AWS Whitepaper

SaaS Architecture Fundamentals

SaaS Architecture Fundamentals: AWS Whitepaper

Table of Contents

SaaS Architecture Fundamentals

Publication date: **August 3, 2022** (*Document revisions*)

The scope, goals, and nature of running a business in a software as a service (SaaS) model can be difficult to define. The terminology and patterns that are used to characterize SaaS vary based on their origin. The goal of this document is to better define the fundamental elements of SaaS and create a clearer picture of patterns, terms, and value systems that are applied when designing and delivering a SaaS system on AWS. The broader goal is to provide a collection of foundational insights that provide customers with a clearer view of the options they should consider as they look to adopt a SaaS delivery model.

This paper is targeted at SaaS builders and architects who are at the beginning of their SaaS journey, as well as more seasoned builders who want to refine their understanding of core SaaS concepts. Some of this information can also be useful to SaaS product owners and strategists who want to get more familiar with the SaaS landscape.

Introduction

The term software as a service (SaaS) is used to describe a business and delivery model. The challenge, however, is that what it means to be SaaS is not universally understood.

While there is some agreement on some of the core pillars of SaaS, there remains some confusion around what it means to be SaaS. It's natural to have some variation in how teams might view SaaS. At the same time, the lack of clarity around SaaS concepts and terms can create some confusion for those exploring a SaaS delivery model.

This document is focused on outlining the terminology that is used to describe core SaaS concepts. Having a shared mindset around these concepts creates a clear picture of the foundational elements of a SaaS architecture, equipping you with a shared vocabulary for describing SaaS architecture constructs. This is especially useful as you dig into additional content that builds on these themes.

This whitepaper steps back from the architecture details of multi-tenancy, and explores how we've defined the fundamentals of what it means to be SaaS. Ideally, this will also provide a clearer set of terminology that allows organizations to more quickly align on the flavor and nature of their SaaS solutions.

Are you Well-Architected?

The AWS Well-Architected Framework helps you understand the pros and cons of the decisions you make when building systems in the cloud. The six pillars of the Framework allow you to learn architectural best practices for designing and operating reliable, secure, efficient, cost-effective, and sustainable systems. Using the AWS Well-Architected Tool, available at no charge in the AWS Management Console, you can review your workloads against these best practices by answering a set of questions for each pillar.

In the SaaS Lens, we focus on best practices for architecting your software as a service (SaaS) workloads on AWS.

For more expert guidance and best practices for your cloud architecture—reference architecture deployments, diagrams, and whitepapers—refer to the AWS Architecture Center.

SaaS is a business model

Defining what it means to be SaaS starts with agreeing on one key principle: SaaS is a business model. This means that—above all else—adoption of a SaaS delivery model is directly driven by a set of business objectives. Yes, technology will be used to realize some of those goals, but SaaS is about putting in place a mindset and model that targets a specific set of business objectives.

Let's look more closely at some of the key business objectives that are associated with adopting a SaaS delivery model.

- **Agility** – This term summarizes the broader goal of SaaS. Successful SaaS companies are built with the idea that they must be ready to continually adapt to the market, customer, and competitive dynamics. Great SaaS companies are structured to continually embrace new pricing models, new market segments, and new customer needs.
- **Operational efficiency** – SaaS companies rely on operational efficiency to promote scale and agility. This means putting into place a culture and tooling that is focused on creating an operational footprint that promotes frequent and rapid release of new features. It also means having a single, unified experience that allows you to manage, operate, and deploy all customer environments collectively. Gone is the idea of supporting one-off versions and customizations. SaaS businesses place a premium on operational efficiency as a core pillar of their ability to successfully grow and scale the business.
- **Frictionless onboarding** – As part of being more agile and embracing growth, you must also place a premium on decreasing any friction in the tenant customer onboarding process. This applies universally to business to business (B2B) and business-to-customer (B2C) customers. No matter which segment or type of customer you support, you still need to be focused on time to value for your customers. The shift to a service-centric model requires the SaaS business to focus on all aspects of the customer experience, with specific emphasis on the repeatability and efficiency of the overall onboarding lifecycle.
- **Innovation** – Moving to SaaS isn't just about addressing the needs of current customers; it's about putting in place the foundational elements that allow you to innovate. You want to react and respond to customer needs in your SaaS model. However, you also want to use this agility to drive future innovations that allow you to unlock new markets, opportunities, and efficiencies for your customers.
- **Market response** – SaaS moves away from the traditional notion of quarterly releases and two-year plans. It relies on its agility to give the organization the ability to react and respond to market dynamics in near real-time. The investment in the organizational, technical, and cultural

elements of SaaS creates the opportunity to pivot business strategy based on the emerging customer and market dynamics.

- **Growth** – SaaS is a growth-centric business strategy. Aligning all the moving parts of the organization around agility and efficiency gives SaaS organizations the ability to target a growth model. This means putting in place the mechanisms that embrace and welcome the rapid adoption of your SaaS offering.

You'll notice that each of these items is focused on a business outcome. There are a wide range of technical strategies and patterns that can be used to build a SaaS system. However, nothing about these technical strategies changes the broader business story.

As we sit down with organizations and ask them what they're trying to achieve as part of their adoption of SaaS, we always start with this business-focused discussion. The technology choices are important, but they must be realized in the context of these business goals. Being multi-tenant without achieving agility, operational efficiency, or frictionless onboarding, for example, would undermine the success of your SaaS business.

With this as our background, let's try to formalize this into a more concise definition of SaaS that conforms to the principles outlined previously:

SaaS is a business and software delivery model that gives organizations the ability to offer their solutions in a low-friction, service-centric model that maximizes value for customers and providers. It relies on agility and operational efficiency as pillars of a business strategy that promotes growth, reach, and innovation.

You should see the alignment between the business objectives and how they rely on having a shared experience for all customers. A big part of moving to SaaS means moving away from the one-off customizations that might be part of a traditional software model. Any effort to offer specialization for customers generally takes us away from the core values we are trying to achieve with SaaS.

You’re a service, not a product

Adopting a “service” model is more than just marketing or terminology. In a service mindset, you will find yourself moving away from aspects of the traditional product-based approach to development. While features and functionality are certainly important to every product, SaaS places more emphasis on the experience customers will have with your service.

What does this mean? In a service-centric model, you think more about how customers are onboarded to your service, how quickly they achieve value, and how rapidly you can introduce features that address customer needs. Details associated with how your service is built, operated, and managed are out of your customer’s view.

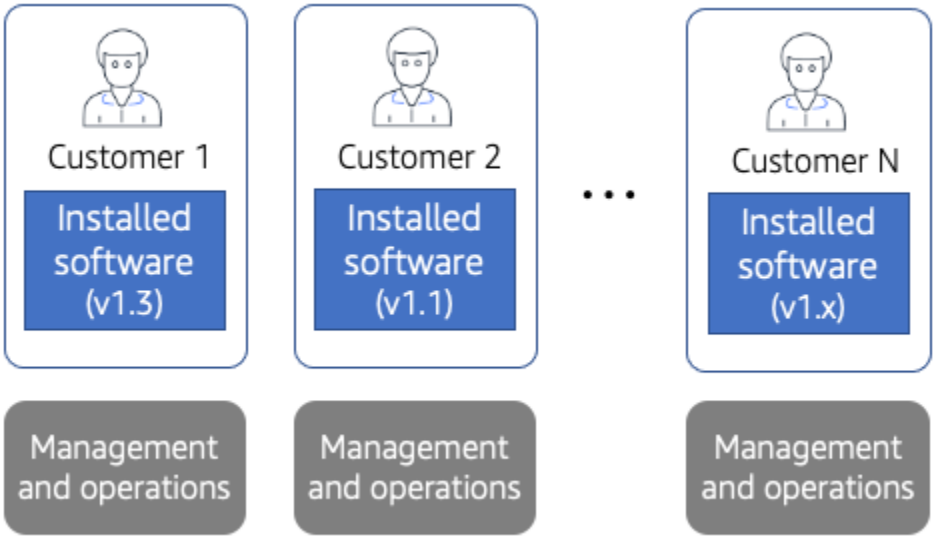
In this mode, we think about this SaaS service as we would with any other service we might consume. If we’re in a restaurant, we certainly care about the food, but we also care about the service. How quickly does your server come to your table, how often do they refill your water, how fast does the food come—these are all measures of the service experience. This is the same mindset and value system that should shape how we think about building a SaaS service.

