SaaS Lens

AWS Well-Architected Framework

SaaS Lens: AWS Well-Architected Framework

Copyright © 2022 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

Abstract and Introduction 1
- Abstract 1
- Introduction 1

Definitions 2
- Tenant 2
- Silo, Pool, and Bridge Models 2
- SaaS Identity 3
- Tenant Isolation 3
- Data Partitioning 3
- Noisy Neighbor 4
- Tenant Onboarding 4
- Tenant Tiers 4
- Tenant Activity and Consumption 4
  - Metering and Billing 4
- Tenant-Aware Operations 5

General Design Principles 6

Scenarios 8
- Serverless SaaS 8
  - Preventing Cross Tenant Access 9
  - Layers Hide Tenant Details 10
- Amazon EKS SaaS 11
- Full Stack Isolation 14
  - Unified Onboarding, Management, and Operations 15
- Hybrid SaaS Deployment 16
- Multi-Tenant Microservices 17
- Tenant Insights 18

The Pillars of the Well-Architected Framework 20
- Operational Excellence 20
  - Design Principles 20
  - Definition 20
  - Best Practices 21
  - Resources 26
- Security 26
  - Design Principles 26
  - Definition 26
  - Best Practices 27
  - Resources 36
- Reliability 37
  - Design Principles 37
  - Definition 37
  - Best Practices 37
  - Resources 41
- Performance Efficiency 41
  - Definition 41
  - Best Practices 42
  - Resources 46
- Cost Optimization 46
  - Definition 47
  - Best Practices 47
  - Resources 51
- Sustainability 51

Conclusion 53

Contributors 54

Document Revisions 55

Notices ... 56

SaaS Lens

Publication date: **December 3, 2020** (*Document Revisions* (p. 55))

Abstract

This paper describes the SaaS Lens for the **AWS Well-Architected Framework**, which enables customers to review and improve their cloud-based architectures and better understand the business impact of their design decisions. We address general design principles as well as specific best practices and guidance in five conceptual areas that we define as the *pillars* of the Well-Architected Framework.

Introduction

The AWS Well-Architected Framework helps you understand the pros and cons of decisions you make while building systems on AWS. By using the Framework you will learn architectural best practices for designing and operating reliable, secure, efficient, and cost-effective systems in the cloud. It provides a way for you to consistently measure your architectures against best practices and identify areas for improvement. We believe that having well-architected systems greatly increases the likelihood of business success.

In this “Lens” we focus on how to design, deploy, and architect your multi-tenant software as a service (SaaS) application workloads in the AWS Cloud. For brevity, we have only covered details from the Well-Architected Framework that are specific to SaaS workloads. You should still consider best practices and questions that have not been included in this document when designing your architecture. We recommend that you read the AWS Well-Architected Framework whitepaper.

This document is intended for those in technology roles, such as chief technology officers (CTOs), architects, developers, and operations team members. After reading this document, you will understand AWS best practices and strategies to use when designing architectures for SaaS applications.

Definitions

The AWS Well-Architected Framework is based on six pillars: operational excellence, security, reliability, performance efficiency, cost optimization, and sustainability. SaaS adds a new dimension of considerations to each of these pillars. The multi-tenant nature of SaaS applications, requires architects to reconsider how they address security, operational efficiency, stability, and agility of their SaaS environments. In this section, we will present an overview of the SaaS concepts that will be used throughout this document. There are 10 areas you should consider when building a SaaS architecture.

Topics

- Tenant (p. 2)
- Silo, Pool, and Bridge Models (p. 2)
- SaaS Identity (p. 3)
- Tenant Isolation (p. 3)
- Data Partitioning (p. 3)
- Noisy Neighbor (p. 4)
- Tenant Onboarding (p. 4)
- Tenant Tiers (p. 4)
- Tenant Activity and Consumption (p. 4)
- Tenant-Aware Operations (p. 5)

Tenant

A tenant is the most fundamental construct of a SaaS environment. As a SaaS provider building an application, you are making this application available to your customers. The customers that are signing up to use your environment are represented as the tenants of your system. Imagine, for example, that your organization has created an accounting service that you want to make available to other companies that will use your service to manage their businesses. Each one of these companies would be viewed as a tenant of your system.

Upon signing up, a tenant will typically provide user information for the tenant administrator. This tenant administrator can then log into the system and configure it based on the needs of their business. This includes having the ability to add users to a given tenant environment.

The software that is provided in this model is referred to as a multi-tenant SaaS system because each of the tenants of the service are consuming a single, shared system that supports the needs of these tenants through a unified experience. An update to the system, for example, would typically be applied to all tenants of that system.

Silo, Pool, and Bridge Models

SaaS applications can be built with a variety of different architectural models. Regulatory, competitive, strategic, cost efficiency, and market considerations all have some influence on the shape of your SaaS architecture. At the same time, there are strategies and patterns that are applied when defining the footprint of a SaaS application. These patterns fall into one of three categories—silo, bridge, and pool.

The **silo model** refers to an architecture where tenants are provided dedicated resources. Imagine, for example, as SaaS environment where each tenant of your system has a fully independent infrastructure

stack. Or, perhaps each tenant of your system has a separate database. When some or all of a tenant's resources are deployed in this dedicated fashion, we refer to this as a silo model. It's important to note that—even though the silo has dedicated resources—a silo environment still relies on a shared identity, onboarding, and operational experience where all tenants are managed and deployed via a shared construct. This differentiates SaaS from a managed service model where customers might be running separate versions of your product with separate onboarding, management, and operational experiences.

In contrast, the **pool model** of SaaS refers to a scenario where tenants share resources. This is the more classic notion of multi-tenancy where tenants rely on shared, scalable infrastructure to achieve economies of scale, manageability, agility, and so on. These shared resources can apply to some or all of the elements of your SaaS architecture, including compute, storage, messaging, etc.

The final pattern is the **bridge model**. *Bridge* is meant to acknowledge the reality that SaaS businesses aren't always exclusively silo or pool. Instead, many systems have a mixed mode where some of the system is implemented in a silo model and some is in a pooled model. For example, some microservices in your architecture might be implemented with silo and others might use pool. The regulatory profile of a service's data and its noisy neighbor attributes might steer a microservice to a silo model. Meanwhile the agility, access patterns, and cost profile of another microservice could tip it toward a pool model.

SaaS Identity

Most systems already rely on an identity provider for authentication. In the world of SaaS, we need to extend the notion of identity to incorporate tenancy into our definition of identity. This means that, after authenticating a user, we need to know who the user is as well as which tenant that user is associated with. This merging of the user identity with the tenant identity is referred to as a SaaS identity. This concept is a foundational element of a SaaS architecture, providing the tenant context that is used to implement the underlying multi-tenant policies and strategies that are part of a SaaS application.

Tenant Isolation

Tenant isolation is one of the foundational topics that every 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 each tenant is prevented from accessing another tenant's resources. Crossing this boundary in any form would represent a significant and potentially unrecoverable 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.

Regardless of how the isolation is implemented, each SaaS architecture needs to ensure that it has put in place the constructs that are needed to ensure that each tenant's resources have been effectively isolated.

Data Partitioning

As you look at different architecture patterns for representing multi-tenant data, you must make choices about how that data is organized. Will the data be stored in separate database, for example, or will it

be comingled in a shared construct? These multi-tenant storage mechanisms and patterns are typically referred to as data partitioning.

Noisy Neighbor

Noisy neighbor is a term that is often applied to general architecture patterns and strategies. The idea behind noisy neighbor is that a user of a system could place load on the system's resources that could have an adverse effect on other users of the system. The end result could be that one user could degrade the experience of another user.

This concept has expanded relevance in a multi-tenant environment where tenants may be consuming shared resources. Adding to the complexity here is that workloads in a multi-tenant environment can be unpredictable. The combination heightens the need for SaaS architectures to employ their own strategies that can manage and minimize the potential impacts of noisy tenants.

Tenant Onboarding

SaaS applications rely on a frictionless model for introducing new tenants into their environment. This often requires the orchestration of a number of components to successfully provision and configure all the elements needed to create a new tenant. This process, in SaaS architecture, is referred to as tenant onboarding. It's important to note that tenant onboarding can be initiated directly by tenants or as part of a provider-managed process.

Tenant Tiers

A SaaS application is often architected to support a range of market segments, providing separate pricing and experiences to a spectrum of customer profiles. These profiles are often referred to as tiers. Supporting the varying needs of these different tiers means introducing architectural constructs that can shape the experience of each tier. These tiering models can influence the cost, operations, management, and reliability footprint of a SaaS solution.

Tenant Activity and Consumption

In multi-tenant SaaS environments, it's important to have visibility into how tenants are using your application and imposing load on your system's architecture. Tracking this information at the tenant level allows you to assess your system's ability to effectively scale and support the constantly evolving workloads being placed on your environment. The metrics and insights that are collected from a SaaS system are frequently referred to as tenant activity and consumption.

Metering and Billing

SaaS products are often sold in a pay-as-you-go model where the cost of a product is determined based on the consumption profile of a customer. This model allows customers to have a pricing model that is more tightly coupled to the value and load they are placing on a SaaS system. In this mode, SaaS providers will define and introduce metering mechanism that will measure consumption. This metering data is typically sent to a billing system that aggregates the billing information and generates a bill. Consumption-based pricing represents one model for pricing that can be combined with additional pricing strategies (subscription, for example).

Tenant-Aware Operations

The operations experience for SaaS environments introduces the need for additional mechanisms and tooling that can be used to create tenant-aware insights into the activity and consumption patterns of individual tenants and tiers. The idea here is that SaaS providers need to be able to view system activity and health through the lens of individual tenants and tenant tiers. This is essential to diagnosing and evaluating the trends and patterns of activity and consumption for individual tenants.

General Design Principles

The Well-Architected Framework identifies a set of general design principles to facilitate good design in the cloud for SaaS applications:

- **There's no one-size-fits-all SaaS architecture**: The needs of SaaS businesses, the nature of their domain, their compliance requirements, the segments of their market, the nature of their solution—all of these factors have a distinct influence on the architecture of a SaaS environment. Every SaaS architecture should be surrounded with an operational and customer experience that realizes the agility and software as a service tenets that are core to succeeding as a SaaS offering. Regardless of how the system is architected, the system should enable tenants to be onboarded, managed, and operated through a single pane of glass that allows the SaaS organization to achieve the agility and economies of scale that are foundational to building a SaaS business.
- **Decompose each service based on its multi-tenant load and isolation profile**: If you're decomposing your system into services, your decomposition strategy should consider how multi-tenant loads, tenant tiers, and isolation requirements will influence the services that are part of your system. In these scenarios, each service needs to be considered separately. Some services might be able to pool data, for example, while others might need to silo the data they manage based on compliance or noisy neighbor considerations. You might also find that some services will be deployed in a silo model to enable tiering strategies. Premium tenants, for example, might have some services that are available in a silo model as part of the value story of the premium tier.
- **Isolate all tenant resources**: The success of a SaaS system relies heavily on a security model that ensures that tenant resources are always protected from any cross-tenant access. A robust SaaS architecture will introduce isolation strategies across all layers of the architecture, providing specific constructs that ensure that any attempt to access a tenant resource is valid for the current tenant context.
- **Design for growth**: The move to a SaaS model is often about growth for SaaS organizations. As you define the architectural and operational footprint of your SaaS offering, you must continually be thinking about how your environment will be able to support an accelerating wave of new tenants. SaaS architects must build a highly agile, frictionless environment that can accommodate spikes in tenant onboarding without adding significant operational overhead. The idea here is to allow for growth in your customer base that doesn't expand the operational or infrastructure footprint of your SaaS environment.
- **Instrument, capture, and analyze tenant metrics**: When you put multiple tenants into an environment—especially a shared environment—it can be challenging to have a clear view of how tenants are using your system. SaaS teams need to invest in metrics instrumentation that can surface insights into the features tenants are using, the load they are putting on your system, the bottlenecks they are facing, the cost profile of their activities, and so on. This data is core to analyzing tenant trends that directly impact the business, architectural, and operational health of a SaaS company and inform its strategy.
- **Onboard tenants through a single, automated, repeatable process**: SaaS is all about agility. A key piece of this agility story is the tenant onboarding process. A robust SaaS system will include a frictionless, repeatable process for onboarding new tenants to your system. This promotes scale and is core to enabling growth. It also ensures that new customers will have a faster path to value.
- **Plan to support multiple tenant experiences**: SaaS markets and customers don't all fit into a single profile. SaaS companies often need to support a range of tenant profiles that can place different demands on your architecture and operations. As a SaaS provider and architect, it's essential to model these tenant personas and build a single environment that includes the constructs and mechanisms needed to span a range of tenant experiences without requiring one-off versions of your product. It's important to identify the value boundaries of your system to enable the business to create tiers of your offering that can reach multiple segments and promote a customer's advancement through these tiers.

- **Support one-off requirements through global customization**: SaaS agility and innovation are achieved by having a single environment that is run by all customers. Being able to update, manage, and operate all customers collectively is foundational to SaaS. The reality is, though, some customers may request customizations. These customizations should be introduced as configuration options that are available to any customer. Keeping these features in the core of the offering enables a SaaS company to support one-off needs without undermining the agility, operational efficiency, and innovation goals of the business.
- **Bind user identity to tenant identity**: Every layer of your architecture is likely to need some notion of tenant context to be able to log data, record metrics, access data, and so on. This means that tenant context needs to become a first-class construct that can be resolved and easily accessed by the layers of your application without invoking another service. The authentication and authorization experience of your solution should bind the tenant identity (and potentially other tenant attributes) to the identity of the authenticated user. This will yield a *SaaS identity* that is passed through all the layers of your system, enabling easy access to tenant context.
- **Align infrastructure consumption with tenant activity**: The activity of tenants in a SaaS environment is often unpredictable. Which resources tenants are consuming, how they're consuming them, and when they are consuming them can vary significantly. The number of tenants in your system can also change regularly. While these factors can present scaling challenges, a robust SaaS architecture will employ policies that limit over-provisioning and align an application's infrastructure consumption with the real-time trends in tenant activity. This promotes tighter alignment between tenant workloads and the cost profile of your overall SaaS infrastructure.
- **Limit developer awareness of multi-tenant concepts**: While tenancy will flow though the layers of your architecture, it should be your goal to limit the degree to which developers have exposure to tenancy. As a rule of thumb, a developer's experience writing a multi-tenant service should not be all that different from writing a service that has no notion of tenancy. If developers need to introduce tenancy throughout their code, this will make it challenging to manage and enforce compliance with your application's multi-tenant policies and mechanisms. This means providing libraries and reusable constructs to developers that hide the details of tenancy.
- **SaaS is a business strategy—not a technical implementation**: SaaS environments and their underlying technology choices are shaped directly by the agility, innovation, and competitive needs of the business. The emphasis and mindset here centers around the creation of a service experience for customers that focuses on zero downtime, regular updates, and closer connection with customers. This means designing an architectural and operational footprint that can promote continual evolution and rapid response to market demands. A technically solid architecture that doesn't enable agility, innovation, and operational efficiency will be unlikely to keep pace with the competitive landscape of the market—especially if you're competing with other SaaS providers.
- **Create tenant-aware operational views**: Operations teams are presented with a new set of challenges in a multi-tenant environment. While having a global view of a system's health and activity remains important in SaaS environments, a robust SaaS operational footprint will also include insights into how specific tenants or tenant tiers are exercising your system. SaaS operations teams should construct dashboards and views that enable them to analyze and profile the activity and load of individual tenants. Being able to view and troubleshoot usage through the lens of individual tenants is essential to building a proactive, efficient multi-tenant operations experience.
- **Measure the cost impact of individual tenants**: The business, architects, and operations teams for a SaaS company often need to have a clear picture of how tenants are impacting the cost footprint of a SaaS environment. For example, are tenants in the basic tier imposing higher costs than tenants in the premium tier? Are tenant consumption patterns or features changing the cost profile of your environment? These are among the questions that can best be answered by have a clear view into tenant cost profiles. This is especially important to understand in environments where tenant resources are shared by multiple tenants. Collecting and surfacing this data often provides a SaaS business with valuable insights that can shape the architecture and business model of a SaaS company.

Scenarios

In this section, we will cover a series of scenarios that represent common patterns and strategies that are used when designing and building SaaS solutions on AWS. We will present the assumptions we made for each of these scenarios, the common drivers for the design, and a reference architecture of how these scenarios are realized using AWS architecture constructs.

