SaaS のテナント分離戦略

マルチテナント環境でリソースを分離する

2020 年 8 月

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。このドキュメントは、(a) 情報提供のみを目的としており、(b) AWS の現行製品とプラクティスを表したものであり、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義務や確約を意味する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、いかなる保証、表明、条件を伴うことなく「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 とお客様の間で行われるいかなる契約の一部でもなく、そのような契約の内容を変更するものでもありません。

目次

要約

テナントの分離は、Software as a Service (SaaS) システムの設計・開発の基礎です。SaaS プロバイダーは、テナントを分離することで、マルチテナント環境であっても、お客様のリソースに他のテナントがアクセスできない状態を担保することができます。このホワイトペーパーでは、SaaS 企業が SaaS 提供モデルの価値提案を実現しながら、自社のシステムでテナントリソースを確実に分離するために使用している一般的な戦略を紹介します。

はじめに

テナントの分離は、すべての Software as a Service (SaaS) プロバイダーが対応する必要がある基礎的なトピックの 1 つです。独立系ソフトウェアベンダー (ISV) は、SaaS に移行し、共有インフラストラクチャモデルを採用してコストと運用の効率を達成する場合、同時にマルチテナント環境でテナントが別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必要があります。テナント間で相互のリソースに何らかの方法でアクセスできる場合、SaaS ビジネスにとって重大で回復不能なイベントが発生する可能性があります。

テナントの分離は、すべての SaaS プロバイダーに不可欠ですが、この分離を実現する戦略やアプローチはプロバイダーごとに異なる場合があります。SaaS 環境でテナントの分離を実現する方法には、さまざまな要因が影響します。テナントの分離方法には、ドメイン、コンプライアンス、デプロイモデル、AWS のサービスの選択ごとに異なる検討事項が発生します。

このホワイトペーパーでは、AWS でテナントの分離を実装するために使用する一般的なパターンと戦略の多くを紹介します。ここでは、目標として、さまざまな SaaS アーキテクチャモデルや AWS の技術に共通するテーマや課題のいくつかと、これらの各環境でテナントの分離を実現するさまざまなアプローチに焦点を置きます。このホワイトペーパーでは、一連のインサイトを提供し、貴社の環境とビジネスモデルの現状に最も適した分離戦略の組み合わせを選択できるようにします。

分離の考え方

ほとんどの SaaS プロバイダーは、概念レベルでは、テナントリソースの保護および分離の重要性と価値について異論はないでしょう。ただし、分離戦略の実装を詳しく見ると、どの程度の分離を**十分**と考えるかについて SaaS ISV ごとに定義が異な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に共通する原則を以下に示します。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役立つ高レベルの分離要件を明確に確立する必要があります。一般的に SaaS におけるテナント分離モデル全体を構成する重要な原則は以下のとおりです。

分離はオプションではない – 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、テナントリソースを確実に分離する手段を講じる必要があります。

認証や認可は分離と同じではない – SaaS 環境へのアクセスは認証と認可で制御できますが、ログ

イン画面や API のエントリポイントを超えても、分離を達成したことにはなりません。これは分離というパズルを構成する 1 つのピースにすぎず、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開発者に任せるべきではない – 開発者が分離を阻害するコードを組み込むことはないとしても、テナントの境界を意図せずに越えてしまうという可能性はあります。この可能性を軽減するには、リソースへのアクセスのスコープ指定を制御する共有メカニズムを使用して (開発者が意識しないところで) 分離ルールを適用します。

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、テナントの分離を簡単に実現できる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統合することで、分離をシームレスな体験にすることができる場合があります。ただし、分離モデルに直接対応するツールやテクノロジーが存在しない場合もあります。明確なソリューションがないからといって、分離要件を引き下げるのではなく、分離要件に合わせて独自のソリューションを構築する必要があります。

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、具体的なインフラストラクチャリソース間の厳密な境界線を引く手法とする考え方もあります。これは、多くの場合、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、または仮想プライベートクラウド (VPC) がテナントごとに異なる分離モデルとなります。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。リソースを共有する場合 (特にリソースが共有されている環境) でも、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では、ランタイムでポリシーを適用することによって、論理的な構造による分離を実施することができます。ここで重要なのは、分離をリソースのサイロ化と同一視しないことです。

ドメインごとに分離要件が異なる場合がある – テナントの分離を実現するアプローチは数多くありますが、ドメインごとの現実に伴う制約により、分離要件が異なる場合があります。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分離アプローチは適切でないことがあります。

分離をセキュリティ対策とするか、ノイジーネイバー対策とするか

分離のトピックは、多くの場合、区分化が困難です。通常、企業はセキュリティとコンプライアンスの視点から分離について考えます。この場合、分離は、リソース間に境界を作成して、クロステ

ナントアクセスの可能性を制限するために使用します。これは、分離の主要な目的です。ただし、分離は他のテナントからの干渉 (ノイジーネイバー) やパフォーマンスの懸念に対処する一環としても実装されます。これは、テナントのデータを分離する別の理由です。このホワイトペーパーの範囲では、この問題のセキュリティ面により注目しながら、両方の項目を取り上げます。

分離の基本概念

分離の課題の 1 つは、テナントの分離に複数の定義があることです。分離とは概してビジネス構造であり、顧客ごとに独自の環境が必要であるとする考え方があります。一方、分離とは概してアーキテクチャ構造であり、マルチテナント環境のサービスと構造を重ね合わせたものとする考え方もあります。以下のセクションでは、さまざまな種類の分離について説明し、分離構造ごとに特定の用語の意味を考えます。

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。ただし、SaaS プロバイダーは、一部 (またはすべて) のテナントが個別に完全にサイロ化したリソースのスタックを実行するようなモデルを使用したい場合もあります。このフルスタックモデルは SaaS 環境に該当しないと言う考え方もあります。ただし、これらの個々のスタックが共有のアイデンティティ、オンボーディング、メータリング (計測)、メトリクス、デプロイ、分析、運用に関連している場合、これは SaaS の有効なバリエーションであり、スケールメリットおよび運用効率を、コンプライアンス、ビジネス、またはドメインに関する考慮事項とトレードするものと言えます。このアプローチの場合、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 1 は、この分離アプローチの概念ビューを示しています。

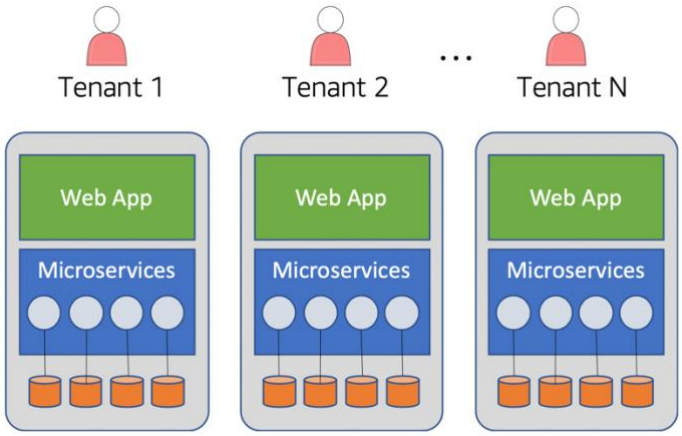

図 *1*- 分離のフルスタックビュー

この図は、サイロ化し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している技術は、ここでは重要ではありません。これは、モノリス、サーバーレス、または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にすることができます。ここでの重要な概念は、テナントが持つスタックのすべてを特定の構造に含めて、そのスタックのすべての変動要素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。完全にカプセル化した環境からテナントが逸脱しないようにできれば、分離を実現したことになります。

一般的に、この分離モデルはより簡単に実施できます。多くの場合、堅牢な分離モデルを実装でき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は達成しにくい場合がありますが、非常に厳格な分離要件に対応する場合は魅力的なモデルとなります。

サイロモデルの長所と短所

SaaS 環境やビジネスドメインごとに異なる一連の要件によっては、サイロが適している場合があります。ただし、サイロモデルを採用する場合は、サイロモデルに伴う課題やオーバーヘッドのいくつかを慎重に検討する必要があります。SaaS ソリューションとしてサイロモデルを使用する場合に検討すべき長所と短所を以下に示します。

長所

- **困難なコンプライアンスモデルのサポート** - 一部の SaaS プロバイダーは、分離要件が厳しい規制環境で販売しています。サイロを使用すると、これらの ISV は、一部またはすべてのテナントに対して専用モデルでデプロイするオプションを提供することが可能になります。

- **ノイジーネイバーの懸念がない** - すべての SaaS プロバイダーはノイジーネイバーの影響を制限することに努めています。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から自分のワークロードが影響を受ける可能性について不安を示します。サイロは、他のテナントからの干渉がない専用の環境を提供することで、この懸念に対処します。
- **テナントコストの追跡** - SaaS プロバイダーは、通常、各テナントがインフラストラクチャコストに与える影響を真剣に把握しようとしています。テナントあたりのコストを計算することは、一部の SaaS モデルでは困難な場合があります。ただし、サイロモデルは性質上大まかな単位で分離されているため、簡単な方法でインフラストラクチャコストを確認し、各テナントと関連付けることができます。
- **影響範囲の縮小** - サイロモデルでは、SaaS ソリューションで機能停止やイベントが発生した場合、一般的に影響範囲が制限されます。SaaS プロバイダーはテナントごとに個別の環境を実行しているため、特定のテナントの環境内で発生した障害は、通常、その環境内に限定されます。1 つのテナントで停止が発生しても、システムを使用している残りのテナントには障害が及ばない場合があります。

短所

- **スケーリングの問題** - プロビジョニングできるアカウントの数には制限があります。この制限により、アカウントベースのモデルを選択できない場合があります。また、アカウント数の急増が SaaS 環境の管理や運用体験を損なう可能性についての一般的な懸念もあります。テナントごとに専用のアカウントを用意する場合、例えば 20 くらいまでは管理可能でも、テナント数が 1,000 にもなると、運用の効率性と俊敏性に影響する可能性が高くなります。
- **コスト** - すべてのテナントを独自の環境で実行している場合、従来の SaaS ソリューションに関連する費用対効果の多くが失われます。これらの環境を動的にスケールした場合でも、アイドル状態のリソースを利用しない期間が発生する可能性があります。これ自体は十分許容可能なモデルですが、組織が SaaS モデルに不可欠なスケールメリットや限界利益を達成する能力は制限されます。

