SaaS Tenant Isolation Strategies

Isolating Resources in a Multi-Tenant Environment

August 2020

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided "as is" without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

© 2020 Amazon Web Services, Inc. or its affiliates. All rights reserved.

Contents

Abstract ... 1

Introduction ... 2

- The Isolation Mindset ... 2
- Isolation: Security or Noisy Neighbor? ... 4

Core Isolation Concepts ... 4

- Silo Isolation ... 4
- Pool Isolation ... 7
- The Bridge Model ... 10
- Tier-Based Isolation ... 11
- Identity and Isolation ... 13

Implementing Silo Isolation ... 14

- Full Stack Isolation ... 14
- Targeted Silo Isolation ... 17
- Silo Compute Considerations ... 19
- Silo for any Resource ... 21

Implementing Pool Isolation ... 21

- Run-time, Policy-Based Isolation with IAM ... 23
- Scaling and Managing Pool Isolation Policies ... 25
- Pooled Storage Isolation Strategies ... 26
- Application-Enforced Pool Isolation ... 29
- Pool for any Resource ... 30
- Hiding the Details of Pooled Isolation ... 30

Isolation Transparency ... 32

Conclusion ... 33

Contributors ... 33

Further Reading ... 33

Document Revisions..34

Abstract

Tenant isolation is fundamental to the design and development of software as a service (SaaS) systems. It enables SaaS providers to reassure customers that—even in a multi-tenant environment—their resources cannot be accessed by other tenants. This paper will look at the full range of strategies that are commonly used by SaaS companies to ensure that their systems are successfully isolating tenant resources while still realizing the value proposition of the SaaS delivery model.

Introduction

Tenant isolation is one of the foundational topics that every software as a service (SaaS) provider must address. As independent software vendors (ISVs) make the shift toward SaaS and adopt a shared infrastructure model to achieve cost and operational efficiency, they also have to take on the challenge of determining how their multi-tenant environments will ensure that tenants are prevented from accessing another tenant's resources. Crossing this boundary in any form would represent a significant and potentially un-recoverable event for a SaaS business.

While the need for tenant isolation is viewed as essential to SaaS providers, the strategies and approaches to achieving this isolation are not universal. There are a wide range of factors that can influence how tenant isolation is realized in any SaaS environment. The domain, compliance, deployment model, and the selection of AWS services all bring their own unique set of considerations to the tenant isolation story.

In this whitepaper, we'll outline many of the common patterns and strategies that are used to implement tenant isolation on AWS. The goal here is to capture some of the common themes and challenges that span the various SaaS architecture models and AWS technologies, while highlighting the various approaches to achieving tenant isolation in each of these environments. This paper should equip you with a collection of insights that will help you select the combination of isolation strategies that best align with the realities of your environment and business model.

The Isolation Mindset

At the conceptual level, most SaaS providers would agree on the importance and value of protecting and isolating tenant resources. However, as you dig into the details of implementing an isolation strategy, you'll often find that each SaaS ISV has their own definition of what is *enough* isolation.

Given these varying perspectives, we have outlined some tenets below that will guide our overall value system for tenant isolation. Every SaaS provider should establish a clear set of high-level isolation requirements that will guide their teams as they define the isolation footprint of their SaaS environment. The following are some key tenets that typically shape the overall SaaS tenant isolation model:

Isolation is not optional – isolation is a foundational element of SaaS and every system that delivers a solution in a multi-tenant model should ensure that their systems take measures to ensure that tenant resources are isolated.

Authentication and authorization are not equal to isolation – while it is expected that you will control access to your SaaS environments through authentication and authorization, getting beyond the entry points of a login screen or an API does not mean you have achieved isolation. This is just one piece of the isolation puzzle and is not enough on its own.

Isolation enforcement should not be left to service developers – while developers are never expected to introduce code that might violate isolation, it's unrealistic to expect that they will never un-intentionally cross a tenant boundary. To mitigate this, scoping of access to resources should be controlled through some shared mechanism that is responsible for applying isolation rules (outside the view of developers).

If there's not out-of-the box isolation solution, you may have to build it yourself – there are a number of security mechanisms such as AWS Identity and Access Management (IAM) that can simplify the path to tenant isolation. These tools and their integration with a broader security scheme often make isolation a somewhat seamless experience. However, there may be scenarios where your isolation model is not directly addressed by a corresponding tool or technology. The absence of a clear solution should not represent an opportunity to lower your isolation requirements—even if that means building something of your own.

Isolation is not a resource-level construct – in the world of multi-tenancy and isolation, some will view isolation as a way to draw a hard boundary between concrete infrastructure resources. This often translates into isolation model where you might have separate databases, compute instances, accounts, or virtual private clouds (VPCs) for each tenant. While these are common forms of isolation, they are not the only way to isolate tenants. Even in scenarios where resources are shared—in fact, especially in environments where resources are shared—there are ways to achieve isolation. In this shared resource model, isolation can be a logical construct that is enforced by run-time applied policies. The key point here is that isolation should not be equated to having siloed resources.

Domains may impose specific isolation requirements – while there are many approaches to achieving tenant isolation, the realities of a given domain may impose constraints that will require specific flavor of isolation. Some high compliance industries, for example, will require that every tenant have its own database. In these cases, the shared, policy-based approaches to isolation may not be adequate.

Isolation: Security or Noisy Neighbor?

The topic of isolation is often challenging to compartmentalize. Typically, companies will think about isolation through a security and compliance lens where isolation is used to create boundaries between resources to limit any exposure to cross-tenant access. This is a key area of focus for isolation. However, isolation is also implemented as part of addressing noisy neighbor and performance concerns. This is another reason data for tenants may be isolated. For the scope of this paper, we will be including coverage of both items with more emphasis on the security dimensions of this problem.

Core Isolation Concepts

Part of the challenge of isolation is that there are multiple definitions of tenant isolation. For some, isolation is almost a business construct where they think about entire customers requiring their own environments. For others, isolation is more of an architectural construct that overlays the services and constructs of your multi-tenant environment. The sections below will explore the different types of isolation, and associate specific terminology with the varying isolation constructs.

Silo Isolation

While SaaS providers are often focused on the value of sharing resources, there are still scenarios where a SaaS provider may choose to have some (or all) of their tenants deployed in a model where each tenant is running a fully siloed stack of resources. Some would say that this full-stack model does not represent a SaaS environment. However, if you've surrounded these separate stacks with shared identity, onboarding, metering, metrics, deployment, analytics, and operations, then we'd still say this is still a valid variant of SaaS that trades economies of scale and operational efficiency for compliance, business, or domain considerations. With this approach, isolation is an end-to-end construct that spans an entire customer stack. The diagram in Figure 1 provides a conceptual view of this view of isolation.

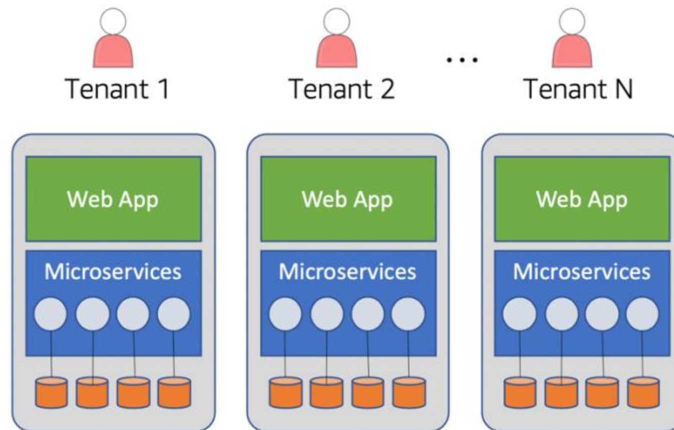


Figure 1 – Full Stack View of Isolation

This diagram highlights the basic footprint of the siloed deployment model. The technologies that are used run these stacks are mostly irrelevant here. This could be a monolith, it could be serverless, or it could be any mix of the various application architecture models. The key concept here is that we're going to take whatever stack the tenant has and surround it with some construct to encapsulate all the moving parts of that stack. This becomes our boundary for isolation. As long as you can prevent a tenant from escaping their fully encapsulated environment, you've achieved the isolation.

Generally, this model of isolation is a much simpler to enforce. There are often well-defined constructs that will enable you to implement a robust isolation model. While this model presents some real challenges to the cost and agility goals of a SaaS environment, it can be appealing to those that have very strict isolation requirements.

Silo Model Pros and Cons

Each SaaS environment and business domain has its own unique set of requirements that may make silo a fit. However, if you're leaning in this direction, you'll definitely want to factor in some of the challenges and overhead associated with the silo model. Below is a list of some of the pros and cons that you need to consider if you are exploring a silo model for your SaaS solution:

Pros

- **Supporting challenging compliance models** – some SaaS providers are selling into regulated environments that impose strict isolation requirements. The silo provides these ISVs with an option that enables them to offer to some or all of their tenants the option of being deployed in a dedicated model.

