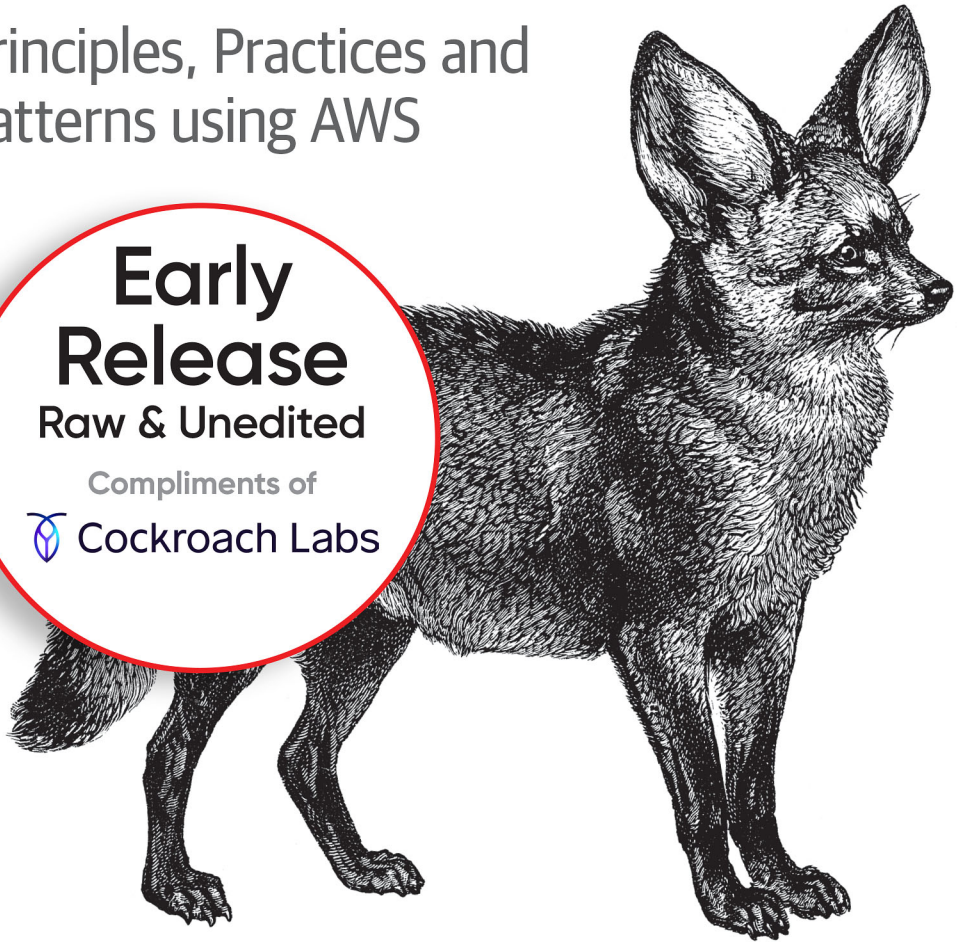
O'REILLY®

Building Multi-Tenant SaaS Architectures

Principles, Practices and Patterns using AWS



Early Release
Raw & Unedited

Compliments of
Cockroach Labs

Tod Golding

CockroachDB

Trusted by

Starburst

Materialize

rapidops

Five9

shopmonkey

OPTIMAL SYSTEMS
A KYOCERA GROUP COMPANY

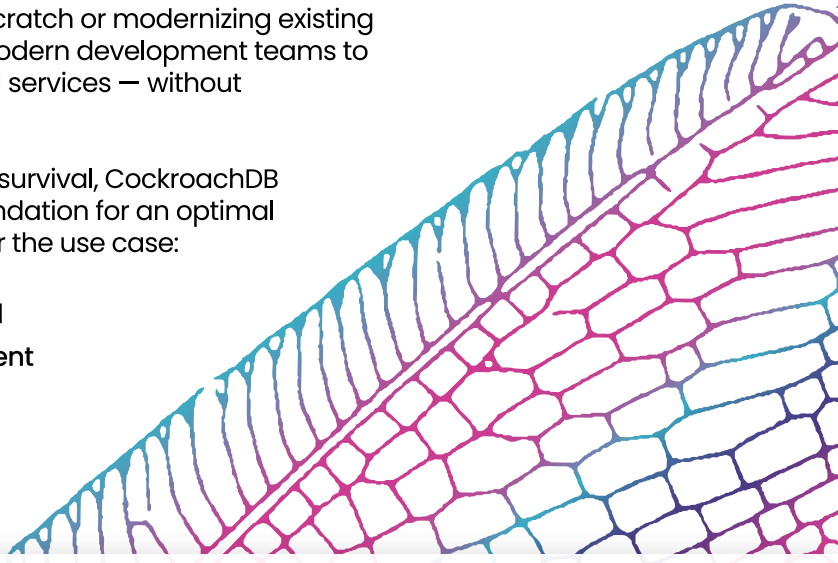
Up(time) in the cloud

Power mission-critical SaaS apps with the global database trusted by the world's most important businesses.

Whether you're building from scratch or modernizing existing apps, CockroachDB enables modern development teams to build highly available apps and services — without operational complexity.

Designed for speed, scale, and survival, CockroachDB provides a highly available foundation for an optimal end-user experience, no matter the use case:

- Customer system of record
- Identity access management
- Transactional analytics
- Metadata management
- GenAI SaaS



Scale horizontally across regions

With architecture that can easily scale as your business grows, you can deploy according to your business needs: multi-cloud, multi-region, and hybrid.

Deliver flawless user experiences

A cloud native resilient database that's always on. 99.99% availability guarantees that your applications are always on, too.

Innovate faster

CockroachDB's automated scale and built-in replication do the work for you. A reliable, scalable back-end service allows dev teams to easily prototype new ideas using familiar SQL.

https://www.cockroachlabs.com/product/

Building Multi-Tenant SaaS Architectures

Principles, Practices and Patterns Using AWS

This excerpt contains Chapters 1, 2, and 3 of *Building Multi-Tenant SaaS Architectures*. The complete book will be available on the O'Reilly Online Learning Platform and through other retailers when it is published.

Tod Golding

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Building Multi-Tenant SaaS Architectures
by Tod Golding

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisition Editor: Louise Corrigan
Development Editor: Melissa Potter
Production Editor: Gregory Hyman
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

March 2024: First Edition

Revision History for the Early Release

2023-01-24: First Release
2023-03-15: Second Release
2023-04-27: Third Release
2023-06-05: Fourth Release
2023-07-17: Fifth Release
2023-08-25: Sixth Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098140649* for release details.

978-1-098-14064-9

Table of Contents

Foreword

The critical advantage of multi-tenant SaaS architecture is its cost-effectiveness, native scalability, and ease of use for both the provider *and* the customer. For the provider, the cost saved by serving multiple customers with a single infrastructure can be significant indeed. The customer, freed from worrying about installation, maintenance, or upgrades, can start creating actual value right away.

Multi-tenancy is a powerful way to increase the efficiency and scalability of software applications while still providing a high level of customization and isolation for each tenant. However, building multi-tenant SaaS applications is a complex task. The benefits of more efficient use of resources are accompanied by the challenge of ensuring that each customer's data is isolated and secure — a challenge that becomes exponentially more difficult given the massive scale of global applications.

This challenge is compounded by the fact that, when composing your SaaS environment, you're not sticking to any one absolute definition of multi-tenancy. You're picking the combinations of shared and dedicated resources that best align with both your business needs and the technical requirements of your system, not to mention satisfying the needs of your customers.

Ultimately, these all converge at the database. In multi-tenancy, each tenant has access only to its own resources and data while the underlying infrastructure remains shared. This can include database tables, file systems, and other data storage mechanisms. Multi-tenant application architecture can be implemented using separate databases, where each tenant has its own separate database instance. This approach simplifies ensuring isolation and security but consumes more resources and is complex to manage. Scaling quickly becomes a problem because these challenges increase exponentially as the number of tenants grows.

A second multi-tenant SaaS architecture uses a single shared database with separate schemas. Each tenant has its own schema within a shared database to create isolation. This approach allows for efficient data sharing and resource utilization but requires a

complex database design and management — which CockroachDB is built to abstract away.

Whichever multi-tenancy design approach best fits your business — single database with separate schemas or multiple databases — CockroachDB's Distributed SQL capabilities provides an efficient, scalable and automatically adaptive DBMS platform on which to build your multi-tenant SaaS application. Overall load from services and tenants is automatically balanced across the cluster to avoid hotspots, and schema changes can be performed online with zero downtime.

CockroachDB, just based on configuration, can serve customers as a single cloud DBaaS for globally distributed workloads running in your own multi-tenant application. Alternatively, the same single database can be consumed in a truly shared multi-tenant serverless footprint (with no noisy neighbor problems). No matter how CockroachDB is implemented in your multi-tenant SaaS architecture, it guarantees atomicity, consistency, isolation, and durability in all transactions. And does this with built-in scalability, all in a single, highly-available database.

Overall, multi-tenancy has become an essential aspect of cloud computing and SaaS applications. It is crucial to designing and developing distributed applications that are performant, efficient, and cost-effective to deploy. As a database designed for cloud native elastic scale and resilience in the face of disasters both trivial and tremendous, CockroachDB is proud to sponsor three chapters of *Building Multi-Tenant SaaS Architectures* ("The SaaS Mindset", "Multi-Tenant Architecture Fundamentals", and "Multi-Tenant Deployment Models"). This is foundational knowledge for evolving a multi-tenant SaaS mindset — the first step in creating a multi-tenant application architected with the strategies and patterns that best align with the realities of your business.

CHAPTER 1
The SaaS Mindset

I've worked with a number of teams that were building software-as-a-service (SaaS) solutions. When I sit down with them to map out their path to SaaS, they tend to start out with what seems like a reasonable, high-level view of what it means to be SaaS. However, as I go a layer deeper and get into the details of their solution, I often discover significant variations in their vision. Imagine, for example, someone telling you they want to construct a building. While we all have some notion of a building having walls, windows, and doors, the actual nature of these structures could vary wildly. Some teams might be envisioning a skyscraper and others might be building a house.

It's kind of natural for there to be confusion around SaaS. As is the case in all technology realms, the SaaS universe has been continually evolving. The emergence of the cloud, shifting customer needs, and the economics of the software domain are in constant motion. How we defined SaaS yesterday may not be the way we'll define it today. The other part of the challenge here is that the scope of SaaS goes well beyond the technical. It is, in many respects, a mindset that spans all the dimensions of a SaaS provider's organization.

With that in mind, I thought the natural place to start this journey was by bringing more clarity to how I define SaaS and how I think this definition shapes your approach to architecting, designing, and building a SaaS solution. The goal in this chapter is to build a foundational mental model that will, ideally, reduce some of the confusion about what it means to be SaaS. We'll move beyond some of the vague notions of SaaS and, at least for the scope of this book, attach more concrete guiding principles to the definition of SaaS that will shape the strategies that we'll explore in the coming chapters.

To get there, we'll need to look at the forces that motivated the move to SaaS and see how these forces directly influenced the resulting architecture models. Following this evolution will provide a more concrete view into the foundational principles that are

used to create a SaaS solution that realizes the full value proposition of SaaS, blending the technical and business parameters that are at the core of developing modern SaaS environments.

While you may feel comfortable with what SaaS means to you, it's possible that the foundational concepts we'll explore here might challenge your view of SaaS and the terminology we use to describe SaaS environments. So, while it may be tempting to treat this chapter as optional, I would say that it may be one of the most important chapters in the book. It's not just an introduction, it's about creating a common vocabulary and mental model that will be woven into the architecture, coding, and implementation strategies that we'll be covering throughout this book.

The Classic Software Model

Before we can dig into the defining SaaS, we need to first understand where this journey started and the factors that have driven the momentum of the SaaS delivery model. Let's start by looking at how software was traditionally built, operated, and managed. These pre-SaaS systems were typically delivered in an "installed software" model where companies were hyper-focused on the features and functions of their offerings.

In this model, software was sold to a customer and that customer often assumed responsibility for installing the system. They might install it in some vendor-provided environment or they might install it on self-hosted infrastructure. The acquisition of these offerings would, in some cases, be packaged with professional services teams that could oversee the installation, customization, and configuration of the customer's environment.

Figure 1-1 provides a conceptual view of the footprint of the traditional software delivery model.

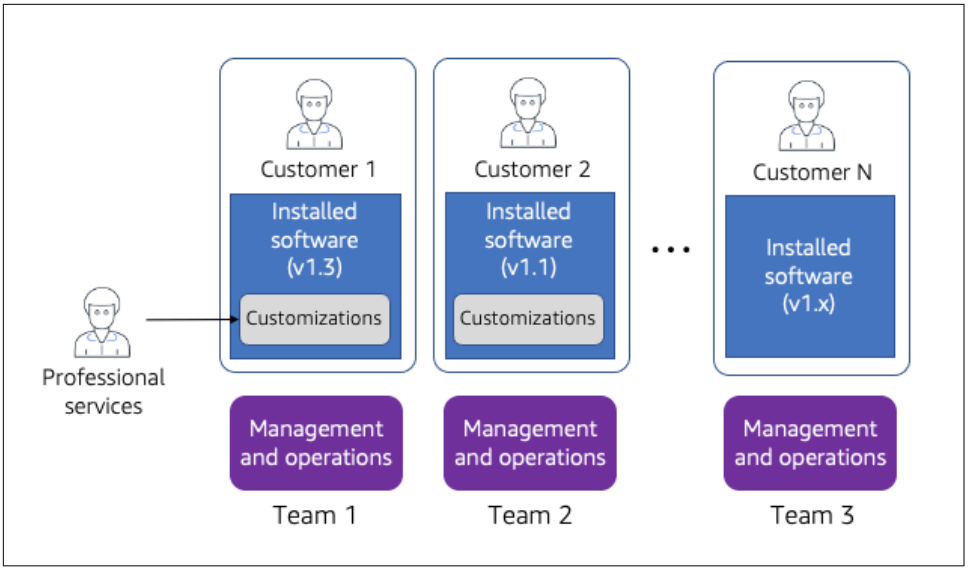


Figure 1-1. The installed software model

Here, you'll see a representation of multiple customer environments. We have Customers 1 and 2 that have installed and are running specific versions of the software provider's product. As part of their onboarding, they also required one-off customizations to the product that were addressed by the provider's professional services team. We also have other customers that may be running different versions of our product that may or may not have any customizations.