This *as-a-service* model should have a heavy influence on how you build your teams and your service. Your backlog of work will now put these experience attributes on equal or higher footing than features and functions. The business will also view these as foundational to the long-term growth and success of your SaaS offering.

The initial motivation

To understand SaaS, let's start with a fairly simple notion of what we're trying to achieve when creating a SaaS business. The best place to start is by looking at how traditional (non-SaaS) software has been created, managed, and operated.

The following diagram provides a conceptual view of how several vendors have packaged and delivered their solutions.



The classic model for packaging and delivering software solutions

In this diagram, we've described a collection of customer environments. These customers represent the different companies or entities that have purchased a vendor's software. Each of these customers is essentially running in a standalone environment where they have installed a software provider's product.

In this mode, each customer's installation is treated as a standalone environment that is dedicated to that customer. This means that customers view themselves as the owners of these environments, potentially requesting one-off customization or unique configurations that support their needs.

One common side effect of this mindset is that customers will control which version of a product they are running. This could happen for a variety of reasons. Customers might be apprehensive about new features, or worried about disruptions associated with adopting a new version.

You can imagine how this dynamic can impact the operational footprint of the software provider. The more you allow customers to have one-off environments, the more challenging it becomes to manage, update, and support the varying configurations of each customer.

This need for one-off environments often requires organizations to create dedicated teams that provide separate management and operations experience for each customer. While some of these resources might be shared across customers, this model generally introduces incremental expenses for each new customer that is onboarded.

Having each customer run their solutions in their own environment (in the cloud or on premises) also impacts cost. While you can attempt to scale these environments, the scaling will be limited to the activity of a single customer. Essentially, your cost optimization is limited to what you can achieve within the scope of an individual customer environment. It also means that you might need separate scaling strategies to accommodate the activity variations between customers.

Initially, some software businesses will view this model as a powerful construct. They use the ability to provide one-off customization as a sales tool, allowing new customers to impose requirements that are unique to their environment. While the customer count and growth of the business remains modest, this model seems perfectly sustainable.

As companies begin to have broader success, however, the constraints of this model begin to create real challenges. Imagine, for example, a scenario where your business reaches a significant growth spike that has you adding lots of new customers at a rapid pace. This growth will begin to add to the operational overhead, management complexity, cost, and a host of other issues.

Ultimately, the collective overhead and impact of this model can begin to fundamentally undermine the success of a software business. The first pain point might be operational efficiency. The incremental staffing and costs associated with bringing on customers begins to erode the margins of the business.

However, operational issues are just part of the challenge. The real issue is that this model, as it scales, directly begins to impact the business's ability to release new features and keep pace with the market. When each customer has their own environment, providers must balance a host of update, migration, and customer requirements as they attempt to introduce new capabilities into their system.

This typically leads to longer and more complex release cycles, which tends to reduce the number of releases that are made each year. More importantly, this complexity has teams devoting more time to analyzing each new feature long before it's released to a customer. Teams begin to be more

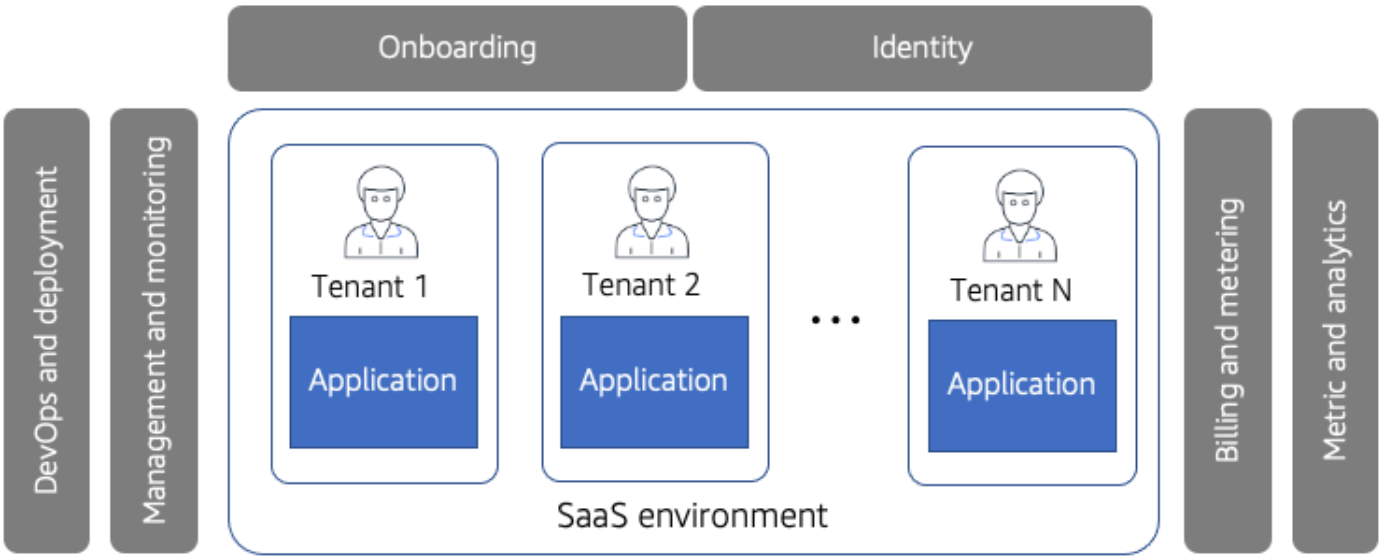
focused on validating new features and less on the speed of delivery. The overhead of releasing new features becomes so significant that teams become more focused on the mechanics of testing, and less on the new functionality that drives the innovation of their offering.

In this slower, more cautious mode, teams tend to have long cycle times that put larger and larger gaps between the inception of an idea and when it lands in the hands of a customer. Overall, this can hinder your ability to react to market dynamics and competitive pressures.

Moving to a unified experience

To address the needs of this classic software dilemma, organizations turn to a model that allows them to create a single, unified experience that allows customers to be managed and operated collectively.

The following diagram provides a conceptual view of an environment where all of the customers are managed, onboarded, billed, and operated through a shared model.



A conceptual view of an environment where all of the customers are managed, onboarded, billed, and operated through a shared model

At first glance, this may not seem all that different than the prior model. However, as we dig in a bit further, you'll see that there are fundamental, significant differences in these two approaches.

First, you'll notice that the customer environments have been renamed to tenants. This notion of a *tenant* is foundational to SaaS. The basic idea is that you have a single SaaS environment, and each one of your customers is viewed as a tenant of that environment, consuming the resources they need. A tenant could be a company with many users, or it could correlate directly to an individual user.