- **No noisy neighbor concerns** – while all SaaS providers should be attempting to limit the impacts of noisy neighbor conditions, some customers will still express reservations about the potential of having their workloads impacted by the activity of other tenants using the system. Silo addresses this concern by offering a dedicated environment with no potential of noisy neighbor scenarios.
- **Tenant cost tracking** – SaaS providers are often highly focused on understanding how each tenant is impacting their infrastructure costs. Calculating a cost per tenant can be challenging in some SaaS models. However, the coarse-grained nature of the silo model provides us with a simple way to capture and associate infrastructure costs with each tenant.
- **Limited blast radius** – the silo model generally reduces your exposure when there may be some outage or event that surfaces in your SaaS solution. Since each SaaS provider is running in its own environment, any failures that occur within a given tenant's environment will likely be constrained to that environment. While one tenant may experience an outage, the error may not cascade through the remaining tenants that are using your system.

Cons

- **Scaling issues** – there are limits on the number of accounts that can be provisioned. This limit may exclude you from selecting the account-based model. There are also general concerns about how a rapidly growing number of accounts might undermine the management and operational experience of your SaaS environment. If you have 20 siloed accounts for each of your tenants, for example, that may be manageable. However, if you have a thousand tenants, that would likely begin to impact operational efficiency and agility.
- **Cost** – with every tenant running in its own environment, we're missing much of the cost efficiency that is traditionally associated with SaaS solutions. Even if these environments scale dynamically, you'll likely have periods of the day when you'll have idle resources that are going un-consumed. While this is a completely acceptable model, it undermines the ability of your organization to achieve the economies of scale and margin benefits that are essential to the SaaS model.

- **Agility** – the move to SaaS is often directly motivated by a desire to innovate at a faster pace. This means adopting a model that enables the organization to respond and react to market dynamics at a rapid pace. A key part of this is being able to unify the customer experience and quickly deploy new features and capability. While there are measures that can be taken with the silo model to try to limit its impact on agility, the highly decentralized nature of the silo model adds complexity that impacts your ability to easily manage, operate, and support your tenants.
- **Onboarding automation** – SaaS environments place a premium on automating the introduction of new tenants. Whether these tenants are being onboarded in a self-service model or using an internally managed provisioning process, you'll still need automated onboarding. And, when you have separate siloes for each tenant, this often becomes a much more heavyweight process. The provisioning of a new tenant will require the provisioning of new infrastructure and, potentially, the configuration of new account limits. These added moving parts introduce overhead that introduces additional dimensions of complexity into the overall onboarding automation, enabling you to focus less time on your customers.
- **Decentralized management and monitoring** – our goal with SaaS is to have a single pane of glass that lets us manage and monitor all tenant activity. This requirement is especially important when you have siloed tenant environments. The challenge here is that you must now aggregate the data from a more decentralized tenant footprint. While there are mechanisms that will enable you to create an aggregate view of your tenants, the effort and energy needed to build and manage this experience is more complex in a siloed model.

Pool Isolation

It's pretty easy to see how the silo model of isolation maps very nicely for many SaaS companies. At the same, many companies that are moving to SaaS are seeking out the efficiency, agility, and cost benefits of being able to have their tenants share some or all of their underlying infrastructure. This shared infrastructure approach, which is referred to as a pool model, adds a level of complexity to the isolation story. The diagram in Figure 2 provides an illustration of the challenge associated with implementing isolation in a pooled model.

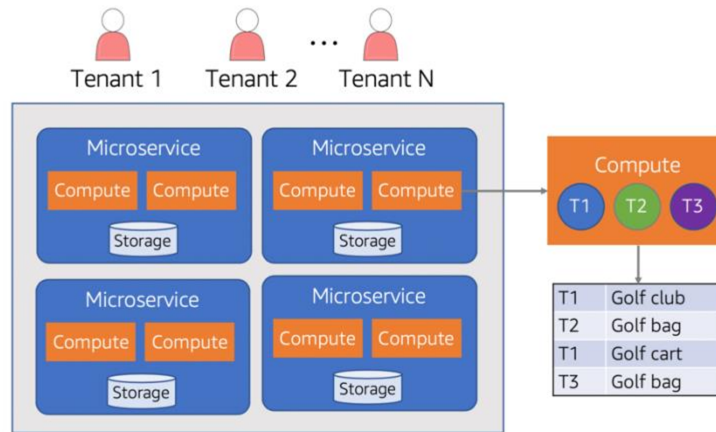


Figure 2 – Pooled Isolation

In this model, you'll notice that our tenants are consuming infrastructure that is shared by all tenants. This enables the resources to scale in direct proportion to the actual load being imposed by the tenants. To the right of the diagram, we've zoomed into the compute of one of the services, highlighting the fact that tenants 1-N may all be running side-by-side within your shared compute at any given time. You'll also notice that the storage in this example is also shared. Here we've represented a table that is indexed by individual tenant identifiers.

Now, while this model is a perfectly good fit for SaaS providers, you can see how this complicates the overall isolation story. With resources being shared, it's unclear what it would mean here to implement isolation. We can't lean on the typical networking and IAM constructs to create boundaries between tenants.

The key here is that—even though this is a more challenging environment to isolation—you cannot use this as a rationale to relax the isolation requirements of your environment. If anything, these shared model increases the chance for cross-tenant access and, as such, it represents an area that requires you to be especially diligent about ensuring that resources are isolated.

As we dig deeper into the pool isolation model (above), you'll see how this architectural footprint introduces a unique blend of challenges—each of which requires its own type of isolation constructs to successfully isolate a tenant's resources.

Pool Model Pros and Cons

While having everything shared enables a lot of efficiency and optimization, it also requires SaaS providers to weigh some of the tradeoffs that come with adopting this model. In many cases, the pros and cons of the pool model end up surfacing as the

inverse of pros and cons we covered for the silo model. The following is an outline of the key pros and cons that are typically associated with the pool isolation model.

Pros

- **Agility** – as you move all tenants into a shared infrastructure model, you get all the natural efficiencies and simplicity that streamlines the agility of your SaaS offering. At its core, the pool model is all about enabling SaaS providers to manage, scale, and operate all of its tenants with one unified experience. Centralizing and standardizing the experience is foundational to enabling SaaS providers to easily manage and apply changes to all tenants without having to perform one-off tasks on a tenant-by-tenant basis. This operational efficiency is key to the overall agility footprint of your SaaS environment.
- **Cost efficiency** – many companies are drawn to SaaS for its cost efficiency. A big part of this cost efficiency is commonly associated with the pool model of isolation. In a pooled environment, your system will scale based on the actual load and activity of all of your tenants. If all the tenants are offline, your infrastructure costs should be minimal. The key concept here is that pooled environments can adjust to tenant load dynamically and enable you to better align tenant activity with resource consumption.
- **Simplified management and operations** – the pool model of isolation gives me one view into all the tenants in my system. I can manage, update, and deploy all of my tenant through a single experience that touches all the tenants in my system. This makes most aspects of the management and operations footprint simpler.
- **Innovation** – the agility that is enabled by the pooled isolation model also tends to be core to enabling SaaS providers to innovate at a faster pace. The more you move away from distributed management and the complexity of the silo model, the more you're freed up to focus on the features and functions of your product.

Cons

- **Noisy neighbor** – the more resources are shared, the more chances there are for one tenant to impact the experience of another. Any activity from one tenant that puts heavy load on the system, for example, has the potential to impact other tenants. A good multi-tenant architecture and design will try to limit these impacts, but there's always some chance of a noisy neighbor condition impact one or more of your tenants in a pooled isolation model.

- **Tenant cost tracking** – in a silo model, it's much easier to attribute consumption of a resource to a specific tenant. However, in a pooled model, the attribution of resources consumption becomes more challenging. This pushes more work to each SaaS provider as they look for ways to instrument their systems and surface the granular data needed to effectively associate consumption with individual tenants.
- **Blast radius** – having all of your resources shared also introduces some operational risk. In the silo model, when one tenant had a failure, the impact of that failure could likely be limited to that one tenant. However, in a pooled environment, an outage will likely impact all the tenants of your system. This can have a significant impact on the business. This usually requires an even deeper commitment to building a resilient environment that can identify, surface, and gracefully recover from failures.
- **Compliance pushback** – while there are measures you can take to isolate your tenants in a pool model, the notion of sharing infrastructure can create situations where customers may be unwilling to run in this model. This is especially true in environments where the compliance or regulatory rules for a domain impose strict constraints on the accessibility and isolation of resources. Even in these cases, though, this may mean some portion of the system will need to be siloed (see the bridge model below).