As each new customer is onboarded here, the provider's operations organization may need to create focused teams that can support the day-to-day needs of these customer installed and customized environments. These teams might be dedicated to an individual customer or support a cross-section of customers.

This classic mode of software delivery is a much more sales driven model where the business focuses on acquiring customers and hands them off to technology teams to address the specific needs of each incoming customer. Here, landing the deal often takes precedence over the need for agility, scale, and operational efficiency. These solutions are also frequently sold with long-term contracts that limit a customer's ability to easily move to any other vendor's offering.

The distributed and varying nature of these customer environments often slowed the release and adoption of new features. Customers tend to have control in these settings, often dictating how and when they might upgrade to a new version. The complexity of testing and deploying these environments could also become unwieldy, pushing vendors toward quarterly or semi-annual releases.

The Natural Challenges of the Classic Model

To be completely fair, building and delivering software in the model described above is and will continue to be a perfectly valid approach for some businesses. The legacy, compliance, and business realities of any given domain might align well to this model.

However, for many, this mode of software delivery introduced a number of challenges. At its core, this approach focused more on being able to sell customers whatever they needed in exchange for trade offs around scale, agility, and cost/operational efficiency.

On the surface, these tradeoffs may not seem all that significant. If you have a limited number of customers and you're only landing a few a year, this model could be adequate. You would still have inefficiencies, but they would be far less prominent. Consider, however, a scenario where you have a significant installed base and are looking to grow your business rapidly. In that mode, the pain points of this approach begin to represent a real problem for many software vendors.

Operational and cost efficiencies are often amongst the first areas where companies using this model start to feel the pain. The incremental overhead of supporting each new customer here begins to have real impacts on the business, eroding margins and continually adding complexity to the operational profile of the business. Each new customer could require more support teams, more infrastructure, and more effort to manage the one-off variations that accompany each customer installation. In some cases, companies actually reach a point where they'll intentionally slow their growth because of the operational burdens of this model.

The bigger issue here, though, is how this model impacts agility, competition, growth, and innovation. By its very nature, this model is anything but nimble. Allowing customers to manage their own environments, supporting separate versions for each customer, enabling one-off customization–these are all areas that undermine speed and agility. Imagine what it would mean to roll out a new feature in these environments. The time between having the idea for a feature, iterating on its development, and getting it in front of all your customers is often a slow and deliberate process. By the time a new feature arrives, the customer and market needs may have already shifted. This also can impact the competitive footprint of these companies, limiting their ability to rapidly react to emerging solutions that are built around a lower friction model.

While the operational and development footprint were becoming harder to scale, the needs and expectations of customers were also shifting. Customers were less worried about their ability to manage/control the environment where their software was running and more interested in maximizing the value they were extracting from these solutions. They demanded lower friction experiences that would be continually inno-

vating to meet their needs, giving them more freedom to move between solutions based on the evolving needs of their business.

Customers were also more drawn to pricing models that better aligned with their value and consumption profile. In some cases, they were looking for the flexibility of subscription and/or pay-as-you-go pricing models.

You can see the natural tension that's at play here. For many, the classic delivery model simply didn't align well with the shifting market and customer demands. The emergence of the cloud also played a key role here. The cloud model fundamentally altered the way companies looked at hosting, managing, and operating their software. The pay-as-you-go nature and operational model of the cloud had companies looking for ways to take advantage of the economies of scale that were baked into the cloud experience. Together, these forces were motivating software providers to consider new business and technology models.

The Move to Shared Infrastructure

By now, the basic challenges of the traditional model should be clear. While some organizations were struggling with this model, others already understood this approach would simply not scale economically or operationally. Larger business-to-consumer (B2C) software companies, for example, knew that supporting thousands or even millions of customers in a one-off model simply wouldn't work.

These B2C organizations really represented the early days of SaaS, laying the groundwork for future SaaS evolution. Achieving scale for these organizations was, from the outset, about building systems from the ground up that could support the massive scale of the B2C universe. They thrived based on their ability to operate in a model where all customers were presented with a single, unified experience.

This shift to sharing infrastructure amongst customers opened all new opportunities for software providers. To better understand this, Figure 1-2 provides a conceptual view of how applications can share infrastructure in a SaaS model.

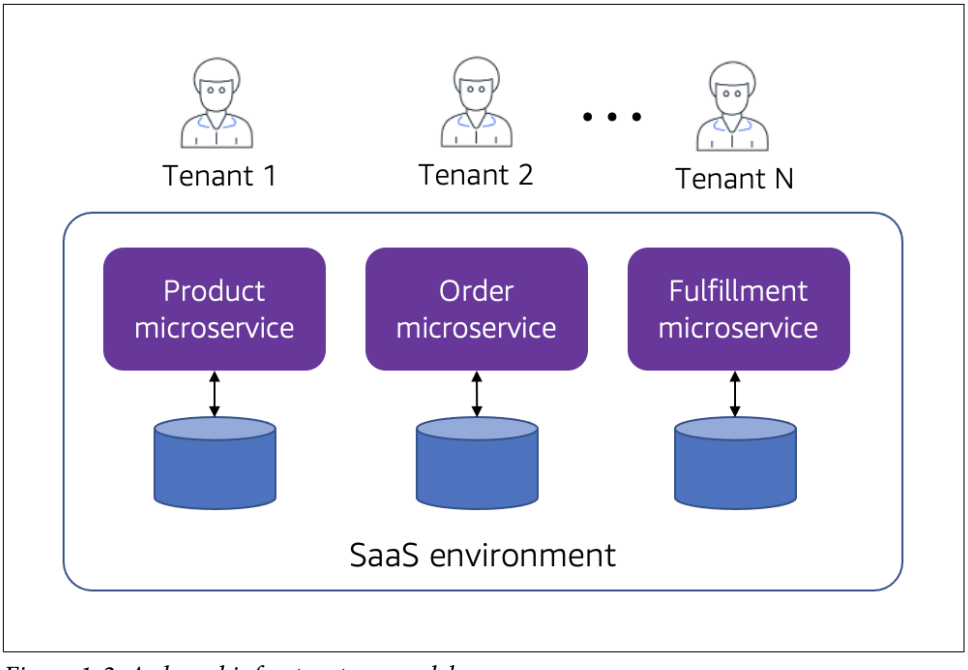


Figure 1-2. A shared infrastructure model

In Figure 1-2 you'll see a simplified view of the traditional notion of SaaS. You'll notice that we've completely moved away from the distributed, one-off, custom nature of the classic model we saw in Figure 1-1. In this approach, you'll see that we have a single SaaS environment with a collection of infrastructure. In this example, I happened to show microservices and their corresponding storage. A more complete example would show all the elements of your system's application architecture.

If we were to take a peek inside one of these microservices at run-time, we could potentially see any number of tenants (aka customers) consuming the system's shared infrastructure. For example, if we took three snapshots of the Product microservice at three different time intervals, we might see something resembling the image in Figure 1-3.

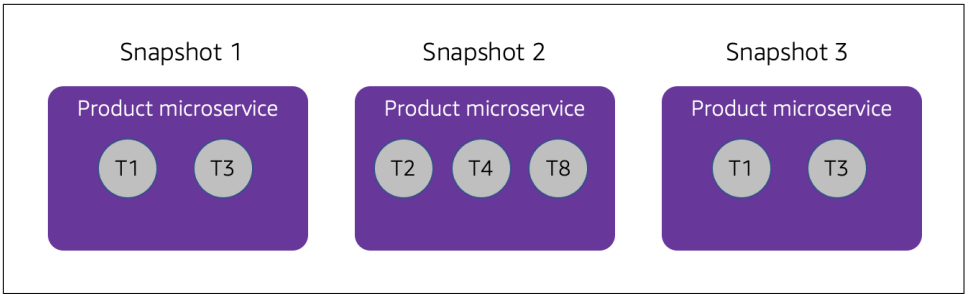


Figure 1-3. Shared microservices

In snapshot 1, our product microservice has two tenants consuming our service (tenant1 and tenant2). Another snapshot could have another collection of tenants that are consuming our service. The point here is that the resource no longer belongs to any one consumer, it is a shared resource that is consumed by any tenant of our system.

This shift to using shared infrastructure also meant we needed a new way to describe the consumers of our software. Before, when every consumer had its own dedicated infrastructure, it was easy to continue to use the term "customer". However, in the shared infrastructure model of SaaS, you'll see that we describe the consumers of our environment as "tenants".

It's essential that you have a solid understanding of this concept since it will span almost every topic we cover in this book. The notion of tenancy maps very well to the idea of an apartment complex where you own a building and you rent it out to different tenants. In this model, the building correlates to the shared infrastructure of your solution and the tenants represent the different occupants of your apartments. These tenants of your building consume shared building resources (power, water, and so on). As the building owner, you manage and operate the overall building and different tenants will come and go.

You can see how this term better fits the SaaS model where we are building a service that runs on shared infrastructure that can accommodate any number of tenants. Yes, tenants are still customers, but the term "tenant" lets us better characterize how they land in a SaaS environment.

The Advantage of Shared Infrastructure

So, we can see how SaaS moves us more toward a unified infrastructure footprint. If we contrast this with the one-off, installed software model we covered above, you can see how this approach enables us to overcome a number of challenges.

Now that we have a single environment for *all* customers, we can manage, operate, and deploy all of our customers through a single pane of glass. Imagine, for example, what it would look like to deploy an update in the shared infrastructure model. We

would simply deploy our new version to our unified SaaS environment and all of our tenants would immediately have access to our new features. Gone is the idea of separately managed and operated versions. With SaaS, every customer is running the same version of your application. Yes, there may be customizations within that experience that are enabled/disabled for different personas, but they are all part of a single application experience.

This notion of having all tenants running the same version of your offering represents a common litmus test for SaaS environments. It is foundational to enabling many of the business benefits that are at the core of adopting a SaaS delivery model.

You can imagine the operational benefits that come with this model as well. With all tenants in one environment, we can manage, operate, and support our tenants through a common experience. Our tools can give us insights into how tenants are consuming our system and we can create policies and strategies to manage them collectively. This brings all new levels of efficiency to our operational model, reducing the complexity and overall footprint of the operational team. SaaS organizations take great pride in their ability to manage and operate a large collection of tenants with modestly sized operational teams.

This focus on operational efficiency also directly feeds the broader agility story. Freed from the burden of one-off, custom versions, SaaS teams will often embrace their agility and use it as the engine of constant innovation. These teams are continually releasing new features, gathering more immediate customer feedback, and evolving their systems in real-time. You can imagine how this model will directly impact customer loyalty and adoption.

The responsiveness and agility of this SaaS model often translates into competitive advantages. Teams will use this agility to reach new market segments, pivoting in real-time based on competitive and general market dynamics.

The shared infrastructure model of SaaS also has natural cost benefits. When you have shared infrastructure and you can scale that infrastructure based on the actual consumption patterns of your customers, this can have a significant impact on the margins of your business. In an ideal SaaS infrastructure model, your system would essentially only consume the infrastructure that is needed to support the current load of your tenants. This can represent a real game-changer for some organizations, allowing them to take on new tenants at any pace knowing that each tenant's infrastructure costs will only expand based on their actual consumption activities. The elastic, pay-as-you-go nature of cloud infrastructure aligns nicely with this model, supporting the pricing and scaling models that fit naturally with the varying workloads and consumption profiles of SaaS environments.

Thinking Beyond Infrastructure

While looking at SaaS through the lens of shared infrastructure makes it easier to understand the value of SaaS, the reality is that SaaS is much more than shared infrastructure. In fact, as we move forward, we'll see that shared infrastructure is just one dimension of the SaaS story. There are economies of scale and agility can be achieved with SaaS–with or without shared infrastructure.

In reality, we'll eventually see that there are actually many ways to deploy and implement your SaaS application architecture. The efficiencies that are attributed to SaaS can certainly be maximized by sharing infrastructure. However, efficiency starts with surrounding your application with constructs that can streamline the customer experience and the management/operational experience of your SaaS environment. It's these constructs that–in concert with your SaaS application architecture–enable your SaaS business to realize its fundamental operational, growth, and agility goals.

To better understand this concept, let's look at these additional SaaS constructs. The diagram in Figure 1-4 provides a highly simplified conceptual view of these common SaaS services.

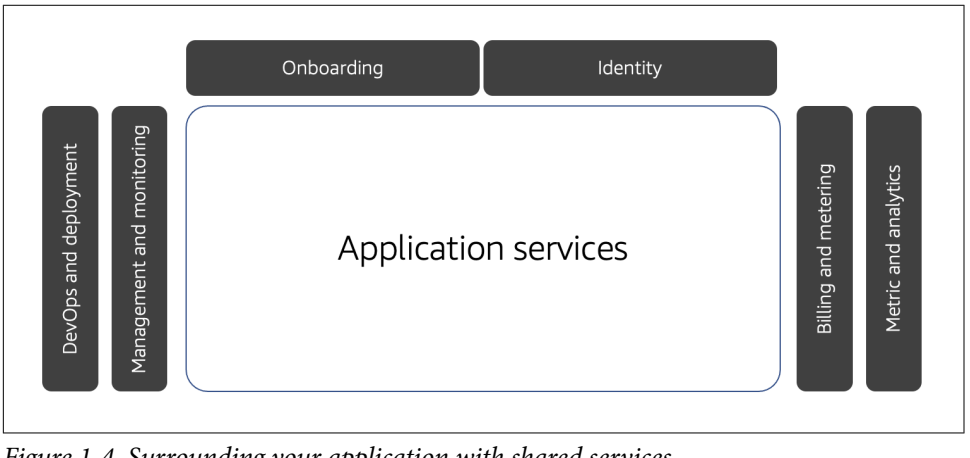


Figure 1-4. Surrounding your application with shared services

At the center of this diagram you'll see a placeholder for application services. This is where the various components of your SaaS application are deployed. Around these services, though, are a set of services that are needed to support the needs of our overall SaaS environment. At the top, I've highlighted the onboarding and identity services, which provide all the functionality to introduce a new tenant into your system. On the left, you'll see the placeholders for the common SaaS deployment and management functionality. And, on the right, you'll see fundamental concepts like billing, metering, metrics, and analytics.

Now, for many SaaS builders, it's tempting to view these additional services as secondary components that are needed by your application. In fact, I have seen teams that will defer the introduction of these services, putting all their initial energy into supporting tenancy in their application services.