Topics

- Serverless SaaS (p. 8)
- Amazon EKS SaaS (p. 11)
- Full Stack Isolation (p. 14)
- Hybrid SaaS Deployment (p. 16)
- Multi-Tenant Microservices (p. 17)
- Tenant Insights (p. 18)

Serverless SaaS

The move to a SaaS delivery model is accompanied by a desire to maximize cost and operational efficiency. This can be especially challenging in a multi-tenant environment where the activity of tenants can be difficult to predict. Finding a mix of scaling strategies that align tenant activity with the actual consumption of resources can be elusive. The strategy that works today might not work tomorrow.

These attributes make SaaS a compelling fit for a serverless model. By removing the notion of servers from your SaaS architecture, organizations are able to rely on managed services to scale and deliver the precise amount of resources your application consumes. This simplifies the architecture and operational footprint of your application, removing the need to continually chase and manage scaling policies. This also reduces the operational overhead and complexity, pushing more of operational responsibility to managed services.

AWS offers a range of services that can be used to implement a serverless SaaS solution. The diagram in Figure 1 provides an example of a serverless architecture.

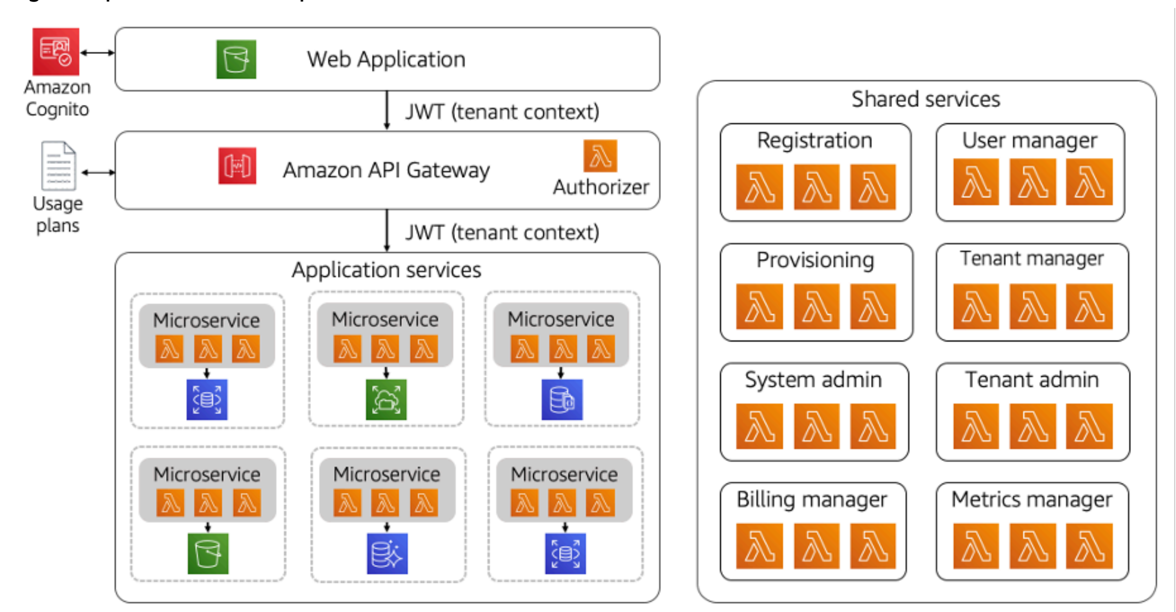


Figure 1: Serverless SaaS architecture

Here you'll see that the moving parts of a Serverless SaaS architecture aren't all that different than a classic serverless web application architecture. On the left of this diagram, you see that we have our web application hosted and served from an Amazon S3 bucket (presumably using one of the modern client frameworks like React, Angular, etc.).

In this architecture, the application is leveraging Amazon Cognito as our SaaS identity provider. The authentication experience here yields a token that includes our SaaS context that is conveyed via a JSON Web Token (JWT). This token is then injected into our interactions with all downstream application services.

The interaction with our serverless application microservices is orchestrated by the Amazon API Gateway. The gateway plays multiple roles here. It validates incoming tenant tokens (via a Lambda authorizer), it maps each tenant's requests to microservices, and it can be used to manage the SLAs of different tenant tiers (via usage plans).

You'll also see that we've represented a series of microservices here that are placeholders for the various services that would implement the multi-tenant IP of your SaaS application. Each microservice here is composed from one or more Lambda functions that implement the contract of your microservice. In alignment with microservices best practices, these services also encapsulate the data that they manage. These services rely on the incoming JWT token to acquire and apply tenant context wherever it is needed.

We've also shown storage here for each microservice. To conform to microservice best practices, each microservice owns the resources that it manages. A database, for example, cannot be shared by two microservices. SaaS also adds a wrinkle here, since the multi-tenant representation of data can change on a service-by-service basis. One service may have separate databases for each tenant (silo) while another might comingle the data in the same table (pool). The storage choices you make here are meant to be driven by compliance, noisy neighbor, isolation, and performance considerations.

Finally, on the right-hand side of this diagram, you'll see a series of shared services. These services deliver all the functionality that is shared by all of the tenants that are running in the left-hand side of the diagram. These services represent common services that are typically built as separate microservices that are needed to onboard, manage, and operate tenants.

More information on general serverless well-architected best practices can be found in the Serverless Applications Lens whitepaper.

Preventing Cross Tenant Access

Each SaaS architecture must also consider how it will prevent tenants from accessing the resources of another tenant. Some wonder about the need to isolate tenants with AWS Lambda since, by design, only one tenant can ever be running a Lambda function at a given moment in time. While this is true, our isolation must also consider ensuring that a tenant running a function is not accessing other resources that might belong to another tenant.

For SaaS providers, there are two basic approaches to implementing isolation in a serverless SaaS environment. This first option follows the silo pattern, deploying a separate set of Lambda functions for each tenant. With this model, you'll define an execution role for each tenant and deploy separate functions for each tenant with their execution role. This execution role will define which resources are accessible to a given tenant. You might, for example, deploy a collection of premium tier tenants in this model. However, this can be difficult to manage and may come up against account limits (depending on how many tenants your system supports).

The other option here aligns more with the pool model. Here, functions are deployed with an execution role that has a scope that's broad enough to accepts calls from all tenants. In this mode, you must apply

isolation scoping at runtime in the implementation of your multi-tenant functions. The diagram in Figure 2 provides an example of how this would be addressed.

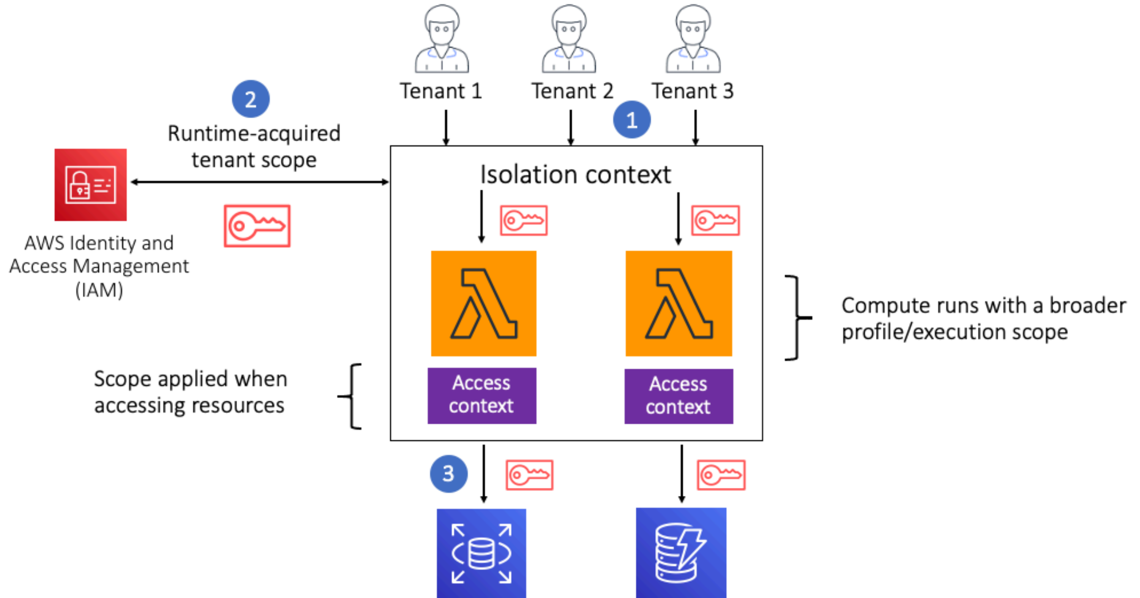


Figure 2: Isolation in a serverless environment

In this example, you'll see that we have three tenants accessing a set of Lambda functions. Because these functions are being shared, they are deployed with an execution role that covers all tenants. Within the implementation of these functions, our code will use the context of the current tenant (supplied via a JWT) to acquire a new set of tenant-scoped credentials from AWS Security Token Service (AWS STS). Once we have these credentials, they can be used to access resources with tenant context. In this example, we're using these scoped credentials to access storage.

It's important to note that this model does push part of the isolation model into the code of your Lambda functions. There are techniques (function wrappers, for example) that can introduce this concept outside the view of developers. The details of acquiring these credentials can also be moved to a Lambda layer to make this a more seamless, centrally managed construct.

Layers Hide Tenant Details

One of our goals with any SaaS architecture is to limit developer awareness of tenant details. In serverless SaaS environments, you can use Lambda layers as a way to create shared code that can implement multi-tenant policies outside the view of developers.

The diagram in Figure 3 provides an example of how Lambda layers can be used to address these multi-tenant concepts. Here you'll see that we have two separate microservices (Product and Order) that have a need to publish log and metrics data. The key detail here is that both services need to inject tenant context into their log messages and metric events. However, it would be less than ideal to have each service implementing these policies on their own. Instead, we've introduced a layer that includes code that manages the publishing of this data.

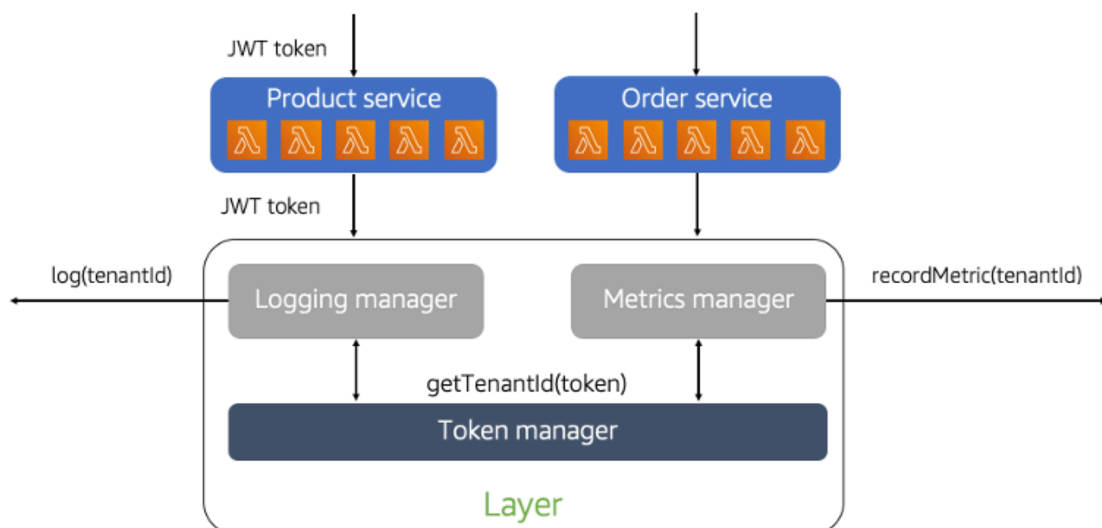


Figure 3: Lambda layers hide away tenant details

You'll notice that our layer includes logging and metrics helpers that accept a JWT token. This allows each microservice to simply call these functions and supply the token without any concern for which tenant they are working with. Then, within the layer, the code uses the JWT token to resolve the tenant context and inject it into the log messages and metric events.

This is just a snippet of how layers can be applied to hide tenant context. The real value is that the policies and mechanisms for injecting tenant context are completely removed from the developer experience. They're also updated, versioned, and deployed separately. This allows these policies to be more centrally managed in your environment without introducing separate microservices that could add latency and create bottlenecks.

Amazon EKS SaaS

For many SaaS providers, the profile of Amazon Elastic Kubernetes Service (Amazon EKS) represents a good fit with their microservices development and architectural goals. It provides a way to build and deploy multi-tenant microservices that can help them realize their agility, scale, cost, and operational goals without requiring a complete shift in their development tooling and mindset. The rich community of Kubernetes tools and solutions also offers SaaS developers a range of different options for building, managing, securing, and operating their SaaS environments.

For container-based environments, much of the architecture is focused on how to successfully ensure that we're preventing cross-tenant access. While there can be a temptation to allow tenants to share containers, this presumes that tenants would be comfortable with a notion of soft multi-tenancy. For most SaaS environments, though, the isolation requirements demand a more robust implementation of isolation.

These isolation factors can have a significant impact on the architectural model that gets built with Amazon EKS. The general guidance for building SaaS architectures with Amazon EKS is to prevent any sharing of containers across tenants. While this adds complexity to the footprint of the architecture, it addresses the fundamental need to ensure that we have created an isolation model that will address the domain, compliance, and regulatory needs of multi-tenant customers.

Let's look at a sample architecture to see the fundamental elements of a SaaS Amazon EKS environment. Since there are lots of moving parts to this solution, let's start by looking at the shared services that are used to support the core, horizontal concepts that span all of our tenants (shown in Figure 4).

First, you'll notice that we have the foundational elements that are part of any highly available, highly scalable AWS architecture. The environment includes a VPC that consists of three Availability Zones. Routing of inbound traffic from tenants is managed by Amazon Route 53, which is configured to direct incoming application requests to the endpoint defined by our NGINX ingress controller. The controller enables selected routing within our Amazon EKS cluster that is essential to the multi-tenant routing that you'll see below.

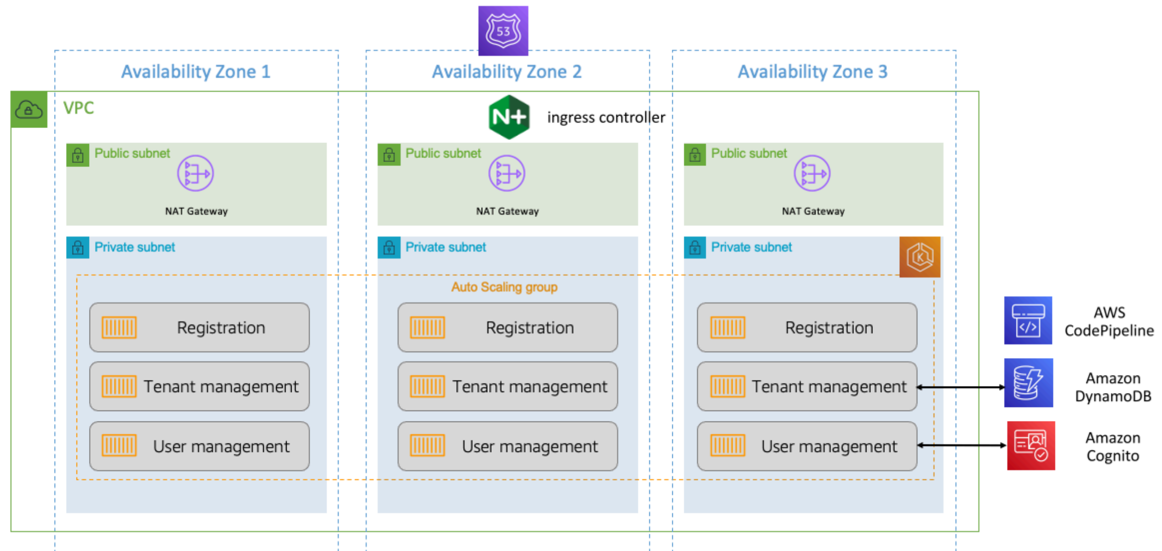


Figure 4: Amazon EKS SaaS shared services architecture

The services running in the Amazon EKS cluster represent a sampling of a few of the common services that are typically part of a SaaS environment. Registration is used to orchestrate the onboarding of new tenants. Tenant management manages the state and attributes of all the tenants in the system, storing this data in an Amazon DynamoDB table. User management provides the basic operations to add, delete, enable, disable, and update tenants. The identities it manages are stored in Amazon Cognito. AWS CodePipeline is also included to represent the tooling that is used to provision each new tenant that is onboarded to the system.

This architecture only represents the foundational elements of our SaaS environment. We now need to look at what it means to introduce tenants into this environment. Given the isolation considerations described previously, our Amazon EKS environment will create separate namespaces for each tenant and secure those namespaces to ensure that we have a robust tenant isolation model.