The Bridge Model

While silo and pool have very distinct approaches to isolation, the isolation landscape for many SaaS providers is less absolute. As you look at real application problems and you decompose our systems into smaller services, you will often discover that your solution will require a mix of the silo and pool models. This mixed model is what we would refer to as a bridge model of isolation. The diagram in Figure 3 provides an example of how the bridge might be realized in a SaaS solution.

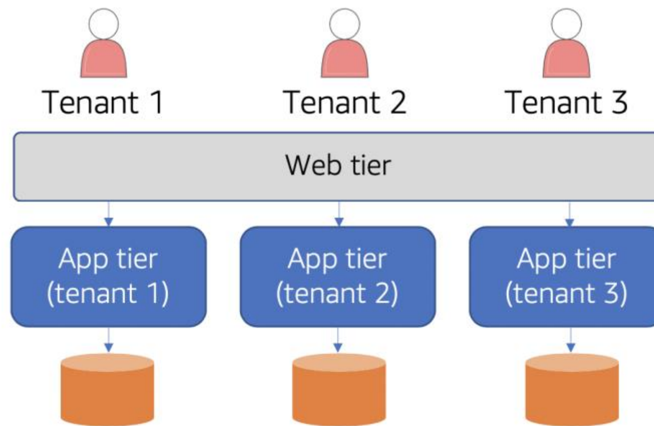


Figure 3 - Bridge Isolation Model

This diagram highlights how the bridge model enables you to combine of the silo and pool models. Here we have a monolithic architecture with classic web and application tiers. The web tier, for this solution, is deployed in a pool model that is shared by all tenants. While the web tier is shared, the underlying business logic and storage of our application are actually deployed in a silo model where each tenant has its own application tier and storage.

Now, imagine we were to break this monolith into microservices. You can imagine that each of the various microservices in our system could leverage combinations of the silo and pool models. We'll dig into that more as we get into the specifics of applying silo and pool with different AWS constructs. The key takeaway here is that your view of silo and pool will be much more granular for environments that are decomposed into a collection of services that have varying isolation requirements.

Bridge Model Pros and Cons

The bridge model is more a hybrid model that focuses on enabling you to apply the silo or pool model where it makes sense. The idea here is that the values and tenets of silo isolation still apply to each of these areas of the system. As you think about pros and cons of the bridge model, then, you should be thinking about the tradeoffs of silo and pool models for each resource or layer of your architecture.

Tier-Based Isolation

While most of our discussion of isolation focuses on the mechanics of preventing cross-tenant access, there are also scenarios where the tiering of your offering might influence your isolation strategy. In this case, it's less about how you're isolating tenants

and more about how you might package and offer different flavors of isolation to different tenants with different profiles. Still, this is another consideration that could determine which models of isolation you'll need to support to address the full spectrum of customers you want to engage. The diagram in Figure 4 provides an example of how isolation might vary across tiers.

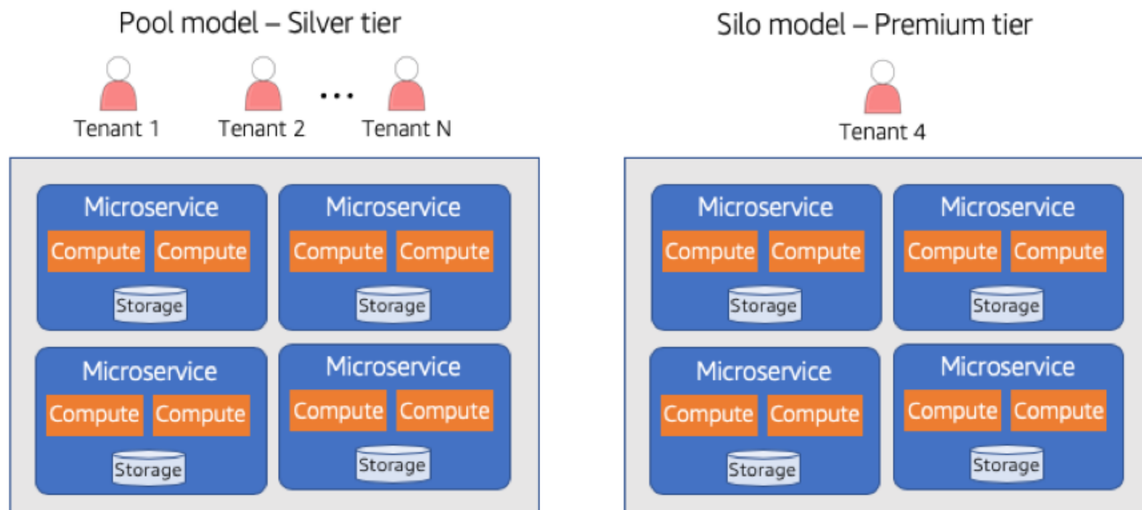


Figure 4 – Tenant Tiering and Isolation

Here you'll see a scenario where we a mix of silo and pool isolation models that have been offered up as tiers to our tenants. Tenants in the silver tier are running in the pooled environment. While these tenants are running in a shared infrastructure model, they still fully expect that their resources will be protected from any cross-tenant access. The tenant on the right has required you to offer them a completely dedicated (silo) environment. To support this, the SaaS provider has created a premium tier model that enables tenants to run in this dedicated model at what we would assume would be a substantially higher price point.

While SaaS providers generally try to limit offering a silo model to their customers, many SaaS businesses have this notion of a private pricing where these tenants offer to pay a premium to be deployed in this model. In fact, SaaS companies will not publish this as an option or identify it as a tier to limit the number of customers that chose this option. If too many of your tenants fall into this model, you'll begin to fall back to a fully siloed model and inherit many of the challenges that we outlined above.

To limit the impact of these one-off environments, SaaS providers will often require these premium customers to run the same version of the product is deployed to the pooled environment. This enables the ISV to continue to manage and operate both

environments through a single pane of glass. Essentially, the silo environment becomes a clone of the pooled environment that happens to be supporting one tenant.

Identity and Isolation

While the scope of your discussion is limited to isolation, it's important to look at how identity connects to the isolation model. The reality is, if you are planning to isolate tenants, you must have some way to represent and identify the tenant that is currently accessing the resources of our SaaS environment. In many cases, identity will be used in combination with other constructs to acquire the policies and scoping rules that are at the core of an isolation scheme. How these policies are defined and applied will vary for each of the isolation models and services you're consuming. Still, the basics of the approach usually follow a pattern similar to what is shown in Figure 5.

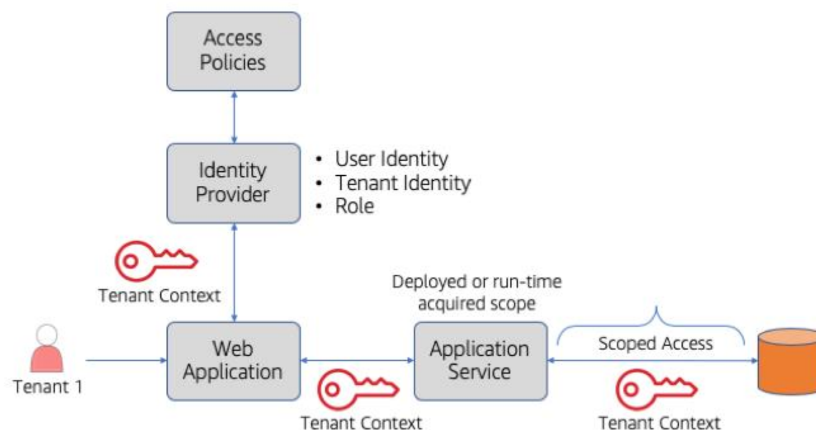


Figure 5 – Connecting Identity and Isolation

This diagram represents a generalization of how identity gets connected to the broader isolation story. Here you'll notice that, as a user is authenticated, the system will return tenant context back to your application that includes the user's binding to a tenant as well as the policies that will be used to enforce isolation for that tenant. This context then flows through all of our interactions and is used by the downstream elements of the SaaS environment to scope access to resource (in this case a database).

How that scope is acquired and applied will vary based on the isolation model and resources you're consuming, but this model provides a view of the core concepts. One key area of variation is in how the tenant scoping is determined. This scoping context could be attached to a service when it is deployed or it could be acquired at run-time. We'll look at both of those models as we get into the specific isolation traits for different architecture models.

Implementing Silo Isolation

Now that we have a clear conceptual picture of isolation, we can turn our attention to how these different models are actually realized with different AWS services and constructs. In the sections below, we'll look at the various ways that silo isolation is used across the different layers of your SaaS architecture.

Full Stack Isolation

The first model we'll look at is full stack silo isolation. In this scenario, where a SaaS ISV requires all the resources of a tenant to be fully isolated, you will rely on more coarse-grained AWS constructs to isolate your tenants. The choices you make here will likely be largely influenced by the management, operations, and scaling profile of your environment. The following is a breakdown of the common full stack silo isolation mechanisms.