In reality, while getting the application services right is certainly an important part of your SaaS model, the success of your SaaS business will be heavily influenced by the capabilities of these surrounding services. These services are at the core of enabling much of the operational efficiency, growth, innovation, and agility goals that are motivating companies to adopt a SaaS model. So, these components–which are common to *all* SaaS environments–must be put front and center when you are building your SaaS solution. This is why I have always encouraged SaaS teams to start their SaaS development at this outer edge, defining how they will automate the introduction of tenants, how they'll connect tenants to users, how they'll manage your tenant infrastructure, and a host of other considerations that we'll be covering throughout this book. It's these building blocks–which have nothing to do with the functionality of your application–that are going to have a significant influence on the SaaS footprint of your architecture, design, code, and business.

So, if we turn our attention back to the diagram in Figure 1-4, we can see this big hole in the middle that represents where we'll ultimately place our application services. The key takeaway here is that, no matter how we design and build what lands in that space, you'll still need some flavor of these core shared services as part of every SaaS architecture you build.

Re-Defining Multi-Tenancy

Up to this point, I've avoided introducing the idea of multi-tenancy. It's a word that is used heavily in the SaaS space and will appear all throughout the remainder of this book. However, it's a term that we have to wander into gracefully. The idea of multi-tenancy comes with lots of attached baggage and, before sorting it out, I wanted to create some foundation for the fundamentals that have driven companies toward the adoption of the SaaS delivery model. The other part of the challenge here is that the notion of multi-tenancy–as we'll define it in this book–will move beyond some of the traditional definitions that are typically attached to this term.

For years, in many circles, the term multi-tenant was used to convey the idea that some resource was being shared by multiple tenants. This could apply in many contexts. We could say that some piece of cloud infrastructure, for example, could be deemed multi-tenant because it was allowing tenants to share some resource under the hood. In reality, many services running in the cloud may be running in a multi-tenant model to achieve their economies of scale. As a cloud consumer, this may be happening entirely outside of your view. Even outside the cloud, teams could build solutions where compute, databases, and other resources could be shared amongst

customers. This created a very tight connection between multi-tenancy and the idea of a shared resource. In fact, in this context, this is a perfectly valid notion of multi-tenancy.

Now, as we start thinking about SaaS environments, it's entirely natural for us to bring the mapping of multi-tenancy with us. Afterall, SaaS environments do share infrastructure and that sharing of infrastructure is certainly valid to label as being multi-tenant.

To better illustrate this point, let's look at a sample SaaS model that brings together the concepts that we've been discussing in this chapter. The image in Figure 1-5 provides a view of a sample multi-tenant SaaS environment.

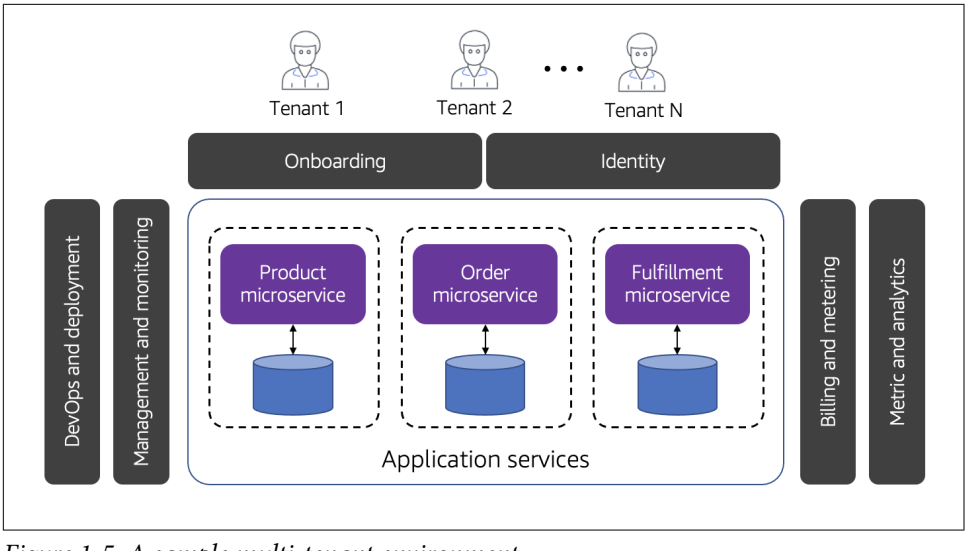


Figure 1-5. A sample multi-tenant environment

Here we have landed the shared infrastructure of our application services inside surrounding services that are used to introduce tenancy, manage, and operate our SaaS environment. Assuming that all of our tenants are sharing their infrastructure (compute, storage, and so on), then this would fit with the classic definition of multi-tenancy. And, to be fair, it would not be uncommon for SaaS providers to define and deliver their solution following this pattern.

The challenge here is that SaaS environments don't exclusively conform to this model. Suppose, for example, I create a SaaS environment that looks like the drawing in Figure 1-6.

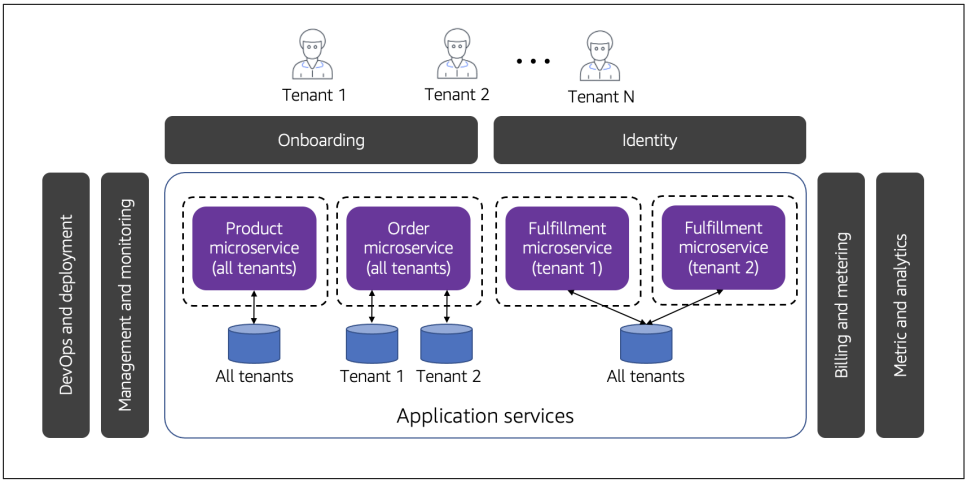


Figure 1-6. Multi-tenancy with shared and dedicated resources

Here you'll see that we've morphed the footprint of some of our application microservices. The Product microservice is unchanged. Its compute and storage infrastructure is still shared by all tenants. However, as we move to the Order microservice, you'll see that we've mixed things up a bit. Our domain, performance, and/ or our security requirements may have required that we separate out the storage for each tenant. So, here the compute of our Order microservice is still shared, but we have separate databases for each tenant.

Finally, our Fulfillment microservice has also shifted. Here, our requirements pushed us toward a model where each tenant is running dedicated compute resources. In this case, though, the database is still shared by all tenants.

This architecture has certainly added a new wrinkle to our notion of multi-tenancy. If we're sticking to the purest definition of multi-tenancy, we wouldn't really be able to say everything running here conforms to the original definition of multi-tenancy. The storage of the Order service, for example, is not sharing any infrastructure between tenants. The compute of our Fulfillment microservices is also not shared here, but the database for this service is shared by all tenants.

Blurring these multi-tenant lines is common in the SaaS universe. When you're composing your SaaS environment, you're not sticking to any one absolute definition of multi-tenancy. You're picking the combinations of shared and dedicated resources that best align with the business and technical requirements of your system. This is all part of optimizing the footprint of your SaaS architecture around the needs of the business.

Even though the resources here are not shared by all tenants, the fundamentals of the SaaS principles we outlined earlier are still valid. For example, this environment

would not change our application deployment approach. All tenants in this environment would still be running the same version of the product. Also, the environment is still being onboarded, operated, and managed by the same set of shared services we relied on in our prior example. This means that we're still extracting much of the operational efficiency and agility from this environment that would have been achieved in a fully shared infrastructure (with some caveats).

To drive this point home, let's look at a more extreme example. Suppose we have a SaaS architecture that resembles the model shown in Figure 1-7. In this example, the domain, market, and/or legacy requirements have required us to have all compute and storage running in a dedicated model where each tenant has a completely separate set of infrastructure resources.

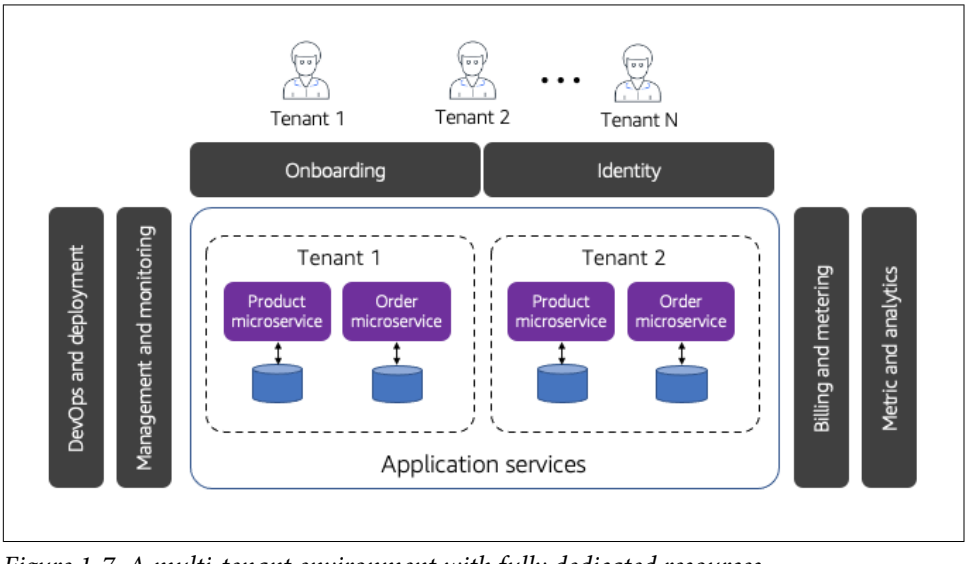


Figure 1-7. A multi-tenant environment with fully dedicated resources

While our tenants aren't sharing infrastructure in this model, you'll see that they continue to be onboarded, managed, and operated through the same set of shared services that have spanned all of the examples we've outlined here. That means that all tenants are still running the same version of the software and they are still being managed and operated collectively.

This may seem like an unlikely scenario. However, in the wild, SaaS providers may have any number of different factors that might require them to operate in this model. Migrating SaaS providers often employ this model as a first stepping stone to SaaS. Other industries may have such extreme isolation requirements that they're not allowed to share infrastructure. There's a long list of factors that could legitimately land a SaaS provider in this model.

So, given this backdrop, it seems fair to ask ourselves how we want to define multi-tenancy in the context of a SaaS environment. Using the literal shared infrastructure definition of multi-tenancy doesn't seem to map well to the various models that can be used to deploy tenant infrastructure. Instead, these variations in SaaS models seem to demand that we evolve our definition of what it means to be multi-tenant.

For the scope of this book, at least, the term multi-tenant will definitely be extended to accommodate the realities I've outlined here. As we move forward, multi-tenant will refer to any environment that onboards, deploys, manages, and operates tenants through a single pane of glass. The sharedness of any infrastructure will have no correlation to the term multi-tenancy.

In the ensuing chapters, we'll introduce new terminology that will help us overcome some of the ambiguity that is attached to multi-tenancy.

Avoiding the Single-Tenant Term

Generally, whenever we refer to something as multi-tenant, there's a natural tendency to assume there must be some corresponding notion of what it means to be single-tenant. The idea of single tenancy seems to get mapped to those environments where no infrastructure is shared by tenants.

While I follow the logic of this approach, this term doesn't really seem to fit anywhere in the model of SaaS that I have outlined here. If you look back to Figure 1-7 where our solution had no shared infrastructure, I also noted that we would still label this a multi-tenant environment since all tenants were still running the same version and being managed/operated collectively. Labeling this a single-tenant would undermine the idea that we aren't somehow realizing the benefits of the SaaS model.

With this in mind, you'll find that the term single-tenant will not be used at any point beyond this chapter. Every design and architecture we discuss will still be deemed a multi-tenant architecture. Instead, we'll attach new terms to describe the various deployment models that will still allow us to convey how/if infrastructure is being shared within a given SaaS environment. The general goal here is to disconnect the concept of multi-tenancy from the sharing of infrastructure and use it as a broader term to characterize any environment that is built, deployed, managed, and operated in a SaaS model.

This is less about what SaaS is or is not and more about establishing a vocabulary that aligns better with the concepts we'll be exploring throughout this book.

Where are the boundaries of SaaS?

We've laid a foundation here for what it means to be SaaS, but there are lots of nuances that we haven't really talked about. For example, suppose your SaaS application

requires portions of the system to be deployed in some external location. Or, imagine scenarios where your application has dependencies on other vendor's solutions. Maybe you are using a third-party billing system or your data must reside in another environment. There are any number of different reasons why you may need to have parts of your overall SaaS environment hosted somewhere that may not be entirely under your control.

So, how would this more distributed footprint fit with the idea of having a single, unified experience for all of your tenants? Afterall, having full control over all the moving parts of your system certainly maximizes your ability to innovate and move quickly. At the same time, it's impractical to think that some SaaS providers won't face domain and technology realities that require them to support externally hosted components/tools/technologies.

This is where we don't want to be too extreme with our definition of SaaS. To me, the boundary is more around how these external dependencies are configured, managed, and operated. If their presence is entirely hidden from your tenants and they are still managed and operated through your centralized experience, this is still SaaS to me. It may introduce new complexities, but it doesn't change the spirit of the SaaS model we're trying to build.

Where this gets more interesting is when SaaS providers have reliance on external resources that are in the direct view of their tenants. If, for example, my SaaS solution stores data in some tenant-hosted database, that's where things get more dicey. Now, you may have a dependency on infrastructure that is not entirely under your control. Updating this database, changing its schema, managing its health–these get more complicated in this model. This is where we start to ask questions about whether this external resource is breaking the third-wall of SaaS, exposing tenants to infrastructure and creating expectations/dependencies that undermine the agility, operations, and innovation of your SaaS environment.