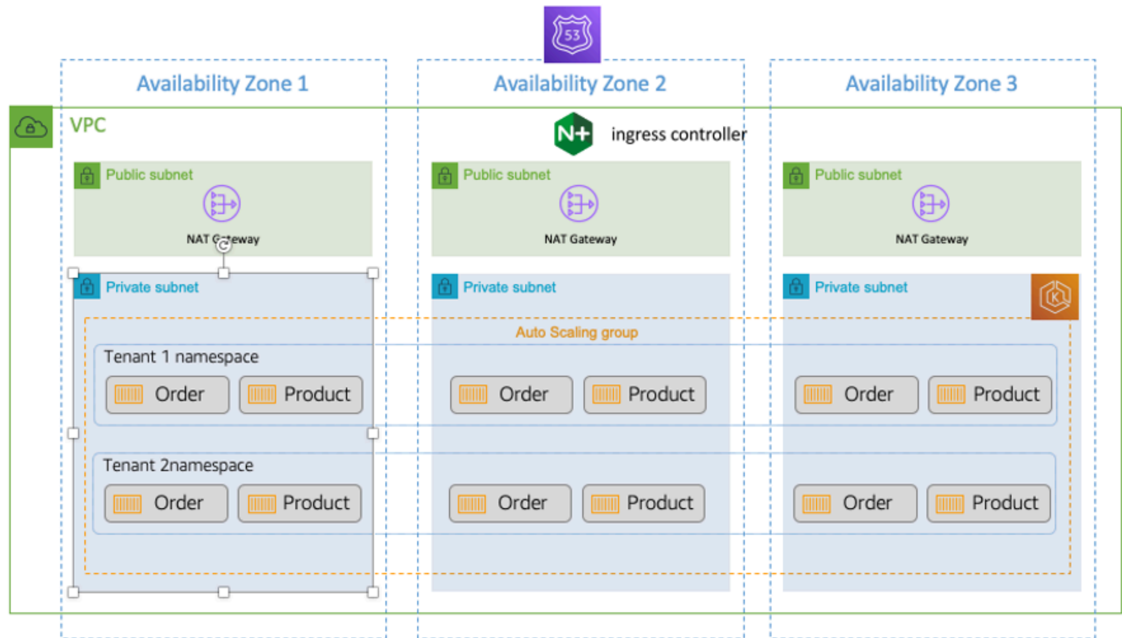


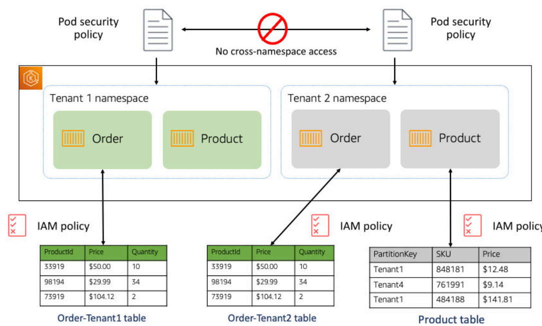
Figure 5: Deploying tenant environments in Amazon EKS

The diagram in Figure 5 provides a view of these namespaces within our SaaS architecture. On the surface, this architecture looks very much like the previous baseline diagram. The key difference is that we've deployed the services that are part of our application into separate namespaces. In this example, there are two tenants with distinct namespaces. Within each, we have deployed some sample services (Order and Product).

Each of the tenant namespaces are provisioned by the registration service that is shown above. This would use continuous delivery services (like AWS CodePipeline) to kick-off a pipeline that creates the namespace, deploys the services, creates tenant resources (databases, etc.), and configures the routing. This is where the ingress controller comes into play. Each provisioned namespace creates a separate ingress resource for each of the microservices in that namespace. This enables tenant traffic to be routed to the appropriate tenant namespace.

While namespaces allow you to have clear boundaries between the tenant resources in your Amazon EKS cluster, these namespaces are more of a grouping construct. The namespace alone does not ensure that your tenant loads are protected from cross-tenant access.

To enhance the isolation story of our Amazon EKS environment, we'll need to introduce different security constructs that can restrict the access of any tenant running in a given namespace. The diagram in Figure 6 provides a high-level illustration of an approach you can take to control the experience of each tenant.



ProductId	Price	Quantity
33919	$50.00	10
98194	$29.99	34
73919	$104.12	2

ProductId	Price	Quantity
33919	$50.00	10
98194	$29.99	34
73919	$104.12	2

PartitionKey	SKU	Price
Tenant1	848181	$12.48
Tenant4	761991	$9.14
Tenant1	484188	$141.81

Figure 6: Isolating tenant resources

There are two specific constructs introduced here. At the namespace level, you'll see that we have created separate pod security policies. These are native Kubernetes networking security policies that can be attached to a policy. In this example, these policies are used to limit network traffic between tenant namespaces. This represents a coarse-grained way to prevent one tenant from accessing the compute resources of another tenant.

In addition to securing the namespaces, you also must ensure that the resources accessed by the services running in a namespace are restricted. In this example, we have two examples of isolation. The Order microservice uses a table per tenant model (silo) and has IAM policies that restrict access to a specific tenant. The Product microservice uses a pooled model where tenant data is comingled and relies on an IAM policy that's applied to each item to restrict tenant access.

Full Stack Isolation

SaaS organizations are motivated by the economies of scale that come with sharing infrastructure resources across tenants. At the same time, there are real-world factors that might result in a SaaS architecture where some or all of the tenants within your environment require dedicated (siloed) infrastructure. Compliance, noisy neighbor, tiering strategies, legacy technologies—these are among the long list consideration that could lead you to offering tenants their own infrastructure. This is referred to as a siloed or full stack isolation SaaS.

This approach is frequently mistaken for a managed service provider (MSP) where a company's product is installed and managed on a one-off basis for individual customers. This approach, while valid, does not align with the SaaS tenets. The key to a SaaS model is to adopt a value system where *all* tenants are managed and operated through a single, unified experience. This means that every tenant is running the same version of the software, they all get deployments at the same time, they are all onboarded through the same process, and they are all managed through a single operational experience, which applies to all tenants.

This approach doesn't deliver the cost efficiencies of a shared infrastructure model (pool). Its distributed nature can also make operations and deployment more complex. Still, done right, a full stack model can still achieve the agility, innovation, and operational efficiency goals that are central to the SaaS mindset.

The diagram in Figure 7 provides a clearer picture of the full stack isolation (silo) model. Each tenant environment in this model is straightforward. You'll see here that we've provided a separate VPC for each tenant. These VPCs hold the full isolated stack that will be used by each tenant. The compute and storage constructs could be composed from any combination of AWS services. The key is that each tenant environment is meant to have the same infrastructure configuration and the same version of your product. So, adding new tenants is as simple as provisioning another VPC with the same infrastructure footprint for each tenant.

You'll also notice that this model use Amazon Route 53 to route the incoming traffic to the appropriate VPC. The routing relies on a model where subdomains are assigned to each tenant. This is a very common pattern in SaaS environments. This routing can also be achieved by inspecting the contents of a tenant JWT token, determining their tenant context, and triggering routing rules through an injected tenant header. This approach, however, can push account limits and might require tuning to address the latency of a runtime resolution of a tenant's identity. Still, it is a valid option for some SaaS environments.

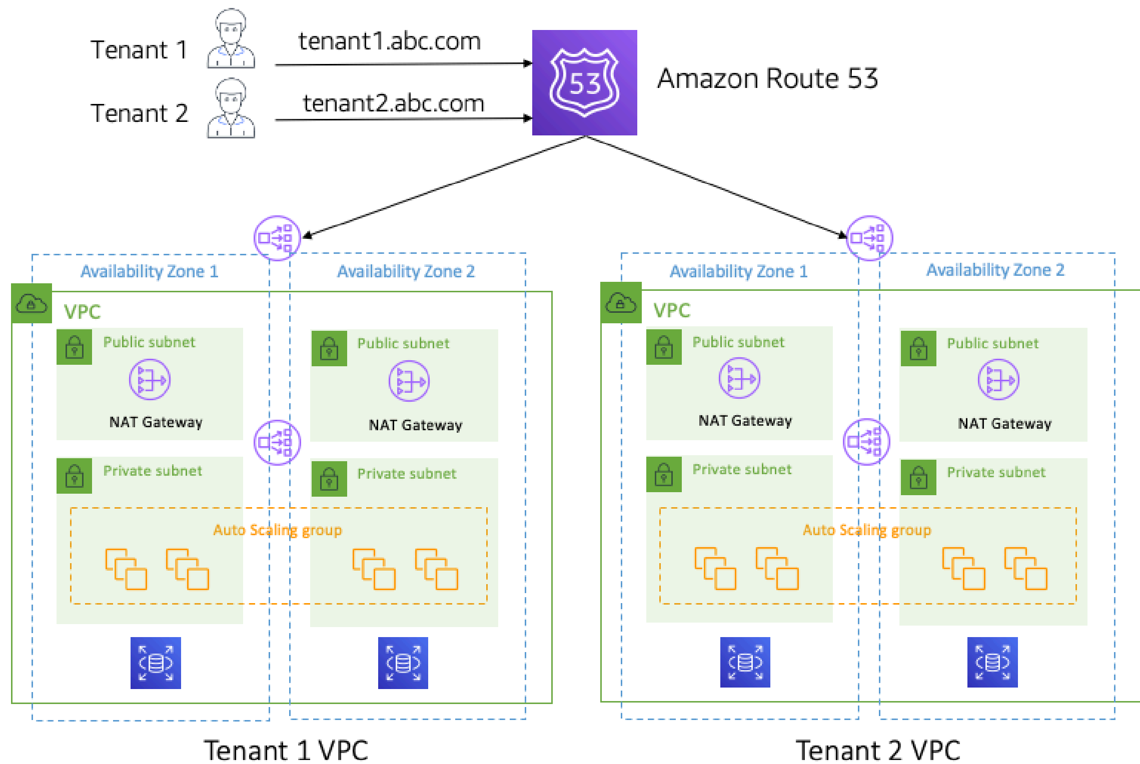


Figure 7: Full stack (silo) isolation

While this example illustrates a VPC-per-tenant model, there are other architectures that can be used to implement the full stack model. Some organizations might choose to use an account-per-tenant model where each tenant environment is deployed in a separate provisioned account. The option you choose will vary depending on the number of tenants you need to support and the overall management experience that you're targeting. In some cases, the number of tenants you need to support could exceed the number of VPCs that can be created within an AWS account.

Unified Onboarding, Management, and Operations

The real SaaS elements of the full stack isolation (silo) model come in when we look at the onboarding, management, and operations of this experience. To realize the full value proposition of SaaS, this model must introduce a collection of services that can be used to manage and operate all of the tenant environments.

The diagram in Figure 8 provides a view of these extra layers of services. What we introduce here is an entirely separate set of services that are built around the needs of the SaaS provider's administration and management experience.

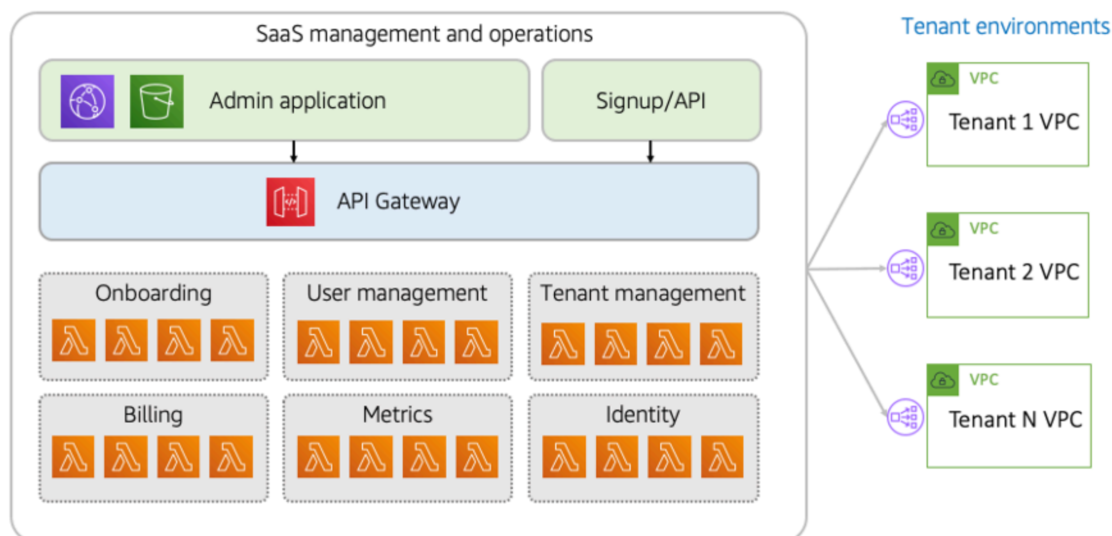


Figure 8: Managing full stack isolated environments

While the stack of our tenant environments is shaped by domain, legacy, and other requirements, the stack of our management experience is completely driven by the goals and usage patterns that can be quite different than those that apply to tenants.

This management very much conforms to a classic serverless SaaS application model. You'll see that we have two entry points into our administration environment. One is the Admin application that uses Amazon CloudFront and Amazon S3 to serve up the application experience that is used by SaaS administrators to manage and configure tenant environment. You'll also see the signup and API experience highlighted here. This represents the ability to trigger tenant onboarding via a public API request. This API can also be part of a SaaS provider's developer API experience.

Finally, we've also highlighted a collection of microservices that represent the shared services that are used to orchestrate the onboarding, management, and operations experience. We've chosen Lambda as the model for our microservices, based on its management, scale, and cost efficiency model. These services could be implemented with other AWS compute services.

On the right-hand side of the diagram, you'll see the individual VPCs that represent each of our tenant environments. In this VPC-per-tenant model, you'll need to open a valid path for your management plane to be able to configure and access the resources of these tenant environments. AWS offers a number of options for this, including AWS PrivateLink, VPC peering, and Amazon EventBridge. The option you choose will depend on the nature of the management experience you are building.

Hybrid SaaS Deployment

The needs of customers in SaaS environments can vary significantly for a business. While there are significant cost and operational advantages to building a SaaS solution that has all customers running in shared infrastructure model (pool), there might be times when customers are not willing to share infrastructure resources with other tenants.

This need for one-off customers to have their own environment can be challenging for SaaS providers. While SaaS providers feel the business pressure to support this model, they also know that supporting variations of this nature will undermine the overall SaaS goals of the business. Each new one-off customer can add operational overhead and complexity. As each new customer is added in this model, you quickly find yourself sliding further away from the innovation, agility, and cost efficiency benefits that come from a shared infrastructure model.

There are, however, architectural and operational strategies that can embrace this model without fully compromising your SaaS vision. The diagram in Figure 9 provides a conceptual view of how you could address this challenge.

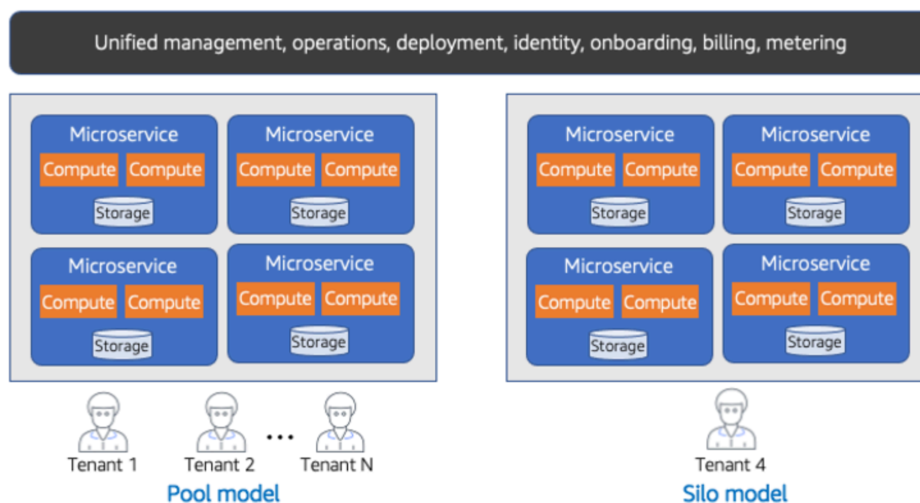


Figure 9: Hybrid deployment model

In this diagram, you'll notice that we have two separate deployments of our SaaS environment. The left side is deployed in a pooled model with shared infrastructure. The majority of our tenants are running in this environment. However, on the right side, we've deployed a separate copy of the multi-tenant environment that is hosting one tenant (tenant 4).

The key to this strategy is that both environments (the left and right) are running the *same* version of the SaaS application. Any new changes or updates are applied to both environments at the same time. More importantly, you'll see that we have a single, unified management experience that spans both of these environments. Onboarding, identity, metering, and billing are shared and common to both environments.