Account Silo Isolation

The AWS account represents one strategy that can be used to isolate tenants in a silo model. The basic approach here is to deploy each tenant into an entirely separate account, linking each tenant account to a parent. All of the moving parts of each tenant stack are deployed with the stack and are operated in complete isolation. This account boundary of isolation is often appealing to those that are running in environments that demand a very easily described boundary each tenant. For a SaaS provider, the account isolation model can provide a compelling story to customers that are especially concerned about any possibility of cross-tenant access.

Let's take a closer look at a sample architecture that uses the account-based pool isolation model. The diagram in Figure 6 illustrates two tenants deployed into two separate accounts.

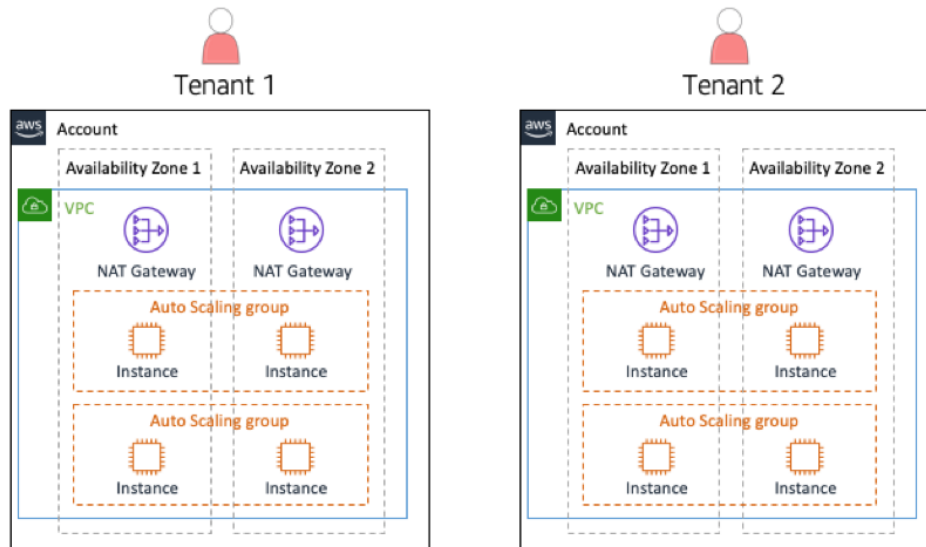


Figure 6 - Account-based Silo and Isolation

The architecture shown here is just an example. The actual infrastructure that lands in your account would vary based on the technologies that were part of your SaaS application's technology stack. This could use containers, be serverless, or any mix of the various AWS architecture models. The key point here is that every tenant is running the same stack in each of these separate accounts.

While there are merits to this model, it presents real challenges when it comes to scale and automation. There's a rich collection of tools that can automate this experience. However, there are constraints on this automation. They key challenge here is often account limits. Some limits can be configured through automation. Others cannot. This issue becomes more complicated if you want to manage and maintain separate limits for each account.

Virtual Private Cloud (VPC) Silo Isolation

The next level of isolation to consider is within a single account. This brings us into the realm of networking constructs where we essentially use the boundaries of the network for each of our siloed tenant environments. The Amazon Virtual Private Cloud (Amazon VPC) provides a natural mechanism for network-based isolation. The diagram shown in Figure 7 provides a sample of a VPC silo model:

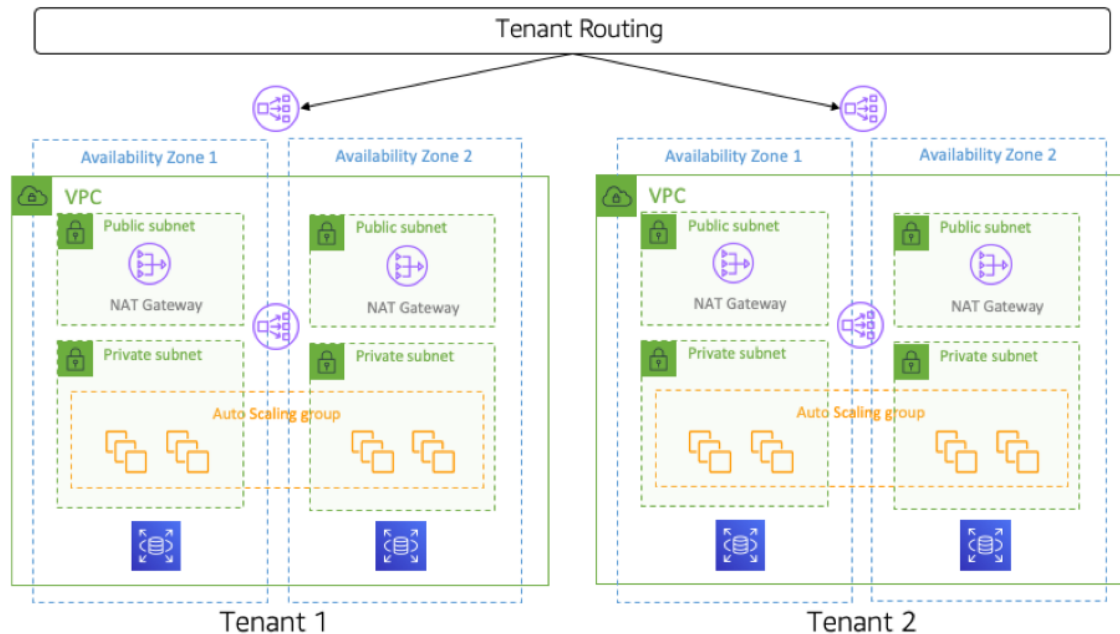


Figure 7 – VPC Silo Isolation

Here you'll notice that we have two separate tenant environments, each hosted in its own VPC. The VPCs represented here using multiple availability zones to convey the typical AWS architecture best practices. As with accounts, the resources configured in each VPC could vary significantly for each SaaS provider. The key aspect of this solution is that the tenants are isolated from one another via the networking constructs that are enforced by the VPC.

The VPC provides a compelling model for those that require siloed isolation. This gets us beyond some of the provisioning challenges of account-based isolation since the limits are owned by the account where these VPCs reside. While this gives creates a more centralized model for managing limits across all tenants, it does not eliminate limits challenges. With the VPC model, you would still need to proactively monitor limits and periodically increase them as needed. There also scenarios where a large number of tenants could exceed the hard limits for your environment. This model does have the upside of enabling you to place tags on your VPC and its resources to calculate tenant costs.

While this model has advantages over account-based isolation, you'll still want to think about scale. As the number of VPCs grows, the management and agility of this approach (and all silo models) decreases. Overall, though, the VPC tends to offer the best combination of options for companies that need to rely on a silo model.

Subnet Silo Isolation

We started with accounts, then moved to VPC as our silo model. You can get even more granular with silo isolation by placing tenants in separate VPC subnets. With this approach, each tenant is placed in a separate subnet within a VPC. The diagram in Figure 8 provides an example of the subnet per tenant model.

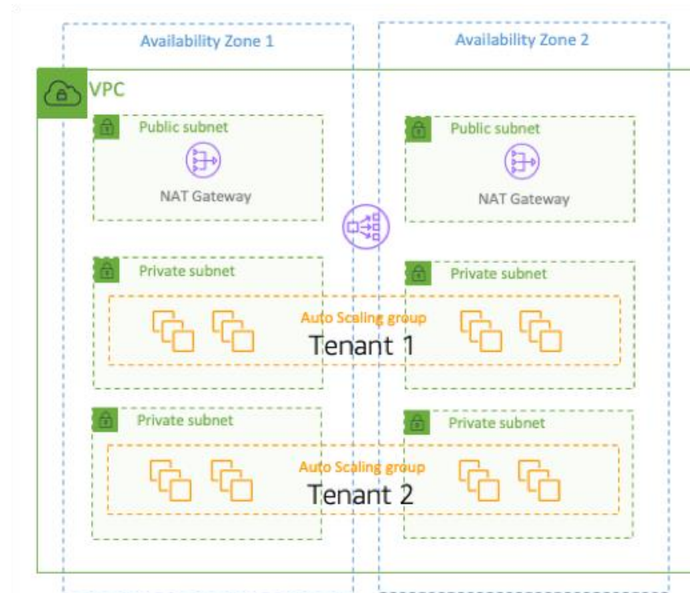


Figure 8 – Subnet Silo Isolation

Here you'll see the same multi-AZ VPC that we had with VPC isolation. However, now you'll notice that the tenants are actually all within the subnets of a single VPC. The isolation of tenants in this model relies on network routing constructs to prevent any cross-tenant access.