My general rule of thumb here (with some exceptions) is that we're providing a service experience. In a service model, our tenant's view is limited to the surface of our service. The tools, technologies, and resources that are used to bring that service to life should be entirely hidden from our tenants. In many respects, this is the hard barrier that prevents our system from falling back into patterns that might lead to one-off dependencies and variations.

The Managed Service Provider Model

There's one last wrinkle that we need to address as we try to refine our view of what it means to be a multi-tenant SaaS environment. Some organizations have opted into what's referred to as a Managed Service Provider (MSP) model. In some cases, they'll categorize MSP as a variant of SaaS. This certainly has created some confusion in the SaaS domain. To better understand the challenges here, let's start by looking at an

MSP environment and see how/where it fits in this discussion. Figure 1-8 provides a conceptual view of an MSP environment.

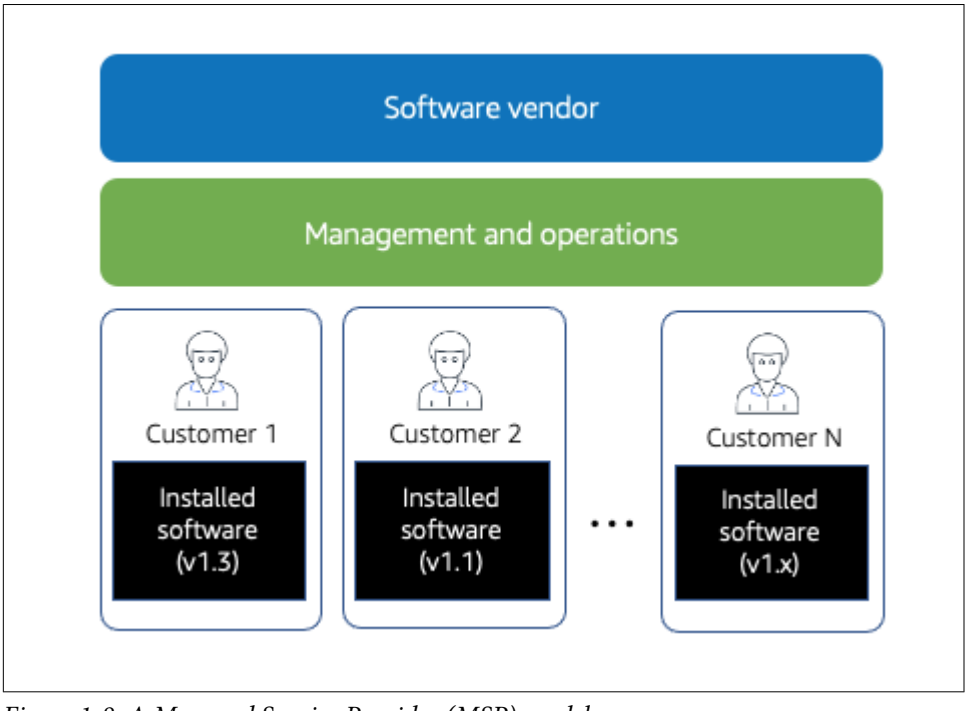


Figure 1-8. A Managed Service Provider (MSP) model

This model definitely resembles the classic installed software model that we outlined earlier. At the bottom of this diagram, you'll see a collection of customers that are running various versions of a software vendor's product. Each one of these customers will be running in its own infrastructure/environment.

With MSP, though, we'll try to get efficiencies and economies of scale out of moving the operations to a centralized team/entity. This is the service that these MSPs provide. They often own responsibility for installing, managing, and supporting each of these customers, attempting to extract some scale and efficiency out of tooling and mechanisms that they use to operate these customer environments.

I've also represented the software vendor at the top of the diagram. This is here to convey the idea that the software provider may have third-party relationships with one or more MSPs that are managing their customer environments.

You can see how some might equate the MSP model to SaaS. Afterall, it does seem to be trying to provide a unified managed and operations experience for all customers. However, if you look back at the principles that we used to describe SaaS, you can see where there are substantial gaps between MSP and SaaS. One of the biggest differ-

ences here is that customers are being allowed to run separate versions. So, while there may be some attempts to centralize management and operations, the MSP is going to have to have one-off variations in their operational experience to support the different footprints of each customer environment. This may require dedicated teams or, at a minimum, it will mean having teams that can deal with the complexities of supporting the unique needs of each customer. Again, MSP still adds lots of value here and certainly creates efficiencies, but it's definitely different than having a single pane of glass that gets its efficiencies from having customers run a single version of a product and, in many cases, getting efficiencies from sharing some or all of their infrastructure. At some level in the MSP model, you're likely to still inherit aspects of the pain that comes with one-off customer variations. MSPs can introduce some measures to offset some of the challenges here, but they'll still face the operational and agility complexities that come with supporting unique, one-off needs of separate customer environments.

The other difference here relates more to how SaaS teams are structured and operated. Generally, in a SaaS organization, we're attempting to avoid drawing hard lines between operations teams and the rest of the organization. We want operations, architects, product owners, and the various roles of our team working closely together to continually evaluate and refine the service experience of their offering.

This typically means that these teams are tightly connected. They're equally invested in understanding how their tenants are consuming their systems, how their imposing load, how they're onboarding, and a host of other key insights. SaaS businesses want and need to have their fingers on the pulse of their systems. This is core to driving the success of the business and being connected more directly to the overall tenant experience. So, while this is a less concrete boundary, it still represents an important difference between SaaS and MSP.

Now, it's important to note that MSP is an entirely valid model. It often represents a good fit for some software providers. MSP can even be a stepping stone for some SaaS providers, providing access to some efficiencies while the team continues to push forward toward its SaaS delivery model. The key here is that we have a clear understanding of the boundaries between SaaS and MSP and avoid viewing SaaS and MSP as somehow being synonymous.

At its Core, SaaS is a Business Model

By now you should have a better sense of how we characterize what it means to be SaaS. It should be clear that SaaS is very much about creating a technology, business, and operational culture that is focused squarely on driving a distinct set of business outcomes. So, while it's tempting to think about SaaS through the lens of technology patterns and strategies, you should really be viewing SaaS more as a business model.

To better understand this mindset, think about how adopting SaaS impacts the business of a SaaS provider. It directly influences and shapes how teams build, manage, operate, market, support, and sell their offerings. The principles of SaaS are ultimately woven into the culture of SaaS companies, blurring the line between the business and technology domains. With SaaS, the business strategy is focused on creating a service that can enable the business to react to current and emerging market needs without losing momentum or compromising growth.

Yes, features and functions are still important to SaaS companies. However, in a SaaS company, the features and functions are never introduced at the expense of agility and operational efficiency. When you're offering a multi-tenant SaaS solution, the needs of the many should always outweigh the needs of the few. Gone are the days of chasing one-off opportunities that require dedicated, one-off support at the expense of long-term success of the service.

This shift in mindset influences almost every role in a SaaS company. The role of a product owner, for example, changes significantly. Product owners must expand their view and consider operational attributes as part of constructing their backlog. Onboarding experience, time to value, agility–these are all examples of items that must be on the radar of the product owner. They must prioritize and value these operational attributes that are essential to creating a successful SaaS business. Architects, engineers, and QA members are equally influenced by this shift. They must now think more about how the solution they're designing, building, and testing will achieve the more dynamic needs of their service experience. How your SaaS offering is marketed, priced, sold, and supported also changes. This theme of new and overlapping responsibilities is common to most SaaS organizations.

So, the question is: what are the core principles that shape and guide the business model of SaaS companies? While there might be some debate about the answer to the question, there are some key themes that seem to drive SaaS business strategies. The following outlines these key SaaS business objectives:

Agility
: This term is often overloaded in the software domain. At the same time, in the SaaS universe, it is often viewed as one of the core pillars and motivating factors of the SaaS business. So many organizations that are moving to SaaS are doing so because they've become operationally crippled by their current model. Adopting SaaS is about moving to a culture and mindset that puts emphasis on speed and efficiency. Releasing new versions, reacting to market dynamics, targeting new customer segments, changing pricing models–these are amongst a long list of benefits that companies expect to extract from adopting a SaaS model. How your service is designed, how it's operated, and how it's sold are all shaped by a desire to maximize agility. A multi-tenant offering that reduced costs without realizing

agility would certainly miss the broader value proposition of what it means to be a SaaS company.

Operational Efficiency

SaaS, in many respects, is about scale. In a multi-tenant environment, we're highly focused on continually growing our base of customers without requiring any specialized resources or teams to support the addition of these new customers. With SaaS, you're essentially building an operational and technology footprint that can support continual and, ideally, rapid growth. Supporting this growth means investing in building an efficient operational footprint for your entire organization. I'll often ask SaaS companies what would happen if 1,000 new customers signed up for their service tomorrow. Some would welcome this and others cringe. This question often surfaces key questions about the operational efficiency of a SaaS company. It's important to note that operational efficiency is also about reacting and responding to customer needs. How quickly new features are released, how fast customers onboard, how quickly issues are addressed–these are all part of the operational efficiency story. Every part of the organization may play a part in building out an operationally efficient offering.

Innovation

With classic software models, teams can feel somewhat hand-cuffed by the realities of their environment. Customers may have one-off customizations, they may have a distributed operational model–there could be any number of factors that make it difficult for them to consider making any significant shifts in their approach. In a SaaS environment, where there's more emphasis on agility and putting customers in a unified environment, teams are freed up to consider exploring new, out-of-the-box ideas that could directly influence the growth and success of the business. In many respects, this represents the flywheel of SaaS. You invest in agility and operational efficiency and this promotes greater innovation. This is all part of the broader value proposition of the SaaS model.

Frictionless Onboarding

SaaS businesses must give careful consideration to how customers get introduced into their environments. If you are trying to remain as agile and operationally efficient as possible, you must also think about how customer onboarding can be streamlined. For some SaaS businesses, this will be achieved through a classic sign-up page where customers can complete the on-boarding process in an entirely self-service manner. In other environments, organizations may rely on an internal process that drives the onboarding process. The key here is that every SaaS business must be focused on creating an onboarding experience that removes friction and enables agility and operational efficiency. For some, this will be straightforward. For others, it may take more effort to re-think how the team builds, operates, and automates its onboarding experience.

Growth

Every organization is about growth. However, SaaS organizations typically have a different notion of growth. They are investing in a model and an organizational footprint that is built to thrive on growth. Imagine building this highly efficient car factory that optimized and automated every step in the construction process. Then, imagine only asking it to produce two cars a day. Sort of pointless. With SaaS, we're building out a business footprint that streamlines the entire process of acquiring, onboarding, supporting, and managing customers. A SaaS company makes this investment with the expectation that it will help support and fuel the growth machine that ultimately influences the margins and broader success of the business. So, when we talk about growth here, we're talking about achieving a level of acceleration that couldn't be achieved without the agility, operational efficiency, and innovation that 's part of SaaS. How much growth you're talking here is relative. For some, growth may be adding 100 new customers and for others it could mean adding 50,000 new customers. While this nature of your scale may vary, the goal of being growth-focused is equally essential to all SaaS businesses.

The items outlined here represent some of the core SaaS business principles. These are concepts that should be driven from the top down in a SaaS company where the leadership places clear emphasis on driving a business strategy that is focused on creating growth through investment in these agility, operational efficiency, and growth goals.

Certainly, technology will end up playing a key role in this business strategy. The difference here is that SaaS is not a technology first mindset. A SaaS architect doesn't design a multi-tenant architecture first then figure out how the business strategy layers on top of that. Instead, the business and technology work together to find the best intersection of business goals and multi-tenant strategies that will realize those strategies.

As we get further into architecture details, you'll see this theme is laced into every dimension of our architecture. As we look at topics like tenant isolation, data partitioning, and identity, for example, you'll see how each of these areas are directly influenced by a range of business model considerations.

Building a Service–Not a Product

Many software providers would view themselves as being in the business of creating products. And, in many respects, this aligns well with their business model. The mindset here is focused on a pattern where we build something, the customer acquires it, and it's, for the most part, theirs to use. There are plenty of permutations and nuances within this product-centric model, but they all gravitate toward a model that is focused on creating something more static and having customers buy it.

In this product-focused mindset, the emphasis is generally on defining the features and functions that will allow a software provider to close gaps and land new opportunities. Now, with SaaS, we shift from creating a product to creating a service. So, is this just terminology or does it have a meaningful impact on how we approach building a SaaS offering? It turns out, this is certainly more than a terminology shift.

When you offer software as a service, you think differently about what success looks like. Yes, your product needs to meet the functional needs of your customers. That dimension of the problem doesn't go away. As a service, though, you are much more focused on the broader customer experience across all dimensions of your business.

Let's look at an example that better highlights the differences between a service and a product. A restaurant provides a good backdrop for exploring these differences. When you go out to dinner, you're certainly looking forward to the food (the product in this example). However, the service is also a part of your experience. How fast you're greeted at the door, how soon the waiter comes to your table, how soon you get water, and how quickly your food arrives are all measures of your service experience. No matter how good the food is, your quality of service will have a lot to do with your overall impression of the restaurant.

Now, think about this through the lens of a SaaS offering. Your SaaS tenants will have similar service expectations. How easily they can onboard your solution, how long it takes to realize value, how quickly new features are released, how easily they can provide feedback, how frequently the system is down–they are all dimensions of a service that must be front-and-center for SaaS teams. Having a great product won't matter if the overall experience for customers does not meet their expectations.

This takes on extra meaning when software is delivered in a SaaS model, where the tenant's only view of your system is the surface of your SaaS solution. SaaS tenants have no visibility into the underlying elements of your system. They don't think about patches, updates, and infrastructure configuration. They only care that the service is providing the experience that lets them maximize the value of your solution.

In this service model, we also often see SaaS companies leveraging their operational agility to drive greater customer loyalty. These SaaS providers will get into a mode where they release new capabilities, respond to feedback, and morph their systems at a rapid pace. Seeing this constant and rapid innovation gives customers confidence that they will be benefactors of this constant evolution. In fact, this is often the tool that allows emerging SaaS companies to take business away from traditional non-SaaS market leaders. While some massive, established market leaders may have a much deeper feature set, their inability to rapidly react to market and customer needs can steer customers to more nimble SaaS-based offerings.