To better understand the idea of a tenant, consider the idea of apartment or commercial buildings. The space in each of these building is rented out to individual tenants. The tenants rely on some of the shared resources of the building (water, power, and so on), paying for what they consume.

SaaS tenants follow a similar pattern. You have the infrastructure of your SaaS environment, and tenants that consume the infrastructure of that environment. The amount of resources consumed by each tenant can vary. These tenants are also managed, billed, and operated collectively.

If you turn back to the diagram, you'll see the notion of tenancy brought to life. Here, tenants no longer have their own environment. Instead, all the tenants are housed and managed within the walls of one collective SaaS environment.

The diagram also includes a range of shared services that surround your SaaS environment. These services are global to all of the tenants of your SaaS environment. This means that onboarding and identity, for example, are shared by all tenants of this environment. The same is true for management, operations, deployment, billing, and metrics.

This idea of a unified set of services that are applied universally to all of your tenants is foundational to SaaS. By sharing these concepts, you're able to address a number of the challenges that were associated with the classic model described above.

Another key, somewhat subtle element of this diagram is that all of the tenants in this environment are running the same version of your application. Gone is the idea of having separate, one-off versions running for each customer. Having all tenants running the same version represents one of the fundamental distinguishing attributes of a SaaS environment.

By having all customers running the same version of your product, you no longer face many of the challenges of a classic installed software model. In the unified model, new features can be deployed to all tenants by a single, shared process.

This approach gives you the ability to employ a single pane of operational glass that can manage and operate all tenants. This lets you manage and monitor your tenants through a common operational experience, allowing new tenants to be added without adding incremental operational overhead. This is a core part of the SaaS value proposition that gives teams the ability to reduce operational expenses, and improve overall organizational agility.

Imagine what it would mean to add 100 or 1,000 new customers in this model. Instead of worrying about how these new customers might erode margins and add complexity, you can view this growth as the opportunity it represents.

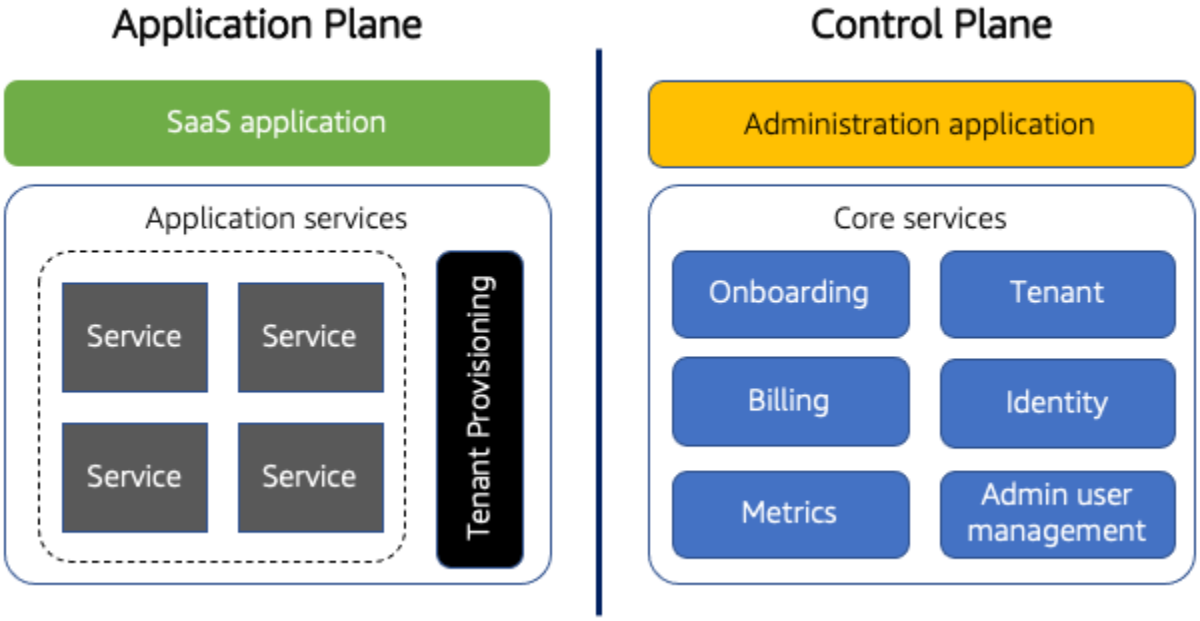
Generally, the focus of SaaS is placed on how the application in the middle of this model is implemented. Businesses want to focus on how data is stored, how resources are shared, and so on. However, the reality is that, while these details are definitely important, there are many ways your application can be built and still present itself as a SaaS solution to your customers.

What's critical is the broader goal of having a single, unified experience that surrounds your tenant environments. Having this shared experience is what allows you to drive the growth, agility, and operational efficiency that is connected to the overall objectives of a SaaS business.

Control plane vs. application plane

The previous diagram provides a conceptual view of the core SaaS architecture concepts. Now let's drill down into that deeper, and better define how your SaaS environment decomposed into distinct layers. Having this clearer picture of the boundaries between SaaS concepts will make it easier to describe the moving parts of a SaaS solution.

The following diagram divides your SaaS environment into two distinct planes. On the right side is the control plane. This side of the diagram includes all the functionality and services that are used to onboard, authenticate, manage, operate, and analyze a multi-tenant environment.



Control plane vs. application plane

This control plane is foundational to any multi-tenant SaaS model. Every SaaS solution—regardless of application deployment and isolation scheme—must include those services that give you the ability to manage and operate your tenants through a single, unified experience.

Within the control plane, we have further broken this down into two distinct elements. The core services here represent the collection of services that are used to orchestrate your multi-tenant experience. We've included some of the common examples of services that are typically part of the core, acknowledging that the core services could vary some for each SaaS solution.

You'll also notice that we show a separate administration application. This represents the application (a web application, a command line interface, or an API) that might be used by a SaaS provider to manage their multi-tenant environment.

An important caveat is that the control plane and its services are not actually multi-tenant. The functionality is not providing the actual functional attributes of your SaaS application (which does need to be multi-tenant). If you look inside any one of the core services, for example, you won't find tenant isolation and the other constructs that are part of your multi-tenant application functionality. These services are global to all tenants.

The left side of the diagram references the application plane of a SaaS environment. This is where the multi-tenant functionality of your application resides. What appears in the diagram needs to remain somewhat vague, because each solution can be deployed and decomposed differently based on the needs of your domain, footprint of your technology, and so on.

The application domain is separated into two elements. There is the SaaS application that represents the tenant experience/application for your solution. This is the surface that tenants touch to interact with your SaaS application. Then there are the backend services that represent business logic and functional elements of a SaaS solution. These could be microservices, or some other packaging of your application services.

You'll also note that we've broken out provisioning. This is done to highlight the fact that any provisioning of resources for tenants during onboarding would be part of this application domain. Some could argue that this belongs in the control plane. However, we've placed it in the application domain, because the resources it must provision and configure are more directly connected to services that are created and configured in the application plane.

Breaking this down into distinct planes makes it easier to think about the overall landscape of a SaaS architecture. More importantly, it highlights the need for a set of services that are entirely outside the scope of your application functionality.