While this is a valid approach, it is not recommended as a preferred model. The scaling and management of this becomes unwieldy relatively quickly. If you have a small population of tenants, this may work out fine. Beyond that, we'd recommend using one of the other silo isolation models described here.

Targeted Silo Isolation

As we mentioned above, silo isolation can also be applied in a much more granular fashion where selective elements of your SaaS solution are deployed in a silo model. Each microservice of your system and each resource those services touch has the option of being configured in a silo model of isolation. How that silo isolation is realized will vary across each service or construct that makes up your application. Let's look at

some sample microservice to better understand how these different dimensions of silo isolation might land in an actual application. The diagram in Figure 9 provides a view of these two models.

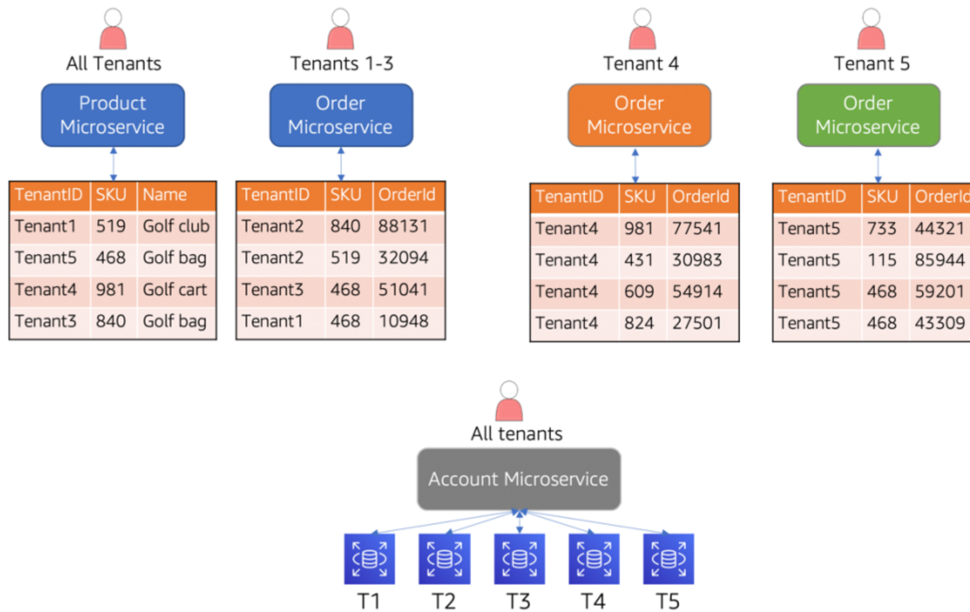


Figure 9 – Microservice Silo Isolation

In this diagram, you'll see a system that has implemented three different microservices: product, order, and account. The deployment and storage models of each of these microservices highlights how isolation (for security or noisy neighbor) could land in a SaaS environment.

Let's review the isolation model for each of these services. The *product* microservice at the top right was deployed in a complete pooled model where both the compute and the storage are shared for all tenants. The table here reflects the fact that tenants all land here as separate items that are indexed in the same table. The assumption here is that we'll be isolating this data with policies that can restrict access to tenant items in this table. The *order* microservice to the right of this item is only for tenants 1 through 3. This microservice is also implemented in a pooled model. The only difference here is that it is supporting a subset of tenants. Essentially, any tenant doesn't get a dedicated (silo) deployment of the *order* microservice would be running in this pooled deployment (think of it as tenants 1..N with the exception of the few that get pulled out as silo microservices).

For the purposes of this discussion, let's focus on the siloed services which are represented by the dedicated *order* microservices (top right) and the *account*

microservice (bottom). You'll notice here that we've deployed standalone instances of the *order* microservice for tenants 4 and 5. The idea here is that these tenants had some requirements for the order processing (compliance, noisy neighbor, and etc.) that required this service to be deployed in a silo model. Here the compute and storage are both dedicated entirely to each of these tenants.

Finally, at the bottom is the *account* microservice. It represents a silo model but only at the storage level. The compute of the microservice is shared by all tenants but each tenant has a dedicated *database* that holds its account data. In this scenario, the isolation concern was focused exclusively on separating the data. The compute was still enabled to be shared.

In looking at this model, you can see how the silo discussion becomes much more granular. Security, noisy neighbor, and a variety of factors will determine how and when you might adopt a silo isolation model for your services. They key takeaway here is that silo is not an all-or-nothing decision. You can think about applying silo to specific *components* of your application and only absorb the challenges of silo where it's actually needed. A potential customer, for example, may be demanding silo. However, after more a detailed discussion, you find out that there are a few specific areas of storage and processing that concern them. This enables you to get the efficiencies of the pool model for those parts of the system that do not require silo isolation. It also gives you the freedom to offer this as a tier to different tenants, supporting a mix of both silo and pool for individual services.

Silo Compute Considerations

As you look to silo the compute resources of your application (like the microservices shown above), you'll want to think about how the isolation models of different compute services might influence your approach. The unique attributes of the various AWS compute services may also require you to take specific measures to ensure that your resources are adequately isolated.

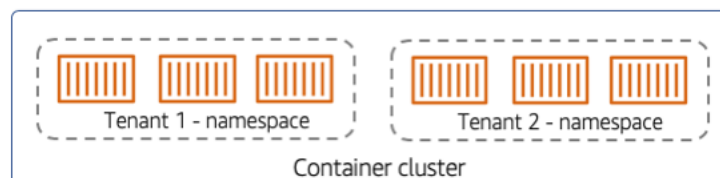


Figure 10 – Container Silo Isolation

Let's start by looking at what it would mean to implement the silo model with containers. The challenge of isolating containers is that there are cases where malicious code or poorly configured environments can *escape* a container and assume privileges that would enable one tenant to access the resources of another tenant. Fortunately, containers offer constructs that, when used properly, can implement a robust isolation model. The mechanisms that are used to prevent cross-tenant access can vary across the different AWS container services. With Amazon Elastic Container Service (Amazon ECS), for example, you'll need to create a separate cluster for each tenant to achieve silo isolation. Amazon Elastic Kubernetes Service (Amazon EKS) introduces some additional mechanisms that will let you silo resources within an EKS cluster. The diagram in Figure 10 provides a look at how you would achieve silo isolation within and EKS cluster.

This example shows two separate groupings of tenants within an EKS cluster where an EKS namespace was used to isolate these compute resources. While namespace provides the foundation of your silo isolation here, namespaces alone don't provide complete isolation. To get full isolation, you need to consider using one of the AWS or partner sidecar solutions that can be used to further lock down the flow between containers. AWS App Mesh and Tigera's Calico represent two examples of solutions that could be used to achieve this added layer of isolation.

AWS Lambda also adds a twist to the silo isolation model. When you think about a Lambda function, you'd presume that it's already isolated since any one tenant can only be executing a function at a given moment in time. However, if a Lambda function is deployed with an execution role that supports all tenants, then there's still the possibility that this function could access a resource that belongs to another tenant. While the pool (as we'll see below) provides us a way around this, a fully siloed version of a Lambda function would mean that this function would not be executed by other tenants. The diagram in Figure 11 provides an example of how you might realize full isolation in a Lambda model.

Figure 11 – Lambda Silo Isolation

This diagram includes two separate tenants that have been deployed in a Lambda silo model. Because we want to ensure that tenant will remain within tenant boundaries, we have deployed separate functions for each tenant where these functions are configured and deployed with a tenant specific role that constrains their access to resources that are associated with that tenant.

This approach has pros and cons. While it can be a compelling isolation story, it is unwieldy and may exceed limits for the Lambda service. Imagine managing and deploying separate functions for 1K tenants. That would become difficult to manage and would undermine the agility of your SaaS goals. At the same time, if you offered this option to a select collection of premium tenants and limited the broader expansion of this model, it would be more reasonable to manage and operate.

The key takeaway here is that, as you consider how to implement your silo model, you'll also need to be thinking about how the silo model is realized on the different AWS compute services. The strategy of silo isolation can change for each service.

Silo for any Resource

While we've focused in on a few key silo models here, it should be clear by now that any resource in your SaaS architecture can typically be deployed in a silo model. However, it should also be clear that the strategies and mechanisms for isolating each AWS service will often vary. Isolating Amazon DynamoDB data, for example, might mean creating separate tables for each tenant. Isolating in other storage models, might require you to create a separate cluster for each tenant. Even Amazon Simple Queue Service (Amazon SQS) and Amazon EventBridge have different approaches to how you might achieve isolation. While it's beyond the scope of this paper to cover isolation strategies for each of these services, it's important for SaaS developers to consider how and where it may be appropriate to silo any one of these services.

The general rule of thumb here is to look at any resource and assess the noisy neighbor and security profile for that resource. You'll have to weigh these isolation options against the added complexities, cost, operational burden, and service limits that may come with adopting a silo strategy for some part of your system. Finding the right balance is often key to the success of your system.