So, while this product vs. service concept may seem like I'm being a bit pedantic, to me it's an important distinction. It connects directly to this idea that SaaS is very

much a mindset that shapes how entire SaaS organizations approach their jobs and their customers. In fact, many SaaS organizations will adopt a series of metrics that measure their ability to meet their service centric goals. It may be tempting to view this as something that can be bolted onto your service at some future date. However, many successful SaaS organizations rely on these metrics as a key pillar of their SaaS business.

The B2B and B2C SaaS Story

The value of SaaS has a natural B2C mapping. Many of the highly visible examples of SaaS show up in the B2C model. The problem here is that some builders and organizations will presume that SaaS somehow only fits in the B2C space. I've seen technology teams and service providers suggest that their business-to-business (B2B) products can't or shouldn't be delivered in a SaaS model purely based on the fact that they are B2B.

For this book, I'll not be assuming anything about whether you're B2C or B2B. The practices, strategies, and patterns here are assumed to apply equally to these domains. Yes, there are areas where I might suggest that a given practice might work slightly differently for B2B or B2C. To me, these are mostly exceptions. The reality is, the goal and mindset of these domains is mostly universal. There are just nuances that might influence how these models will address the needs of their customers.

It is important, however, to note that the architecture patterns that are used in some B2C offerings are required to address unique scaling and availability challenges. If you're supporting millions of tenants and onboarding thousands of new tenants each day, the design and strategies you employ are going to be highly specialized. In fact, these B2C SaaS providers are often required to invent new breakthrough technologies that can address their massive scaling and operational challenges. While there's lessons to be learned from these models, the exotic and inventive strategies that are used in these environments are often well outside the needs of the average SaaS builder. Our focus will be more on those B2C and B2B environments that are leveraging the scale and availability of existing cloud technologies to support the scaling and availability needs of their environment. Where it makes sense, I'll highlight areas where you may need to explore more targeted scaling models to support loads that might exceed the natural scaling boundaries of existing tooling, service, and so on.

Defining SaaS

I've devoted the bulk of this chapter to bringing more clarity to the boundaries, scope, and nature of what it means to be SaaS. It only seems fair to take all the information we discussed here and attempt to provide an explicit definition of SaaS that, ideally, incorporates the concepts and principles that we have covered here. Here's the

definition I think best summarizes the view of SaaS I'll be using across the rest of this book:

SaaS is a business and software delivery model that enables organizations 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'll see here that this definition sticks to the theme of SaaS being a business model. There's no mention here of any technologies or architecture considerations. It's your job as a SaaS architect and builder to create the underlying patterns and strategies that enable the business to realize its objectives. While that may seem like the job of any architect, it should be clear that the unique blend of business and technology demands for SaaS environments will be infused directly into the design, architecture, and implementation of your SaaS solution.

Conclusion

This chapter was all about establishing the foundational elements of the SaaS mindset, providing you with a clearer view of what I mean when I talk about multi-tenancy and the core terms I use to describe a SaaS model. It should also make it clear that your job as a SaaS architect and builder goes well beyond the technology domain. Before you can choose any architecture for your SaaS system, you'll need to have a firm grasp on the nature of key insights from your business to know which strategies and patterns are going to best align with the realities of your business. This reality will become clearer as we get into the details of how you architect and design SaaS systems. The challenges and needs of SaaS architecture will require you to add all new dimensions to your toolbag. In some cases, you may actually need to be the evangelist of these concepts, driving the teams around you to think differently about how they approach their jobs.

While having a clear view of the SaaS mindset is essential, our goal in this book is to dig into the technical dimensions of SaaS. In the next chapter, we'll start to look at the various architectural constructs and concepts that are used when designing SaaS environments. Having a firm grip on these constructs will provide you with a foundation of terminology and concepts that will be essential as we move into more detailed architecture models and implementation strategies. It will also expose you to the range of fundamental considerations that are part of every SaaS architecture.

CHAPTER 2

Multi-tenant Architecture Fundamentals

As you progress through this book, you'll realize that SaaS architecture comes in many shapes and sizes. There are countless permutations of architecture patterns and strategies that are composed to create the SaaS architecture that best aligns with the domain, compliance, and business realities of a SaaS company.

There are, however, some core themes that span all SaaS architectures. In this chapter, we'll start our architecture journey by building a foundation of architecture terminology and constructs that will be used as we explore specific architecture models. In many respects, the concepts we'll cover here will represent your playbook for building SaaS systems, providing you with a core set of topics and questions you'll need to address as part of any SaaS environment.

Getting a firm grasp on these core architecture principles is key to developing a solid understanding of the elements of SaaS environments, equipping you with a sense of the landscape of SaaS architecture challenges. The goal here is to bring clarity to these concepts and illustrate how multi-tenancy extends or alters your approach to your existing architecture strategies. You already likely have strong notions of what it means to scale, secure, design, and operate robust architecture. With SaaS, we have to look at how multi-tenancy influences and overlays these key concepts, often introducing new principles that introduce an entirely new set of constraints and considerations that will shape how you approach architecting a SaaS offering.

Adding Tenancy to your Architecture

Let's start our exploration of SaaS architecture concepts by looking at a traditional non-SaaS application. In classic applications, the environment is constructed from the ground up with the assumption that it will be installed and run by individual cus-

tomers. Each customer essentially has its own dedicated footprint. Figure 2-1 provides a conceptual view of how one of these applications might be designed and built.

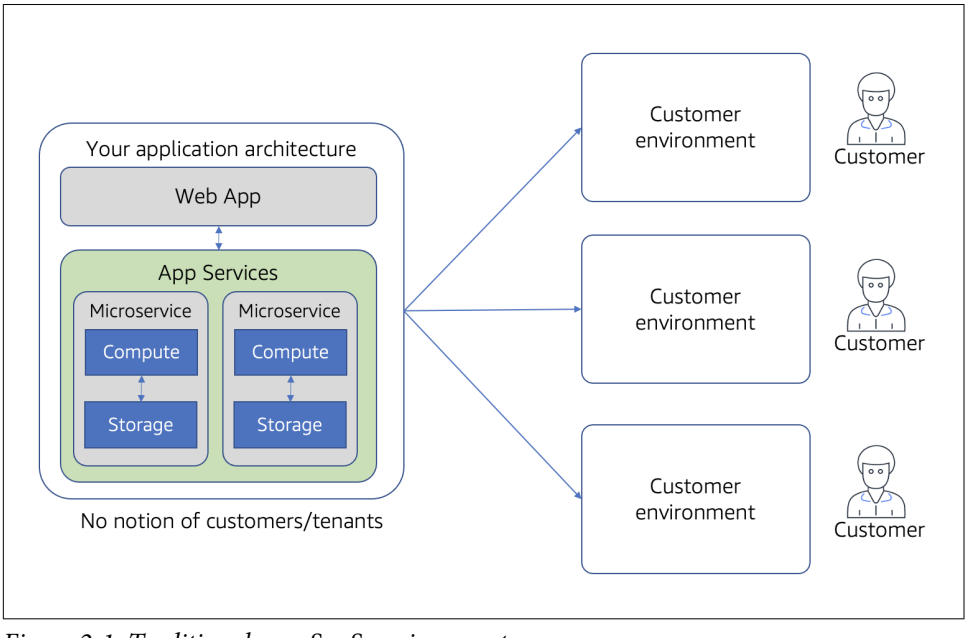


Figure 2-1. Traditional non-SaaS environments

On the left here, you'll see that we have a simplified view of a class application. Here, this application is built and then sold to individual customers. These customers might install the software in their own environment or it might run in the cloud. This approach simplifies the entire architectural model of this environment. The choices about how customers enter the environment, how they access our resources, and how they consume the services of our environment are much simpler when we know that they will be running in an environment that is dedicated to each customer. The general mindset here is that you have a piece of software and you're just stamping out copies of it for each new customer.

Now, let's think about what it means to deliver this same application in a multi-tenant SaaS environment. The diagram in Figure 2-2 provides a conceptual view of what this might look like. You see here that our customers, which are now tenants, are all consuming the same application.

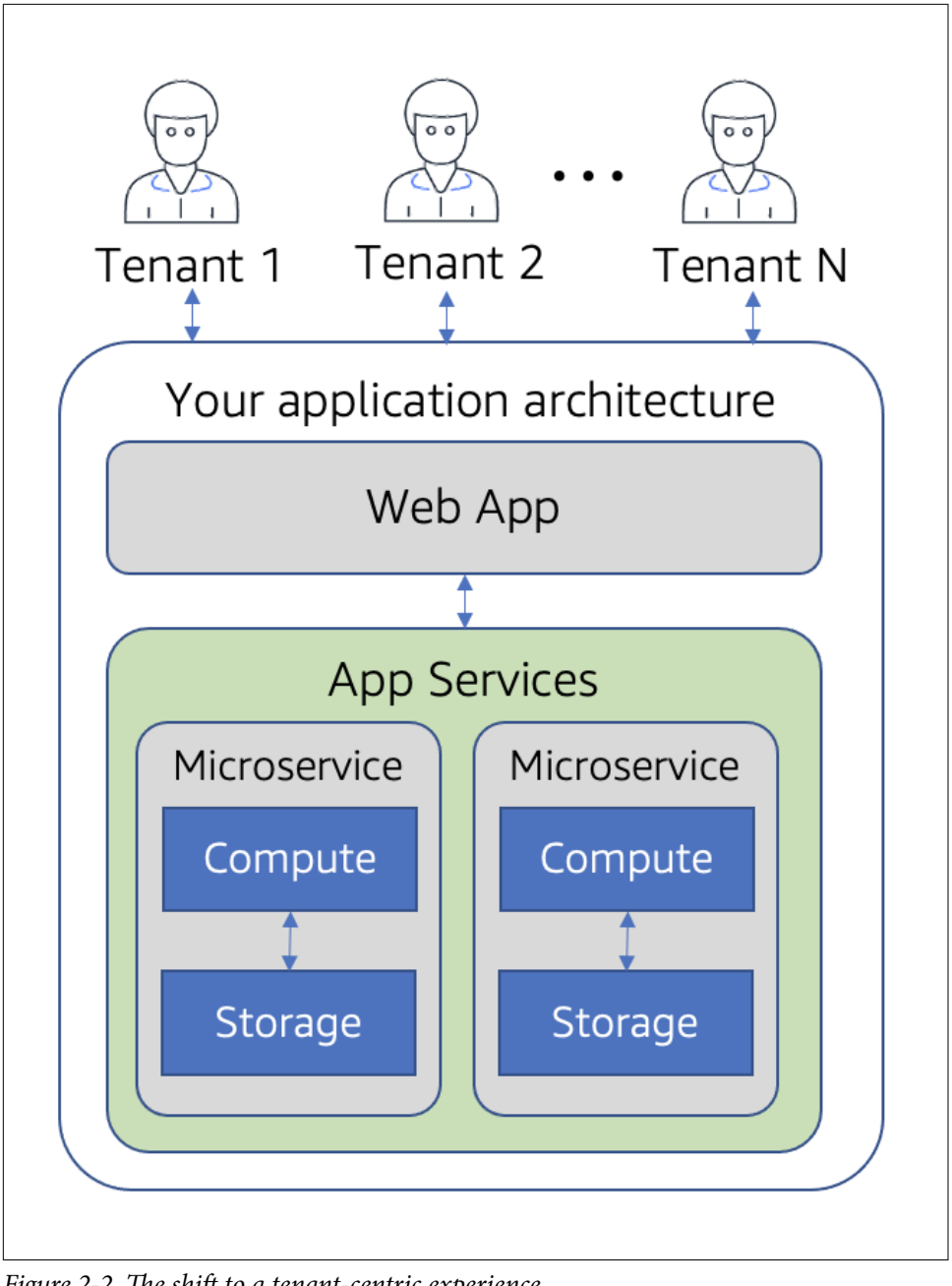


Figure 2-2. The shift to a tenant-centric experience

This shift may seem fairly simple in the diagram. However, it has a profound impact on how we design, build, secure, and manage this environment. We've essentially

made the transition from a per customer dedicated model to a multi-tenant architecture. Supporting this model reaches into every dimension of the underlying implementation of your system. It affects how you implement authentication, routing, scaling, performance, storage, and, in targeted areas, how you code the application logic of your system.

As a SaaS architect and builder, it becomes your job to determine how your system will support this notion of tenancy. In some cases, the influence here may be introduced as an extension of existing patterns and strategies. In other cases, it may require you to build and design all new constructs that must be added to support the needs of a multi-tenant setting.

Part of the challenge here is that there's a wide range of parameters that will influence how this notion of tenancy ends up landing in your environment. The key, at this stage, is to understand the fundamental architecture concepts and then look at how the different dimensions of your environment might influence how these concepts are expressed in a given environment. So, for the moment at least, let's stay up a level from these environmental considerations and focus squarely on building an SaaS architecture foundation that will give us the terminology and taxonomy of constructs that we can use to characterize the different moving parts and mechanisms that should be in the vocabulary of every SaaS architect and builder.

The Two Halves of Every SaaS Architecture

If we step back from the details of SaaS, we typically find that every SaaS environment–independent of its domain or design–can be broken down into two very distinct halves. In fact, across our entire discussion of SaaS across this book, you'll find that we'll use these two halves as the lens through which we'll look at how a multi-tenant system is built, deployed, and operated.

Figure 2-3 provides a conceptual representation of the two halves of SaaS. On the right-hand side of the diagram, you'll see what is labeled as the control plane. The control plane is where we'll place all of the cross-cutting constructs, services, and capabilities that support the foundational needs of a multi-tenant SaaS environment.