Core services

The control plane referenced previously mentions a series of core services that represent the typical services that are used to onboard, manage, and operate a SaaS environment. It may be helpful to further highlight the role of some of these services to highlight their scope and purpose in a SaaS environment. The following provides a brief summary of each of these services:

- **Onboarding** – Every SaaS solution must provide a frictionless mechanism for introducing new tenants into your SaaS environment. This can be a self-service sign-up page or an internally managed experience. Either way, a SaaS solution should do all that it can remove internal and external friction from this experience and ensure stability, efficiency, and repeatability for this process. It plays an essential role in supporting the growth and scale of a SaaS business. Generally, this service orchestrates other services to create users, tenant, isolation policies, provision, and per-tenant resources.
- **Tenant** – The tenant service provides a way to centralize the policies, attributes, and state of tenants. The key is that tenants are not individual users. In fact, a tenant is likely associated with many users.
- **Identity** – SaaS systems need a clear way to connect users to tenants that will bring tenant context to the authentication and authorization experience of their solutions. This influences both the onboarding experience and the overall management of user profiles.
- **Billing** – As part of adopting SaaS, organizations often embrace new billing models. They may also explore integration with third-party billing providers. This core service is largely focused on supporting the onboarding of new tenants, and collecting consumption and activity data that is used to generate bills for tenants.
- **Metrics** – SaaS teams rely heavily on their ability to capture and analyze rich metric data that brings more visibility to how tenants use their system, how they consume resources, and how their tenants engage their systems. This data is used to shape operational, product, and business strategies.
- **Admin user management** – SaaS systems must support both tenant users and admin users. The admin users represent the administrators of a SaaS provider. They will log into your operational experience to monitor and manage your SaaS environment.

Re-defining multi-tenancy

The terms *multi-tenancy* and *SaaS* are often tightly connected. In some instances, organizations describe SaaS and multi-tenancy as the same thing. While this might seem natural, equating SaaS and multi-tenancy tends to lead teams to take a purely technical view of SaaS when, in reality, SaaS is more of a business model than an architecture strategy.

To better understand this concept, let's start with the classic view of multi-tenancy. In this purely infrastructure-focused view, multi-tenancy is used to describe how resources are shared by tenants to promote agility and cost efficiency.

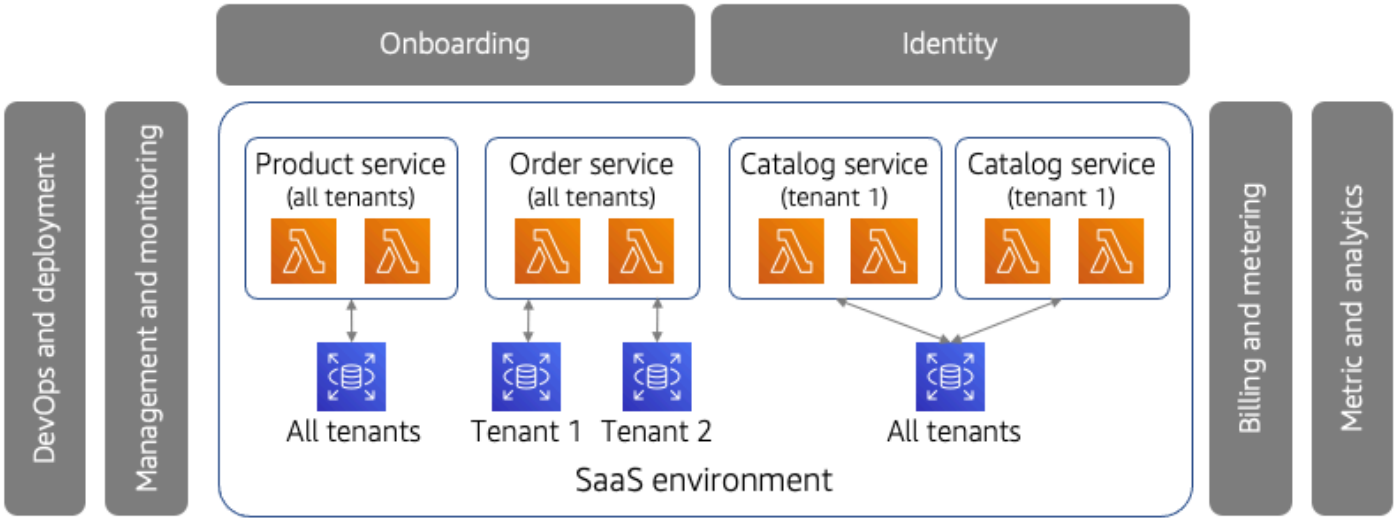
Suppose, for example, you have a microservice or an Amazon Elastic Compute Cloud (Amazon EC2) instance that is consumed by multiple tenants of your SaaS system. This service would be viewed as running in a multi-tenant model, because tenants are sharing use of the infrastructure that runs this service.

The challenge of this definition is that it too directly attaches the technical notion of multi-tenancy to SaaS. It presumes that the defining characteristic of SaaS is that it must have shared, multi-tenant infrastructure. This view of SaaS begins to fall apart as we look at the various ways that SaaS is realized in different environments.

The following diagram provides a view of a SaaS system that exposes some of the challenges we have with defining multi-tenancy.

Here you'll see the classic SaaS model described previously, with a series of application services surrounded by the shared services that allow you to manage and operate your tenants collectively.

What's new is the microservices that we've included. The diagram includes three sample microservices: *product*, *order*, and *catalog*. If you look closely at the tenancy model of each of these services, you'll notice they all employ slightly different patterns of tenancy.



SaaS and multi-tenancy

The product service shares all of its resources (compute and storage) with all tenants. This aligns with the classic definition of multi-tenancy. However, if you look at the order service, you'll see that it has shared compute, but separate storage for each tenant.

The catalog service adds another variation where the compute is separate for every tenant (a separate microservice deployment for each tenant), but it shares the storage for all tenants.

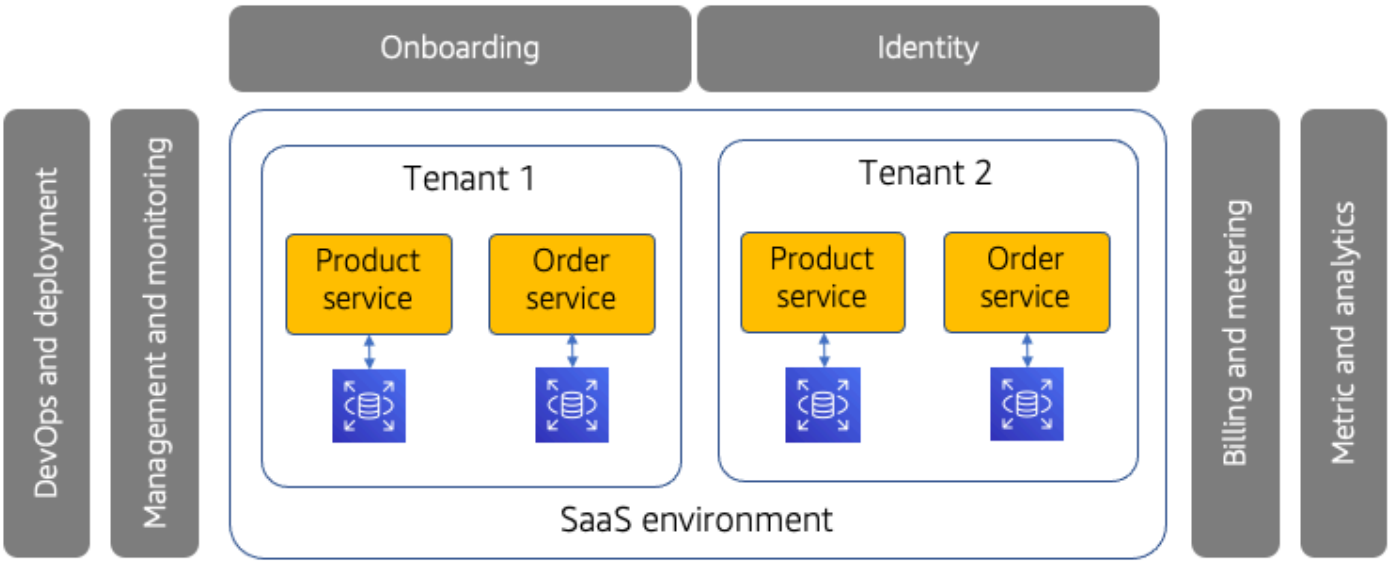
Variations of this nature are common in SaaS environments. Noisy neighbor, tiering models, isolation needs—these are amongst a list of reasons that you might selectively share or silo parts of your SaaS solution.