Implementing Pool Isolation

The pool model is often the most appealing to SaaS providers. The efficiency, agility, and cost profile of pool is frequently what motivates providers to deliver in this model. Of

course, as we move resources into a shared model, we have a much more challenging isolation story to tell. There is often a fundamental mismatch between the tools and mechanisms that provide isolation and the nature of tenants consuming a shared resource. This is further complicated by the fact that each resource we need to isolate in the pool model may require a different approach to enforcing isolation. While these challenges are real, they should *not* represent an opportunity to somehow relax your isolation requirements. This just means you'll have to work a bit harder to find the right combination of tools and construct to isolate some resources in a pooled model.

Before we dig into some specific pool isolation techniques, let's get a clear picture of how the pool model changes our approach to isolation. Generally, when we talk about isolating AWS resources, we focus on how AWS Identity and Access Management (IAM) can be used to control the interactions between resources. For a silo model, in fact, IAM represents a perfectly good model for expressing your tenant isolation policies. With the pool model, though, using these IAM constructs can be a bit more involved. The diagram in Figure 12 provides illustration of how silo and pool require separate isolation mindsets.

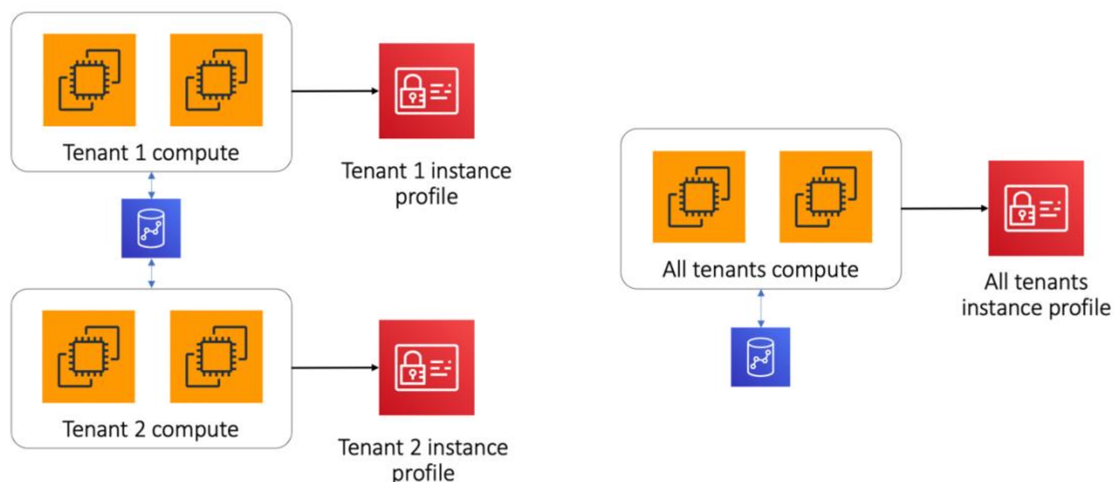


Figure 12 – IAM and Scoping Access

Here's you'll see two different ways of apply IAM policies to scope access of compute constructs. On the left we have two siloed deployments where tenants are running in their own infrastructure. These tenants are both accessing some other resource (in this case storage). When these instances were deployed, they were configured with separate IAM instance profiles for each tenant (tenant 1 and tenant 2). Since this binding was created at deployment time, we can be sure that these instances will be prevented from accessing the resources of another tenant.

On the right you'll see an example where we've deployed compute nodes in a pooled model. The compute that is running here will be running on behalf of all tenants. This reality directly impacts how we can scope the IAM profile for the compute that is deployed here. Instead of constraining the compute to a specific tenant, we must deploy these compute nodes with a profile that is open enough to support all tenants. This wider scope is where we run into the real challenges of the pool model. Now, we'll need to come up with new ways to implement the scoping of access that is enforced by your SaaS solution.

Given this unique aspect of pool isolation, you'll find that the options for implementing pool isolation will vary significantly. While it's beyond the scope of this paper to explore all the permutations of pool isolation, we can examine some common patterns to get a better feel for the different strategies that are often applied. The sections that follow will provide an overview of these strategies.

Run-time, Policy-Based Isolation with IAM

In the pooled environment, SaaS providers will typically turn to IAM to find a strategy to isolate their resources. However, as noted above, you'll need to be creative with how you apply IAM to achieve isolation in a pooled model. Instead of inheriting the IAM scoping of your compute node, you'll need to introduce your own code that will provide run-time enforcement of your pooled isolation model. The diagram in Figure 13 provides a conceptual view of this model.

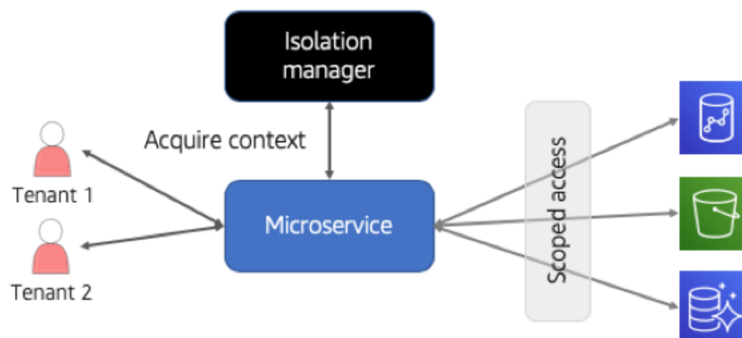


Figure 13 – Runtime Acquired Scoping

In this diagram, you'll see that we have a microservice that needs to access some downstream resources (databases, S3 buckets, etc.). This microservice was deployed in a pooled model, which means that it will be processing requests from multiple tenants. The job of this microservice is to ensure that, as it processes these requests, it

will apply constraints that will prevent tenants from crossing a boundary to another tenant's resources. In the diagram, you'll see that our microservice reaches out to the isolation manager to acquire a scoping context that is used to control interactions with and resources that are accessed by the code running in this microservice.

This conceptual model provides some view of the moving parts. However, to see this in action, we need to look at a more concrete strategy that explain how this context is express and applied. The diagram in Figure 14 provides a more in-depth look at how IAM can be used as part of this run-time scoping of access to tenant resources.

Here you'll see the full lifecycle of configuring and applying policies in a run-time model. In the first step of this process, the tenant onboards to your system. During this process, they setup the user for our tenant as well as the IAM policies for that tenant (steps 2 and 3). Once the tenant has onboarded, we then hit the microservice of our application (step 4). Because this microservice is running in a pooled model, it has been deployed with a broad IAM scope that enables it to access resources for all tenants. Our job, then, is to look at each request that is sent to this service and narrow the scope of that request based to a single tenant. We do that by asking the isolation manager for a set of credentials that are specific to the current tenant (step 5). This isolation manager will look-up the IAM policies for the tenant (step 6) and generate a tenant scoped set of credentials that are returned back to the calling microservice. Finally, this microservice uses these credentials to access a database (step 7). With these new tenant-scoped credential, the code of the microservice will be prevented from accessing the resources of another tenant.

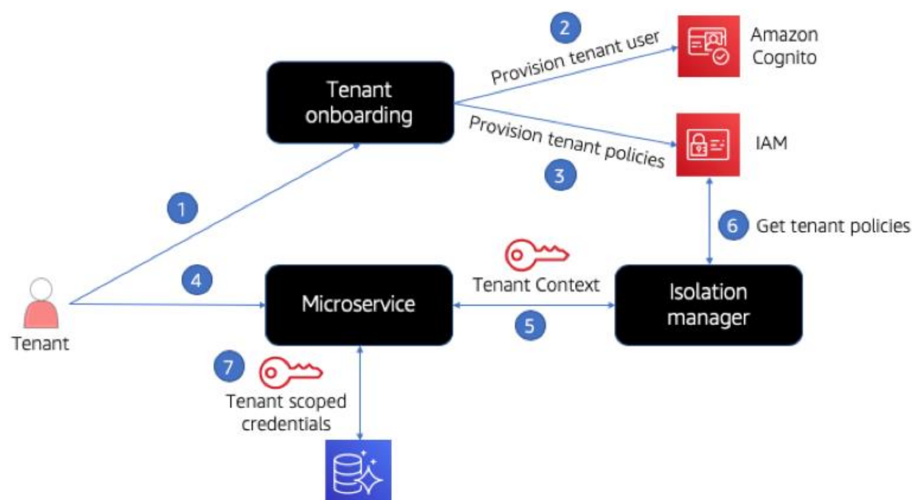


Figure 14 – Scoping with IAM Policies

In this model, we’re essentially saying that our microservice will have this tenant context applied each time it attempts to access another resource. This scoping is applied as a matter of an agreed upon convention where the microservice is expected to always acquire new credentials before accessing a tenant resource.