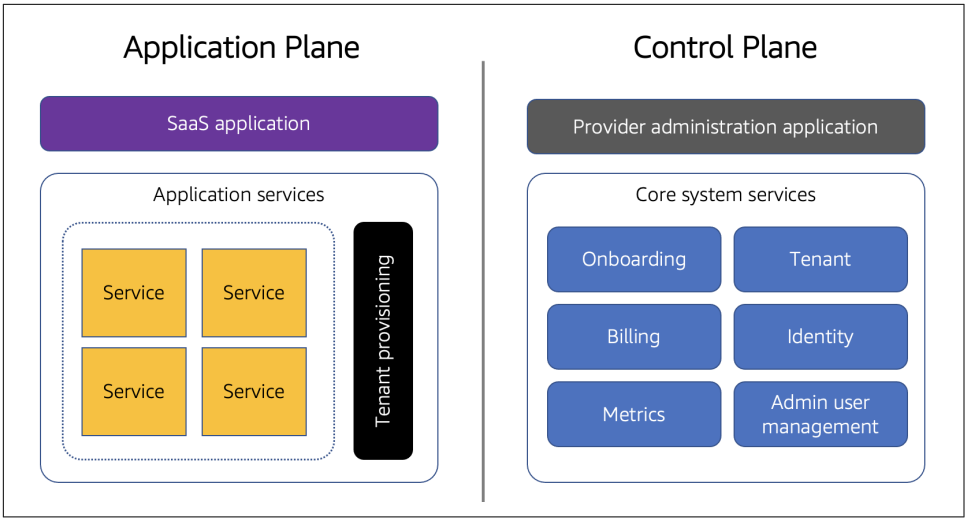


Figure 2-3. SaaS application and control planes

We often describe the control plane as the single pane of glass that is used to orchestrate and operate all the moving parts of your SaaS solution. It is at the core of enabling many of the principles that are essential to the success of your SaaS business. Concepts like tenant onboarding, billing, metrics, and a host of other services live in this control plane. You'll also see that our control plane includes an administration application. This represents the console or administration experience that is used by a SaaS provider to configure, manage, and operate their SaaS environment.

One interesting caveat here is that the services running in the control plane are not built or designed as multi-tenant services. If you think about it, there's actually nothing multi-tenant about the capabilities of the control plane. It doesn't have functionality that supports the needs of individual tenants. Instead, it provides the services and functionality that spans all tenants.

While architects and builders are often tempted to start the SaaS discussion with the multi-tenant aspects of their application, the foundations of your SaaS architecture often start with the control plane. In many respects, the control plane provides a forcing function, requiring engineers to inject and support the nuances of tenancy from the outset of their development.

In contrast, the application plane is where the features and functionality of your SaaS service are brought to life. This is where we see the manifestation of all the multi-tenant principles that are classically associated with SaaS environments. It's here that we focus more of our attention on how multi-tenancy will shape the design, functionality, security, and performance of our service and its underlying resources. Our time and energy in the application plane is focused squarely on identifying and choosing

the technologies, application services, and architecture patterns that best align with the parameters of your environment, timelines, and business. This is where you pour your energy into building out an application footprint that embraces agility and enables the business to support a range of personas and consumption models.

It's important to note that there is no single design, architecture, or blueprint for the application plane. I tend to view the application plane as a blank canvas that gets painted based on the unique composition of services and capabilities that my SaaS service requires. Yes, there are themes and patterns that we'll see that span SaaS application architectures. In fact, by the end of this book, you'll certainly have a healthy respect for the degree to which SaaS applications can vary from one solution to the next. Business, domain, and legacy realities are amongst a long list of factors that can influence the shape of your multi-tenant application strategy.

This view of the two halves of SaaS aligns with the mental model of multi-tenancy that we discussed in Chapter 1. Our application plane could share all tenant infrastructure or it could have completely dedicated infrastructure and it wouldn't matter. As long as we have a control plane that manages and operates these tenant environments through a unified experience, then we're considering this a multi-tenant environment.

This separation of concerns also influences our mental model for how the elements of our SaaS environment are updated and evolved. The services and capabilities of the control plane are versioned, updated, and deployed based on the needs of the SaaS provider, providing a range of services that reduce friction, enable centralized operations, and provide system-wide insights that are used to analyze the activity, scaling, and consumption patterns of all of your tenants. Meanwhile, our application plane is being driven more by the needs and experience of the system's tenants. Here, updates and deployments are introduced to provide new features, enhance tenant performance, support new tiering strategies, and so on.

Together, these two halves of SaaS represent the most fundamental building blocks of any SaaS environment. Understanding the roles of these planes will have a significant influence on how you'll approach the architecture, design, and decomposition of your SaaS offering.

Inside the Control Plane

Now that we have a better sense of the roles of the control and application planes, let's take a high-level pass at exploring the core concepts that commonly live within the scope of the control plane. We'll dig into each of these topics in much greater detail later in this book, exploring real-world implementation and architecture strategies. At this stage, though, we need to start a level up and develop an understanding of the different components that are part of any control plane you might build. Having a

higher level grasp of these components, the roles they play, and how they are related will allow us to explore these building blocks of multi-tenancy without getting distracted by the different nuances that show up when we pivot to the specific influences of technologies, languages, and domain considerations. Having this foundational view will allow you to see the landscape of options and begin to see the different components that span all SaaS architecture models.

The following is a breakdown of the different services and capabilities that are likely to show up in the control plane of your SaaS architecture, which covers onboarding, identity, metrics, billing, and tenant management.

Onboarding

The control plane is responsible for managing and orchestrating all the steps needed to get a new tenant introduced into your SaaS environment. On the surface, this may seem like a simple concept. However, as you'll see in Chapter 4, there are lots of moving parts to the onboarding experience. The choices you make here, in many respects, are at the core of enabling many of the multi-tenant business and design elements of your SaaS environment.

At this stage, let's stick with a high-level view of the key elements of the onboarding experience. In Figure 2-4 you'll see a conceptualized representation of the components that play a role in the onboarding experience. Here we show a tenant signing up for our SaaS service and triggering the onboarding process via the control plane. After this initial request, the control plane owns the rest of the onboarding flow, creating and configuring our tenant and its corresponding identity footprint. This includes assigning a unique identifier to our tenant that will be leveraged across most of the moving parts of our multi-tenant architecture.

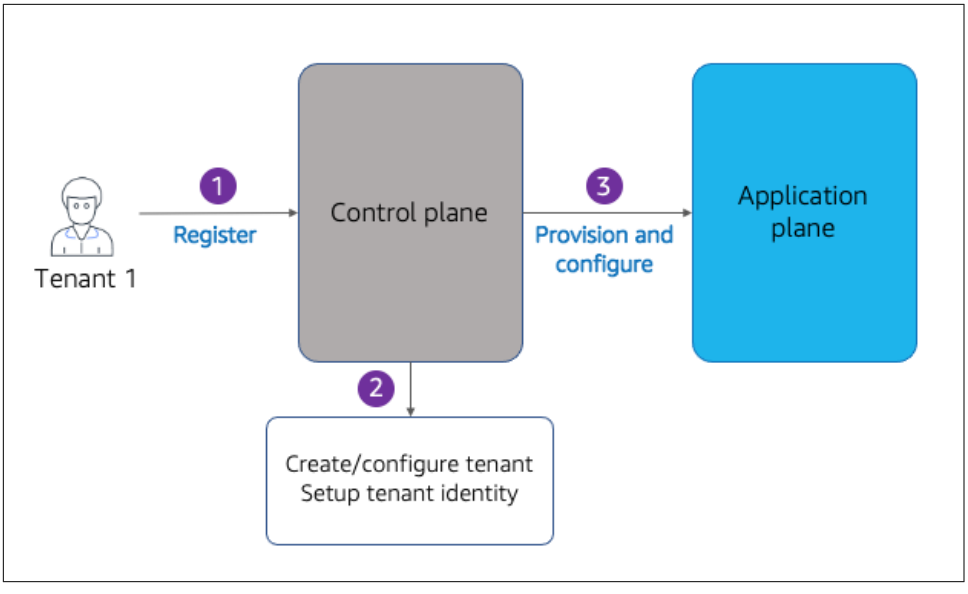


Figure 2-4. Onboarding tenants

You'll also notice that we show the control plane interacting with the application plane, provisioning and configuring any application specific resources that may be needed for each tenant. When we get into more detailed views of sample onboarding principles, we'll see how this part of the onboarding experience can get quite involved.

While there are common themes in the onboarding experience, the actual implementation of onboarding can vary significantly based on the domain you're in, the business goals of your solution, and the footprint of your application architecture. The key here, though, is that onboarding represents a foundational concept that sits at the front door of your SaaS experience. Business teams can and should take great interest in shaping and influencing how you approach building out this aspect of your system.

The higher level takeaway here is that onboarding is at the center of creating and connecting the most basic elements of a multi-tenant environment: tenants, users, identity, and tenant application resources. Onboarding weaves these concepts together and establishes the foundation for introducing tenancy to all the moving parts of your SaaS environment.

Identity

At first glance, you might wonder why identity belongs in the SaaS story. It's true that there are any number of different identity solutions that you can use to construct your SaaS solution. You could even suggest that your identity provider belongs some-

how outside the scope of our control plane discussion. However, it turns out that multi-tenancy and the control plane often have a pretty tight binding to your SaaS architecture. The diagram in Figure 2-5 provides a simplified view of how identity is applied in multi-tenant environments.

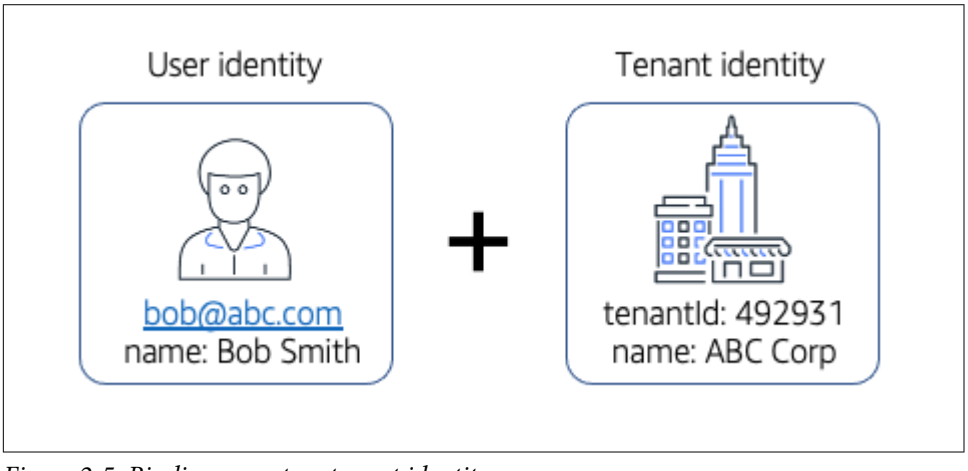


Figure 2-5. Binding users to a tenant identity

On the left you'll see the classic notion of user identity that is typically associated with authentication and authorization. It's true that our SaaS user will authenticate against our SaaS system. However, in a multi-tenant environment, being able to authenticate a user is not enough. A SaaS system must know who you are as a user *and* it must also be able to bind that user to a tenant. In fact, every user that is logged into our system must be attached in some way to a tenant.

This user/tenant binding ends up adding a wrinkle to our system's overall identity experience, requiring architects and builders to develop strategies for binding these two concepts in a way that still conforms to the requirements of your overall authentication model. This gets even more complicated when we start thinking about how we might support federated identity models in multi-tenant environments. We'll see that, the more the identity experience moves outside of our control, the more complex and challenging it becomes to support this binding between users and tenants. In some cases, you may find yourself introducing constructs to stitch these two concepts together.

When we dig into onboarding and identity in Chapter 4, you'll get a better sense of the key role identity plays in the broader multi-tenant story. Getting identity right is essential to building out a crisp and efficient strategy for introducing tenants into your SaaS architecture. The policies and patterns you apply here will have a cascading impact across many of the moving parts of your design and implementation.

Metrics

When your application is running in a multi-tenant model, it becomes more difficult to create a clear picture of how your tenants are using your system. If you're sharing infrastructure, for example, it's very hard to know which tenants are currently consuming that infrastructure and how the activity of individual tenants might be impacting the scale, performance, and availability of your solution. The population of tenants that are using your system may also be constantly changing. New tenants may be added. Existing tenants might be leaving. This can make operating and supporting multi-tenant environments particularly challenging.

These factors make it especially important for SaaS companies to invest in building out a rich metrics and analytics experience as part of their control plane. The goal here is to create a centralized hub for capturing and aggregating tenant activity that allows teams to monitor and analyze the usage and consumption profile of individual tenants.

The role of metrics here is very wide. The data collected will be used in an operational context, allowing teams to measure and troubleshoot the health of the system. Product owners might use this data to assess the consumption of specific features. Customer success teams might use this data to measure a new customer's time to value. The idea here is that successful SaaS teams will use this data to drive the business, operational, and technology success of their SaaS offering.

You can imagine how metrics will impact the architecture and implementation of many of the moving parts of your multi-tenant system. Microservice developers will need to think about how and where they'll add metrics instrumentation. Infrastructure teams will need to decide how and where they'll surface infrastructure activity. The business will need to weigh in and help capture the metrics that can measure the customer experience. These are just a few examples from a long list of areas where metrics might influence your implementation.

The tenant must be at the center of this metrics strategy. Having data on consumption and activity has significantly less value if it cannot be filtered, analyzed, and viewed through the lens of individual tenants.

Billing

Most SaaS systems have some dependency on a billing system. This could be a home grown billing system or it could be any one of the commercial SaaS billing systems that are available from different billing providers. Regardless of the approach, you can see how billing is a core concept that has a natural home within the control plane.

Billing has a couple touch points within the control plane. It's typically connected to the onboarding experience where each new tenant must be created as a "customer"

within your billing system. This might include configuring the tenant's billing plan and setting up other attributes of the tenant's billing profile.

Many SaaS solutions have billing strategies that meter and measure tenant activity as part of generating a bill. This could be bandwidth consumption, number of requests, storage consumption, or any other activity-related events that are associated with a given tenant. In these models, the control plane and your billing system must provide a way for this activity data to be ingested, processed, and submitted to your billing system. This could be a direct integration with the billing system or you could introduce your own services that process this data and send it to the billing system.

We'll get more into the details of billing integration in Chapter 16. The key here is to realize that billing will likely be part of your control plane services and that you'll likely be introducing dedicated services to orchestrate this integration.

Tenant Management

Every tenant in our SaaS system needs to be centrally managed and configured. In our control plane, this is represented by our Tenant Management service. Typically, this is a pretty basic service that provides all the operations needed to create and manage the state of tenants. This includes tracking key attributes that associate tenants with a unique identifier, billing plans, security policies, identity configuration, and an active/inactive status.

In some cases, teams may overlook this service or combine it with other concepts (identity, for example). It's important for multi-tenant environments to have a centralized service that manages all of this tenant state. This provides a single point of tenant configuration and allows tenants to easily manage them through a single experience.

We'll explore the elements and permutations of implementing tenant management more in Chapter 5.

Inside the Application Plane

Now that we have a better sense of the core concepts with the control plane, let's start looking at the common areas where multi-tenancy shows up in the application plane. While the control plane typically has a consistent set of common services, the application plane is a bit more abstract. How and where multi-tenancy is applied within the application plane can vary significantly based on a wide range of factors. That being said, there are still a range of themes that will surface, albeit in different forms, within your application plane. So, even though there is variation here, every SaaS architect will need to consider how/where they will introduce these themes into the application plane of their solution.