Given these variations—and many of the other possibilities—it becomes more challenging to figure out how you should use the term *multi-tenant* to characterize this environment. Overall, as far the customer is concerned, this is a multi-tenant environment. However, if we use the most technical definition, some parts of this environment are multi-tenant and some are not.

This is why it becomes necessary to move away from using the term *multi-tenant* to characterize SaaS environments. Instead, we can talk about how multi-tenancy is implemented within your application, but avoid using it to characterize a solution as being SaaS. If the term *multi-tenant* is to be used, it makes more sense to use it to describe the entire SaaS environment as being multi-tenant, knowing that some parts of the architecture may be shared and some may not. Overall, you are still operating and managing this environment in a multi-tenant model.

The extreme case

To better highlight this notion of tenancy, let's look at a SaaS model that has tenants that share zero resources. The following diagram provides a sample SaaS environment that is employed by some SaaS providers.



Stack per tenant

In this diagram, you'll see that we still have our common environment surrounding these tenants. However, each tenant is deployed with a dedicated collection of resources. Nothing is shared by the tenants in this model.

This example challenges what it means to be multi-tenant. Is this a multi-tenant environment even though none of the resources are shared? The tenants that consume this system have all the same expectations you'd have of a SaaS environment with shared resources. In fact, they may have no awareness of how their resources are deployed under the hood of the SaaS environment.

Even though these tenants are running in a siloed infrastructure, they are still managed and operated collectively. They share one unified onboarding, identity, metrics, billing, and operational experience. Also, when a new version is released, it is deployed to all tenants. For this to work, you cannot allow one-off customization for individual tenants.

This extreme example provides a good model for testing the notion of multi-tenant SaaS. While it may not realize all the efficiencies of shared infrastructure, it is an entirely valid multi-tenant SaaS

environment. For some customers, their domain may dictate that some or all of their customers run in this model. That doesn't mean they are not SaaS. If they use these shared services and all the tenants run the same version, this still complies with the foundational principles of SaaS.

Given these parameters and this broader definition of SaaS, you can see the need to evolve the use of the term *multi-tenant*. It makes more sense to refer to any SaaS system that is managed and operated collectively as being *multi-tenant*. Then, you can defer to more granular terminology to describe how resources are shared or dedicated within the implementation of a SaaS solution.

Removing the single-tenant term

As part of using the term *multi-tenant*, it's only natural for people to want to use the term *single-tenant* to describe SaaS environments. However, given the backdrop outlined previously, the term *single-tenant* creates confusion.

Is the preceding diagram a single-tenant environment? While each tenant technically has its own stack, these tenants are still being operated and managed in a multi-tenant model. This is why the term *single-tenant* is generally avoided. Instead, all environments are characterized as multi-tenant, as they are just implementing some variation of tenancy where some or all of the resources are shared or dedicated.

Introducing silo and pool

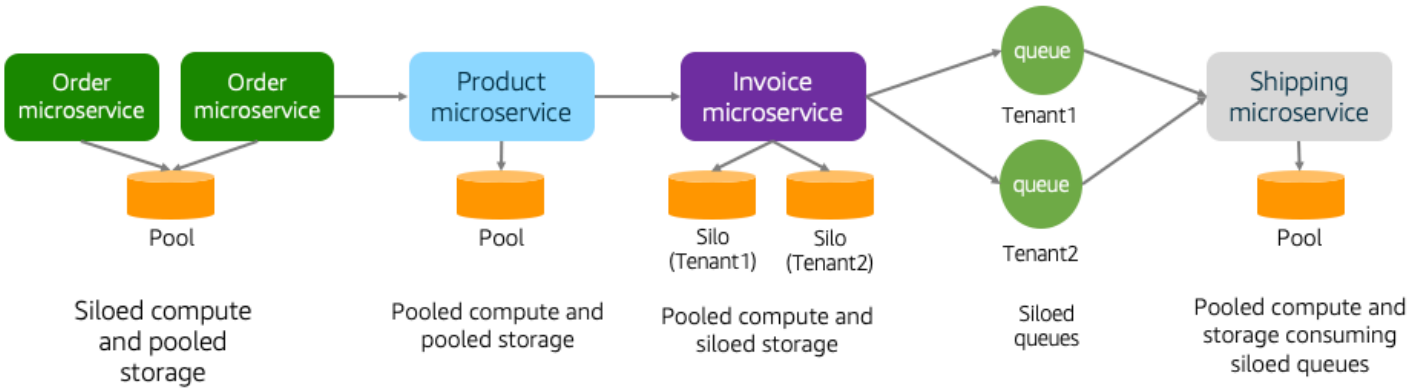
With all these variations of models and given the challenges around the term *multi-tenancy*, we have introduced some terminology that lets us more accurately capture and describe the different models that are used when building a SaaS.

Two terms that we use to characterize the use of resources in a SaaS environment are *silo* and *pool*. These terms allow us to label the nature of SaaS environments, using *multi-tenant* as an over-arching description that can be applied to any number of underlying models.

At the most basic level, the term *silo* is meant to describe scenarios where a resource is dedicated to a given tenant. Conversely, the *pool* model is used to describe scenarios where a resource is shared by tenants.

As we look at how the silo and pool terms are used, it's important to be clear that silo and pool are not all-or-nothing concepts. Silo and pool could apply to an entire tenant's stack of resources, or it could be applied selectively to parts of your overall SaaS environment. So, if we say some resource is using a *silo* model, that does not mean all resources in that environment are siloed. The same holds true for how we would use the term *pooled*.

The following diagram provides an example of how siloed and pooled models are used more granularly in a SaaS environment:



Silo and pool models

This diagram includes a series of samples that are meant to illustrate the more targeted nature of the silo and pool models. If you follow this from left to right, you'll see that we start out with an order microservice. This microservice has siloed compute and pooled storage. It interacts with a product service that has pooled compute and pooled storage.

The product service then interacts with an invoice microservice that has pooled compute and siloed storage. This service sends messages via queues to the shipping service. The queues are deployed in a siloed model.