Scaling and Managing Pool Isolation Policies

While IAM policies provide powerful isolation constructs, they can also present SaaS providers with scaling challenges. If your system has a large number of tenants with a large population of policies, you may find that you will exceed the limits of the IAM service. You may also find it difficult to manage these polices as the number of tenants and the compexity of these policies grow. In these situations, some SaaS companies will attempt to alternate approach to how they generate and manage their IAM policies at run-time.

One approach to this challenge is to shift to a model where your IAM policies are generated in at run-time. The idea here is to have your system implement a mechanism that will examine the current context of a call and generate the required IAM policy on-the-fly. This moves the policies out of IAM (since they are transient) and enables you to address potential limits on the number of policies that are needed to support all of your tenants. The diagram in Figure 15 provides an overview of this dynamic policy generation mechanism.

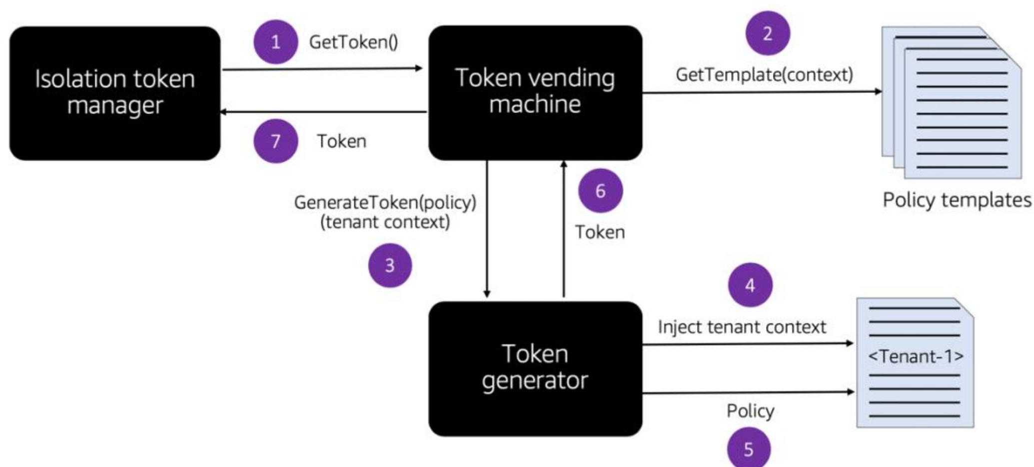


Figure 15 – Dynamic Policy Generation

In this flow, you’ll see that we start with the same isolation manager that we used in our prior example. However, instead of going directly the IAM to retrieve the policies need to scope access, we have a series of steps that are used to generate a policy. The

isolation manager first makes a request to the token vending machine to get a tenant scoped token (step 1). It's the job of the vending machine to go to the templates that you have pre-defined for your tenant isolation model (step 2). Think of these as template files that have all of the moving parts of a traditional IAM policy. However, key elements of the file are not filled in (those that represent our tenant context). You might, for example, fill in a table name or the leading key condition of an Amazon DynamoDB table with a tenant identifier.

Once you have the template that's needed, you now call out to the token generator to request a token (step 3). In this step, we also provide the current tenant context. The token generator then fills the tenant details into the template, leaving us with a fully hydrated IAM policy (steps 4 and 5). Finally, the token generator uses this policy to generate a token that is scoped according to the provided policy. This token is returned back to the isolation manager (steps 6 and 7). Now, this token can be used to access resources with the tenant context applied.

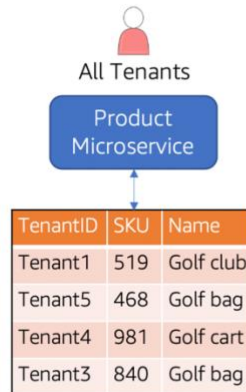
By moving these policies into templates, you take on the added responsibility of assuring that these policies enforce your tenant isolation requirements. Ideally, the details of this mechanism will be mostly outside the view of developers so the potential for something to go wrong is reduced.

One upside here is the management profile of this model. Should you choose to change something about your isolation policies, the path to applying these changes will be much more straightforward since there won't be a separate policy for each tenant. That, and you'll own the content lifecycle of these policy templates (versioning and deploying them through your own pipeline).

Pooled Storage Isolation Strategies

Isolating data in a pooled model is an area that gets lots of attention from SaaS providers. As data is co-mingled, SaaS developers become hyper-focused on identifying ways to ensure that each tenant's data is protected. In fact, while many SaaS providers are intrigued by the cost, management, and agility profiles of the pool model, they will often default to a silo model purely to address expected pushback they may get from customers that will may be hesitant to accept pooling of their data.

The general notion of pool storage isolation (for any storage service) is that the data for all tenants is represented in a shared storage construct. The diagram in Figure 16 provides an illustration of pooled storage.



TenantID	SKU	Name
Tenant1	519	Golf club
Tenant5	468	Golf bag
Tenant4	981	Golf cart
Tenant3	840	Golf bag

Figure 16 – Pooled Storage

Here you'll see that we have a product microservice that is storing its data in a pooled model. The table has an index in the first column that represents the key for each tenant. All of the tenant product data resides in this one table.

With this model, the challenge of isolating the data becomes much more complex. How do you create some virtual view of this table that is constrained to just those rows that belong to a given tenant? Also, how will this isolation be realized spanning each of the AWS storage services? The reality is, each service may require its own unique approach to implement isolation in the pooled model.

To get a better sense of this variation, let's start by looking at one example of how you might use IAM to implement pooled isolation with DynamoDB. As a fully managed storage service, DynamoDB offers you a rich collection of IAM mechanisms to control access to resources. This includes the ability to define a *leading key* condition in your IAM policy that can restrict access to the items in a DynamoDB table. The IAM policy shown in Figure 17 provides an example policy that demonstrates this approach to isolation.

The key area to focus on in this policy is the condition. This condition indicates that, when this policy is applied, all attempts to access the DynamoDB table will be limited to items that have key that matches the value of this leading key. So, in this case, the tenant identifier would be in the leading key, constraining access to data for a given tenant.

```
{
  "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"
          ]
      }
  }
}
```

Figure 17– DynamoDB Isolation with Leading Keys

Now, if we look at employing this same isolation model to Amazon Aurora PostgreSQL, you'll see that the mechanism is quite different. With Aurora PostgreSQL, you cannot use IAM to scope access to data at the row level. Instead, you'll need to use the row level security (RLS) feature of PostgreSQL to isolate your tenant data. The diagram in Figure 18 provides a simple example of how you'd setup RLS for a product table in your system.

```
-- 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);
```

Figure 18 - Pooled Isolation with PostgreSQL RLS

The first step in configuring RLS is to alter your table to enable row level security for that table. Then, you'll create an isolation policy for that that requires the `tenant_id` column to match the value of the current user (which is supplied contextually). Now, with these changes in place, all interactions with this table will be restricted to the rows that are valid for the current tenant.

In contrasting the DynamoDB and Aurora PostgreSQL approaches, you can see that you'll need to do some exploration with each storage service that you are using to find a

model that will let you achieve isolation. There are also cases where services may not offer a more granular isolation model. In these cases, you'll have to introduce your own mechanisms to enforce your pool isolation policies.

Application-Enforced Pool Isolation

Most of our attention so far has been on strategies for using IAM as the foundation our pooled isolation model. And, while IAM often represents a great fit for isolating resources, there can also be scenarios where IAM may not support the flavor of isolation that your application requires. This is where you may have to fall back and look at introducing other frameworks or tools to control access to your application's pooled resources.

Application-enforced isolation typically includes some model where you express policies (much like you do with IAM). These same frameworks often include policy enforcement mechanisms that will sit between you and your resources, authorizing your access to the resources. The diagram in Figure 19 provides a high-level conceptual view of the moving parts that might be part of an application-enforced policy model.