This shared experience allows a SaaS organization to manage and operate these environments through a single pane of glass. By fitting these one-off tenants into this model, you're able to minimize the impact to the overall agility and operational efficiency of your SaaS environment. This model impacts cost efficiency and adds operational complexity. However, it can represent a reasonable compromise for many SaaS companies.

Multi-Tenant Microservices

The microservices running in a multi-tenant environment must address additional considerations. These microservices must be able to reference and apply tenant context within each service. At the same time, it's also our goal to limit the degree to which developers need to introduce any tenant awareness into their code.

To achieve this goal, SaaS microservices—regardless of their compute model—should introduce libraries, modules, and shared constructs that can push tenant-specific processing into code. These constructs hide the policies and mechanisms that are needed to resolve and apply tenant context.

The diagram in Figure 10 provides a view into how you can streamline multi-tenant development of your microservices. The key idea here is that we've taken anything that relies on tenant context and used libraries to apply multi-tenant policies.

This example illustrates a number of scenarios where a SaaS microservice might need access to tenant context. The flow starts with a call into our microservice with a request to get a list of products. Somewhere in the code of our service, your service logs a message. The microservice developer simply logs a message to a logging wrapper. This wrapper gets the tenant context from the Token Manager and injects that context into our message that is, in this example, published to Amazon S3. Our log policies and how tenant data lands in logs in managed by whichever policies we choose to include in our logging helper.

This theme continues across the rest of the experience here. Our call to getProducts() first gets the tenant identifier from the Token Manager. It then uses this context to get tenant-scoped credentials from an isolation manager before using these credentials to get the product data from DynamoDB.

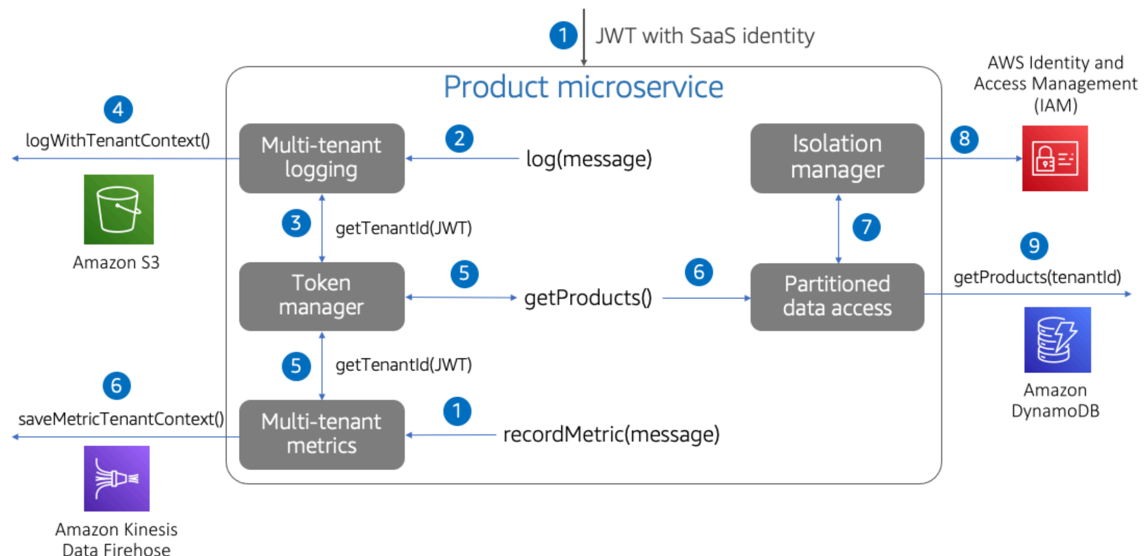


Figure 10: Developing multi-tenant microservices

Finally, our service records a metric (perhaps execution time) using the same mechanism for resolving the tenant identifier and injecting tenant context into the metrics messages.

The practices outlined here are not meant to introduce heavyweight constructs into your microservices. This abstraction of tenancy follows the general design practices of pushing common concepts into shared code. It's important to note that all of the concepts represented in this diagram are running within the context of a single microservice. Each of these blocks simply represent code that is reused by your microservice. Breaking these concepts into separate services would add latency and complexity that would typically not be justified.

Tenant Insights

As companies move into a multi-tenant model, it becomes increasingly challenging to capture and surface insights into how tenants are using your application. This is especially true with the resources of your SaaS environment that are shared by tenants.

At the same time, with all your tenants potentially running in one shared environment, the ability to have a clear view into tenant activity and consumption is essential to building a robust SaaS offering. Knowing what features of the product tenants are using, knowing how tenants are pushing the architecture of your environment, and having visibility into how different tiers of tenants are scaling are among the insights that SaaS companies need to be able to effectively operate a healthy SaaS business.

The scope of SaaS metrics is intentionally broad. This is because metrics are consumed in multiple contexts by different roles within a SaaS organization. This means you need to be thinking beyond the infrastructure health. SaaS metrics often include business agility, feature consumption, microservice activity, scaling trends, and so on. The idea is to capture and correlate tenant and tier usage trends with application and consumption activity.

You can imagine having various dashboards for the different roles that would consume this data. A product manager, for example, might want insights into feature-oriented metrics. Meanwhile, an operations person might be using this data to assess the health and consumption trends of individual tenants and tiers.

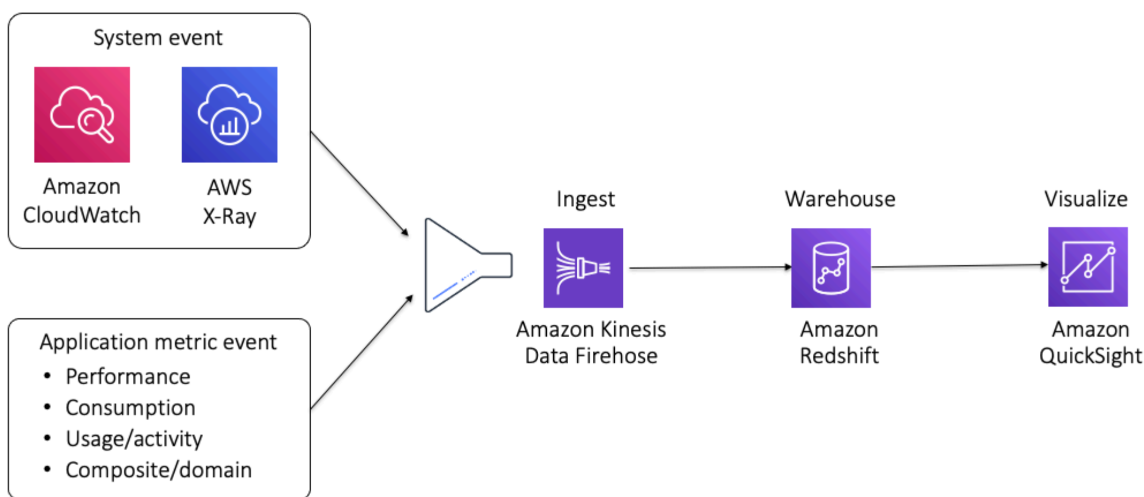


Figure 11: Ingest and visualize SaaS metrics

The architecture for capturing and surfacing this data is relatively straightforward. It includes all the mechanisms needed to convey, publish, ingest, aggregate, and visualize these SaaS metrics. The diagram in Figure 11 provides a view of an architecture that could be used to building your SaaS metrics environment on AWS.

The sources of the metric data you'll want to capture could be somewhat diverse. As the diagram suggests, your solution might need to capture some of the metrics from native AWS sources (CloudWatch, for example). However, a significant portion of this data will come from the code that you instrument into your SaaS application. It's here that you'll want to invest in publishing metrics that best capture the insights that can maximize the business and operational value of your domain's data.

The data from these sources is typically represented with a common schema that can represent the different types of consumption in your application (counts, durations, features, etc.). In this example, this data is ingested with Amazon Kinesis Data Firehose, which publishes the data to Amazon Redshift. Finally, Amazon QuickSight is used to build the different dashboards that are used across the organizations to view the trends of tenants and tiers.

The Pillars of the Well-Architected Framework

This section describes each of the pillars, and includes definitions, best practices, questions, considerations, and key AWS services that are relevant when architecting solutions for multi-tenant SaaS applications.

For brevity, we have only included questions that are specific to SaaS workloads. Questions that have not been included in this document should still be considered when designing your architecture. We recommend that you read the AWS Well-Architected Framework whitepaper.

Pillars

- Operational Excellence Pillar (p. 20)
- Security Pillar (p. 26)
- Reliability Pillar (p. 37)
- Performance Efficiency Pillar (p. 41)
- Cost Optimization Pillar (p. 46)
- Sustainability pillar (p. 51)

Operational Excellence Pillar

The **operational excellence** pillar includes the ability to run and monitor systems to deliver business value and to continually improve supporting processes and procedures.

The operational excellence pillar provides an overview of design principles, best practices, and questions. You can find prescriptive guidance on implementation in the Operational Excellence Pillar whitepaper.

Design Principles

In the cloud, there are a number of principles that drive operational excellence.

Definition

There are three best practice areas for operational excellence in the cloud:

- Prepare
- Operate
- Evolve

Operations teams need to understand their business and customer needs so that they can effectively and efficiently support business outcomes. Operations creates and uses procedures to respond to operational

events and validates their effectiveness to support business needs. Operations collects metrics that are used to measure the achievement of desired business outcomes. Everything continues to change—your business context, business priorities, customer needs, etc. It's important to design operations to support evolution over time in response to change and to incorporate lessons learned through their performance.

Best Practices

Topics

- Prepare (p. 21)
- Operate (p. 21)
- Evolve (p. 25)

Prepare

There are no operational practices unique to SaaS applications.

Operate

SaaS OPS 1: How are you able to effectively monitor and manage the operational health of a multi-tenant environment?

In multi-tenant environments, where all of an organization's customers might be deployed in a shared infrastructure model, the need for a more detailed operational view of health is often essential to the success and growth and of a SaaS business. Any outage or health issue has the potential of cascading across all the tenants of your system and taking a SaaS service down for all customers. This means that SaaS organizations must place a premium on creating an operational experience that will enable operations teams to effectively analyze and respond to continually shifting workloads of a SaaS environment.

Building a robust multi-tenant operational experience requires SaaS companies to create or customize tooling to introduce the more granular views of health and activity that are needed in a SaaS environment. This often means using a combination of existing tools and custom solutions to create an experience that supports tenancy and tenant tiers as first-class concepts within the operational experience.

A SaaS operations dashboard, for example, should include views of health that are presented through the lens of tenants and tenant tiers. While viewing the global view of health is still part of this experience, a SaaS operations team also needs the ability to see the health and activity of tenants. The diagram in Figure 12 provides a conceptual view of one way a SaaS provider might surface tenant activity as a first-class construct.

The simplified view in this diagram highlights a few samples of operational views that would add value in multi-tenant environment. At the top-left of the page, you'll see a view of health for the most active tenants. The color indicators shown could focus attention to tenants that might be experiencing issues that, when looking at a global view of health, have not been surfaced. This allows operations to react and respond proactively to any issues that might not be entirely apparent to a tenant.

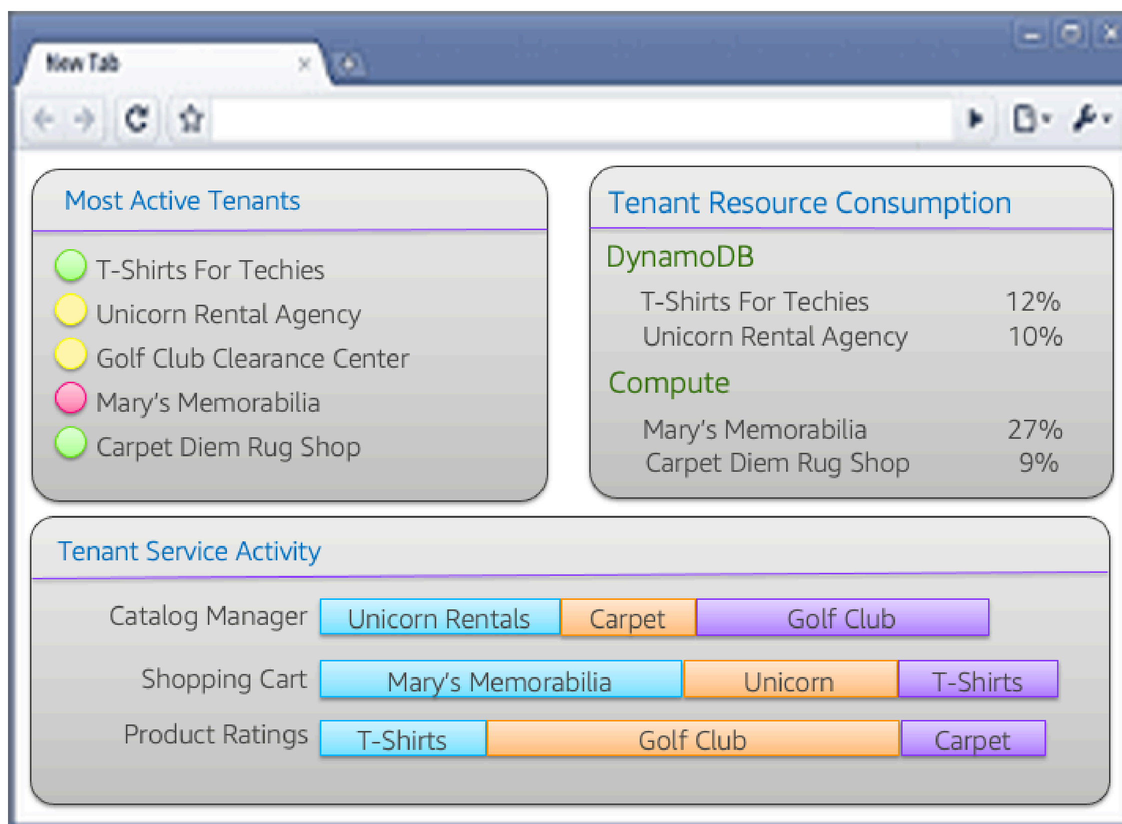


Figure 12: Tenant-aware operations views

In the top-right of this view, you'll see data on tenant resource consumption. The idea here is to view consumption of AWS resources on a tenant-by-tenant basis, allowing you to see how particular tenants are exercising specific services. At the bottom, you'll see a view of consumption that illustrates how tenants are consuming the various microservices of your application. The data here lets operations have views into how tenants are consuming the various services of your system and determine if a particular tenant or tier might be saturating a particular microservice.

In addition to providing this general view of tenant activity, your operational experience should also include the ability to drill into the operational data for individual tenants and tiers. Imagine support scenarios where a specific tenant or tenant tier is experiencing issues. In these cases, you'll want to be able to access operational data and view it through the lens of that particular tenant or tier. This is essential to being able to troubleshoot and diagnose issues in a multi-tenant setting.

Creating these operational views relies on having access to operational data that includes the tenant and tiering context that's required to create the operational views that can analyze insights by tenant or tenant tier. This requires SaaS architects to give care consideration to how and where they can inject tenant context into the various mechanisms that are used to record health and activity events. For example, logs should include mechanisms to ensure that tenant context (such as tenant identifier and tier) are injected into your system's log data.

The views that you choose to construct, however, will vary based on the nature of your application's design and architecture. Generally, teams should be thinking about the operational views that will enable operations teams to effectively monitor tenant trends and build the instrumentation into their environments from the outset. Adding these concepts into your application later in your development process is more challenging and will likely undermine both development and operational experiences.

Note

The numbering of the questions in this whitepaper has been changed to match the order in the SaaS Lens of the AWS Well-Architected Tool. **SaaS OPS 2** is described in the following best practice, Evolve (p. 25).

SaaS OPS 3: How are new tenants onboarded to your system?

SaaS solutions are highly focused on maximizing agility and innovation. Tenant onboarding plays a key role in this agility story. By creating an architecture that promotes a frictionless, repeatable onboarding process, SaaS organizations are able to streamline, optimize, and scale their ability to introduce new tenants into their SaaS environment. This enables SaaS companies to support rapid growth and offer customers an experience the accelerates their overall time to value.

It's important to note that automated onboarding applies to both B2C and B2B SaaS environments. While some SaaS offerings might not include a self-service onboarding experience, this does not reduce the need for a frictionless onboarding experience. Even when onboarding is an internally executed process, it should still automate all the elements of tenant creation. The need for reduced friction is foundational to creating a solution that aligns with SaaS best practices.