- **俊敏性** - SaaS への移行の直接的な原動力となるのは、多くの場合、イノベーションを加速したいという願望です。つまり、市場力学に迅速に反応して対応できるモデルを組織が採用することを意味します。この場合、重要となるのは、カスタマーエクスペリエンスを一元化し、新しい機能や性能をすばやくデプロイできることです。サイロモデルでは、俊敏性への影響を軽減する対策を講じることはできますが、高度に分散管理されているモデルの性質上複雑さが増し、テナントを簡単に管理、運用、サポートする能力が影響を受けます。
- **オンボーディングのオートメーション** -SaaS 環境では、新しいテナントの導入を自動化することが重要です。これらのテナントのオンボーディングに、セルフサービスモデルを使用する場合でも、内部で管理しているプロビジョニングプロセスを使用する場合でも、オンボーディングを自動化する必要があります。テナントごとに個別のサイロを使用している場合、通常、このプロセスは非常に複雑になります。新規テナントのプロビジョニングには、新しいインフラストラクチャのプロビジョニングが必要であり、新しいアカウント制限の設定が必要になる場合もあります。これらの新しい変動要素がもたらすオーバーヘッドに伴って、オンボーディングのオートメーション全体に新たな次元の複雑さが追加され、顧客に注力できる時間が少なくなります。
- **分散型の管理とモニタリング** - 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を管理およびモニタリングできるようにすることです。この要件は、テナント環境をサイロ化している場合に特に重要です。この場合の課題は、分散化したテナントフットプリントからデータを集約する必要があることです。テナントの集約ビューを作成できるメカニズムはありますが、このメカニズムを構築および管理するために必要な労力とエネルギーは、サイロ化したモデルではより複雑になります。

プール分離

サイロ分離のモデルは、多くの SaaS 企業に適していることが容易にわかります。同時に、SaaS に移行している多くの企業は、基盤となるインフラストラクチャの一部またはすべてをテナント間で共有できることの効率性、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、分離に別のレベルの複雑さをもたらします。図 2 は、プールモデルで分離を実装する場合の課題を示しています。

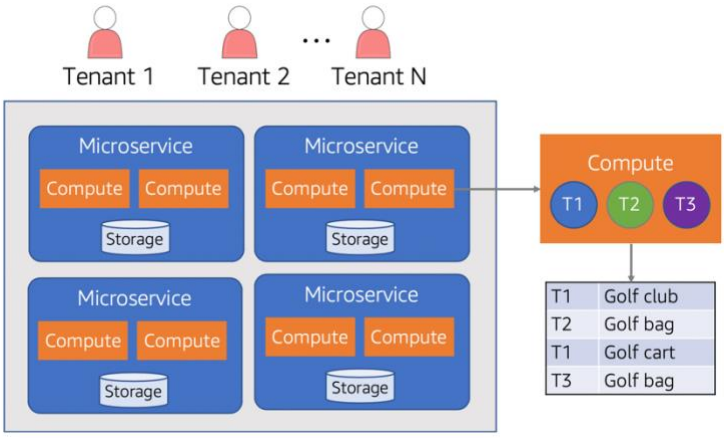

図 2 - プール分離

このモデルの場合、テナントが実際に利用しているインフラストラクチャは、すべてのテナントによって共有されていることがわかります。これにより、リソースは、テナントの実際の利用に伴う負荷に正比例してスケールできます。図の右側は 1 つのサービスのコンピューティングを拡大したものです。テナント 1 から N までのすべてが、共有コンピューティング内で同時に並列実行している可能性があることを示しています。また、この例でストレージも共有していることがわかります。ここでは、個々のテナント ID でインデックス付けしたテーブルを表しています。

このモデルは、SaaS プロバイダーに最適である反面、分離の全体像が複雑になることがわかります。リソースが共有されている環境では、分離の実装が何を指すのかが不明瞭になります。一般的なネットワークと IAM 構造を利用して、テナント間の境界を作成することはできません。

ここで重要な点は、分離の達成が困難な環境であったとしても、それを理由にして環境の分離要件を緩和することはできないということです。これらの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、リソースを分離することに特に入念に取り組む必要があります。

プール分離モデル (上図) を詳しく調べると、このアーキテクチャのフットプリントに特有の複数の課題があることがわかります。テナントのリソースを適切に分離するには、課題ごとに独自のタイプの分離構造が必要です。

プールモデルの長所と短所

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を採用した場合のトレードオフのいくつかを検討する必要があります。多くの場合、プールモデルの長所と短所は、サイロモデルで取り上げた長所と短所の逆になります。プール分離モデルの一般

的な長所と短所は、以下のとおりです。

長所

- **俊敏性** – すべての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、SaaS サービスの俊敏性をもたらす効率性とシンプルさのすべての特性を利用できます。プールモデルの要点は、SaaS プロバイダーが体験を一元化して、すべてのテナントを管理、スケーリング、運用できるようにすることです。体験の一元化と標準化は、SaaS プロバイダーがテナントごとに 1 回限りのタスクを実行することなく、すべてのテナントへの変更を簡単に管理および適用できるようにするための基礎です。この運用効率は、SaaS 環境の俊敏性のフットプリント全体の鍵となります。
- **コスト効率** – 多くの企業は SaaS の高いコスト効率に魅力を感じています。このコスト効率に関しては、プール分離モデルによるところが大きいです。プール化した環境では、すべてのテナントの実際の負荷とアクティビティを反映して、システムがスケールされます。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のコストは最小限になります。ここでの重要な概念は、プール化した環境はテナントの負荷に合わせて動的に調整できるため、テナントのアクティビティに合わせてリソースを適切に使用できることです。
- **管理と運用の簡素化** – 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 1 つのビューで確認できます。一元化したビューで、システム内のすべてのテナントを管理、更新、デプロイできます。これにより、管理と運用のフットプリントのほとんどの側面が簡素化されます。
- **イノベーション** – プール化した分離モデルがもたらす俊敏性を活用して、SaaS プロバイダーのイノベーションを迅速化できます。分散型の管理や、サイロモデルの複雑さから解放されるほど、製品の特徴と機能に集中できるようになります。

短所

- **ノイジーネイバー** – リソースを共有するほど、特定のテナントが他のテナントの体験に影響を及ぼす可能性が高くなります。例えば、特定の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が及ぶ場合があります。マルチテナントの適切なアーキテクチャや設計では、これらの影響を抑制しようとしますが、プール化した分離モデルでは、常に他のテナントからの干渉が他の複数のテナントに影響を及ぼす可能性があります。

- **テナントコストの追跡** – サイロモデルでは、リソースを利用したテナントを簡単に特定できます。一方、プールモデルでは、リソースを利用したテナントを特定することが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詳細なデータを取得し、リソースの使用量と特定のテナントを効果的に関連付ける必要があるため、プロバイダーの作業量が増大します。
- **影響範囲** – すべてのリソースを共有すると、運用上のリスクも生じます。サイロモデルでは、あるテナントに障害が発生しても、通常、その障害の影響はそのテナントに限定されます。一方、プール化した環境では、機能停止がシステムのすべてのテナントに影響する可能性があります。これは、ビジネスに多大な影響を与える場合があります。このため、障害を特定して顕在化させ、障害からスムーズに復旧できるよう、回復性の高い環境を構築するためのより徹底した取り組みが必要となります。
- **コンプライアンスの反動** – プールモデルで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が、このモデルでの実行を顧客に躊躇させる場合があります。特に、ドメインのコンプライアンスや規制のルールによって、リソースのアクセシビリティと分離が厳格な制約を受けるような環境では、これが顕著になります。ただし、このような場合でも、システムの一部をサイロ化するだけで済むことがあります (以下のブリッジモデルを参照)。

ブリッジモデル

サイロとプールは分離に対して明確に異なるアプローチを取りますが、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に注目し、システムをより小さいサービスに分解すると、多くの場合、サイロモデルとプールモデルを混合したソリューションが必要であることに気付きます。この混合モデルを分離のブリッジモデルと呼びます。図 3 は、SaaS ソリューションでブリッジを実現する方法の例を示しています。

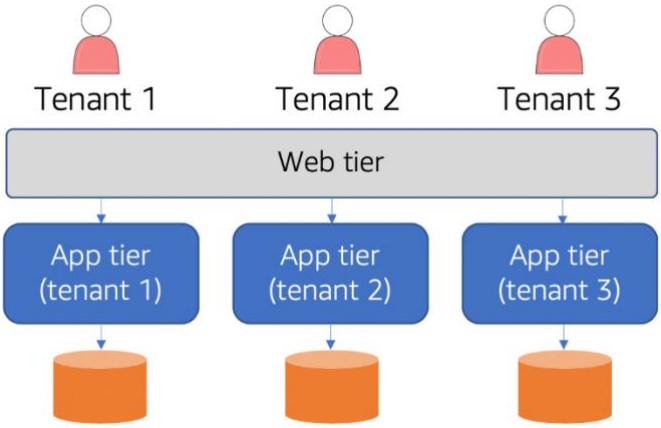

図 *3* - ブリッジ分離モデル

この図は、ブリッジモデルによってサイロモデルとプールモデルを組み合わせる方法を示しています。これは、従来のウェブ層とアプリケーション層を備えたモノリシックアーキテクチャです。このソリューションのウェブ層は、すべてのテナントが共有するプールモデルでデプロイされています。ウェブ層は共有していますが、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で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。

ここで、このモノリスを複数のマイクロサービスに分割するとします。システム内のさまざまなマイクロサービスごとに、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。この点に関しては、AWS の構成ごとにサイロとプールを適用する方法を説明するときに、より詳しく掘り下げます。ここで重要な点は、分離要件が異なる複数のサービスに分解した環境では、サイロとプールの見え方がより詳細になることです。