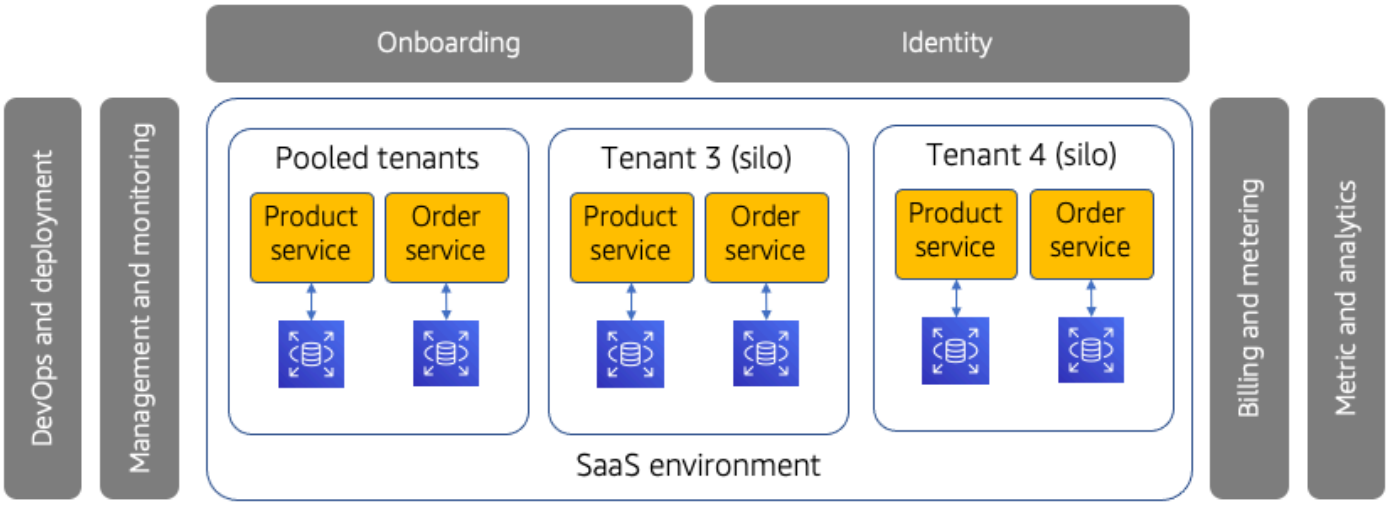
Finally, the shipping microservice acquires messages from the siloed queues. It uses pooled compute and storage.

While this may seem a bit convoluted, the goal is to highlight the granular nature of the silo and pool concepts. As you look to design and build your SaaS solution, it's expected that you would make these silo and pool decisions based on the needs of your domain and your customers.

Noisy neighbor, isolation, tiering, and a host of other reasons might influence how and when you choose to apply the silo or pooled model.

Full stack silo and pool

Silo and pool can also be used to describe an entire SaaS stack. In this approach, all the resources for a tenant are deployed in either a dedicated or shared manner. The following diagram provides an example of how this might land in a SaaS environment.



Full stack silo and pool models

In this diagram, you'll see that there are three different models for your full stack tenant deployments. First, you'll see that there is a full stack pool environment. Tenants in this pool share all the resources (compute, storage, and so on).

The other two stacks shown are meant to represent full stack siloed tenant environments. In this case, Tenant 3 and Tenant 4 are shown as each having their own dedicated stacks where none of the resources are shared with other tenants.

This mix of silo and pooled models in the same SaaS environment isn't all that atypical. Imagine, for example, that you have a set of basic tier tenants that pay a moderate price for using your system. These tenants are placed in the pooled environment.

Meanwhile, you may also have premium tier tenants that are willing to pay more for the privilege of running in a silo. These customers are deployed with separate stacks (as shown in the diagram).

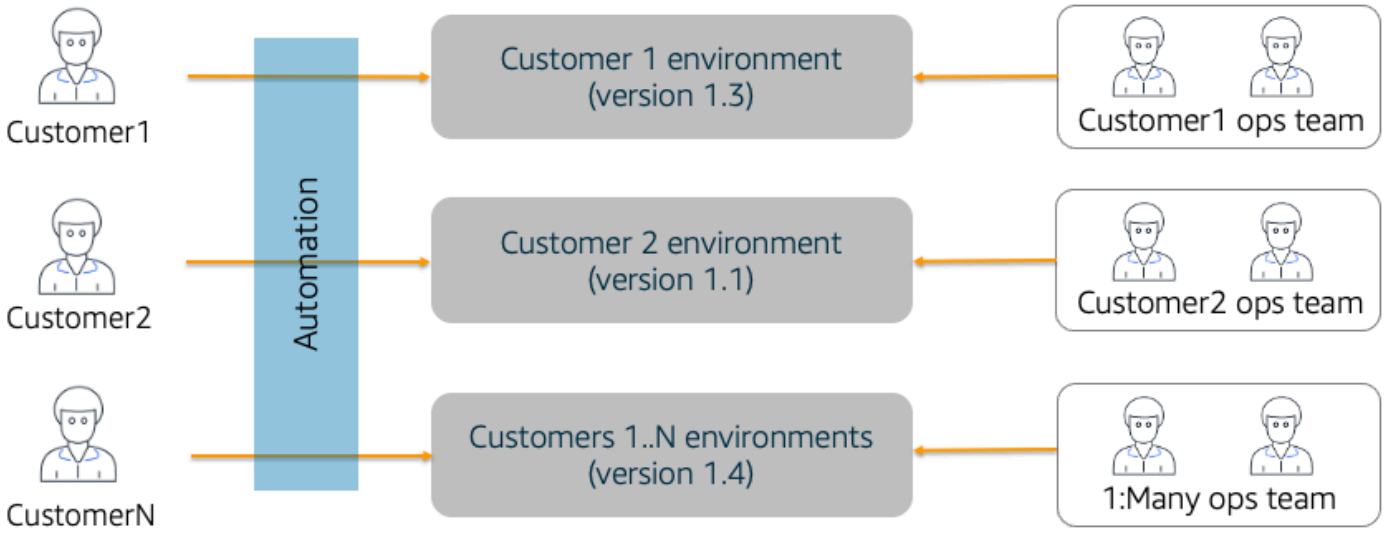
Even in this model, where you may have allowed tenants to run in their own full stack silo, it would be essential that these siloes don't allow any one-off variation or customization for these tenants.

In all respects, each of these stacks should be running the same configuration of the stack, with the same version of the software. When a new version is released, it's deployed to the pooled tenant environment, and each of the siloed environments.

SaaS vs. Managed Service Provider (MSP)

There is also some confusion that surrounds the lines between SaaS and the Managed Service Provider (MSP) models. If you look at an MSP model, it can appear to have some similar goals as the SaaS model.

If you dig a bit more into MSP, however, you'll find that MSP and SaaS are actually different. The following diagram provides a conceptual view of an MSP environment.



Managed Service Provider (MSP) model

This diagram represents one approach to the MSP model. On the left you'll see customers that run in the MSP model. Generally, the approach here would be to use whatever automation is available to provision each customer environment, and install the software for that customer.

On the right is some approximation of the operational footprint that the MSP would provide to support these customer environments.

It's important to note that the MSP is often installing and managing a version of the product that a given customer wants to run. All customers could be running the same version, but this is not typically required in an MSP model.

The general strategy is to simplify the life of a software provider by owning the installation and management of these environments. While this makes life simpler for the provider, it does not map directly to the values and mindset that are essential to a SaaS offering.

The focus is on offloading the management responsibility. Making that move is not the equivalent of having all customers run on the same version with a single, unified management and operations experience. Instead, MSP often allows for separate versions, and often treats each of these environments as operationally separate.

There are certainly areas where MSP might begin to overlap with SaaS. If the MSP essentially required all customers to run the same version and the MSP was able to centrally onboard, manage, operation, and bill all the tenants through one experience, that might begin to be more SaaS than MSP.

The broader theme is that automating the install of environments does not equate to having a SaaS environment. Only when you add all the other caveats previously discussed would this represent more of a true SaaS model.

If we move back from the technology and operations aspects of this story, the line between MSP and SaaS become even more distinct. Generally, as a SaaS business, the success of your offering relies on your ability to be deeply involved in all the moving parts of the experience.