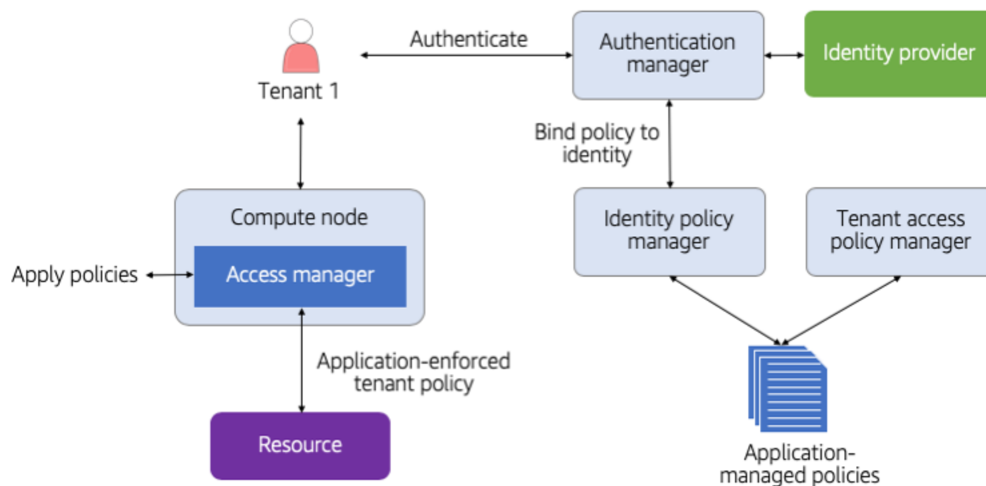


Figure 19 – Application-enforced Pool Isolation

In this example, your tenant would authenticate against an identity provider and introduce some construct that will identify the policies that were defined for this specific user (this could also happen in a downstream process). The key here is that the policies would then be connected to your user's identity, enabling downstream operations apply these policies in the context of a given user. Once you've authenticated, your identity would flow through the services of your system. Here there would need to be a library or

process that would sit between your code and the resource you're attempting to access, applying the policies that were bound to you as a user.

> **Note:** This approach is only meant to represent a conceptual model. The strategies that are employed by each framework may take a different approach to expressing and applying their policies.

It's worth noting that the boundaries of policy-based isolation and role-based access control (RBAC) often get blurred as part of this discussion. The tooling here, in fact, often contributes to this confusion. As a generality, though, we wouldn't want to equate RBAC to tenant isolation. In many cases, RBAC has a functional mapping where user roles (defined by an application) are used to control access to a system's functionality. That scope is different than drawing boundaries of isolation between the tenants of your system, which is less about a functional goal and more about preventing one tenant from accessing another tenant's data.

Pool for any Resource

Our coverage of pool here highlights the fundamental moving parts of implementing a pooled strategy. However, it does not touch on how pool might land in every AWS service. That is beyond the scope of this paper. That being said, the concepts and tradeoffs of pool isolation tend to be similar for most resources. As you look at the range of additional AWS services, you'll find yourself balancing the available isolation mechanisms with the efficiency of having a resource that is shared by tenants. In an ideal scenario, you could use a pooled model for every resource and still achieve all of your isolation goals. The reality is, though, you'll find scenarios where the isolation model for some resources will be challenging. In these cases, this may push you toward a silo model. That, or you'll absorb the effort to use some flavor of application-enforced isolation to realize your isolation goals.

Hiding the Details of Pooled Isolation

As we mentioned above, one key aspect of the pool model is that it relies on developers to conform to the overall model. Developers must, as a matter of convention, acquire the scoped context before accessing resources. Given the importance of compliance with this model, you'll often see companies creating mechanisms that simplify a developer's ability to align with the isolation policies adopted as part of a SaaS offering.

The general approach here fits very much with common design best practices. This usually translates into the creation libraries, modules, or lightweight frameworks that are shared by teams. The goal here is to move the mechanics of acquiring a scoped context into shared constructs that can be leveraged across your team. This diagram in Figure 20 provides a conceptual view of this notion of hiding away the details of isolation.

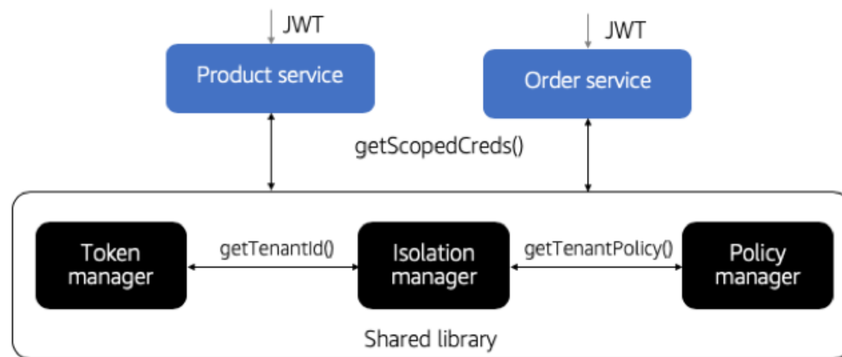


Figure 20 - Using Libraries for Isolation Standardization

Here you'll see that we have two microservices (product and order) that need to acquire credential to comply with the pooled isolation model of our system. What we've done here is moved all of the code and details of this process to shared libraries (these are not separate microservices). When our microservice needs scoped credentials, it will call into the isolation manager, passing in a JWT token that that was supplied to the microservices. This isolation manager will then get the `tenantId` from the token manager, which owns all the logic associated with cracking open the JWT and extracting tenant information. It will then get the policy for this tenant from the policy manager and use that policy to get a set of tenant-scoped credentials. These credentials would then be returned to the calling service.

There's nothing especially unique about this approach. This is simply applying the basic strategy of ensuring that reusable constructs are extracted so they can be versioned and shared more universally by your team. The key concept here is that you should attempt to push as much of the details of tenant isolation away from the view of your developers, making as simple as possible for them to apply your isolation scheme.

How you choose to implement this could also be influenced by the stack or compute construct that your application uses. With Lambda, for example, it may make sense to move these libraries to Lambda Layers where these horizontal concepts are versioned separately and universally referenced by you Lambda functions.

You may also look to introduce mechanisms that will take this completely outside of the view of your developers, intercepting and acquiring these scoped credentials before you get into the implementation of your microservices. With some languages, for example, you could use aspects to intercept incoming requests, acquire the scoped credentials, and inject them into the microservice. With Lambda functions, there are various open source *wrapper* libraries that could be used to inject scoped credentials into a Lambda functions. For some, these strategies may provide a stricter model for enforcing isolation

Isolation Transparency

We've talked about isolation mostly based on how it is realized within the design and architecture of your application. However, it's important to also think about isolation from the perspective of the tenants of your system. Even though SaaS developers and architects are constantly weighing their isolation options, it's still important to present tenants with a clear and consistent story around isolation that takes them away from the underlying details of your isolation strategy. The diagram in Figure 21 provides a conceptual view isolation that we want to present to customers.

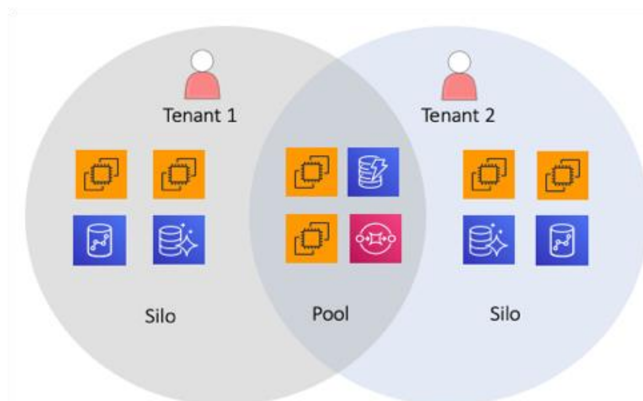


Figure 21 – Making Isolation Transparent

Here you'll notice that we have two tenants that have resources. Some of the resources are deployed in a silo model (on the left and right). Other resources for these tenants are deployed in a pool model (in the overlapping portion of these two circles). The idea here is that, despite the fact that there is a mix of silo and pool here, your system offers a comprehensive approach to isolation that prevents any cross-tenant access. To your customer, they just need assurance that this isolation is in place. Ideally, they won't need to know which resources are pooled and which are siloed.

Conclusion

After reviewing the isolation concepts outlined here, you should have a good sense of the landscape of isolation options you'll need to consider as you build out a SaaS solution on AWS. We explored a number of key patterns here, highlighting different models for implementing isolation that are directly influenced by the domain, compliance, operations, and performance profile of your SaaS application. We focused much of this discussion on the silo and pool isolation models, exploring the nuances of how these models are realized in different SaaS models. We also looked at how your isolation strategies can be influenced by the AWS services that are used to build your SaaS environment.

While implementing isolation can add layers of complexity to your SaaS solution, the need for a robust isolation model is core to implementing any best practices SaaS application. Any scenario where a tenant could end up accessing another tenant's resource—even inadvertently—could represent a significant setback to a SaaS business. This requires organizations to be hyper-vigilant about implementing isolation models that minimize their reliance authentication or well-behaved code as the pillars of their isolation strategy.

Contributors

Contributors to this document include:

- Tod Golding, Principal Partner Solutions Architect, AWS SaaS Factory

Further Reading

For additional information, see:

- Saas Tenant Isolation Patterns
- SaaS Storage Strategies Whitepaper
- Managing SaaS Identity Through Custom Attributes and Amazon Cognito
- SaaS and OpenID Connect: The Secret Sauce of Multi-tenant Identity and Isolation

Document Revisions

Date	Description
August 2020	First Publication