ブリッジモデルの長所と短所

ブリッジモデルは、サイロモデルまたはプールモデルを適切に使い分けることに重点を置いたハイブリッドモデルです。ここでの考え方は、サイロ分離の価値と原則が引き続きシステムの該当する各領域に適用されるということです。ブリッジモデルの長所と短所について考える場合、アーキテクチャのリソースやレイヤーごとに、サイロモデルとプールモデルのトレードオフを考えることになります。

階層 (ティア) 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仕組みに焦点を当てていますが、サービスの階層化が分離戦略に影響を与える場合もあります。この場合は、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テナントごとに異なるタイプの分離をどのようにパッケージ化して提供するかが重要となります。また、これは、幅広い顧客層に対応するために、どの分離モデルが必要かを判断する際の考慮事項ともなります。図 4 は、階層間で分離がどのように異なるかを示しています。

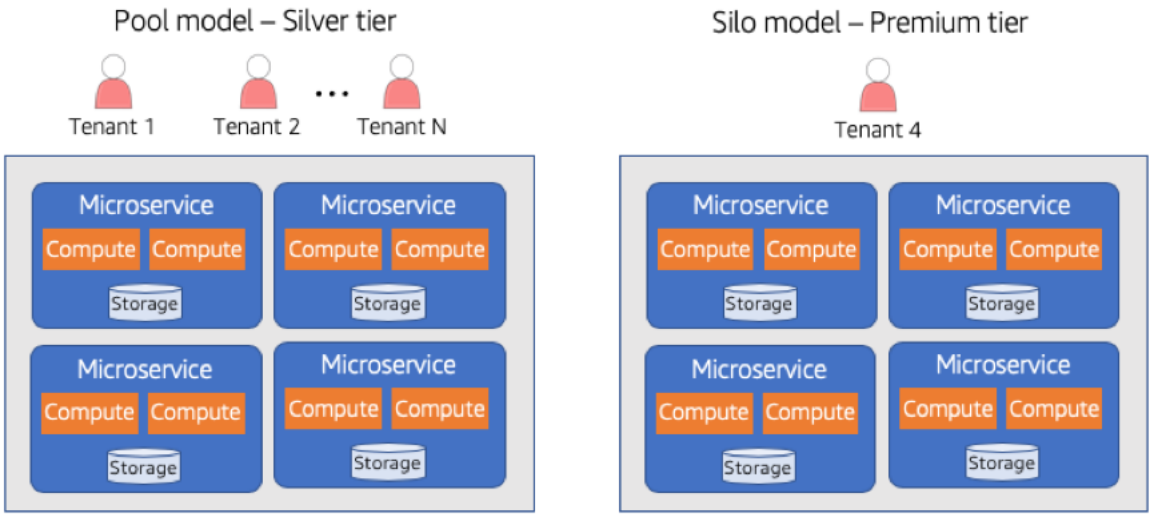

図 4 - テナントの階層化と分離

ここでは、それぞれをティアとしてサイロ分離モデルおよびプール分離モデルの両方をテナントに提供しています。シルバーティアのテナントは、プール化した環境で実行しています。これらのテナントは共有インフラストラクチャモデルで実行していますが、自分のリソースがクロステナントアクセスから完全に保護されることを期待できます。右側のテナントは、完全な専用 (サイロ) 環境の提供を要求しています。それをサポートするため、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。これは非常に高価になることが想定されます。

SaaS プロバイダーは一般的に顧客へのサイロモデルの提供を制限しようとしています。また、多くの SaaS ビジネスでは個別の料金設定を採用し、このモデルでのデプロイにテナントがプレミアム料金を支払うようにしています。実際、SaaS 企業はこの方式を選択する顧客の数を制限するために、このオプションを明示的に公開したりティアとして位置付けることはしないでしょう。この

モデルに該当するテナントが多すぎると、完全にサイロ化したモデルに戻り、上で説明した課題の多くを継承することになります。

これらの個別の環境の影響を制限するために、SaaS プロバイダーは通常、これらのプレミアム顧客に対して、プール化した環境にデプロイするのと同じバージョンの製品を実行するよう要求します。これにより、ISV は引き続き 1 つの画面で両方の環境を管理および運用することができます。基本的に、サイロ環境は、1 つのテナントだけをサポートしているプール環境のクローンと考えることができます。

アイデンティティと分離

分離に限定して説明を進めていますが、アイデンティティが分離モデルにどのように関連するかを検討することも重要です。実際には、テナントを分離する場合、SaaS 環境のリソースに現在アクセスしているテナントを表現および特定する方法が必要です。多くの場合、分離スキームの中核であるポリシーおよびスコープ指定ルールを取得するには、アイデンティティと他の構造を組み合わせて使用します。これらのポリシーを定義および適用する方法は、使用する分離モデルやサービスごとに異なります。ただし、このアプローチの基本は、通常、図 5 に示すようなパターンに従います。

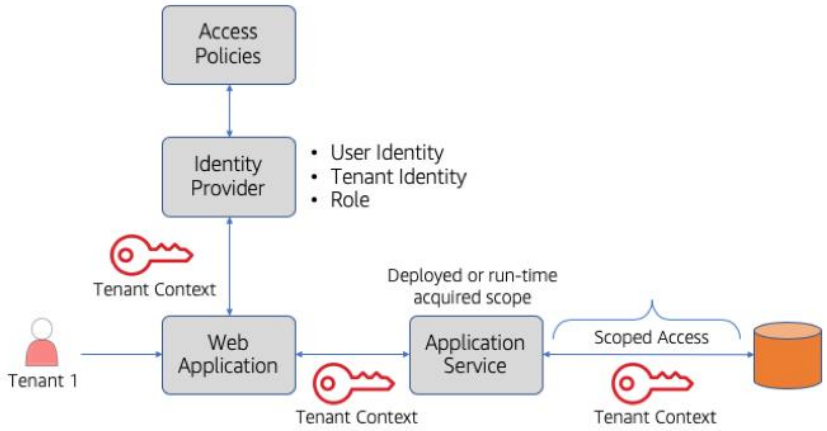

図 5 - アイデンティティと分離の関連性

この図は、アイデンティティが分離を考える上でどのくらい広範囲に関連しているかを一般化して示しています。ここでは、ユーザーが認証されると、システムはテナントのコンテキストをアプリケーションに返します。このコンテキストには、テナントへのユーザーのバインドと、このテナントに分離を適用するために使用するポリシーが含まれています。次に、このコンテキストは、すべ

てのやり取りを通過し、SaaS 環境のダウンストリーム要素によってリソース (この場合はデータベース) へのアクセスをスコープ指定するために使用されます。
このスコープを取得および適用する方法は、使用する分離モデルとリソースによって異なりますが、このモデルから主要な概念を確認できます。方法によって異なる 1 つの重要な要素は、テナントのスコープ指定をどのように決定するかです。このスコープ指定コンテキストは、デプロイ時にサービスにアタッチすることも、実行時に取得することもできます。アーキテクチャモデルごとの分離特性を詳しく掘り下げる際に、これらの両方のモデルについて検討します。

サイロ分離の実装

分離の概念が明確になったので、次に、これらの各モデルを AWS のサービスや構造ごとに実現する方法に注目します。以下のセクションでは、SaaS アーキテクチャのレイヤーごとにサイロ分離を使用するさまざまな方法について説明します。

フルスタック分離

最初に検討するモデルは、フルスタックのサイロ分離です。このシナリオの場合、SaaS ISV はテナントのすべてのリソースを完全に分離する必要があり、より大きな単位での AWS の構成を使用してテナントを分離します。ここでの選択は、環境の管理、運用、スケーリングのプロファイルの影響を受ける傾向があります。一般的なフルスタックのサイロ分離メカニズムの内訳を以下に示します。

アカウントのサイロ分離

AWS アカウントは、サイロモデルでテナントを分離するために使用できる 1 つの戦略を表します。ここでの基本的なアプローチは、各テナントを完全に別個のアカウントにデプロイし、各テナントアカウントを親アカウントにリンクすることです。各テナントスタックのすべての変動要素は、スタックと共にデプロイされ、完全に分離して運用されます。このアカウントによる分離の境界は、通常、テナントの境界を最も楽な方法で記述する必要がある場合に魅力的です。SaaS プロバイダーにとって、アカウント分離モデルは、クロステナントアクセスの可能性を特に懸念している顧客向けの説得力のあるオプションとなります。

アカウントベースのサイロ分離モデルを使用するサンプルアーキテクチャを詳しく見てみましょう。図 6 は、2 つの別個のアカウントにデプロイした 2 つのテナントを示しています。

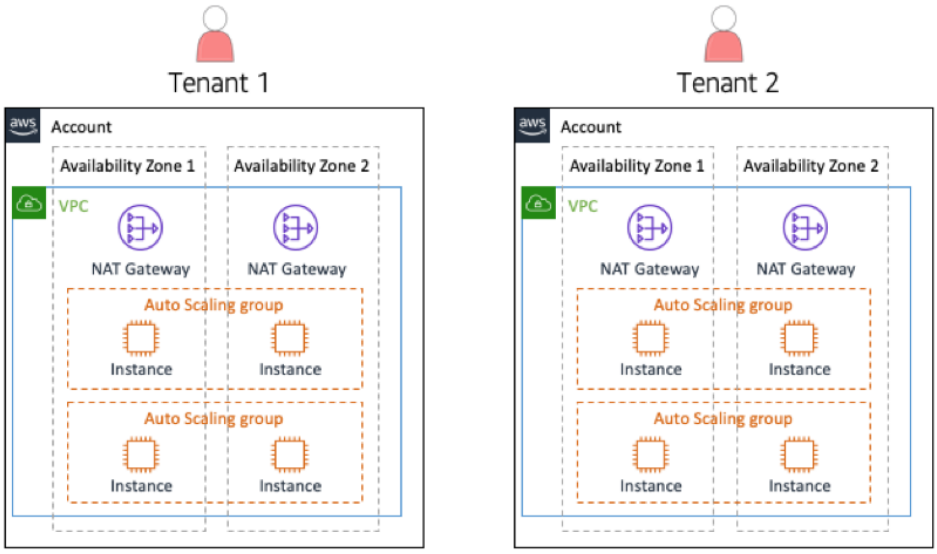

図 6 - アカウントベースのサイロと分離

ここに示すアーキテクチャは単なる例です。アカウントに実際に配置されるインフラストラクチャは、SaaS アプリケーションのスタックを構成する技術によって異なります。このテクノロジーは、コンテナを使用する場合、サーバーレスである場合、またはさまざまな AWS アーキテクチャモデルの任意の組み合わせである場合があります。ここで重要な点は、すべてのテナントがこれらの個別のアカウントで同じスタックを実行していることです。