This usually means having your finger on the pulse of the onboarding experience, understanding how operational events impact tenants, tracking key metrics and analytics, and being close to your customer. In an MSP model where this is handed over to someone else, you may end up being a level removed from the key details that are core to operating a SaaS business.

SaaS migration

Many of the providers that adopt SaaS migrate to SaaS from a traditional installed software model (described previously). For these providers, it's especially important to have good alignment on the core principles of SaaS.

Here, again, is where there can be some confusion about what it means to migrate to a SaaS model. Some, for example, view moving to the cloud as migrating to SaaS. Others view adding automation to their installation and provisioning process as achieving migration.

It's fair to say that every organization may start at a different spot, have different legacy considerations, and is likely facing different market and competitive pressures. This means that every migration will look different.

Still, while every path is different, there are some areas where there are disconnects around the core principles that shape migration strategies. Having good alignment around the concepts and principles can have a significant impact on the overall success of your SaaS migration.

Based on the concepts outlined previously, it should be clear that the move to SaaS starts with the business strategy and goals. This point can get lost in a migration setting where there is pressure to get to SaaS as quickly as possible.

In this mode, organizations often view migration primarily as a technical exercise. The reality is, every SaaS migration should start with a clear view of the target customers, the service experience, the operational goals, and so on. Having a clearer focus on what your SaaS business needs to look like will have a profound impact on the shape, priorities, and path you take to migrate your solution to SaaS.

Having this clear vision from the outset of your migration lays the foundation for how you migrate both the technology and the business as part of your move to SaaS. As you set out on this path, focus on those questions that can tell you the most about where you're headed.

The following table provides a view of the contrasting nature of technical and business migration mindsets.

Table 1 — Technical first vs. business first migration

The tech first mindset	The business first mindset
How do we isolate tenant data?	How can SaaS help us grow our business?
How do we connect users to tenants?	Which segments are we targeting?
How do we avoid noisy neighbor conditions?	What is the size and profile of these segments?
How do we do A/B testing?	What tiers will we need to support?
How do we scale based on tenant load?	What service experience are we targeting?
Which billing provider should we use?	What is our pricing and packaging strategy?

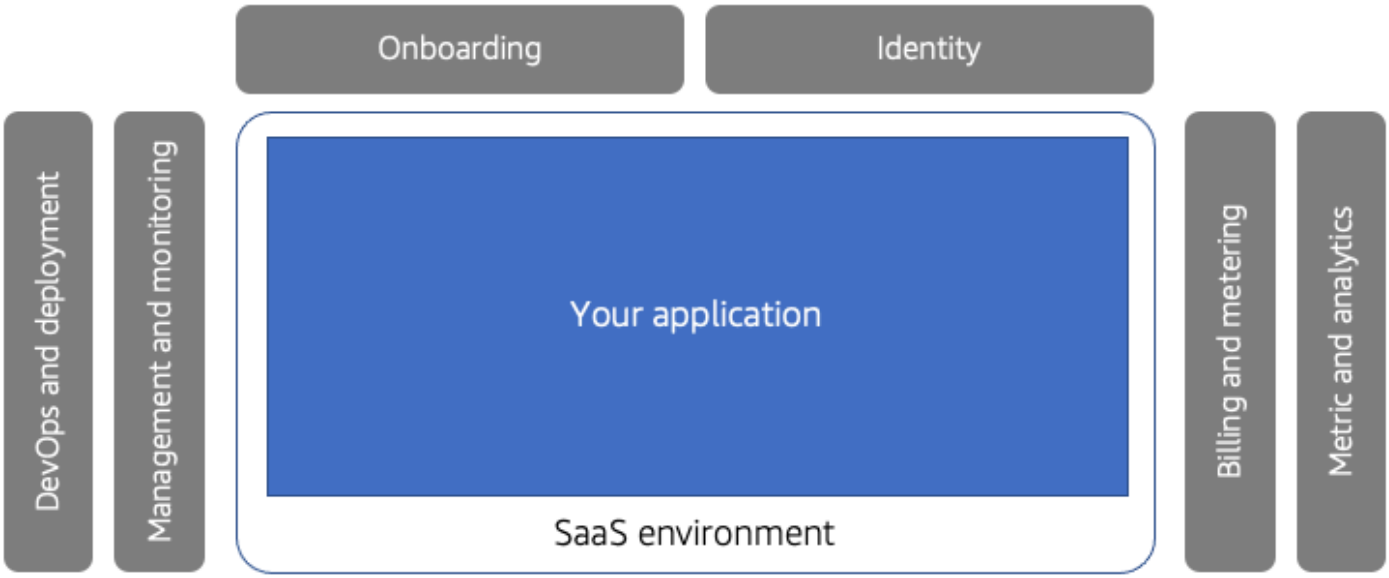
On the left is an example of what a tech first migration mindset might look like. The engineering team is hyper-focused on chasing the classic multi-tenant topics that are certainly important to any SaaS architecture.

The problem is that the answers to many of the questions on the left are often directly influenced by the answers to the question on the right. It's unlikely that this point is new to anyone looking at migration. However, the reality is that many organizations start out by chasing the operational and cost efficiency as their first step, assuming the business bits will work themselves out.

Within this migration strategy, there can also be confusion about how your legacy environment might be evolved to fit into a SaaS model. This too is an area where there are a multitude of options for migrating to SaaS. However, there is a common value system that we generally advocate for any migration.

In our previous discussion of SaaS principles, we outlined different patterns and terminologies that are used to describe SaaS environments. One common theme spanning all of those solutions is this idea of having shared services that surround your application. Identity, onboarding, metrics, billing—these are all called out as common elements that are core to any SaaS environment.

Now, as we look at migration, you'll see that these same shared services play a key role in any migration story. The following diagram provides a conceptual view of the migration landscape.



Migrating to SaaS

This diagram represents the target experience for any migration path. It includes all the same shared services that were described previously. In the middle is a placeholder for your application.

The key idea is that you can land any number of application models in the middle of this environment. Your first step in migration might have each tenant running in its own silo. Or, you may have some hybrid architecture where elements are siloed and other bits of functionality are addressed through a collection of modernized microservices.

How much you initially modernize your application will vary based on the nature of your legacy environment, market needs, cost considerations, and so on. What doesn't change is the introduction of these shared services.

Any SaaS migration needs to support these foundational shared services to give your business the ability to operate in a SaaS model. All variations of the application architecture need SaaS identity, for example. You'll need tenant-aware operations to manage and monitor your SaaS solution.

Putting these shared services in place at the front of your migration allows you to present a SaaS experience to your customers—even if the underlying application is still running in a full stack silo for each tenant.

The general goal is to get your application running in a SaaS model. Then, you can turn your attention to further modernization and refinement of your application. This approach also allows

you to move the other parts of your business (marketing, sales, support, and so on) at a faster pace. More importantly, this allows you to begin to engage and collect customer feedback that can be used to shape the ongoing modernization of your environment.

It's important to note that the shared services that you put in place may not include every feature or mechanism you'll ultimately need. The main goal is to create the shared mechanisms that are needed at the outset of your migration. This allows you to focus on the elements of the system that are essential to the evolution of your application architecture and your operational evolution.

SaaS identity

SaaS adds new considerations to your application's identity model. As each user is authenticated, they must be connected to a specific tenant context. This tenant context provides essential information about your tenant that is used throughout your SaaS environment.

This binding of tenants to users is often referred to as the *SaaS identity* of your application. As each user authenticates, your identity provider will typically yield a token that includes both the user identity and tenant identity.