As you dig into the application pane, you'll find that your technology stack and deployment footprint will have a significant influence on how these concepts are applied. In some cases, there may be ready-made solutions that fit your use case precisely. In other cases, you may find yourself inventing solutions to fill gaps in your technology stack. While building out something to fill these gaps may add complexity and overhead to the build of your solution, in most cases you'll want to take on this added work to ensure that your SaaS solution is not compromising on important elements of your multi-tenant architecture.

In subsequent chapters we'll look at real-world working examples that provide a more concrete view of how these constructs are realized within your application plane. For now, though, let's come up a level and establish a core set of application plane principles that should span every SaaS architecture.

Tenant Context

One of the most fundamental concepts in our application plane is the notion of tenant context. Tenant context does not map to any one specific strategy or mechanism. Instead, it's a broader concept that is meant to convey the idea that our application plane is always functioning in the context of specific tenants. This context is often represented as a token or some other construct that packages all the attributes of your tenant. A common example that you'll use here is a JSON Web Token (JWT) which combines your user and tenant information in one construct that is shared across all the moving parts of your multi-tenant architecture. This JWT becomes our passport for sharing tenant information (context) with any service or code that relies on this context. It's this token that is referred to as your tenant context.

Now, you'll see that this tenant context has a direct influence on how your application architecture processes tenant requests. This may affect routing, logging, metrics, data access, and a host of other constructs live within the application plane. Figure 2-6 provides a conceptual view of tenant context in action.

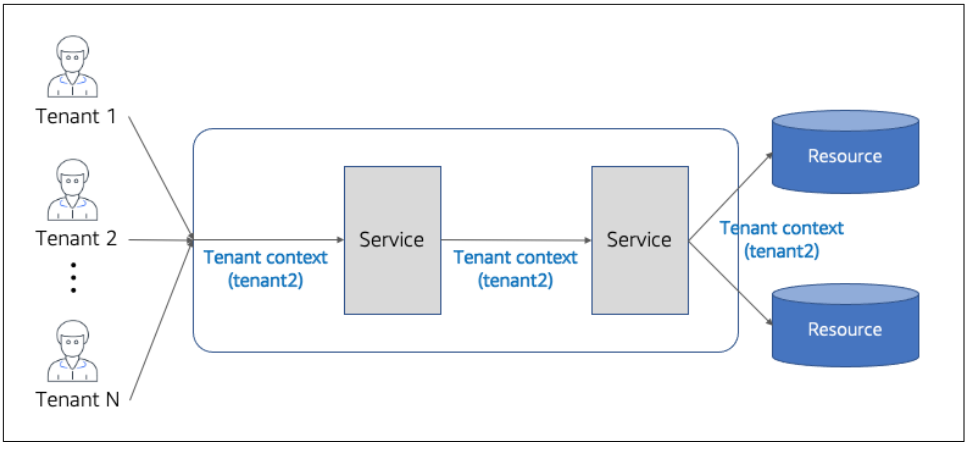


Figure 2-6. Applying tenant context

The flow in Figure 2-6 shows tenant context being applied across the different services and resources that are part of a multi-tenant environment. This starts on the left-hand side of the diagram where my tenants authenticate against the identity that was created during onboarding and acquire their tenant context. This context is then injected into a service of my application. This same context flows into each downstream interaction of my system, enabling you to acquire and apply that context across a range of different use cases.

This represents one of the most fundamental differences of a SaaS environment. Our services don't just work with users–they must incorporate tenant context as part of the implementation of all the moving parts of our SaaS application. Every microservice you write will use this tenant context. It will become your job to figure out how to apply this context effectively without adding too much complexity to the implementation of your system. This, in fact, is a key theme that we'll address when we dig into SaaS microservice in Chapter 8.

As a SaaS architect, this means that you must be always thinking about how tenant context will be conveyed across your system. You'll also have to be thinking about the specific technology strategies that will be used to package and apply this tenant context in ways that limit complexity and promote agility. This is a continual balancing act for SaaS architects and builders.

Tenant Isolation

Multi-tenancy, by its very nature, focuses squarely on placing our customers and their resources into environments where resources may be shared or at least reside side-by-side in common infrastructure environments. This reality means that multi-tenant

solutions are often required to apply and implement creative measures to ensure that tenant resources are protected against any potential cross-tenant access.

To better understand the fundamentals of this concept, let's look at a simple conceptual view of a solution running in our application plane (shown in Figure 2-7).

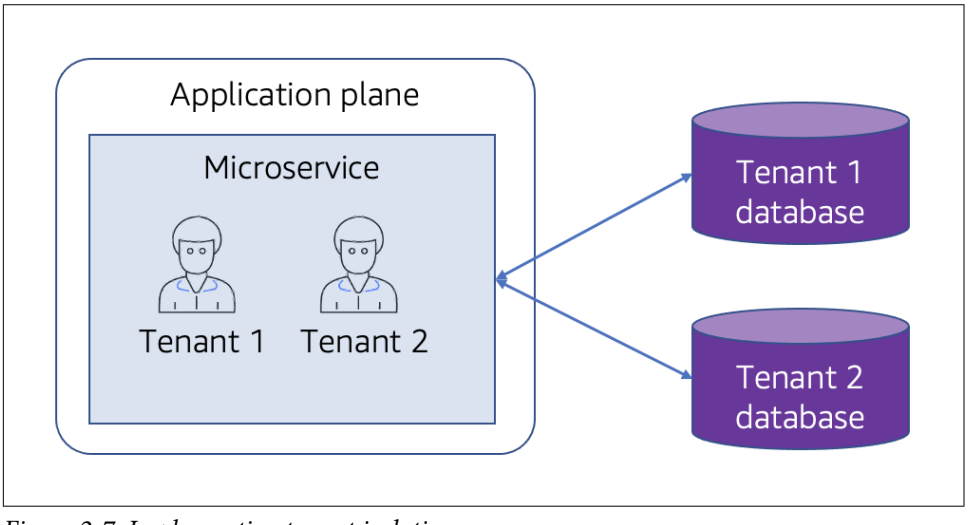


Figure 2-7. Implementing tenant isolation

Here you'll see we have the simplest of application planes running a single microservice. For this example, our SaaS solution has chosen to create separate databases for each tenant. At the same time, our microservice is sharing its compute with all tenants. This means that our microservice can be processing requests from tenants 1 and 2 simultaneously.

While the data for our tenants are stored in separate databases, there is nothing in our solution that ensures that tenant 1 can't access the database of tenant 2. In fact, this would be the case even if our tenants weren't running in separate, dedicated microservices.

To prevent any access to another tenant's resources, our application plane must introduce a construct to prevent this cross-tenant access. The mechanisms to implement this will vary wildly based on a number of different considerations. However, the basic concept–which is labeled Tenant Isolation–spans all possible solutions. The idea here is that every application plane must introduce targeted constructs that strictly enforce the isolation of individual tenant resources–even when they may be running in a shared construct.

We'll dig into this concept in great detail in Chapter 10. It goes without saying that tenant isolation represents one of the most fundamental building blocks of SaaS architecture. As you build out your application plane you'll need to find the flavor

and approach that allows you to enforce isolation at the various levels of your SaaS architecture.

Data Partitioning

The services and capabilities within our application plane often need to store data for tenants. Of course, how and where you choose to store that data can vary significantly based on the multi-tenant profile of your SaaS application. Any number of factors might influence your approach to storing data. The type of data, your compliance requirements, your usage patterns, the size of the data, the technology you're using–these are all pieces of the multi-tenant storage puzzle.

In the world of multi-tenant storage, we refer to the design of these different storage models as data partitioning. The key idea here is that you are picking a storage strategy that partitions tenant data based on the multi-tenant profile of that data. This could mean the data is stored in some dedicated construct or it could mean it lands in some shared construct. These partitioning strategies are influenced by a wide range of variables. The storage technology you're using (object, relational, nosql, etc.) obviously has a significant impact on the options you'll have for representing and storing tenant data. The business and use cases of your application can also influence the strategy you select. The list of variables and options here are extensive.

As a SaaS architect, it will be your job to look at the range of different data that's stored by your system and figure out which partitioning strategy best aligns with your needs. You'll also want to consider how/if these strategies might impact the agility of your solution. How data impacts the deployment of new features, the up-time of your solution, and the complexity of your operational footprint are all factors that require careful consideration when selecting a data partitioning strategy. It's also important to note that, when picking a strategy, this is often a fine-grained decision. How you partition data can vary across the different services within your application plane.

This is a much deeper topic that we'll cover more extensively in Chapter 9. By the end of that chapter, you'll have a much better sense of what it means to bring a range of different strategies to life using a variety of different storage technologies.

Tenant Routing

In this simplest of SaaS architecture models, you may find that all tenants are sharing their resources. However, in most cases, your architecture is going to have variations where some or all of your tenant's infrastructure may be dedicated. In fact, it would not be uncommon to have microservices that are deployed on a per tenant basis.

The main point here is that SaaS application architectures are often required to support a distributed footprint that has any number of resources running in a combination of shared and dedicated models. The image in Figure 2-8 provides a simplified

sample of a SaaS architecture that supports a mix of shared and dedicated tenant resources.

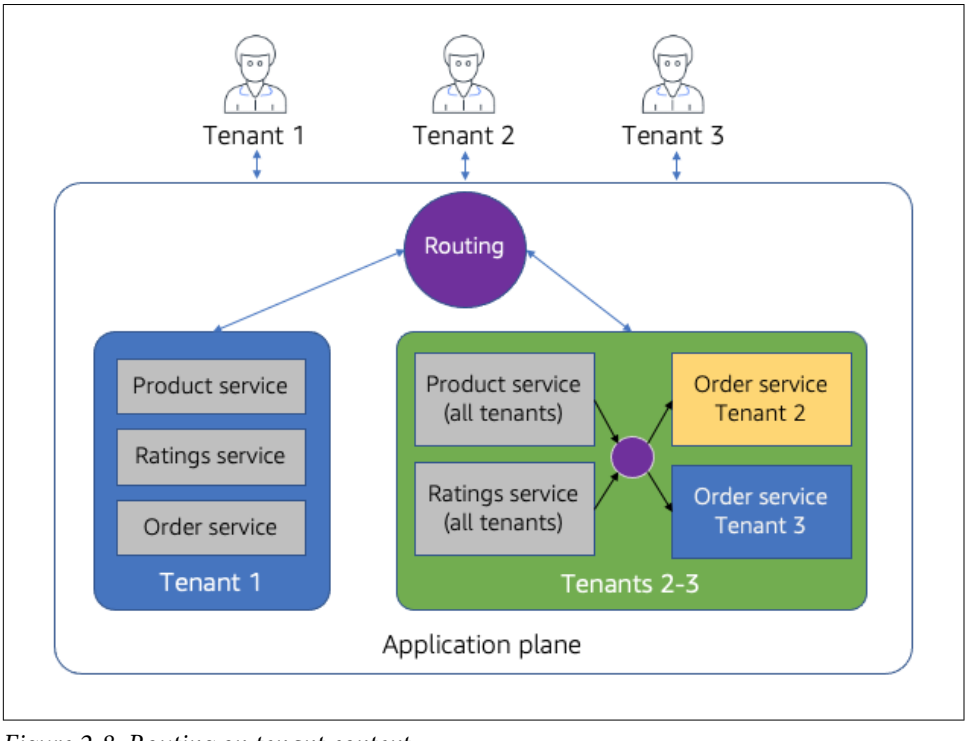


Figure 2-8. Routing on tenant context

In this example, we have three tenants that will be making requests to invoke operations on our application services. In this particular example, we have some resources that are shared and some that are dedicated. On the left, tenant 1 has an entirely dedicated set of services. Meanwhile, on the right-hand side, you'll see that we have the services that are being used by tenants 2 and 3. Here, note that we have the product and rating services that are being shared by both of these tenants. However, these tenants each have dedicated instances of the order service.

Now, as you step back and look at the overall configuration of these services, you can see where our multi-tenant architecture would need to include strategies and constructs that would correctly route tenant requests to the appropriate services. This happens at two levels within this example. If you start at the top of our application plane where our application plane is receiving requests from three separate tenants, you'll notice that there's a conceptual placeholder for a router here. This router must accept requests from all tenants and use the injected tenant context (that we discussed earlier) to determine how and where to route each request. Also, within the tenant 2/3 box on the right, you'll see that there is another placeholder for routing that will

determine which instance of the order service will receive requests (based on tenant context).

Let's look at a couple concrete examples to sort this out. Suppose we get a request from tenant 1 to look up a product. When the router receives this request, it will examine the tenant context and route the traffic to the product service on the left (for tenant 1). Now, let's say we get a request from tenant 3 to update a product that must also update an order. In this scenario, the top-level router would send the request to the shared product service on the right (based on the tenant 2 context). Then, the product service would send a request to the order service via the service-to-service router. This router would look at the tenant context, resolve it to tenant 2, and send a request to the order service that's dedicated to tenant 2.

This example is meant to highlight the need for multi-tenant aware routing constructs that can handle the various deployments footprint we might have in a SaaS environment. Naturally, the technology and strategy that you apply here will vary based on a number of parameters. There are also a rich collection of routing tools and technologies, each of which might approach this differently. Often, this comes down to finding a tool that provides flexible and efficient ways to acquire and dynamically route traffic based on tenant context.

We'll see these routing constructs applied in specific solutions later in this book. At this stage, it's just important to understand that routing in multi-tenant environments often adds a new wrinkle to our infrastructure routing model.

Multi-Tenant Application Deployment

Deployment is a pretty well understood topic. Every application you build will require some DevOps technology and tooling that can deploy the initial version of your application and any subsequent updates. While these same concepts apply to the application plane of our multi-tenant environment, you'll also discover that different flavors of tenant application models will add new considerations to your application deployment model.

We've already noted here that tenants may have a mix of dedicated and shared resources. Some may have fully dedicated resources, some may have fully shared, and others may have some mix of dedicated and shared. Knowing this, we have to now consider how this will influence the DevOps implementation of our application deployment.