このモデルにはメリットがありますが、スケーリングと自動化に関しては大きな課題があります。この体験を自動化できる豊富なツールのコレクションがあります。ただし、この自動化には制約があります。ここで重要な課題は、多くの場合、アカウントの制限です。自動化で設定できる制限もありますが、設定できない制限もあります。アカウントごとに別個の制限を管理および維持する場合、この問題はより複雑になります。

Virtual Private Cloud (VPC) のサイロ分離

次に検討すべきレベルの分離は、単一のアカウント内での分離です。これにより、ネットワーク構造の領域に入り、サイロ化したテナント環境ごとにネットワークの境界を基本的に使用します。Amazon Virtual Private Cloud (Amazon VPC) は、ネットワークベースの分離のための自然なメ

カニズムを提供します。図 7 は、VPC サイロモデルのサンプルを示しています。

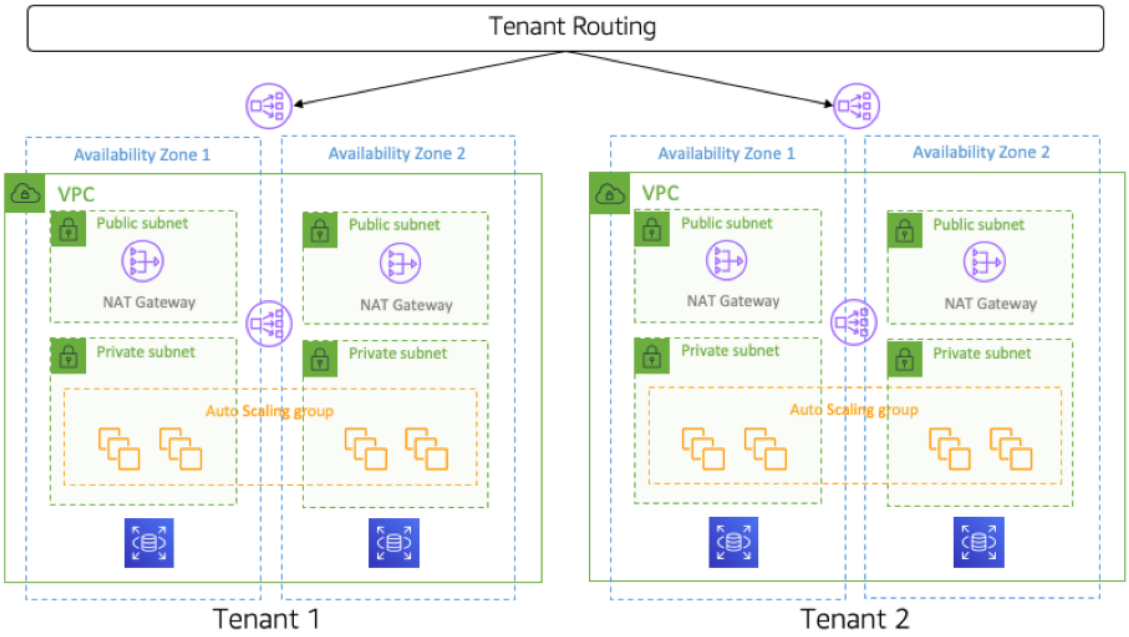

図 7 - VPC サイロ分離

ここでは、2 つの別個のテナント環境があり、各環境を独自の VPC でホストしています。ここでの VPC は、複数のアベイラビリティーゾーンを使用して AWS アーキテクチャの一般的なベストプラクティスを示しています。アカウントと同様に、各 VPC で設定するリソースは SaaS プロバイダーごとに大きく異なる場合があります。このソリューションの重要な点は、VPC が強制するネットワーク構造を介してテナントを相互に分離していることです。

VPC は、サイロ化した分離を必要とするユーザーに魅力的なモデルを提供します。制限を管理しているのは、これらの VPC が存在するアカウントであるため、アカウントベースの分離に伴うプロビジョニングの課題の一部は克服されます。これにより、すべてのテナントにわたって制限を一元的に管理するモデルが作成されますが、制限の課題がなくなるわけではありません。VPC モデルでは、制限を事前にモニタリングし、必要に応じて定期的に制限を増やす必要があります。また、テナント数が多いために環境のハードリミットを超える場合もあります。このモデルには、VPC とそのリソースにタグを付けてテナントのコストを計算できるというメリットがあります。

このモデルはアカウントベースの分離に優る利点がありますが、スケーリングについては引き続き検討が必要です。VPC の数が増えると、このアプローチ (およびすべてのサイロモデル) の管理と俊敏性が低下します。全体としては、サイロモデルに依存する必要がある企業にとって、VPC は最適なオプションの組み合わせを提供すると言えます。

サブネットのサイロ分離

サイロモデルとして、アカウントから始めて VPC に進みました。サイロ分離をさらにきめ細かく進めて、テナントを別個の VPC サブネットに配置できます。このアプローチでは、各テナントを VPC 内の別個のサブネットに配置します。図 8 は、テナントごとのサブネットモデルの例を示しています。

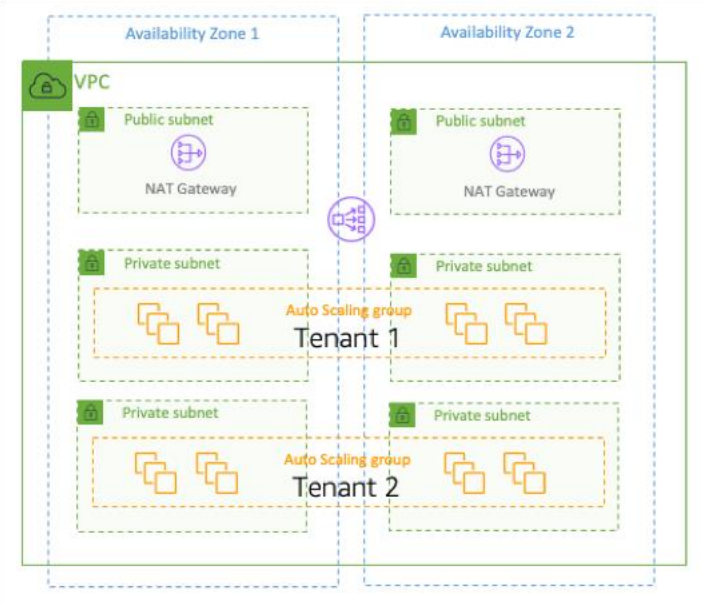

図 8 - サブネットのサイロ分離

ここでは、VPC の分離と同じマルチ AZ の VPC を使用しています。ただし、実際には、すべてのテナントが単一の VPC のサブネット内にあることがわかります。このモデルでのテナント分離は、クロステナントアクセスを防止するために、ネットワークルーティング構造に依存しています。

これは有効なアプローチですが、あまり推奨するモデルではありません。このモデルのスケーリングと管理は、比較的すぐに手に負えなくなります。テナント数が少ない場合は、正常に機能する可能性があります。テナント数が多い場合は、ここで説明する他のサイロ分離モデルのいずれかを使用することをお勧めします。

ターゲットを絞ったサイロ分離

前述したように、サイロ分離はよりきめ細かい方法で適用し、SaaS ソリューションの限定した要素をサイロモデルにデプロイできます。システムの各マイクロサービスおよびこれらのサービスが

使用する各リソースは、サイロ分離モデルで設定することができます。サイロ分離を実現する方法は、アプリケーションを構成するサービスや構造ごとに異なります。サンプルのマイクロサービスをいくつか見ていきましょう。これらの特質の異なるサイロ分離を実際のアプリケーションでどのように実現するかをより良く理解できます。図 9 は、これら 2 つのモデルのビューを示しています。

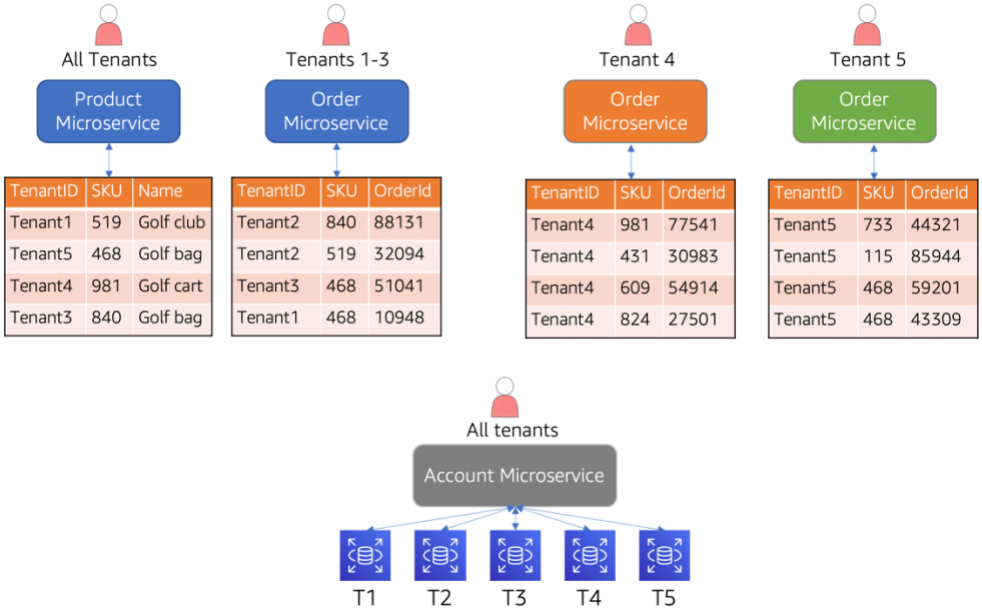

図 9 - マイクロサービスのサイロ分離

この図では、製品、注文、アカウントという 3 つの異なるマイクロサービスを実装しているシステムを示しています。これらの各マイクロサービスのデプロイモデルおよびストレージモデルは、SaaS 環境で分離 (セキュリティ対策またはノイジーネイバー対策) をどのように実現しているかを示しています。