Generally, the onboarding process for a SaaS application requires the orchestration of a series of shared services that can configure and provision all the resources needed to introduce a new tenant. The diagram in Figure 13 provides a high-level view of how you could implement this onboarding via a series of microservices.

The flow of the onboarding experience represented in this diagram covers all the steps needed to introduce a tenant and have them begin using your SaaS system. At the front of this process, a tenant calls our registration service requesting to create a new tenant. From this point forward, the registration service will then sit at the middle of the onboarding process, orchestrating all the services needed to create the new tenant environment.

The next step in this process is to create a new user. This new user will represent the administrator for this new tenant. To support this process, we've included a user management service. This service doesn't hold data about the user, but it creates the user in an identity provider (in this case Amazon Cognito). It also creates any IAM policies that are needed to support the isolation requirements of this tenant.

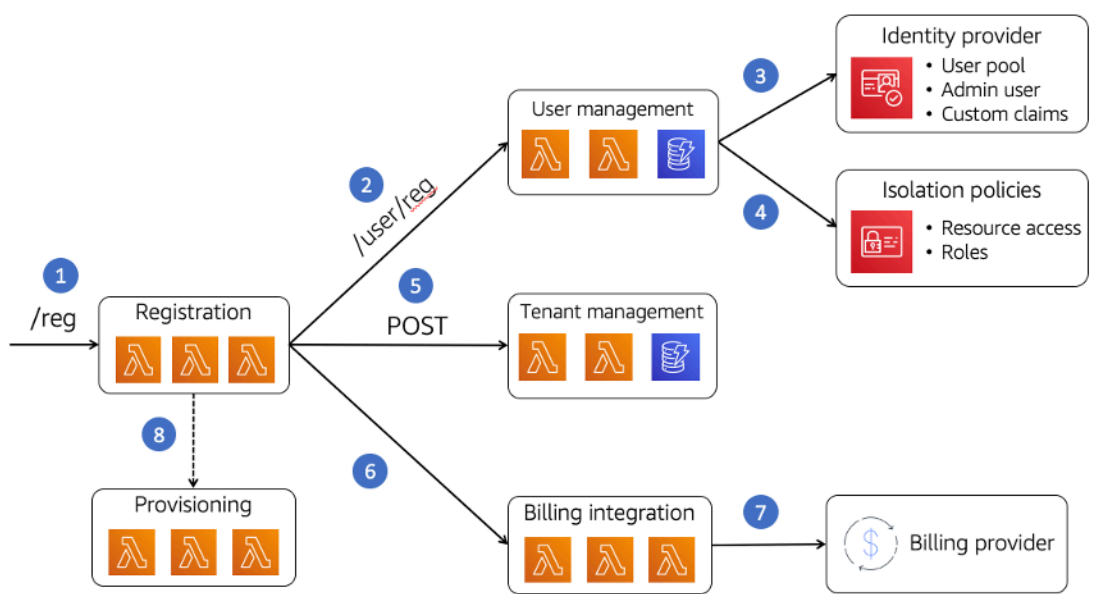


Figure 13: Frictionless tenant onboarding

In this model, we've also relied on a strategy that use a separate Amazon Cognito user pool for each tenant. This allows us to customize the identity experience for each tenant of our system (password policies, multi-factor authentication, etc.). By selecting this approach, we must map each user ID to a corresponding user pool. This mapping is managed by the user management service.

After the user is created, our process now creates a tenant. This separation of tenant as a distinct service is essential to SaaS environments. It provides a centralized way to manage the state and attributes of a tenant completely separate from the users that are associated with that tenant. A tenant's tier or the status, for example, would be controlled by the tenant management service.

As part of onboarding, a SaaS system must often establish a footprint in a billing system. In this example, you'll see that we inform the billing integration service that we're creating a new tenant. This service then assumes responsibility for creating a new account with the billing provider. This includes configuring the plan or tier for the tenant (free, bronze, platinum, etc.).

The last step in the process shown in the diagram relates to the provisioning of tenant infrastructure. Some SaaS architectures will include dedicated tenant resources. In these scenarios, our onboarding process must provision these new resources before our tenant can be activated.

While the flow represented here could vary depending on the nature of your environment, the concepts represented here are common to the onboarding process. Automating the creation, configuration, and provisioning of tenant resources is fundamental to creating a rich, scalable multi-tenant experience.

SaaS OPS 4: How do you support the need for tenant-specific customizations?

One of the significant challenges SaaS architects face is the need to ensure that all tenants are running the same version of their product. This is especially true for companies that have migrated to SaaS and have grown accustomed to supporting unique customer requirements through one-off versions of their product.

While this might seem tempting, any movement away from a unified customer management, operations, support, and deployment experience directly undermines the overall agility of a SaaS organization. As each new custom environment is introduced, a SaaS organization slowly makes its way to a traditional software model. Ultimately, this ends up eroding the cost, operational efficiency, and general innovation goals that are core the SaaS business model.

The challenge, then, is to find a strategy that allows you to meet these occasional one-off needs without creating a forked version of your product. The compromise here is achieved through the introduction of customization options that are added to the overall offering. So, instead of spinning off a separate version, you'd invest the extra time and effort into figuring out how these new features can be added in a way that makes them available to *all* customers. Then, through tenant configuration, you can determine which tenants will have these new capabilities enabled.

A common approach to this problem is to use feature flags. Feature flags are commonly used by application developers as a way to have multiple paths of execution in a common code base with flags that enable or disable each of the different capabilities at runtime. This technique, which is often used as a general development strategy, provides an effective way to introduce customization into your SaaS environment. Each feature flag would correlate to a tenant configuration option. This configuration would be evaluated at runtime and influence the features that are enabled for each tenant.

The diagram in Figure 14 provides a conceptual view of how these flags would be applied. A series of flags will be turned on and off for individual tenants, determining which capabilities are enabled for a tenant. These configuration options would be changed as a tenant signs up for new features of a SaaS offering. In some cases, these flags can be associated with tiers (instead of individual tenants).

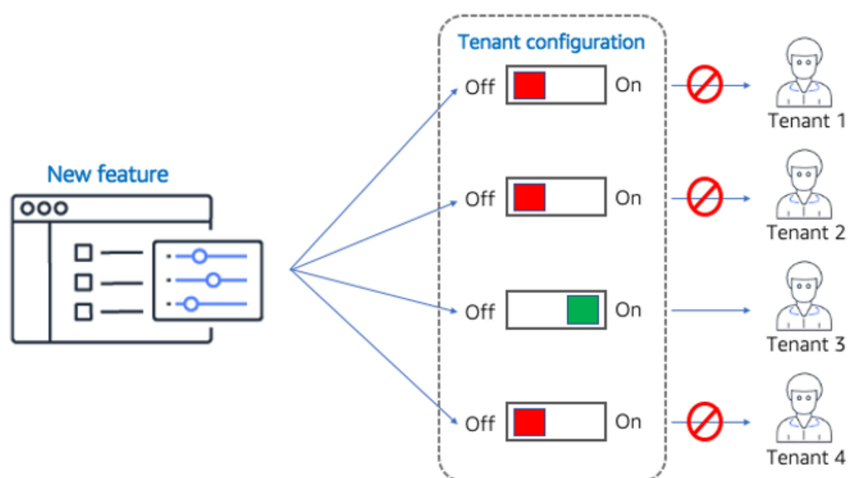


Figure 14: Managing tenant needs with feature flags

Feature flags are just one way to achieve this. The key takeaway here is that—even if a single tenant needs a unique feature—that feature should be introduced as a customization to the core platform. How you apply that customization can vary based on the stack and design of your system. The intent is to ensure that we can deploy and manage a single version of our product.

While this can be a powerful construct, it should be applied with caution. If, by introducing feature flags, you create a complex maze of options that end up presenting every tenant a unique experience, your system will quickly become unmanageable. Try to be selective about how and when you introduce these flags. As a rule of thumb, the business should not see this as a sales tool that enables the organization to offer one-off customizations to new customers.

Evolve

SaaS OPS 2: How are you capturing and surfacing metric data that can be used to analyze the usage and consumption trends of individual tenants?

To continually evolve your SaaS operations experience, you'll need to have access to a rich collection of operational data that can be used to analyze your multi-tenant SaaS environment. This often means instrumenting and publishing a much richer collection of tenant metrics from your application that can accurately capture the activity and consumption patterns of the tenants that are exercising your environment.

SaaS metrics go beyond the fundamental of infrastructure consumption (CPU, memory, etc.), identifying specific operational usage patterns that are fundamental to understanding the operation of multi-tenant loads in your environment. Analysis of this data will allow SaaS organizations to assess trends that are happening across their system and identify opportunities to introduce policies or changes to the underlying architecture that will continually improve the reliability, scalability, cost efficiency, and overall agility of a SaaS offering.

There are two distinct elements of capturing and surfacing metric data. First, your application must publish the metric data that can provide useful operational insights into your SaaS environment. You'll need to identify key points in your application where you'll want to capture and publish these metrics.

The other piece of the puzzle here is the ingestion, aggregation, and surfacing of this data. There is a wide spectrum of tools that you can choose here from AWS or one of the AWS Partner Network

(APN) Partners. Ultimately, the ingestion component of this becomes more of a data warehouse and business intelligence question. The SaaS architecture scenarios listed previously in this document includes a Tenant Insights scenario. This scenario outlines an architecture for ingesting metric data. That architecture represents one of the patterns you can apply to address this need.

While our focus here is on the operational view of these tenant metrics, you can imagine that these metrics have usage in multiple contexts across a SaaS business. The operational requirement is for your architecture to ensure that the organization is actively collecting and analyzing tenant activity and consumption trends to identify opportunities to evolve your SaaS system. This fits with the broader theme of building the foundational tooling that can be used to assess the continually shifting mix of tenants and tenant workloads.

Resources

Refer to the following resources to learn more about our best practices related to operational excellence.

Documentation & Blogs

- GPSTEC309-SaaS Monitoring Creating a Unified View of Multi-tenant Health featuring New Relic Slides
- Feature Toggles (aka Feature Flags)

Videos

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

Security Pillar

The **security** pillar includes the ability to help protect information, systems, and assets while delivering business value through risk assessments and mitigation strategies.

Design Principles

In the cloud, there are a number of principles that can help you strengthen your system security.

Definition

There are five best practice areas for security in the cloud:

- Identity and access management
- Detective controls
- Infrastructure protection
- Data protection
- Incident response

Multi-tenancy adds a layer of additional considerations to your SaaS architecture. With SaaS, you have users that are now accessing a shared environment in the context of a given tenant. This context must be captured and conveyed across all the layers of your application’s architecture and plays a fundamental role in securing the overall footprint of your environment.

From a security perspective, you need to look at how tenancy is introduced into your environment and how it is used to secure tenant resources. Overall, you need to ensure that each tenant has a carefully constrained experience that prevents them from accessing any other tenant's resources.

Best Practices

Topics

- Identity and Access Management (p. 27)
- Detective Controls (p. 28)
- Infrastructure Protection (p. 29)
- Data Protection (p. 36)
- Incident Response (p. 36)

Identity and Access Management

SaaS SEC 1: How are you associating tenant context with users and applying that context within your SaaS architecture?

The move to a multi-tenant architecture often begins with identity. Each user that accesses your application must be connected with a tenant. This binding of a user identity to a tenant is informally referred to as a *SaaS identity*. The key attribute of the SaaS identity is that it elevates tenant context to a first-class construct, connecting it directly to the overall authentication and authorization model of your SaaS application.

This approach allows tenant context to flow through all the layers of the architecture using the same architecture constructs that are used to convey and access user identity. For example, if you have 100 microservices in your application, you want each of those services to be able to acquire and apply tenant context without requiring a roundtrip to another service. Managing this context through another service adds latency and often creates bottlenecks in your architecture.

Injecting tenant context into your identity can be achieved through multiple patterns. The identity provider and technology you select for your application will directly shape the approach and strategies that you'll end up applying to introduce this context into your experience. While the tools might change, the fundamental need is to introduce tenancy into the overall authentication experience of your environment where tenancy is injected at the point where a user enters your application.

The diagram in Figure 15 provides an example of how this is commonly achieved using AWS services. This example includes the common components and technologies that would be used to inject tenant context into a SaaS environment. This is illustrated on the left side of the diagram, where a tenant completes a sign-up form, and triggers a call to your application's registration service.

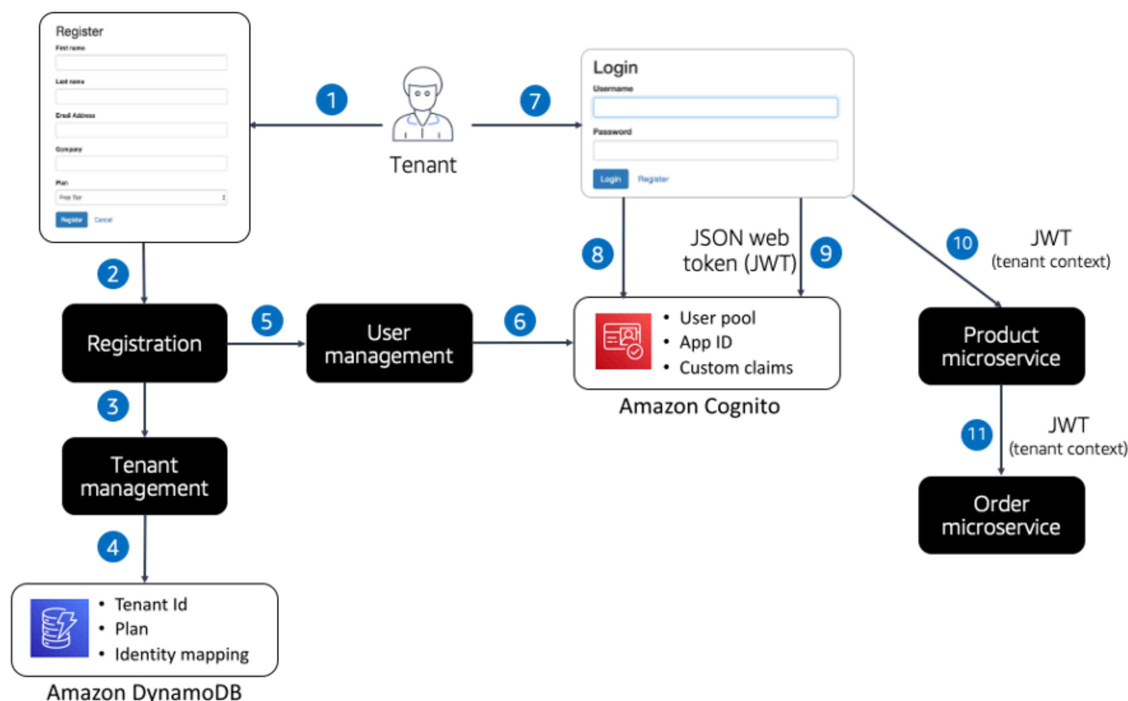


Figure 15: Injecting tenant content

This registration service creates a tenant and then creates a user in Amazon Cognito. As part of this process, you introduce custom claims into user's attributes that hold information about the user's relationship to a tenant. These custom claims become part of the identity signature of your user, connecting them directly with a tenant as a first-class construct.

After the onboarding is completed, you then can look at how these user and tenant attributes are applied when a user logs in (the right-hand side of the diagram). Here you'll see that the user authenticates against Amazon Cognito and, as part of that process, returns a JSON web token (JWT) that includes the custom claims created during the onboarding process.

Now you have a token that has all the information that you need to inject tenant context into the interactions with multi-tenant application services. In this example, we show the JWT being passed as a bearer token in the header of each request to the Product microservice. This service can now acquire and apply the context from this token without calling another service.

Finally, this Product microservice makes a call to an Order microservice, passing the JWT in the header of the request. This illustrates how the tenant context can flow across all of your microservice calls without adding any additional lookups or latency.

This example happens to rely on Amazon Cognito to connect the user and tenant identities. However, this same model could be implemented with other identity providers or alternate authentication schemes. The key here is that you're building an authentication experience that can yield a representation that connects tenant and users. This representation should then be available to all the layers of your solution.

Detective Controls

There are no security practices unique to SaaS applications.

Infrastructure Protection

SaaS SEC 2: How are you ensuring that tenant resources are protected from cross-tenant access?

Tenant isolation is one of the foundational topics that every 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 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 unrecoverable 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.

Topics

- The Isolation Mindset (p. 29)
- Core Isolation Concepts (p. 30)

The Isolation Mindset

At the conceptual level, many 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 help guide your 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 unintentionally 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 an 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 help you simplify the path to tenant isolation. Combining these tools with your broader security scheme can help make isolation an easier process.. However, there might 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 runtime 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 might impose constraints that will require a specific flavor of isolation. For example, some high compliance industries may require that every tenant have its own database. In these cases, the shared, policy-based approaches to isolation may not be adequate.

