各テナントの消費の割合を求める方法が必要です (この消費データの収集方法の詳細については、上述を参照してください)。消費データを入手したら、このデータと AWS の請求書のコスト情報を関連付けて、テナントごとのコスト計算を行う必要があります。

請求データのコレクションへのアクセスには、いくつかのオプションが用意されています。AWS は、この課金データを取り込んで集計するための API を提供しています。また、APN パートナーの

さまざまなソリューションを利用してデータを取り込み、これらのソリューションを介してコストにアクセスすることもできます。ここでは、多くの場合、AWS のコストを取り込んでまとめるために APN パートナーと連携するのが最短ルートになります。

このエクスペリエンスの概要を図 28 に示します。収集する必要のある 2 つのデータがあることがわかります。あるプロセスで、テナントごとのコストモデルに関連するコストの詳細度に合わせて、コストをまとめた AWS の請求書のデータが集約され、取り込まれます。次に、テナントのアクティビティを分析し、各テナントに消費の割合を割り振ったテナント消費集計が表示されます。最後に、これらの消費の割合をインフラストラクチャコストに適用して、テナントあたりのコストを算出します。

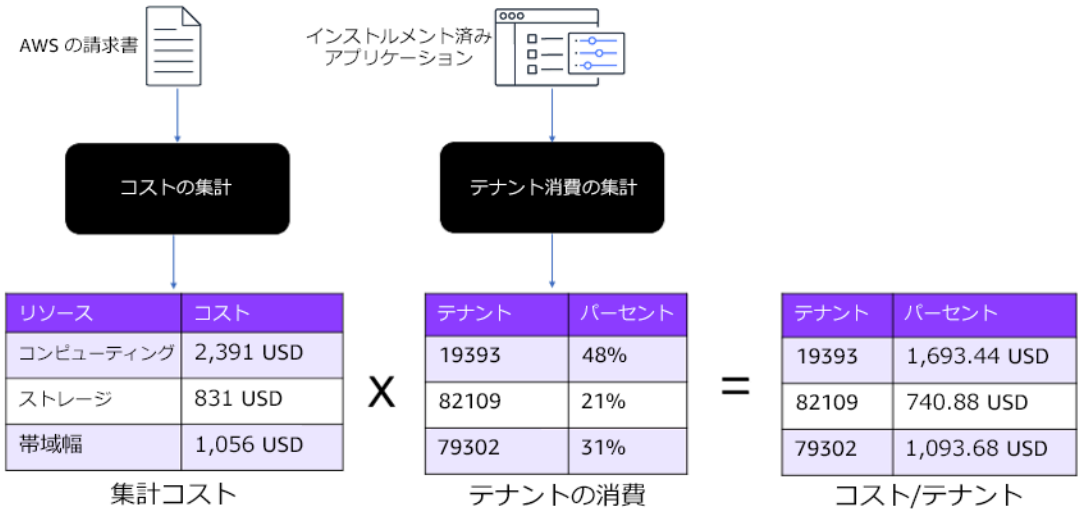

図 28: テナントあたりのコストの計算

このデータを取得したら、結果として得られるコストを表現する最適な方法を選択することができます。一般的には、ビジネスの中核となるサービスの範囲を対象とした、テナントあたりのコストを持つことが重要であると考えられますが、他の分析を行う場合には、重み付けを使用することによって、テナントごとの全体的なコストを計算することができます。

長期的な最適化

SaaS アプリケーションに固有のコストに関する運用方法はありません。

リソース

コスト最適化に関する AWS のベストプラクティスの詳細については、以下のリソースを参照してください。

ドキュメントとブログ

- Calculating Tenant Costs in SaaS Environments
- Calculating SaaS Cost Per Tenant: A PoC Implementation in an AWS Kubernetes Environment
- SaaS metrics deep dive: A look inside multi-tenant analytics
- SaaS Analytics and Metrics: Capturing and Surfacing the Data That's Fundamental to Your Success
- AWS のモニタリングツール

動画

- SaaS Metrics: The Ultimate View of Tenant Consumption

まとめ

AWS Well-Architected フレームワークでは、SaaS アプリケーション向けクラウドに、主要アーキテクチャのベストプラクティスが提供されるため、信頼性、安全性、効率性、コスト効率に優れたシステムを設計して運用することができます。このフレームワークでは、既存のアーキテクチャや提案されたアーキテクチャを評価するための一連の質問に加え、それぞれの柱についての一連の AWS ベストプラクティスも提供されています。このフレームワークをアーキテクチャに適用し、安定した効率のよいシステムを構築することにより、機能面の要件を満たすことに専念できます。

寄稿者

本ドキュメントは、次の人物および組織が寄稿しました。

- AWS SaaS ファクトリ、プリンシパルパートナーソリューションアーキテクト、Tod Golding

改訂履歴

このホワイトペーパーの更新に関する通知を受け取るには、RSS フィードをサブスクライブしてください。

変更	説明	日付
マイナーな更新	わかりやすくするために図が修正されました。	December 11, 2020
初版発行	SaaS レンズが初めて公開されました。	December 3, 2020

注意

お客様は、この文書に記載されている情報を独自に評価する責任を負うものとします。本書は、(a) 情報提供のみを目的としており、(b) AWS の現行製品と慣行ついて説明していますが、予告なしに変更されることがあり、(c) AWS およびその関連会社、サプライヤーまたはライセンサーからの契約上の義務や保証をもたらすものではありません。AWS の製品やサービスは、明示または暗示を問わず、一切の保証、表明、条件なしに「現状のまま」提供されます。お客様に対する AWS の責任は、AWS 契約により規定されます。本書は、AWS とお客様の間で締結される一切の契約の一部ではなく、その内容を修正することもありません。