これらの各サービスの分離モデルを見ていきましょう。左上の**製品**マイクロサービスは、完全なプールモデルで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よって共有されています。ここでは、すべてのテナントが同じテーブル内のインデックス付けされた個別の項目となっています。これらのデータを分離するには、このテーブルのテナント項目へのアクセスを制限できるポリシーを使用することを、ここでは前提としています。この項目の右側にある**注文**マイクロサービスは、テナント 1～3 のみを対象としています。このマイクロサービスも、プールモデルで実装しています。ここでの唯一の違いは、テナントのサブセットをサポートしていることです。基本的には、**注文**マイクロサービスの専用 (サイロ) デプロイ以外のあらゆるテナント

は、このプール化し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取り出した少数のテナントを除いては、テナント 1～N と考えます)。

この点をわかりやすくするため、サイロ化したサービスを示す専用の**注文**マイクロサービス (右上) と、**アカウント**マイクロサービス (下) に注目しましょう。テナント 4 と 5 に対して、**注文**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、ノイジーネイバーなど) 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るためです。コンピューティングとストレージは両方とも、完全にこれらの各テナント専用です。

最後に、**アカウント**マイクロサービスが下部にあります。これは、ストレージレベルでのみサイロモデルになっています。この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が共有していますが、各テナントにはアカウントデータを保持する専用の**データベース**があります。このシナリオでは、分離に関する懸念はデータの分離のみに限定されます。コンピューティングは依然として共有できます。

このモデルを見ると、サイロに関する議論がどのように細分化していくかがわかります。セキュリティ、ノイジーネイバー、およびさまざまな要因によって、どのような場合にどのような方法でサイロ分離モデルをサービスで採用するかが決まります。ここでの重要なポイントは、サイロモデルは採用するか、採用しないかの二者択一ではないということです。サイロは、アプリケーションの特定の**コンポーネント**に適用することを検討できます。サイロを実際に必要とするコンポーネントに限り、サイロの課題に対処すれば済みます。例えば、見込み顧客がサイロを要求している場合があります。ただし、よく話し合ってみると、見込み顧客が懸念しているのはストレージと処理の特定の領域に限られていることがわかります。これにより、サイロ分離を必要としないシステム部分にはプールモデルを採用し、このモデルの効率性を活用できます。また、これをティアとしてさまざまなテナントに提供することで、個々のサービスに合わせてサイロとプールの両方を組み合わせて使用することもできます。

サイロモデルのコンピューティングに関する考慮事項

アプリケーションのコンピューティングリソースをサイロモデルで分離する場合 (上図のマイクロサービスなど)、さまざまなコンピューティングサービスの分離モデルが手法に与える影響について検討する必要があります。AWS コンピューティングサービスごとの固有の属性により、リソースを適切に分離するために特定の対策が必要になる場合もあります。

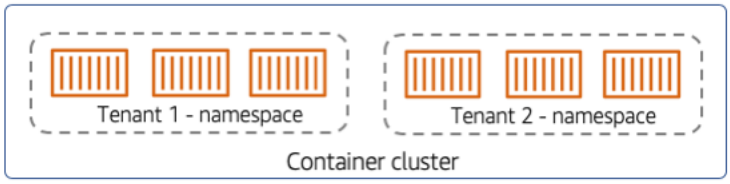

図 10 - コンテナのサイロ分離

コンテナを使用してサイロモデルを実装するとはどういうことであるかをまず考えましょう。コンテナを分離する際に課題となるのは、悪意のあるコードや設定の不十分な環境がコンテナを**エスケープ**して特権を獲得し、あるテナントが別のテナントのリソースにアクセスできるようになる場合があることです。さいわい、コンテナが提供する構造を適切に使用すると、堅牢な分離モデルを実装できます。クロステナントアクセスの防止に使用するメカニズムは、AWS のコンテナサービスごとに異なる場合があります。例えば、Amazon Elastic Container Service (Amazon ECS) では、サイロ分離を達成するために、テナントごとに個別のクラスターを作成する必要があります。Amazon Elastic Kubernetes Service (Amazon EKS) には、EKS クラスター内でリソースをサイロ化するためのいくつかの追加のメカニズムがあります。図 10 は、EKS クラスター内でサイロ分離を達成する方法を示しています。

この例では、EKS クラスター内に 2 つのテナントグループがあり、EKS 名前空間を使用してこれらのコンピューティングリソースを分離しています。名前空間はサイロ分離の基盤を提供しますが、名前空間だけでは完全な分離を達成できません。完全な分離を達成するには、AWS やパートナーのサイドカーソリューションの 1 つを使用して、コンテナ間のフローをさらにロックダウンすることを検討する必要があります。AWS App Mesh と Tigera の Calico は、この分離の追加レイヤーを達成するために使用できる 2 つのソリューションの例です。

また、AWS Lambda がサイロ分離モデルを複雑にする場合もあります。Lambda 関数について考えると、1 つのテナントが関数を実行できるのは特定の時点に限られるため、既に分離が達成されていると見なすことができます。ただし、すべてのテナントをサポートする実行ロールとともに Lambda 関数がデプロイされている場合は、この関数が別のテナントに属するリソースにアクセスできる可能性があります。後続の章で説明するプールモデルの方法を使うと、これを回避できます。また、完全にサイロ化された Lambda 関数を使うことで、他のテナントによる実行は起こりません。図 11 の図は、Lambda モデルで完全な分離を達成する方法の例を示しています。

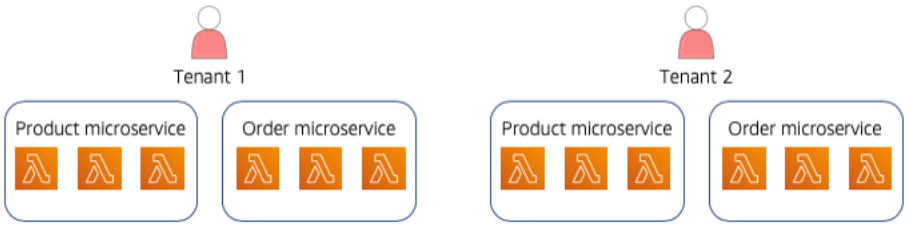

図 11 - Lambda のサイロ分離

この図には、Lambda のサイロモデルでデプロイした 2 つの異なるテナントが含まれています。テナントがテナントの境界内に留まるようにするため、テナントごとに別々の関数をデプロイしています。これらの関数の設定とデプロイに使用したテナント固有のロールは、テナントに関連付けられたリソースへのアクセスを制限します。

このアプローチには長所と短所があります。魅力的な分離方法ですが、煩雑となり、Lambda サービスの制限を超える可能性があります。1,000 件のテナントがあり、テナントごとに個別の関数を管理およびデプロイすることを想像してみてください。管理は困難となり、SaaS の目標の俊敏性が損なわれることでしょう。一方、このオプションを特定のプレミアムテナントのコレクションに提供し、このモデルのさらなる拡張を制限すれば、より合理的に管理および運用できます。

ここで重要な点は、サイロモデルの実装方法を検討する場合、AWS のコンピューティングサービスごとにサイロモデルをどのように実現するかを検討する必要があるということです。サイロ分離の戦略は、サービスごとに異なる場合があります。

あらゆるリソースのサイロモデル

いくつかの重要なサイロモデルに着目しましたが、SaaS アーキテクチャのどのリソースでも、たいていはサイロモデルにデプロイできることが明らかです。ただし、AWS のサービスごとに分離の戦略とメカニズムが異なる場合があることも明らかです。例えば、Amazon DynamoDB のデータを分離する場合は、テナントごとに別個のテーブルを作成することになります。他のストレージモデルで分離する場合は、テナントごとに別個のクラスターの作成が必要になることがあります。Amazon Simple Queue Service (Amazon SQS) と Amazon EventBridge でも、分離を実現する方法は異なります。これらの各サービスの分離戦略については、このホワイトペーパーで扱う範囲を超えていますが、SaaS 開発者は、これらの各サービスをサイロ化する方法と場所について検討することが重要です。

ここでの一般的な経験則は、リソースを確認し、そのリソースに関するノイジーネイバーとセキュ

リティのプロファイルを評価することです。これらの分離オプションは、システムにサイロ戦略を採用することで増える複雑さ、コスト、運用上の負荷、およびサービスの制限と照らし合わせて検討する必要があります。多くの場合、適切なバランスを見つけることが、システムの成功の鍵となります。

プール分離の実装

多くの場合、プールモデルは SaaS プロバイダーにとって最も魅力的です。プールの効率性、俊敏性、コストのプロファイルは、このモデルの採用をプロバイダーに促す大きな要因となっています。無論、リソースを共有モデルに移行することに伴う分離の課題も複雑になります。多くの場合、分離を提供するツールやメカニズムと、テナントが共有リソースを使用することの間には、根本的な不一致があります。これをさらに複雑にするのは、プールモデルで分離するリソースごとに分離を適用するアプローチが異なることです。これらの課題は現実であり、これを分離要件を緩和する口実に**しない**ことが重要です。より多くの努力をして、プールモデルでリソースを分離するためのツールと構造の適切な組み合わせを見つける必要があります。

特定のプール分離手法を詳しく検討する前に、プールモデルに伴って分離へのアプローチがどのように変わるかを確認しましょう。一般的に、AWS のリソースを分離する場合は、AWS Identity and Access Management (IAM) を使用してリソース間のやり取りを制御する方法に注目します。サイロモデルの場合、実際、IAM はテナントの分離ポリシーを示す完璧なモデルとなります。ただし、プールモデルでは、これらの IAM 構造の使用が少し複雑になることがあります。図 12 は、サイロとプールでは分離について異なる考え方が必要であることを示しています。