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 their multi-tenant environment. The sections below will explore the different types of isolation, and associate specific terminology with the varying isolation constructs.

Topics

- Silo Isolation (p. 30)
- Pool Isolation (p. 32)
- The Bridge Model (p. 34)
- Tier-Based Isolation (p. 34)
- Targeted Isolation (p. 35)

Silo Isolation

While SaaS providers are often focused on the value of sharing resources, there are still scenarios where a SaaS provider might 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 this is 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 16 provides a conceptual view of this view of isolation.

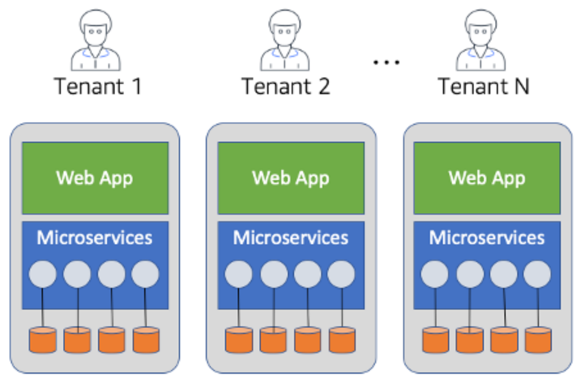


Figure 16: Silo isolation model

This diagram highlights the basic footprint of the siloed deployment model. The technologies that are used to 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 is to take whatever stack the tenant has and surround it with some construct to encapsulate all the moving parts of that stack. This becomes the 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 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 might 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. These are 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 model 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. The silo model addresses this concern by offering a dedicated environment with no potential for 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 you with a simpler way to capture and associate infrastructure costs with each tenant.
- **Reduced scope of impact** – The silo model generally reduces your exposure when there might 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 cannot 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 might prevent 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. For example, having 20 siloed accounts for each of your tenants might be manageable. However, if you have a thousand tenants, that number would likely begin to impact operational efficiency and agility.
- **Cost** – With every tenant running in its own environment, much of the cost efficiency that is traditionally associated with SaaS solutions is not realized. Even if these environments scale dynamically, you'll likely have periods of the day when you'll have idle resources that are going unconsumed. 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 capabilities. While there are measures you can take 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 will still need to automate 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** – The goal with SaaS is to have a single pane of glass that enables you to 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

You can see how the silo model of isolation maps very nicely for many SaaS companies. 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 17 provides an illustration of the challenge associated with implementing isolation in a pooled model.

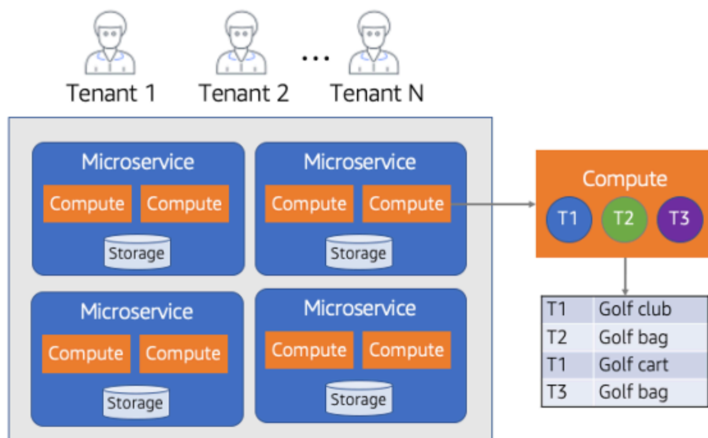


Figure 17: Pool isolation model

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. The right side of the diagram narrows in on the compute aspect of one of the services, highlighting the fact that tenants 1-N might all be running side-by-side within your shared compute at any given time. The storage in this example is also shared and is represented as a table indexed by individual tenant identifiers.

This model can work well for SaaS providers, however, it has the potential to complicate the overall isolation story. With shared resources, implementing isolation is not as clear and typical networking and IAM constructs cannot be relied upon to create boundaries between tenants.

The key here is that—even though this is a more challenging environment to isolate—you cannot use this as a rationale to relax the isolation requirements of your environment. The 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, 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 can enable 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. These are the key pros and cons that are typically associated with the pool isolation model.

Pros

- **Agility** – When you move your tenants into a shared infrastructure model, you can leverage the natural efficiencies and simplicity that help streamline 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 their tenants with one unified experience. Centralizing and standardizing the experience is foundational to enabling SaaS providers to more 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 you one view into all the tenants in a system. You can manage, update, and deploy all of your tenants through a single experience that touches all your tenants in the 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 free 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. For example, any activity from one tenant that puts a heavy load on the system 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 impacting 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. Each SaaS provider should look for ways to instrument their systems and surface the granular data needed to effectively associate consumption with individual tenants.
- **Reduced scope of impact** – Sharing all resources shared also introduces some operational risk. In the silo model, when one tenant has 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 in your system, which can have a significant impact on your 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 you may be unwilling to run in this model. This is especially true in environments with compliance or regulatory rules for a domain impose strict constraints on the accessibility and isolation of resources. Even in these cases, though, this might mean some portion of the system will need to be siloed.

The Bridge Model

While the silo and pool models 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 your 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 the bridge model of isolation. The diagram in Figure 18 provides an example of how the bridge model might be realized in a SaaS solution.

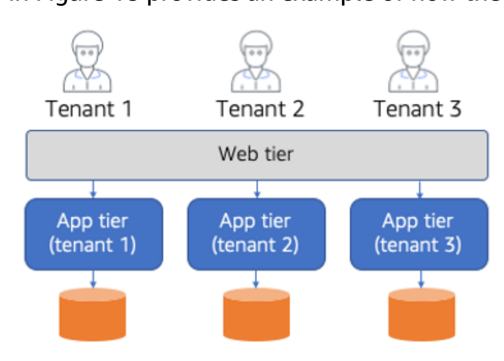


Figure 18: Bridge isolation model

This diagram highlights how the bridge model enables you to combine 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.

If the monolith was broken into microservices, each of the various microservices in your system could leverage combinations of the silo and pool models. More detail on this approach will follow in the description of specifics of applying silo and pool models with different AWS constructs. The key takeaway here is that your view of the silo and pool models will be much more granular for environments that are decomposed into a collection of services that have varying isolation requirements.

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 19 provides an example of how isolation might vary across tiers.

The below example uses a mix of silo and pool isolation models that have been offered up as tiers to the 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 that a completely dedicated (silo)

environment be offered. To support this, the SaaS provider has created a Premium tier model that enables tenants to run in this dedicated model likely at 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 are outlined previously.

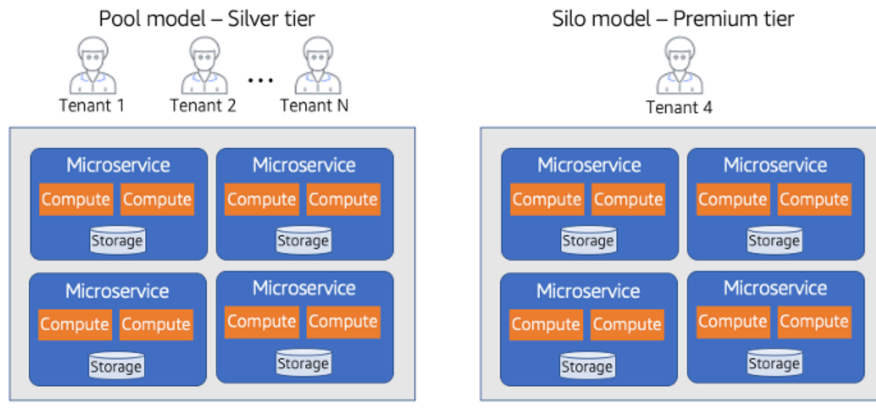


Figure 19: Tier-based isolation

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 that 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.

Targeted Isolation

It's important to note that the isolation choices in your system can be quite granular. Each microservice of your system and each resource those services touch has the option of being configured with a different model of isolation. Let's look at some sample microservices to better understand how you might vary the isolation model across a varying microservices. The diagram in Figure 20 provides a view of microservices that use both the silo and pool models of 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 that tenants all land here as separate items that are indexed in the same table. The assumption is that the data will be isolated with policies that can restrict access to tenant items in this table. The Order microservice is only for tenants 1 through 3 and also implemented in a pooled model. The only difference here is that it's supporting a subset of tenants. Essentially, any tenant that 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 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, 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 which 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 is focused exclusively on separating the data. The compute is still enabled to be shared.

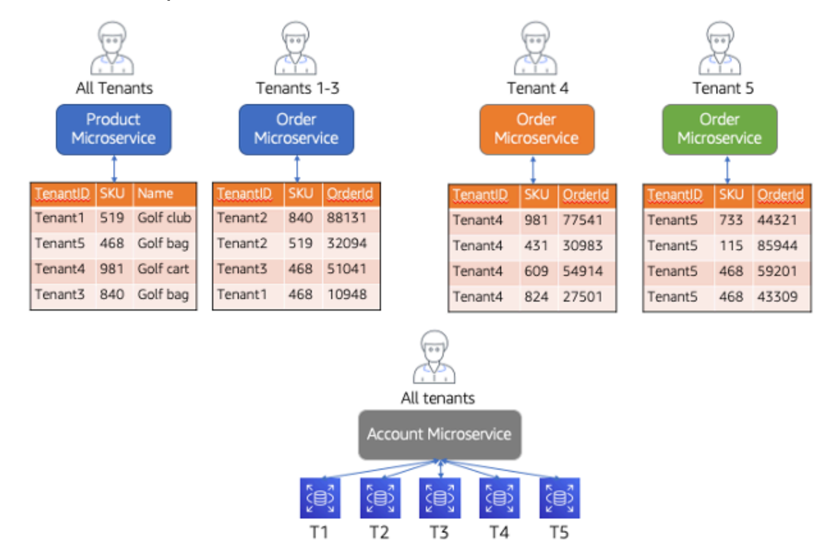


Figure 20: Targeted isolation

This model shows 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. The key takeaway here is that the silo model is not an all-or-nothing decision. You can think about applying the silo model to specific components of your application and only absorb this model's challenges where it's actually needed, such as when a potential customer demands its use. In this case, a more detailed discussion with the customer, you find out that there are only a few specific areas of storage and processing that are of concern. Doing so will enable you to get the efficiencies of the pool model for those parts of the system that do not require silo isolation and also give you the flexibility to offer a tiered structure to support a mix of both silo and pool models for individual services.

Data Protection

There are no security practices unique to SaaS applications.

Incident Response

There are no security practices unique to SaaS applications.

Resources

Refer to the following resources to learn more about our best practices for security.

Documentation & Blogs

- Isolating SaaS Tenants with Dynamically Generated IAM Policies
- Partitioning Pooled Multi-Tenant SaaS Data with Amazon DynamoDB

- Multi-tenant data isolation with PostgreSQL Row Level Security
- Identity Federation and SSO for SaaS on AWS
- Managing SaaS Identity Through Custom Attributes and Amazon Cognito
- Onboarding and Managing Agents in a SaaS Solution
- Amazon Cognito API Reference
- Building Serverless SaaS with Lambda layers
- Modeling SaaS Tenant Profiles on AWS

Whitepapers

- SaaS Tenant Isolation Strategies Isolating Resources in a Multi-Tenant Environment Whitepaper
- SaaS Storage Strategies whitepaper

Videos

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

Reliability Pillar

The **reliability** pillar includes the ability of a system to recover from infrastructure or service disruptions, dynamically acquire computing resources to meet demand, and mitigate disruptions such as misconfigurations or transient network issues.

Design Principles

In the cloud, there are a number of principles that can help you increase reliability.

Definition

There are three best practice areas for reliability in the cloud:

- Foundations
- Change management
- Failure management

To achieve reliability, a system must have a well-planned foundation and monitoring in place, with mechanisms for handling changes in demand or requirements. The system should be designed to detect failure and automatically heal itself.

Best Practices

Topics

- Foundations (p. 38)
- Change Management (p. 40)
- Failure Management (p. 41)

Foundations

SaaS REL 1: How do you limit an individual tenant's ability to impose load that might impact availability for other tenants of your system?

The workloads of tenants in a multi-tenant environment can be continually shifting. Tenants may impose different types of load on the system. New tenants with new workload profiles might also be continually added to the system. These factors can make it very challenging for SaaS companies to create an architecture that is resilient enough to react and respond to these evolving needs.

These variations in load can have an adverse impact on a tenant's experience. Imagine a scenario where a single tenant ends up placing an extreme load on some aspect of your system. This can be especially pronounced if they can interact with your system via an API. In this case, the load of this tenant could end up consuming a disproportionate level of the system's resources which, in turn, could end up impacting the reliability of the overall system or another tenant's experience.

This ability for one tenant to impact another can get more complicated for systems that have a tiered offering. For example, a system might have basic, advanced, and premium tiers. If each of these tiers are allowed to impose any level of load on the system, we might find that the basic tier is impacting the reliability of a premium tier tenant.

In a multi-tenant environment, you need to be especially proactive in your efforts to identify workloads and patterns of consumption that could impact the reliability of your system. Reliability issues in a multi-tenant environment can easily cascade through your system and, potentially, impact the experience of *all* of your customers.

To address these workload issues, your system must introduce mechanisms that can detect and resolve workload issues before they can impact the reliability of your application.

The architecture of your application and your application stack will influence the strategies you apply to prevent tenant's from impacting the reliability of your system and the experience of other tenants. One common approach is to use throttling to prevent tenants from consuming excess resources. The diagram in Figure 21 provides an example of doing this using Amazon API Gateway.

In this example, you'll see that we have leveraged usage plans with API Gateway to define separate usage experiences for each tenant of the system. This approach uses separate API keys for the basic and advanced tiers of the system and these keys are connected to usage plans that have different SLAs. This approach enables two different levels of control. First, it can ensure that *all* tenants are prevented from saturating our system with requests through the configuration of the usage plans. The other benefit is that we can use these usage plan configurations to prevent lower tier tenants from impacting the experience of higher tier tenants.

While this model relies on API Gateway to implement a throttling policy, this core concept can be applied to other infrastructure constructs. The fundamental goal of this approach is to have some ability to monitor and manage access at the entry point to your application, detecting and throttling any tenant that may impose load that could impact the overall reliability of your environment.

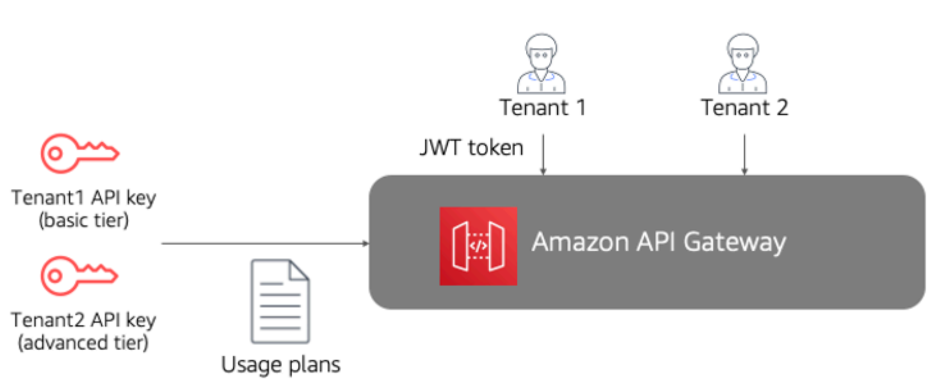


Figure 21: Throttling tenants by tier

SaaS REL 2: How do you proactively detect and maintain tenant health?

Reliability in a SaaS environment is often very much influenced by a provider's ability to proactively identify and remediate issues before they impact a tenant. Building a proactive view of health in a multi-tenant environment requires you to surface additional reliability data that provides more detailed, tenant-aware insights into the health trends of your tenant workloads.

These tenant-aware insights are used to identify tenant specific trends, activities, or insights that can effectively capture condition that could impact the reliability for that tenant or the entire system. Having this data and surfacing it proactively, allows you to build alarms, policies, and automation that can attempt to heal the system without incurring an outage.

To make this possible, you'll need to introduce code into your application that will publish health insights with tenant context. This starts by identifying the workflows and events in your architecture that represent useful health data. This could be consumption data, scaling insights, latency metrics, and so on. Each of these insights would be published via log file or to a data warehouse that would aggregate the data.

Once you have this health data aggregated in a repository, you can then introduce tooling that can surface alarms or trigger policies based on an analysis of the aggregated health data.

SaaS REL 3: How are you testing the multi-tenant capabilities of your SaaS application?