Connecting tenants to users represents a foundational aspect of your SaaS architecture that has many downstream implications. The token from this identity process flows into the microservices of your application and is used to create tenant aware logs, record metrics, meter billing, enforce tenant isolation, and so on.

It's essential that you avoid scenarios that rely on separate, standalone mechanisms that map users to tenants. This can undermine the security of your system, and often creates bottlenecks in your architecture.

Tenant isolation

The more you move customers into a multi-tenant model, the more they will be concerned about the potential for one tenant to access the resources of another tenant. SaaS systems include explicit mechanisms that ensure that each tenant's resources—even if they run on shared infrastructure—are isolated.

This is what we refer to as *tenant isolation*. The idea behind tenant isolation is that your SaaS architecture introduces constructs that tightly control access to resources, and block any attempt to access resources of another tenant.

Note that tenant isolation is separate from general security mechanisms. Your system will support authentication and authorization; however, the fact that a tenant user is authenticated does not mean that your system has achieved isolation. Isolation is applied separately from the basic authentication and authorization that may be part of your application.

To better understand this, imagine you've used an identity provider to authenticate access to your SaaS system. The token from this authentication experience may also include information about a user's role which could be used to control that user's access to a specific application functionality. These constructs provide security, but not isolation. In fact, a user could be authenticated and authorized, and still access the resources of another tenant. Nothing about authentication and authorization will necessarily block this access.

Tenant isolation focuses exclusively on using tenant context to limit access to resources. It evaluates the context of the current tenant, and uses that context to determine which resources are accessible for that tenant. It applies this isolation for all users within that tenant.

This gets more challenging as we look at how tenant isolation is realized across all the different SaaS architecture patterns. In some cases, isolation may be achieved by having entire stacks of resources dedicated to a tenant where network (or more coarse-grained) policies prevent cross-tenant access. In other scenarios, you may have pooled resources (items in an Amazon DynamoDB table) that require more fine-grained policies to control access to the resources.

Any attempt to access a tenant resource should be scoped to just those resources that belong to that tenant. It's the job of SaaS developers and architects to determine which combination of tools and technologies will support the isolation requirements of your specific application.

Data partitioning

Data partitioning is used to describe different strategies used to represent data in a multi-tenant environment. This term is used broadly to cover a range of different approaches and models that can be used to associate different data constructs with individual tenants.

Note that there is often a temptation to view data partitioning and tenant isolation as interchangeable. These two concepts are not meant to be equivalent. When we talk about *data partitioning*, we are talking about how tenant data is stored for individual tenants. Partitioning data does not ensure that the data is isolated. Isolation must still be applied separately to ensure that one tenant can't access the resources of another tenant.

Each AWS storage technology brings its own set of considerations to the data partitioning strategy. For example, isolating data in Amazon DynamoDB will look very different than isolating data with Amazon Relational Database Service (Amazon RDS).

Generally, when you think about data partitioning, you start by thinking about whether the data will be siloed or pooled. In a siloed model, you have a distinct storage construct for each tenant with no co-mingled data. For pooled partitioning, the data is co-mingled and partitioned based on a tenant identifier that determines which data is associated with each tenant.

As an example, with Amazon DynamoDB, a siloed model uses a separate table for each tenant. Pooling data in Amazon DynamoDB is achieved by storing the tenant identifier in the partition key of each Amazon DynamoDB table that manages data for all tenants.

You can imagine how this might vary across the range of AWS services, with each one introducing its own constructs that may require a different approach to realizing silo and pooled storage models with each service.

While data partitioning and tenant isolation are separate topics, the data partitioning strategies you choose are likely to be influenced by the isolation model of your data. For example, you might silo some storage because that approach best aligns with the requirements of your domain or customers. Or, you might choose silo because the pool model may not allow you to enforce isolation with the level of granularity that your solution requires.

Noisy neighbor can also impact your approach to isolation. Some workloads or use cases in your application may need to be kept separate to limit impacts from other tenants or to meet service level agreements (SLAs).

Metering, metrics, and billing

Discussions of SaaS also tend to include the notion of *metering*, *metrics*, and *billing*. These concepts often get folded together into one concept. However, it's important to distinguish the different roles that metering, metrics, and billing play in a SaaS environment.

The challenge of these concepts is that they often have overlapping uses of the same word. For example, we can talk about metering that is used to generate your bill. At the same time, we can also talk about metering that is used to track internal consumption of resources that is not connected to billing. We also talk about metrics and SaaS in many contexts that can get mixed into this discussion.

To help sort this out, let's associate some specific concepts with each of these terms (knowing there are not absolutes here).

- **Metering** – This concept, while it has many definitions, fits best in the SaaS billing domain. The idea is that you meter tenant activity or consumption of resources to collect the data needed to generate a bill.
- **Metrics** – Metrics represent all the data that you capture to analyze trends across your business, operations, and technology domains. This data is used to across many contexts and roles within the SaaS team.

This distinction isn't critical, but helps simplify how we think about the role of metering and metrics in a SaaS environment.

Now, if we connect these two concepts to examples, you can think of instrumenting your application with specific metering events used to surface the data needed to generate a bill. This could be number of requests, number of active users, or it could map to some aggregate of consumption (requests, CPU, memory) that correlates to some unit that makes sense to your customers.

In your SaaS environment, you'll publish these billing events from your application and they will be ingested and applied by the billing construct employed by your SaaS system. This could be a third-party billing system or something custom.

In contrast, the mindset behind metrics is to capture those actions, activities, consumption patterns, and so on that are essential to evaluating the health and operational footprint imposed on your system by different tenants. The metrics you publish and aggregate here are dictated more

by the need of different personas (operational teams, product owners, architects, and so on). Here, this metric data is published and aggregated into some analytics tooling that allows these different users to build the views of system activity that analyze the aspects of the system that best align with their persona. A product owner may want to understand how different tenants are consuming features. An architect might need views that help them understand how tenants are consuming infrastructure resources, and so on.

B2B and B2C SaaS

SaaS offerings are created for both B2B and B2C markets. While the markets and customers certainly have different dynamics, the overall principles of SaaS do not somehow change each of these markets.

If you look at onboarding, for example, B2B and B2C customers may have different onboarding experiences. It's true that B2C systems may focus more on a self-service onboarding flow (although B2B system may support this as well).

While there may be differences in how the onboarding is offered to customers, the fundamental underlying values for onboarding are mostly the same. Even if your B2B solution relies on an internal onboarding process, we would still expect that process to be as frictionless and automated as possible. Being B2B doesn't somehow mean that we're changing our expectations around time to value for our customers.

Conclusion

The goal of this document is to outline fundamental SaaS architecture concepts, providing some clarification around the models and terminology that are used to characterize SaaS patterns and strategies. The hope is that this will equip organizations with a clearer view of the overall SaaS landscape.

Much of what's covered here focused on what it means to be SaaS, with significant emphasis on the creating an environment that lets you manage and operate all of your SaaS tenants through a unified experience. This connects to the core idea that SaaS is first and foremost a business model. The SaaS architecture you build is meant to promote these foundational business goals.

Further reading

There are a number of resources that go into greater detail around SaaS architecture patterns that conform to the patterns described here.

For additional information, see:

- SaaS Tenant Isolation Strategies (AWS whitepaper)
- SaaS Storage Strategies (AWS whitepaper)
- Well-Architected SaaS Lens (AWS whitepaper)

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
Initial publication	Whitepaper published.	August 3, 2022

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.

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.