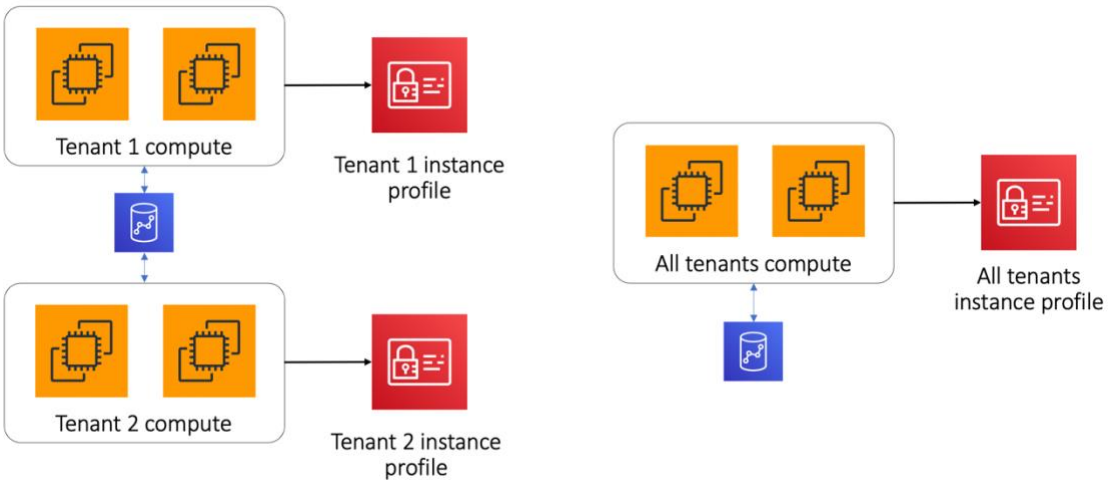

図 12 - IAM とアクセスのスコープ指定

ここでは、2 つの異なる方法で IAM ポリシーを適用し、コンピューティングリソースへのアクセスをスコープ指定しています。左側には、2 つのサイロ化したデプロイがあり、各テナントが独自のインフラストラクチャで実行しています。これらのテナントは、両方とも他のリソース (この例ではストレージ) にアクセスしています。これらのインスタンスは、デプロイしたときに、テナントごと (テナント 1 とテナント 2) に個別の IAM インスタンスプロファイルで設定しています。このバインドはデプロイ時に作成しているため、これらのインスタンスは別のテナントのリソースにアクセスできないようになっています。

右側は、プールモデルでコンピューティングノードをデプロイした例です。ここで実行しているコンピューティングは、すべてのテナントのために実行しています。この状況は、ここでデプロイしているコンピューティングの IAM プロファイルをどのようにスコープ指定できるかに直接影響します。コンピューティングを特定のテナントに制限する代わりに、すべてのテナントをサポートするために十分広域なプロファイルを使用して、これらのコンピューティングノードをデプロイする必要があります。プールモデルの真の課題に直面するのは、この広域なスコープの部分です。ここで、SaaS ソリューションで適用するアクセスのスコープ指定を実装する新しい方法が必要になります。このプール分離に特有である側面に注目すると、プール分離を実装するオプションは大きく異なることがわかります。プール分離のすべての組み合わせを検討するのはこのホワイトペーパーで扱う範囲を超えていますが、一般的なパターンを検討することで、通常適用する戦略について理解を深めることができます。以降のセクションでは、これらの戦略の概要を示します。

IAM を使用したポリシーベースのランタイム分離

プール化した環境の場合、通常、SaaS プロバイダーは IAM を使用してリソースを分離する戦略を見つけます。ただし、前述したように、プールモデルで分離を達成するには IAM を適用する方法について独自の考え方が必要です。コンピューティングノードの IAM スコープ指定を継承する代わりに、独自のコードを導入して、プール化した分離モデルをランタイムに適用する必要があります。図 13 は、このモデルの概念図を示しています。

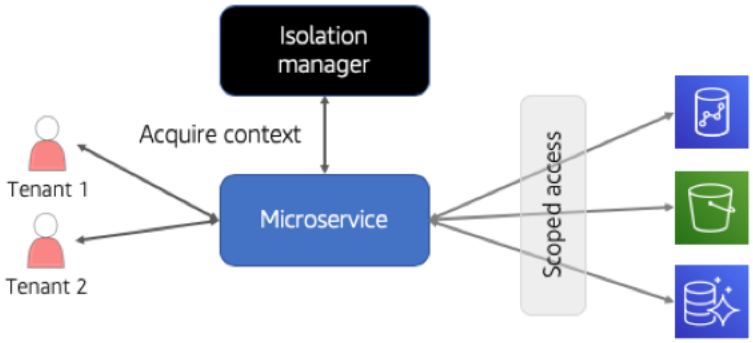

図 13 - ランタイムに取得したスコープ指定

この図では、マイクロサービスがいくつかのダウンストリームリソース (データベース、S3 バケットなど) にアクセスする必要があります。このマイクロサービスは、プールモデルでデプロイしています。つまり、複数のテナントからのリクエストを処理します。このマイクロサービスの役目は、これらのリクエストを処理する際に制約を適用し、テナントが別のテナントのリソースとの境界を越えないようにすることです。この図では、マイクロサービスが分離マネージャー (Isolation manager) を呼び出してスコープ指定コンテキストを取得し、このコンテキストを使用して、このマイクロサービスでコードを実行してアクセスするリソースとのやり取りを制御します。

この概念モデルは、変動要素のビューを示しています。ただし、これを実際に確認するには、このコンテキストがどのように表現され適用されるかを説明する、より具体的な戦略を見る必要があります。図 14 は、テナントリソースにアクセスする際に、ランタイムスコープを指定する一部として IAM を使用する方法を詳しく示しています。

ランタイムモデルでポリシーを設定および適用するライフサイクル全体を確認できます。このプロセスの最初のステップでは、テナントがシステムにオンボードします。このプロセス中に、テナントのユーザーと、そのテナントの IAM ポリシーをセットアップします (ステップ 2 とステップ 3)。テナントがオンボードしたら、アプリケーションのマイクロサービスを呼び出します (ステップ 4)。このマイクロサービスはプールモデルで実行しているため、すべてのテナントのリソースにアクセスできる幅広い IAM スコープでデプロイしています。次に、このサービスに送信される各リクエストを確認し、そのリクエストのスコープを 1 つのテナントに絞り込みます。そのためには、現在のテナントに固有の一連の認証情報を分離マネージャーに要求します (ステップ 5)。この分離マネージャーは、テナントの IAM ポリシーを検索し (ステップ 6)、テナントのスコープ指定された一連の認証情報を生成して、呼び出し元のマイクロサービスに返します。最後に、このマイクロサービス

はこれらの認証情報を使用してデータベースにアクセスします (ステップ 7)。これらのテナントのスコープ指定された新しい認証情報により、マイクロサービスのコードが別のテナントのリソースにアクセスするのを防ぐことができます。

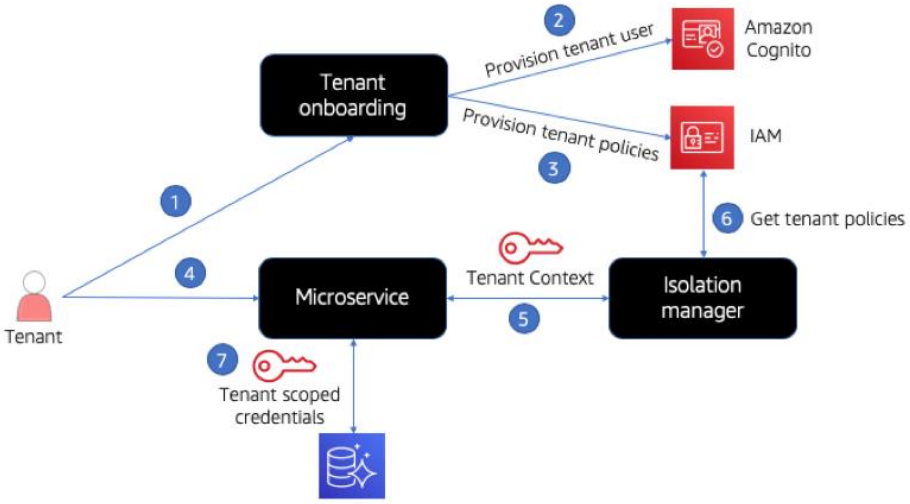

図 14 - IAM ポリシーを使用したスコープ指定

このモデルでは、基本的に、マイクロサービスが別のリソースにアクセスしようとするたびに、このテナントコンテキストが適用されるということです。このスコープ指定は、暗黙的な定義として適用され、マイクロサービスはテナントリソースにアクセスする前に常に新しい認証情報を取得する必要があります。

プール分離ポリシーのスケーリングと管理

IAM ポリシーは強力な分離構造を提供しますが、SaaS プロバイダーにスケーリングの課題ももたらします。システムに多数のテナントがあり、ポリシー数も多い場合は、IAM サービスの制限を超えることがあります。また、テナントの数やポリシーの複雑さが増大するに従って、これらのポリシーを管理することが難しくなる場合もあります。このような状況では、IAM ポリシーをランタイムに生成および管理する方法へのアプローチを変えようとする SaaS 企業も出てきます。

この課題に対する 1 つのアプローチは、IAM ポリシーを実行時に生成するモデルに移行することです。ここでは、呼び出しの現在のコンテキストを調べ、必要な IAM ポリシーをオンザフライで生成するメカニズムをシステムで実装します。これにより、IAM からポリシーが削除され (一時的であるため)、すべてのテナントをサポートするために必要なポリシー数に対する潜在的な制限に対処で