The testing model for a multi-tenant environment goes beyond simply ensuring that the functionality of your application works as expected. SaaS providers must also create tests that will validate that your system can effectively address common reliability challenges that are associated with a multi-tenant solution.

There are a number of different dimensions that you'll want to include as part of your multi-tenant testing strategy. In many cases, these tests are targeted at validating the constructs that you have in place to address the scale, operations, and reliability footprint of your SaaS product.

It's important to note that SaaS testing is often about simulating the extremes that your application might encounter. You should be focused on building a suite of tests that can effectively model and evaluate how your system will respond to the expected and the unexpected. In addition to ensuring that customers have a positive experience, your tests must also consider how cost effectively it is achieving scale. If you are over-allocating resources in response to activity, you're likely impacting the bottom line for the business.

Some specific areas where you might augment your load and performance testing strategy in a SaaS environment are:

- **Cross-tenant impact tests** – Create tests that simulate scenarios where a subset of your tenants place a disproportionate load on your system. This will allow you to determine how the system responds when load is not distributed evenly among tenants, and assess how this might affect overall tenant experience. If your system is decomposed into separately scalable services, you'll want to create tests that validate the scaling policies for each service to ensure that they're scaling on the right criteria.
- **Tenant consumption tests** – Create a range of load profiles (for example, flat, spikey, and random) that track both resource and tenant activity metrics, and determine the delta between consumption and tenant activity. You can ultimately use this delta as part of a monitoring policy that could identify suboptimal resource consumption. You can also use this data with other testing data to determine if you've sized your instances correctly, have IOPS configured correctly, and are optimizing your AWS footprint.
- **Tenant workflow tests** – Use these tests to assess how the different workflows of your SaaS application respond to load in a multi-tenant context. The idea is to pick well-known workflows of your solution, and concentrate load on those workflows with multiple tenants to determine if these workflows create bottlenecks or over-allocation of resources in a multi-tenant setting.
- **Tenant onboarding tests** – As tenants sign up for your system, you want to be sure that they have a positive experience and that your onboarding flow is resilient, scalable, and efficient. This is especially true if your SaaS solution provisions infrastructure during the onboarding process. You'll want to determine that a spike in activity doesn't overwhelm the onboarding process. This is also an area where you might have dependencies on third-party integrations (billing, for example). You'll likely want to validate that these integrations can support their SLAs. In some cases, you might implement fallback strategies to handle potential outage for these integrations. In these cases, you'll want to introduce tests that verify that these fault tolerance mechanisms are performing as expected.
- **API throttling tests** – The idea of API throttling is not unique to SaaS solutions. In general, any API you publish should include the notion of throttling. With SaaS, you also need to consider how tenants at different tiers can impose load via your API. A tenant in a free tier, for example, might not be allowed to impose the same load as a tenant in the gold tier. This enables you to verify that the throttling policies associated with each tier are being successfully applied and enforced.
- **Data distribution tests** – In most cases, SaaS tenant data will not be uniformly distributed. These variations in a tenant's data profile can create an imbalance in your overall data footprint, and might affect both the performance and cost of your solution. To offset this dynamic, SaaS teams will typically introduce sharding policies that account for and manage these variations. Sharding policies are essential to the performance and cost profile of your solution, and, as such, they represent a prime candidate for testing. Data distribution tests allow you to verify that the sharding policies you've adopted will successfully distribute the different patterns of tenant data that your system may encounter. Having these tests in place early might help you avoid the high cost of migrating to a new partitioning model after you've already stored significant amounts of customer data.
- **Tenant isolation testing** – SaaS customers expect that every measure will be taken to ensure that their environments are secured and inaccessible by other tenants. To support this requirement, SaaS providers build in a number of policies and mechanisms to secure each tenant's data and infrastructure. Introducing tests that continually validate the enforcement of these policies is essential to any SaaS provider.

As you can see, this test list is focused on ensuring that your SaaS solution will be able to handle load in a multi-tenant context. Load for SaaS is often unpredictable, and you will find that these tests often represent your best opportunity to uncover key load and performance issues before they impact one or all of your tenants. In some cases, these tests might also surface new points of inflection that may merit inclusion in the operational view of your system.

Change Management

There are no reliability practices unique to SaaS applications.

Failure Management

There are no reliability practices unique to SaaS applications.

Resources

Refer to the following resources to learn more about our best practices related to reliability.

Documentation & Blogs

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

Videos

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

Performance Efficiency Pillar

The **performance efficiency** pillar focuses on the efficient use of computing resources to meet requirements and maintaining that efficiency as demand changes and technologies evolve.

Definition

There are four best practice areas for performance efficiency in the cloud:

- Selection (compute, storage, database, network)
- Review
- Monitoring
- Tradeoffs

Take a data-driven approach to selecting a high-performance architecture. Gather data on all aspects of the architecture, from the high-level design to the selection and configuration of resource types.

By reviewing your choices on a cyclical basis, you will ensure that you are taking advantage of the continually evolving AWS Cloud.

Monitoring will ensure that you are aware of any deviance from expected performance and can take action on it. Finally, your architecture can make tradeoffs to improve performance, such as using compression or caching, or relaxing consistency requirements.

Best Practices

Topics

- Selection (p. 42)
- Review (p. 45)
- Monitoring (p. 45)
- Tradeoffs (p. 46)

Selection

SaaS PERF 1: How do you prevent one tenant from adversely impacting the experience of another tenant?

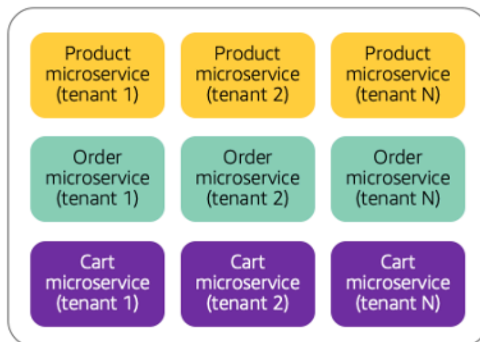
In a multi-tenant environment, tenants might have profiles and use cases that impose significantly different loads on your system. New tenants with new workload profiles might also be continually added to the system. These factors can make it very challenging for SaaS companies to build an architecture that can meet the rapidly evolving performance requirements each of these tenants.

Handling and managing these variations in tenant load is key to the performance profile of a SaaS environment. A SaaS architecture must be able to successfully detect these tenant consumption trends and apply strategies that can scale effectively to meet tenant demands or restrict the activity of individual tenants.

There are a variety of strategies that can be used to manage these scenarios where a tenant might be placing a disproportionate load on your system. This can be achieved through isolation of high-demand resources, introduction of scaling strategies, or the application of throttling policies.

In the simplest and most extreme case, you might consider creating tenant-specific deployments for parts of your application. The diagram in Figure 22 illustrates one way that you might decompose your system to address performance challenges.

Services that must be isolated for each tenant



Services that are shared by all tenants

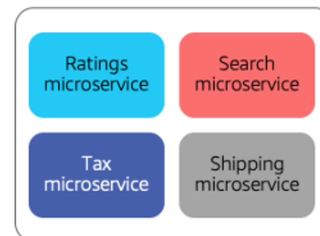


Figure 22: Addressing performance with siloed services

In this example, you'll notice that we have two distinct deployment footprints (some in a silo model and some in a pool model). On the left side of the diagram, you'll see that separate instances of Product, Order, and Cart microservices have been deployed for each tenant. Meanwhile, on the right side of the diagram, you'll see a collection of microservices that are shared by all tenants.

The basic idea behind the approach is to carve out specific services that are seen as critical to the performance profile of our application. By separating them out, your system can ensure that the load of any one tenant won't impact the performance of other tenants (for this set of services). This strategy can increase costs and decrease the operational agility of your environment, but still represents a valid way to target performance areas. This same approach may also be applied to address compliance and isolation requirements.

You might, for example, deploy an order management microservice for each tenant to limit any ability for one tenant to adversely impact another tenant's order processing experience. This adds operational complexity and reduces cost efficiency, but can be used as a brute force way to selectively target cross-tenant performance issues for key areas of your application.

Ideally, you should try to address these performance requirements through constructs that can address tenant load issues without absorbing the overhead and cost of separately deployed services. Here you would focus on creating a scaling profile that allows your shared infrastructure to effectively respond to shifts in tenant load and activity.

A container-based architecture, such as Amazon EKS or Amazon ECS, could be configured to scale your services based on tenant demand without requiring any significant over-provisioning of resources. The ability for containers to scale rapidly enhances your system's ability to respond effectively to spikey tenant loads. Combining the scaling speed of containers with the cost profile of AWS Fargate often represents a solid blend of elasticity, operational agility, and cost efficiency that can help organizations address the spikey loads of tenants without over-provisioning environments.

A serverless SaaS architecture built with AWS Lambda could also be a good fit for addressing the spikey tenant loads. The managed nature of AWS Lambda allows your application's services to scale rapidly to address spikes in tenant load. There might be concurrency and cold start factors you'd need to factor into this approach. However, it can represent an effective strategy for limiting cross-tenant performance impacts.

While a responsive scaling strategy can help with this problem, you might want to put other measures in place to simply prevent tenants from imposing loads that would have cross-tenant impacts. In these scenarios, you might choose to detect and constrain the activity of the tenants by setting limits (potentially by tier) that would apply throttling to control the level of load the place on your system. This would be achieved by introducing throttling policies that would examine the load of tenants, identify and activity that exceeds limits, and throttle their experience.

SaaS PERF 2: How are you ensuring that the consumption of infrastructure resources aligns with the activity and workloads of tenants?

The business model of SaaS companies often relies heavily on a strategy that allows them to align the costs of their infrastructure with the actual activity of their tenants. Since the load and makeup of the tenants in a SaaS system is continually changing, you need an architecture strategy that can effectively scale the consumption of resources in a pattern a pattern that very much mirrors these real-time, unpredictable patterns of consumption that are part of the SaaS experience.

The graph in Figure 23 provides a hypothetical example of an environment that has aligned infrastructure consumption and tenant activity. Here the blue solid line represents the actual activity trends of tenants spanning a window of time. The red dashed line represents the actual infrastructure that's being provisioned to address the load of tenants.

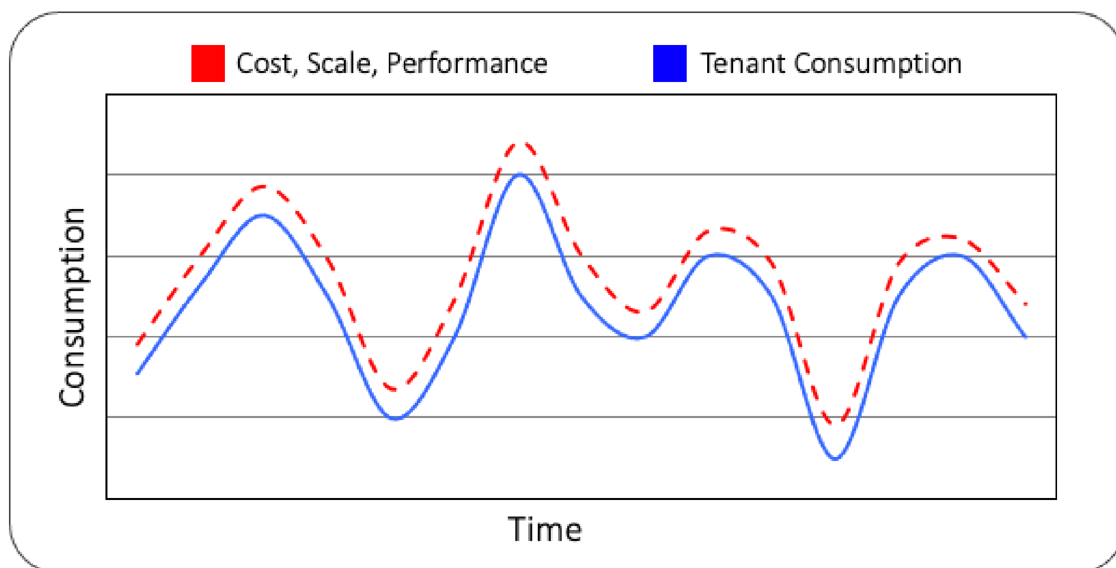


Figure 23: Aligning tenant activity and consumption

Our strategy here, in an ideal environment, would be to keep the gap between the red a blue line as small as possible. You'll always have some margin for error here where you have some cushion to ensure that you're not impacting the availability or performance of the system. At the same time, you want to be able to deliver just enough infrastructure to support the *current* performance needs of your tenants.

The key challenge here is that the load shown in this diagram is often unpredictable. While there may be some general trends, your architecture and scaling strategies can't assume that the load today will be the same tomorrow or even in the next hour.

The simplest approach to aligning consumption with activity is to use AWS services that provide a serverless experience. The classic example of this would be AWS Lambda. With AWS Lambda, you can build a model where servers and scaling policies are no longer your responsibility. With serverless, your SaaS application will only incur those charges that are directly correlated with tenant consumption. If there's no load on your SaaS system, there will be no AWS Lambda costs.

AWS Fargate also enables a container-based version of this this serverless mindset. By using Fargate with Amazon EKS or Amazon ECS, you only pay for the container compute costs that are actually consumed by your application.

This ability to use a serverless model extends beyond compute constructs. For example, the storage pieces of your solution can rely on serverless technology as well. Amazon Aurora Serverless allows you to store relational data without needing to size the instances that are running your database. Instead, Amazon Aurora Serverless will size your environment based on actual load and only charge for what your application consumes.

Any model that lets you move away from a need to create scaling policies is going to streamline your operational and cost experience. Instead of continually chasing the elusive perfect automatic scaling configuration, you can focus more of your time and energy on the features and functions of your application. This also enables the business to grow and accept new tenants without being concerned about unexpected jumps in its AWS bill.

For scenarios where serverless may not be an option, you'll need to fall back to traditional scaling strategies. In these scenarios, you'll need to capture and publish tenant consumption metrics and define scaling policies based on these metrics.

Review

There are no performance practices unique to SaaS applications.

Monitoring

SaaS PERF 3: How do you enable varying levels of performance for different tenant tiers and plans?

SaaS solutions are often offered in a tiered model where tenants will have access to different experiences. Performance can often be an area that is used to differentiate tiers of a SaaS environment, using performance as a way to create a value boundary that would compel tenants to move to higher level tiers.

In this model, your architecture will introduce constructs that will monitor and control the experience of each tier. This isn't just about maximizing performance—it's also about limiting the consumption of lower tiered tenants. Even if your system could accommodate the load of these tenants, you might choose to limit this load purely based on cost or business considerations. This is often part of ensuring that the cost footprint of a tenant correlates with the revenue that tenant contributes to the business.

The least complex way to approach this problem is to introduce throttling policies that are associated with individual tenant tiers. As a tenant reaches a limit, you would apply the throttling and limit their consumption.

There are also scenarios where you can use specific AWS configurations to configure the consumption profile of a tenant tiers. For example, in AWS Lambda, you can use reserve concurrency to limit the consumption of a given tenant tier. The diagram in Figure 24 provides an example of how this could be realized.

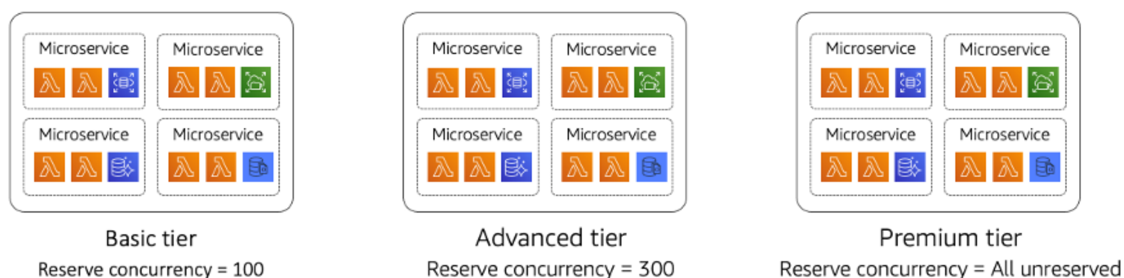


Figure 24: Controlling tenant performance with reserve concurrency

In this example, we've created three separate tenant tiers and deployed three separate collections of our SaaS application's microservices for each of these tiers. These collections are also configured with separate reserve concurrency settings which are used to determine how many concurrent function invocations can be running for that group of functions. The Basic tier has a reserve concurrency of 100 and the Advanced tier has 300. The idea here is that the consumption of my lower end tiers will be capped, leaving all the remain concurrency for the premium tier.

This approach aligns nicely with our goal of offering the best experience our preferred tiers while also limiting a lower tier's ability to consume excess resources and impact the performance of our higher tier tenants.

Containers also have unique strategies for addressing tiering for performance. Within Amazon EKS, for example, you can configure separate `ResourceQuotas` and `LimitRanges` to control the amount of resources that are available in a namespace.

While these constraints are helpful in configuring a tenant's performance experience, some SaaS applications will actually address performance through application design and decomposition strategies. This might be achieved by deploying siloed microservices for higher tier tenants, eliminating any noisy neighbor considerations for these specific services. In fact, you might find that the decomposition of your system into microservices might be directly influenced by the tiering and performance profile you are targeting.

In some cases, your SaaS application might also introduce architectural constructs that optimize the experience of higher tier tenants. Imagine, for example, offering caching of key data to premium tier tenants. By limiting the cache to just these users, you avoid the expense of having a cache that must support all users. The effort to introduce these optimizations should be offset with enough value to the customer and the business to warrant the investment.

Tradeoffs

There are no performance practices unique to SaaS applications.

Resources

Refer to the following resources to learn more about our best practices related to performance efficiency.

Documentation & Blogs

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

Whitepapers

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

Videos

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

Cost Optimization Pillar

The **cost optimization** pillar includes the continual process of refinement and improvement of a system over its entire lifecycle. From the initial design of your very first proof of concept to the ongoing operation of production workloads, adopting the practices in this paper will enable you to build and operate cost-aware systems that achieve business outcomes and minimize costs, thus allowing your business to maximize its return on investment.

Definition

There are four best practice areas for cost optimization in the cloud:

- Cost-effective resources
- Matching supply and demand
- Expenditure awareness
- Optimizing over time

As with the other pillars, there are tradeoffs to consider. For example, do you want to optimize for speed to market or for cost? In some cases, it's best to optimize for speed—going to market quickly, shipping new features, or simply meeting a deadline—rather than investing in upfront cost optimization.

Design decisions are sometimes guided by haste as opposed to empirical data, as the temptation always exists to overcompensate "just in case" rather than spend time benchmarking for the most cost-optimal deployment.

This often leads to drastically over-provisioned and under-optimized deployments. The following sections provide techniques and strategic guidance for the initial and ongoing cost optimization of your deployment

Best Practices

Topics

- Cost-Effective Resources (p. 47)
- Matching Supply and Demand (p. 47)
- Expenditure Awareness (p. 47)
- Optimizing Over Time (p. 51)

Cost-Effective Resources

There are no cost practices unique to SaaS applications.

Matching Supply and Demand

There are no cost practices unique to SaaS applications.

Expenditure Awareness

SaaS COST 1: How do you measure the resource consumption of individual tenants?

Measuring and attributing costs in a multi-tenant environment begins with having a solid strategy for attributing consumption to tenants. This will require teams to design and develop a consumption mapping model that represents a clear view of how tenants are consuming the resources of your system. The ultimate goal is to arrive at a collection of insights that will allow you to allocate a percentage of consumption to each tenant of your system.

Assembling this view of consumption can be particularly challenging in a multi-tenant environment where tenants might be sharing some or all of the system's resources. This more fine-grained consumption model removes many of the options and tooling strategies that are often used to attribute consumption in an AWS environment (tagging, for example).

While there's no single model for defining how tenant consumption is captured in a SaaS architecture, there are some common strategies that should be considered when selecting a strategy for your application. First, you'll want to look at the overall cost profile of your SaaS environment and determine how your application is influencing the costs in your AWS bill. For some environments, your costs might be heavily concentrated in a few areas of your application. In these scenarios, you might get a better ROI on gathering consumption data for just those areas that contribute most to your bill. For example, if Amazon S3 represents 1% of your bill, there might be little value in calculating your tenant's Amazon S3 consumption.

The other factor you'll want to consider here is granularity. There are less invasive approaches that can approximate tenant consumption that may be adequate for your environment. This really comes down to striking a balance between the level of consumption detail you're after and the complexity of instrumenting and capturing the data that's need to attribute consumption.

Let's start by looking at the simplest model for approximating tenant consumption. The diagram in Figure 25 provides a conceptual view of one way you could capture tenant activity in a minimally invasive model. The basic approach here is to inspect each call that is made to the API using as AWS Lambda authorizer. The authorizer would extract the tenant context from the incoming JWT and publish an event that records this activity for the tenant. An alternate approach to this would be to use AWS X-Ray to capture this data (instead of the Amazon API Gateway).

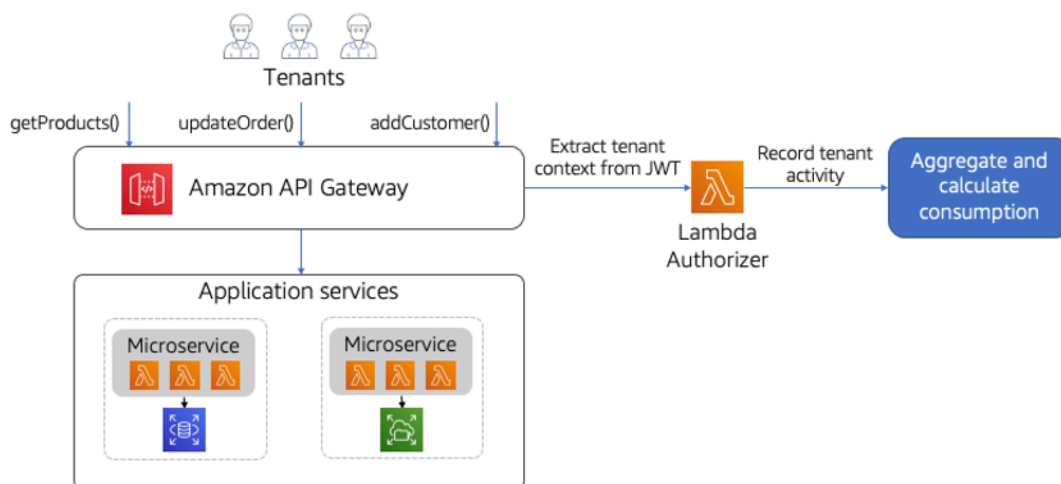


Figure 25: Minimally invasive capture of tenant consumption

There's also a placeholder on the diagram for aggregating and calculating tenant consumption. The strategy and tools you choose to fill in this gap will depend on the nature of the data, its lifecycle, and how it fits into your broader SaaS metrics and analytics story. You may choose to include this data as part of the general metrics footprint of your SaaS environment, pulling out the insights that are essential to allocating consumption to tenants.

This particular approach relies on tracking frequency of calls for each tenant as a way to infer the level of consumption for each tenant. While the number of calls to a service might not precisely correlate to consumption, for some environments this may represent a reasonable compromise.

A more granular view of consumption can be created by introducing more specialized instrumentation into the details of your application. The diagram in Figure 26 provides a view of how you might introduce metrics instrumentation into the microservices of your SaaS application.

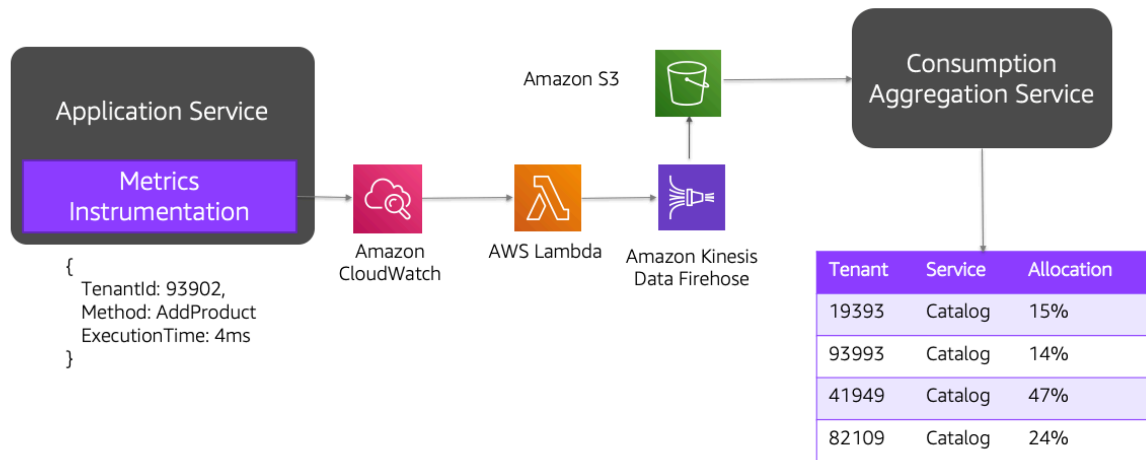


Figure 26: Instrumenting microservices with tenant consumption events

In this example, we've introduced metrics instrumentation into each microservice of our application. These microservices will capture more detailed data about how tenants are consuming this service and its related resources. This detailed data is published as an event and aggregated. Here we've shown Amazon CloudWatch, AWS Lambda, Amazon Kinesis Data Firehose, and Amazon S3 being used to publish and ingest the data. This data would then be analyzed and, based on your own modeling, arrive at a distribution of consumption across tenants.

Attributing tenant consumption gets more challenging as you look beyond the microservices of your application. You might have to develop specific, targeted strategies on a service-by-service basis. Storage services, for example, might require a separate service that can profile storage consumption. This might require looking at IOPS, data footprint, and other factors to analyze consumption for tenants.

SaaS COST 2: How are you correlating tenant consumption with the costs of your infrastructure?

The dynamic nature of SaaS environments can make it challenging to understand how the cost profile of your system's infrastructure might be changing. The shifting needs and mix of tenants in your system will likely lead to significant fluctuations in the cost of operating your SaaS environment. At the same time, a SaaS business needs to have a clear picture of how tenants are influencing costs to make strategic decisions about how to build, sell, and operate their SaaS application.

To understand the business value of having better insights into how tenants are influencing costs for the business, let's look at one example of how cost data could be applied in a SaaS environment. The graph in Figure 27 provides an example of a scenario where the costs of a SaaS environment were correlated with revenue from those tenants and the size of the ecommerce catalog being managed by these tenants.

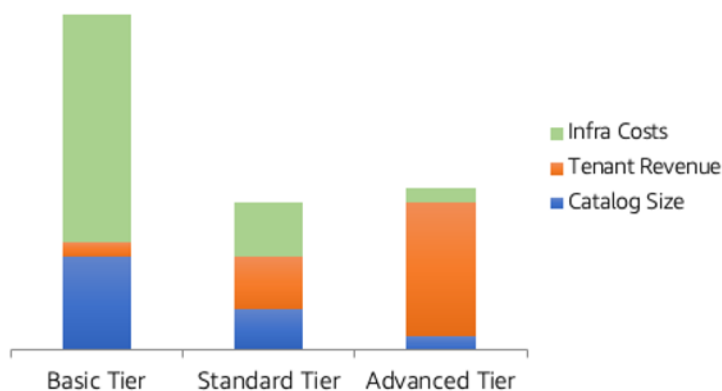


Figure 27: Costs per tenant by tier

This graph illustrates the distribution of costs across the tiers of a SaaS offering. Here you'll see a large difference between the infrastructure costs of Basic and Advanced tier tenants. The key observation is that the Basic tier, which generates the smallest revenue, is responsible for the largest portion of the system's infrastructure costs. Meanwhile, the Advanced tier, which generates the most revenue, has a much smaller cost footprint. This imbalance likely means there's something wrong with our model.

This is just one example of how having access to costs broken down by tenants and tiers is essential to SaaS providers. Access to this cost per tenant data allows a SaaS organization to assess a wide range of architectural considerations that can be influencing the cost profile of your environment. It can also help guide pricing and tiering strategies.

There are two fundamental aspects associated with assembling this cost per tenant data. First, you'll need some way to attribute and calculate tenant consumption to arrive at a percentage of consumption for each tenant (see above for details on how this consumption data is collected). After you have the consumption data, you'll need to correlate this data with cost information from your AWS bill to arrive at a cost per tenant calculation.

There are a number of options available for accessing the collection billing data. AWS provides APIs that can be used to ingest and aggregate this billing data or you can explore a range of APN Partner solutions to ingest this data and access the costs through these solutions. The shortest path here is often to engage with an APN Partner to deal with the nuances of ingesting and summarizing AWS cost.

The high-level view of this experience is captured in Figure 28. You'll see that we have two distinct sets of data that we need to collect. One process will aggregate and ingest the data from your AWS bill summarizing the costs in a manner that aligns with the granularity of costs that are relevant to your cost per tenant model. Next, you'll see the tenant consumption aggregation which analyzed tenant activity and assigns a percentage of consumption to each tenant. Finally, these consumption percentages are applied to the infrastructure costs to arrive at the cost per tenant.

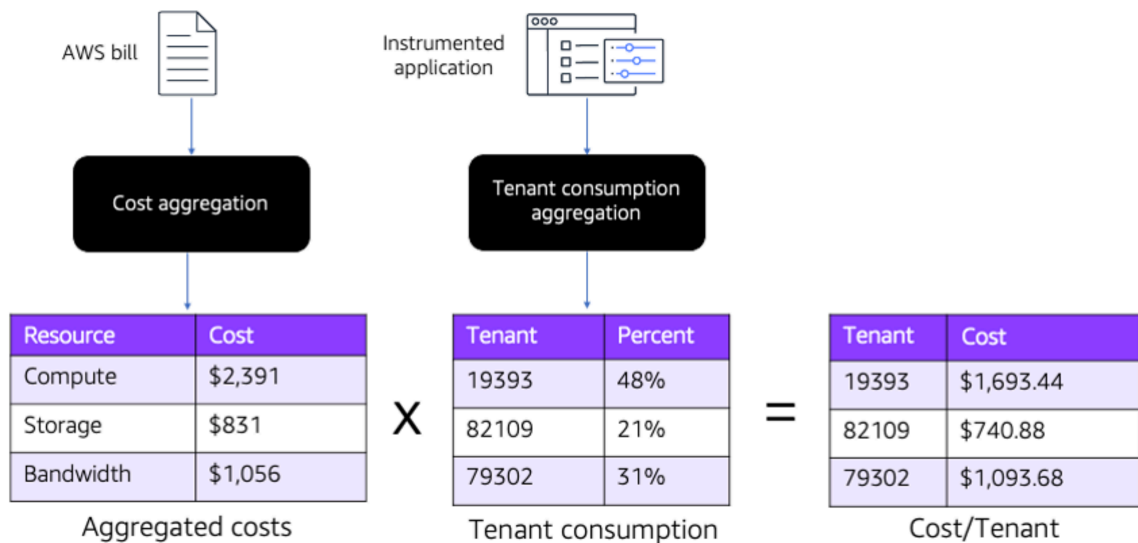


Figure 28: Calculating cost per tenant

After you have this data, you can choose how best to represent the resulting costs. Generally, it would seem valuable to have costs per tenant across a range of services that are core to your business. However, you could use weighting and calculate an overall cost per tenant for other analyses that you might perform.

Optimizing Over Time

There are no cost practices unique to SaaS applications.

Resources

Refer to the following resources to learn more about AWS best practices for cost optimization.

Documentation & Blogs

- Calculating Tenant Costs in SaaS Environments
- Calculating SaaS Cost Per Tenant: A PoC Implementation in an AWS Kubernetes Environment
- SaaS metrics deep dive: A look inside multi-tenant analytics
- SaaS Analytics and Metrics: Capturing and Surfacing the Data That's Fundamental to Your Success
- Monitoring tools in AWS

Videos

- SaaS Metrics: The Ultimate View of Tenant Consumption

Sustainability pillar

The sustainability pillar includes the ability to continually improve sustainability impacts by reducing energy consumption and increasing efficiency across all components of a workload by maximizing the benefits from the provisioned resources and minimizing the total resources required.

There are no sustainability practices unique to this lens. For information on Sustainability, refer to the Sustainability Pillar whitepaper.

Conclusion

The AWS Well-Architected Framework provides architectural best practices across the pillars for designing and operating reliable, secure, efficient, and cost-effective systems in the cloud for SaaS applications. The framework provides a set of questions that allows you to review an existing or proposed architecture, and also a set of AWS best practices for each pillar. Using the framework in your architecture will help you produce stable and efficient systems, which allows you to focus on your functional requirements.

Contributors

The following individuals and organizations contributed to this document:

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

Document Revisions

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Minor update (p. 55)	Corrected numbering of questions to match the SaaS Lens in the AWS Well-Architected Tool.	September 23, 2022
Minor update (p. 55)	Figure 28 corrected.	August 1, 2022
Minor update (p. 55)	Figure corrected for clarity.	December 11, 2020
Initial publication (p. 55)	SaaS Lens first published.	December 3, 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.

© 2021 Amazon Web Services, Inc. or its affiliates. All rights reserved.