きるようになります。図 15 は、このポリシーを動的に生成するメカニズムの概要を示しています。

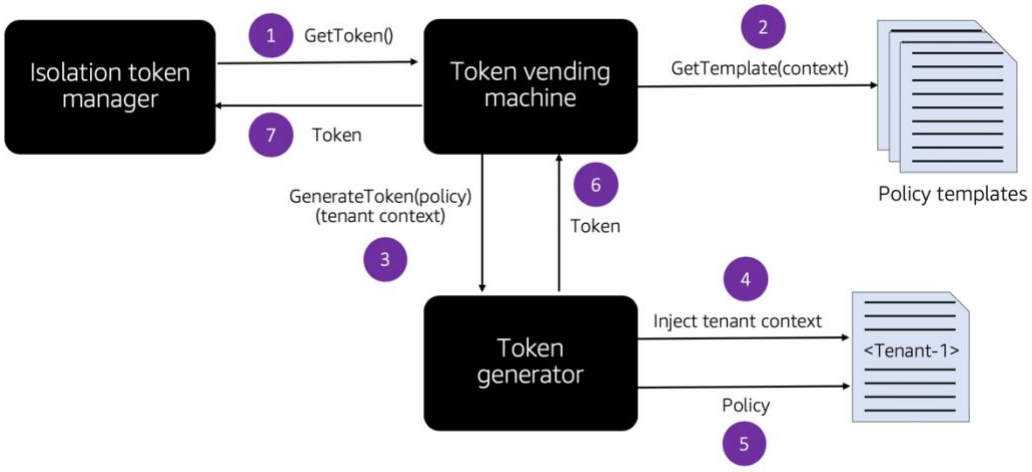

図 15 - ポリシーの動的生成

このフローでは、前の例で使用したのと同じ分離マネージャーから開始します。ただし、IAM に直接移動してアクセスをスコープ指定するポリシーを取得する代わりに、一連のステップを使用してポリシーを生成します。分離マネージャー (Isolation token manager) は、最初にトークン発行機 (Token vending machine) にリクエストを行い、テナントのスコープ指定されたトークンを取得します (ステップ 1)。トークン発行機は、テナント分離モデル用に事前に定義済みのテンプレートを参照します (ステップ 2)。これらは、従来の IAM ポリシーのすべての変動要素を含むテンプレートファイルと考えてください。ただし、ファイルのキー要素 (テナントのコンテキストを表す要素) は入力されていません。例えば、Amazon DynamoDB テーブルのテーブル名やリーディングキー条件 (dynamodb:LeadingKeys) にテナント識別子を入力できます。

必要なテンプレートを取得したら、トークン生成機 (Token generator) を呼び出してトークンをリクエストします (ステップ 3)。このステップでは、現在のテナントコンテキストも提供します。次に、トークン生成機はテナントの詳細をテンプレートに入力し、完全に一体化された IAM ポリシーを提供します (ステップ 4 とステップ 5)。最後に、トークン生成機はこのポリシーを使用して、指定されたポリシーに従ってスコープ指定されたトークンを生成します。このトークンが分離マネージャーに返されます (ステップ 6 とステップ 7)。これで、このトークンを使用して、テナントコンテキストが適用されたリソースにアクセスできます。

これらのポリシーをテンプレート内に移動することで、これらのポリシーがテナントの分離要件を確実に実施するという責任を負うことになります。理想的には、このメカニズムの詳細の大半は開発者の目に触れないところにあるため、問題が発生する可能性は低くなります。

ここでの利点は、このモデルの管理プロファイルです。分離ポリシーの一部を変更することを選択した場合、ポリシーはテナントごとに分かれていないため、これらの変更は簡単に適用できます。これにより、これらのポリシーテンプレートのコンテンツライフサイクルの所有 (独自のパイプラインを使用したバージョニングとデプロイ) が可能になります。

プール化したストレージの分離戦略

プールモデルでのデータ分離は、SaaS プロバイダーから多くの注目を集めている分野です。データが混合している場合、SaaS 開発者は各テナントのデータを確実に保護する方法を特定することを非常に重視します。実際、多くの SaaS プロバイダーはプールモデルのコスト、管理、俊敏性のプロファイルに強い関心を寄せていながら、多くの場合、サイロモデルを選択する結果となっています。この唯一の理由は、データのプール化を受け入れることをためらう顧客からの予想されるプッシュバックに対処することにあります。

プールストレージ分離の一般的な概念 (ストレージサービスを問わず) は、すべてのテナントのデータを共有ストレージ構造で表現することです。図 16 は、プール化したストレージを示しています。

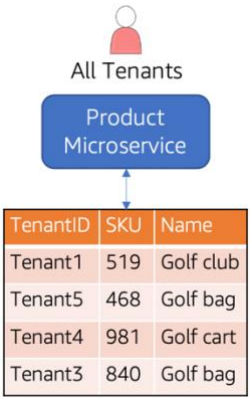

図 16 - プール化したストレージ

ここでは 1 つの製品マイクロサービスがあり、データをプールモデルで保存しています。テーブルの最初の列には、各テナントのキーを示すインデックスがあります。すべてのテナント製品データは、この 1 つのテーブル内にあります。

このモデルでは、データの分離に伴う課題が非常に複雑になります。特定のテナントに属する行のみに限定して、このテーブルの仮想ビューを作成するには、どうしたらよいでしょうか? また、AWS の各ストレージサービスにわたって、この分離を実現するには、どうしたらよいでしょうか? 実はプールモデルで分離を実装するには、サービスごとに独自のアプローチが必要になる場合があ

ります。

このバリエーションをより良く理解するために、IAM を使用して Amazon DynamoDB でプール化した分離を実装する方法の例を見てみましょう。Amazon DynamoDB は、フルマネージドのストレージサービスとして、リソースへのアクセスを制御するための豊富な IAM メカニズムのコレクションを提供しています。これには、Amazon DynamoDB テーブルの項目へのアクセスを制限できる**リーディングキー**条件 (dynamodb:LeadingKeys) を IAM ポリシーで定義する機能も含まれています。図 17 の IAM ポリシーは、分離に対するこのアプローチを示すサンプルポリシーです。

このポリシーで重要なのは条件です。この条件は、このポリシーを適用したときに、Amazon DynamoDB テーブルへのすべてのアクセス試行を、このリーディングキーの値と一致するキーを持つ項目に制限することを指定します。したがって、この場合、テナント識別子はリーディングキーにあり、特定のテナントのデータへのアクセスを制限します。

```
{
  "Sid": "TenantReadOnlyOrderTable",
  "Effect": "Allow",
  "Action": [
      "dynamodb:GetItem",
      "dynamodb:BatchGetItem",
      "dynamodb:Query",
      "dynamodb:DescribeTable"
  ],
  "Resource": [
      "arn:aws:dynamodb:us-east-1:000000000000:table/Order"
  ],
  "Condition": {
      "ForAllValues:StringEquals": {
          "dynamodb:LeadingKeys": [
              "5bd24c40d66c4755819d28ceab9f0826"
          ]
      }
  }
}
```

図 17 - リーディングキーを使用した DynamoDB の分離

この同じ分離モデルを Amazon Aurora PostgreSQL で使用した場合は、メカニズムがまったく異なることがわかります。Aurora PostgreSQL では、IAM を使用して行レベルでデータへのアクセスをスコープ指定することはできません。代わりに、PostgreSQL の行レベルセキュリティ (RLS) 機能を使用して、テナントデータを分離する必要があります。図 18 は、システムで製品テーブルの RLS を設定する方法の簡単な例を示しています。

```
-- Turn on RLS
ALTER TABLE product ENABLE ROW LEVEL SECURITY;

-- Restrict read and write actions so tenants can only see their rows
CREATE POLICY product_isolation_policy ON product
USING (tenant_id = current_tenant);
```

図 18 - PostgreSQL RLS を使用したプール分離

RLS を設定する最初のステップは、テーブルの行レベルセキュリティを有効にするようにテーブルを変更することです。次に、tenant_id 列を現在のユーザーの値 (コンテキストから提供) と一致させるための分離ポリシーを作成します。これらの変更を適用すると、このテーブルとのすべてのやり取りは、現在のテナントで有効な行に制限されます。

DynamoDB や Aurora PostgreSQL のアプローチとは対照的に、使用しているストレージサービスではある程度の調査を個別に行って、分離を実現するモデルを見つける必要があります。また、サービスがよりきめ細かな分離モデルを提供しない場合もあります。このような場合は、独自のメカニズムを導入してプール分離ポリシーを適用する必要があります。

アプリケーションで適用するプール分離

これまで注目したほとんどの戦略では、プール化した分離モデルの基盤として IAM を使用しています。IAM は、多くの場合、リソースを分離する最適な手段ですが、アプリケーションが必要とする形態の分離を IAM がサポートしていない場合もあります。このような場合は、他のフレームワークやツールを使用して、アプリケーションのプール化したリソースへのアクセスを制御する必要があります。

アプリケーションで適用する分離には、通常、ポリシーを表現するモデルが含まれます (IAM での場合とほぼ同様です)。これらの同じフレームワークに通常含まれるポリシー適用メカニズムは、ユーザーとリソースの間に位置して、リソースへのアクセスを許可します。図 19 は、アプリケーションで適用するポリシーモデルの一部を構成する変動要素の概念ビューを示しています。

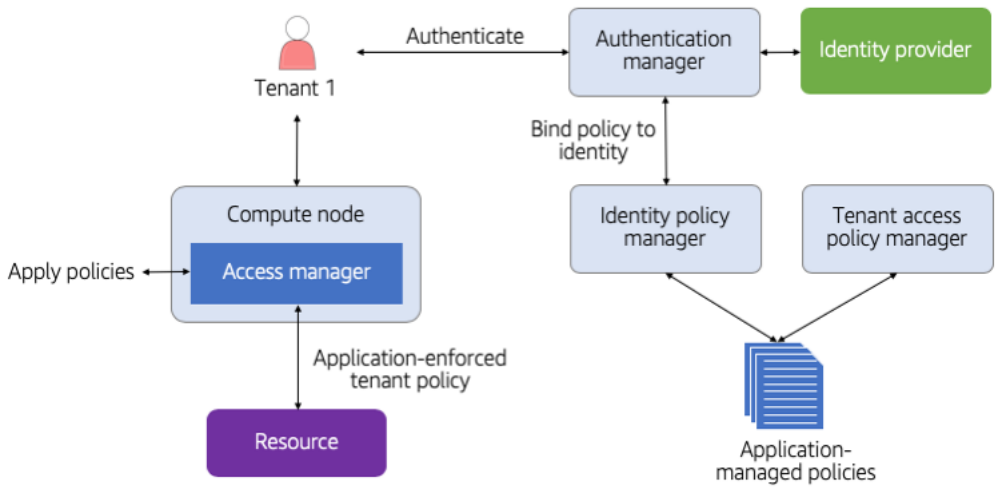

図 19 - アプリケーションで適用するプール分離

この例で、テナントは ID プロバイダーに対して認証を行い、この特定のユーザーに対して定義されたポリシーを特定する構造を導入します (これはダウンストリームプロセスでも発生する可能性があります)。ここで重要な点は、ポリシーがユーザーの ID に接続され、これらのポリシーをダウンストリームオペレーションで特定のユーザーのコンテキストに適用できるようになることです。認証が完了すると、ID はシステムのサービスを通じて流れます。ここで、コードおよびアクセスしようとしているリソースの間にライブラリまたはプロセスを介在させ、ユーザーにバインドされたポリシーを適用する必要があります。

> **注意:** このアプローチは、概念モデルを示すことのみを目的としています。各フレームワークで使用する戦略ごとに、ポリシーを表現および適用するアプローチは異なる場合があります。

ここで注意する点として、ポリシーベースの分離とロールベースのアクセスコントロール (RBAC) の境界は明確でない場合があります。ここでのツールは、この混乱を助長する可能性があります。ただし、一般的に RBAC とテナント分離を同等と見なさないようにします。多くの場合、RBAC には機能マッピングがあり、ユーザーのロール (アプリケーションで定義) を使用してシステム機能へのアクセスを制御します。この方法は、システムのテナント間に分離の境界を設定することとは異なります。分離の境界は、機能的な目標というよりも、あるテナントが別のテナントのデータにアクセスするのを防ぐことを主眼とします。

あらゆるリソースのプールモデル

ここで取り上げるプールは、プール戦略の実装の基本的な変動要素に焦点を当てています。ただし、AWS の各サービスでプールを実現する方法についてはこのホワイトペーパーのスコープ外のため、取り上げません。プール分離の概念とトレードオフは、ほとんどのリソースで似ています。その他の AWS のサービスでも、利用可能な分離メカニズムと、テナント間でリソースを共有する効率性とのバランスを取ることが必要になります。すべてのリソースにプールモデルを使用しながら、すべての分離目標を達成することが理想的なシナリオとなります。ただし、実際には、一部のリソースでは分離モデルの達成が困難である場合があります。このような場合は、サイロモデルの採用を検討します。あるいは、アプリケーションで適用する分離を使用するように努力し、分離の目標を達成します。

プール分離の詳細を隠す

前述のように、プールモデルの重要な側面の 1 つは、モデル全体に準拠するには開発者に依存するということです。開発者は、慣例として、リソースにアクセスする前にスコープ指定されたコンテキストを取得する必要があります。このモデルに準拠することが重要であるため、多くの場合、企業は特定のメカニズムを作成して、SaaS サービスの一部として導入している分離ポリシーに開発者が簡単に適合できるようにしています。

ここでの一般的なアプローチは、一般的な設計のベストプラクティスに忠実に従っています。これに従い、チーム間で共有するライブラリ、モジュール、または軽量フレームワークを作成することになります。ここでの目標は、スコープ指定されたコンテキストを取得する仕組みを、チーム全体で活用できる共有構造に移行することです。図 20 は、分離の詳細を隠すという考え方の概念ビューを示しています。

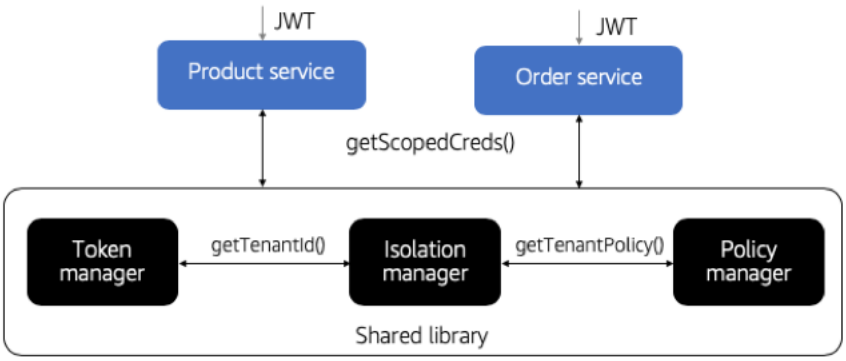

図 20 - ライブラリを使用した分離の標準化

ここに 2 つのマイクロサービス (製品と注文) があり、各サービスはシステムのプール分離モデルに準拠するために認証情報を取得する必要があります。ここで行っているのは、このプロセスのすべてのコードと詳細を共有ライブラリに移動することです (これらは個別のマイクロサービスではありません)。マイクロサービスは、スコープ指定された認証情報を必要とする場合、分離マネージャー (Isolation manager) を呼び出し、マイクロサービスに提供されている JWT トークンを渡します。この分離マネージャーは、トークンマネージャー (Token manager) から `tenantId` を取得します。トークンマネージャーは、JWT の解析とテナント情報の抽出に関連するすべてのロジックを所有しています。次に、分離マネージャーはポリシーマネージャー (Policy manager) からこのテナントのポリシーを取得し、そのポリシーを使用してテナントのスコープ指定された一連の認証情報を取得します。これらの認証情報は、呼び出し元のサービスに返されます。

このアプローチには特にユニークな点はありません。単に基本的な戦略を適用して再利用可能な構造を抽出し、これらをバージョン管理して、チームがより広範に共有できるようにしているだけです。ここでの重要な概念は、テナント分離の詳細を極力開発者の目に触れないようにして、開発者が分離スキームをできるだけ簡単に適用できるようにすることです。

これを実装する方法は、アプリケーションで使用しているスタックやコンピューティング構造から影響を受ける場合もあります。例えば、Lambda では、これらのライブラリを Lambda レイヤーに移動するのが適切です。移動先のレイヤーで、これらの水平概念を個別にバージョン管理し、Lambda 関数で広範に参照します。

また、これを開発者から完全に隠し、マイクロサービスの実装を開始する前に、これらのスコープ指定された認証情報をインターセプトおよび取得するメカニズムを導入することもできます。例えば、一部の言語では、アスペクトを使用して受信リクエストをインターセプトし、スコープ指定さ

れた認証情報を取得して、マイクロサービスに挿入できます。Lambda 関数には、スコープ指定された認証情報を Lambda 関数に挿入するために使用できるさまざまなオープンソースの**ラッパー**ライブラリがあります。これらの戦略は、分離を適用するためのより厳格なモデルを提供する場合があります。

分離の透過性

分離については、アプリケーションの設計とアーキテクチャ内でどのように実現するかを中心に説明してきました。ただし、システムのテナントの視点から分離について考えることも重要です。SaaS 開発者やアーキテクトは分離オプションを常に検討していますが、テナントに対して分離に関する明確で一貫した説明を提示し、テナントから分離戦略の基礎的な詳細を隠すことが依然として重要です。図 21 は、顧客に提示する分離の概念ビューを示しています。

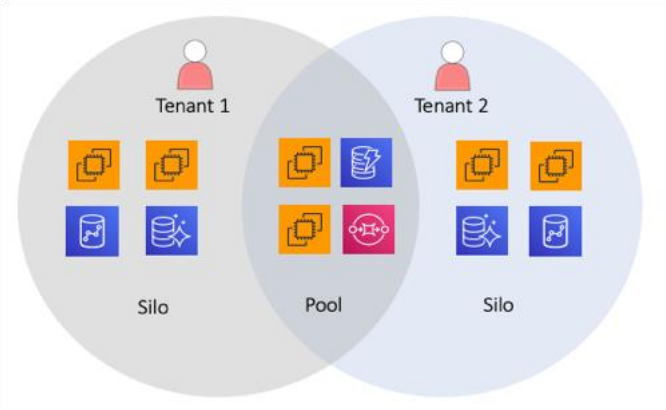

図 21 - 分離を透過的にする

ここでは、2 つのテナントがリソースを持っています。一部のリソースは、サイロモデル (左と右) にデプロイしています。これらのテナントの他のリソースは、プールモデル (2 つの円の重複部分) にデプロイしています。ここでは、サイロとプールが混在しているにもかかわらず、システムはクロステナントアクセスを防止する包括的な分離アプローチを提供しています。顧客にとって必要なことは、この分離が有効であるという保証のみです。理想的には、どのリソースがプール化され、どのリソースがサイロ化されているかを顧客は知る必要がありません。

まとめ

ここで説明している分離の概念に注目すると、AWS で SaaS ソリューションを構築する際に考慮すべき分離オプションについてよく理解できるはずです。いくつかの主要なパターンを紹介し、分離を実装するためのさまざまなモデルを示しました。これらのモデルは、SaaS アプリケーションのドメイン、コンプライアンス、運用、パフォーマンスのプロファイルから直接影響を受けます。サイロとプールの分離モデルについて主に説明し、これらをさまざまな SaaS モデルで実現する際の方法の相違点に注目しました。また、分離戦略が SaaS 環境の構築に使用する AWS のサービスによってどのような影響を受けるかについても検討しました。

分離を実装すると、SaaS ソリューションが複雑化しますが、堅牢な分離モデルは、ベストプラクティスに従った SaaS アプリケーションを実装するために重要です。テナントが別のテナントのリソースにアクセスできるようなシナリオは、偶発的に発生した場合でも、SaaS ビジネスにとって重大な阻害要因となります。そのため、組織は慎重を期して分離モデルを実装し、分離戦略の柱としての認証や正常に動作するコードにできるだけ依存しないようにする必要があります。

寄稿者

この文書の寄稿者は次のとおりです。

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

参考資料

詳細については、次を参照してください。

- SaaS Tenant Isolation Patterns (Saas テナント分離パターン)
- SaaS ストレージ戦略に関するホワイトペーパー
- Managing SaaS Identity Through Custom Attributes and Amazon Cognito (カスタム属性と Amazon Cognito による SaaS アイデンティティの管理)
- SaaS と OpenID Connect: マルチテナントのアイデンティティと分離に関する秘密の情報源

ドキュメントの改訂

日付	説明
2020 年 8 月	初版発行