Imagine deploying an application that had two dedicated microservices and three shared microservices. In this model, our deployment automation code will have to have some visibility into the multi-tenant configuration of our SaaS application. It won't just deploy updated services like you would in a classic environment. It will need to consult the tenant deployment profile and determine which tenants might

need a separate deployment of a microservice for each dedicated microservice. So, microservices within our application plane might be deployed multiple times. That, and our infrastructure automation code may need to apply tenant context to the configuration and security profile of each of these microservices.

Technically, this is not directly part of the application plane. However, it has a tight connection to the design and strategies we apply within the application plane. In general, you'll find that the application plane and the provisioning of tenant environments will end up being very inter-connected.

The Gray Area

While the control and application planes cover most of the fundamental multi-tenant architecture constructs, there are still some concepts that don't fit so cleanly into either of these planes. At the same time, these areas still belong in the discussion of foundational SaaS topics. While there are arguments that could be made for landing these in specific planes, to steer clear of the debate, I'm going to handle these few items separately and address the factors that might push them into one plane or the other.

Tiering

Tiering is a strategy most architects have encountered as part of consuming various third-party offerings. The basic idea here is that SaaS companies use tiers to create different variations of an offering with separate price points. As an example, a SaaS provider could offer their customers Basic, Advanced, and Premium tiers where each tier progressively adds additional value. Basic tier tenants might have constraints on performance, number of users, features, and so on. Premium tier tenants might have better SLAs, a higher number of users, and access to additional features.

The mistake some SaaS architects and builders make is that they assume that these tiers are mostly pricing and packaging strategies. In reality, tiering can have a significant impact on many of the dimensions of your multi-tenant architecture. Tiering is enabled by building a more pliable SaaS architecture that offers the business more opportunities to create value boundaries that they may not have otherwise been able to offer.

Tiering naturally layers onto our discussion of tenant context, since the context that gets shared across our architecture often includes a reference to a given tenant's tier. This tier is applied across the architecture and can influence routing, security, and a host of other aspects of the underlying implementation of your system.

In some implementations of tiering, we'll see teams place this within their control plane as a first class concept. It's true that onboarding often includes some need to map a tenant's profile to a given tier. Tiers are also often correlated to a billing plan,

which would seem natural to maintain within the scope of the control plane. At the same time, tiers are also used heavily within the application plane. They can be used to configure routing strategies or they could also be referenced as part of the configuration of throttling policies. The real answer here is that tiering has a home in both planes. However, I would probably lean toward placing it in the control plane since the tier can be managed and returned by interactions with the control plane (authentication, for example). The returned tier can be attached to the tenant context and applied through that mechanism within the application plane.

Tenant, Tenant Admin, and System Admin Users

The term "user" can easily get overloaded when we're talking about SaaS architecture. In a multi-tenant environment, we have multiple notions of what it means to be a user–each of which plays a distinct role. Figure 2-9 provides a conceptual view of the different flavors of users that you will need to support in your multi-tenant solution.

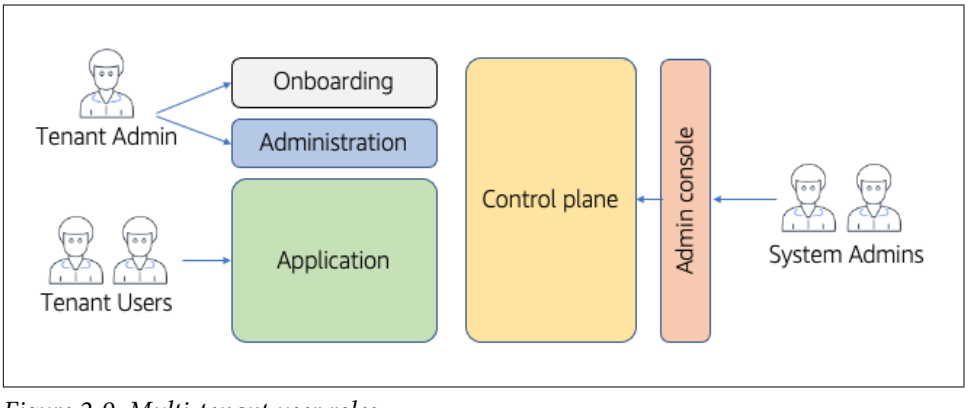


Figure 2-9. Multi-tenant user roles

On the left-hand side of the diagram, you'll see that we have the typical tenant-related roles. There are two distinct types of roles here: tenant administrators and tenant users. A tenant administrator represents the initial user from your tenant that is onboarded to the system. This user is typically given admin privileges. This allows them to access the unique application administration functionality that is used to configure, manage, and maintain application level constructs. This includes being able to create new tenant users. A tenant user represents users that are using the application without any administrative capabilities. These users may also be assigned different application-based roles that influence their application experience.

On the right-hand side of the diagram, you'll see that we also have system administrators. These users are connected to the SaaS provider and have access to the control plane of your environment to manage, operate, and analyze the health and activity of a SaaS environment. These admins may also have varying roles that are used to char-

acterize their administrative privileges. Some may have full access, others may have limits on their ability to access or configure different views and settings.

You'll notice that I've also shown an administration console as part of the control plane. This represents an often overlooked part of the system admin role. It's here to highlight the need for a targeted SaaS administration console that is used to manage, configure, and operate your tenants. It is typically something your team needs to build to support the unique needs of your SaaS environment (separate from other tooling that might be used to manage the health of your system). Your system admin users will need an authentication experience to be able to access this SaaS admin console.

SaaS architects need to consider each of these roles when building out a multi-tenant environment. While the tenant roles are typically better understood, many teams invest less energy in the system admin roles. The process for introducing and managing the lifecycle of these users should be addressed as part of your overall design and implementation. You'll want to have a repeatable, secure mechanism for managing these users.

The control plane vs. application plane debate is particularly sticky when it comes to managing users. There's little doubt that the system admin users should be managed via the control plane. In fact, the initial diagram of the two planes shown at the outset of this chapter (Figure 2-3) actually includes an admin user management service as part of its control plane. It's when you start discussing the placement of tenant users that things can get more fuzzy. Some would argue that the application should own the tenant user management experience and, therefore, management of these users should happen within the scope of the application plane. At the same time, our tenant onboarding process needs to be able to create the identities for these users during the onboarding process, which suggests this should remain in the control plane. You can see how this can get circular in a hurry.

My general preference here, with caveats, is that identity belongs in the control plane–especially since this is where tenant context gets connected to the user identities. This aspect of the identity would never be managed in the scope of the application plane.

A compromise can be had here by having the control plane manage the identity and authentication experience while still allowing the application to manage the non-identity attributes of the tenant outside of the identity experience. The other option here would be to have a tenant user management service in your control plane that supports any additional user management functionality that may be needed by your application.

Tenant Provisioning

So far, we've highlighted the role of onboarding within the control plane. We also looked at how the onboarding process may need to provision and configure application infrastructure as part of the onboarding experience. This raises an important question: should tenant provisioning live within the control plane or the application plane?

Figure 2-10 provides a conceptual view of the two options. On the left, you'll see the model where tenant provisioning runs within the application plane. In this scenario, all the elements of onboarding (tenant creation, billing configuration, and identity setup) still happen within the scope of the control plane. The provisioning step is triggered and orchestrated by the onboarding service, but runs within the application plane.

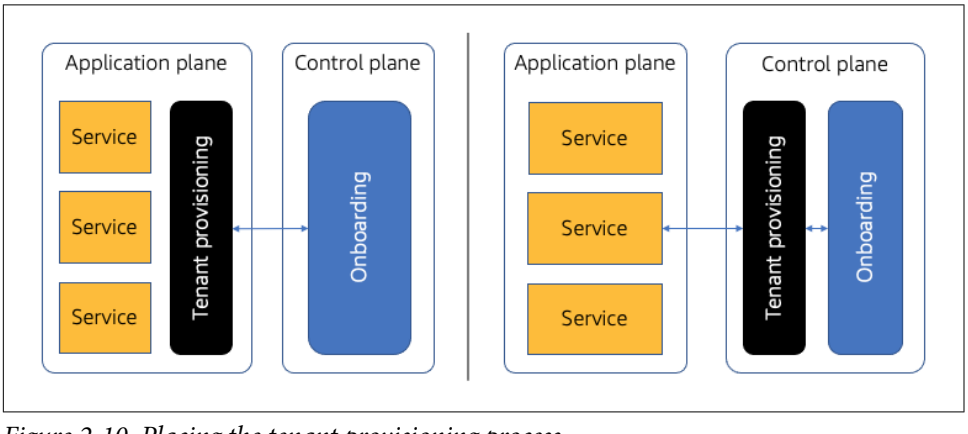


Figure 2-10. Placing the tenant provisioning process

The alternate approach is shown on the right side of this diagram. Here, tenant provisioning is executed from within the control plane. This means that the tenant provisioning would execute infrastructure configuration scripts that are applied within the application plane. This puts all the moving parts of onboarding within the control plane.

The tradeoffs, to me, center around the encapsulation and abstraction of the application plane. If you believe the structure and footprint of application infrastructure should be unknown to the control plane, then you'll favor the model on the left. If you feel strongly that onboarding is already owned by the control plane, you could argue that it's natural for it to also own the application provisioning process.

My bias here leans toward keeping provisioning closest to the resources that are being described and configured. I'd prefer not to make updates to the control plane based on changes in the architecture of the application plane. The tradeoff here is that the

control plane must support a more distributed onboarding experience and rely on messaging between the control and application planes to track the provisioning progress/status. Both models have their merits. The key here is that provisioning should be a standalone part of the onboarding experience. So, if at some point you choose to move it, it would be somewhat encapsulated and could move without significant rethink.

Integrating Control and Application Planes

Some organizations will create very specific boundaries between the control and application planes. This might be a network boundary or some other architectural construct that separates these two planes. This has advantages for some organizations in that it allows these planes to be configured, managed, and operated based on the unique needs of each plane. It also introduces opportunities to design more secure interactions between these planes.

With this in mind, we can then start to consider different approaches to integrating the control and application planes. The integration strategy you choose here will be heavily influenced by the nature of the interactions between the planes, the geographic footprint of your solution, and the security profile of your environment.

Some teams may opt for a more loosely-coupled model that is more event/message driven while others may require a more native integration that enables more direct control of the application plane resources. There are few absolutes here and there are a wide range of technologies that bring different possibilities to this discussion. The key here is to be thoughtful in picking an integration model that enables the level of control that fits the needs of your particular domain, application, and environment.

Much of the discussion here leans toward a model where the application and control planes are deployed and managed in separate infrastructure. While the merits of separating these planes is compelling, it's important to note that there is no rule that suggests that these planes must be divided along some hard boundary. There are valid scenarios where a SaaS provider may choose to deploy the control and application planes into a shared environment. The needs of your environment, the nature of your technology, and a range of other considerations will determine how/if you deploy these planes with more concrete architectural boundaries. They key here to ensure that you divide your system into these distinct planes–regardless of how/where they are deployed.

As we dig into the specifics of the control plane, we can look at the common touch points between these two planes and get into the specific integration use cases and

their potential solutions. For now, though, just know that integration is a key piece of the overall control/application plane model.

Picking Technologies for Your Planes

SaaS teams pick the technologies for implementing their SaaS solutions based on any number of different variables. Skill sets, cloud providers, domain needs, legacy considerations–these are just a few of the many parameters that go into selecting a technology for your multi-tenant SaaS offering.

Now, as we look at SaaS through the lens of our control and application planes, it's also natural to think about how the needs of these two planes might influence your choice of technologies. If you choose an entirely container-based model for your application plane, should that mean your control plane must also be implemented with containers? The reality here is that the planes will support different needs and different consumption profiles. There is nothing that suggests that the technologies they use must somehow match.

Consider, for example, the cost and consumption profile of your control plane. Many of these services may be consumed on a more limited basis than services running in our application plane. We might favor choosing a different technology for our control plane that yields a more cost-efficient model. Some teams might choose to use serverless technologies to implement their control plane.

The decisions can also be much more granular. I might choose one technology for some types of services and different technologies for other services. The key here is that you should not assume that the profile, consumption, and performance profile of your control and application planes will be the same. As part of architecting your SaaS environment, you want to consider the technology needs of these two planes independently.

Avoiding the Absolutes

This discussion of SaaS architecture concepts devoted lots of attention to defining SaaS architecture through the lens of the control and application planes. The planes equip us with a natural way to think about the different components of a multi-tenant architecture and they give us a good mental model for thinking about how the different features of a multi-tenant architecture should land in your SaaS environment.

While these constructs are useful, I would also be careful about attaching absolutes to this model. Yes, it's a good way to think about SaaS and it provides us with a framework for talking about how we can approach building multi-tenant solutions. It's certainly provided me with a powerful construct for engaging teams that are trying to design and architect their SaaS systems. It has also put emphasis on the need for a set

of shared services that are outside the scope of the multi-tenant architecture of your application.

The key here, though, is to use these concepts to shape how you approach your SaaS architecture, allowing for the fact that there may be nuances of your environment that may require variations in your approach. It's less about being absolute about what's in each plane and more about creating an architecture that creates a clear division of responsibility and aligns with the security, management, and operational profile of your SaaS offering.

Conclusion

This chapter was all about building a foundation of SaaS architecture concepts. We looked at the core elements of SaaS architecture with the goal of framing multi-tenant architecture patterns and strategies without getting into the specifics of any particular technology or domain. The concepts we covered here should apply to any SaaS environment and provide any team with any technology a mental model for approaching SaaS architecture.

We've really only touched the surface of multi-tenant architecture here. As we move forward, we'll start mapping these concepts to concrete examples that tease out all the underlying details and add another layer of design considerations that we'll build on the mental model that we've created here. This added layer of detail will start to illustrate the web of possibilities you'll need to navigate as you consider how best to connect the needs of your SaaS business with the realities that come with realizing these principles with languages, technology stacks, and tools that bring their own set of variables to your multi-tenant equation.

Our next step is to start looking at SaaS deployment models. This will shift us from thinking about concepts to mapping these concepts to seeing those concepts landed in different patterns of deployment. The goal here is to start thinking about and bringing more clarity to the strategies that are used to support the various SaaS models that you'll need to consider as you shape your SaaS architecture.

CHAPTER 3
Multi-Tenant Deployment Models

Selecting your multi-tenant deployment model is one of the first things you'll do as a SaaS architect. It's here that you step back from the details of the multi-tenant implementation and ask yourself broader questions about the fundamental footprint of your SaaS environment. The choices you make around the deployment model of your application will have a profound impact on the cost, operations, tiering, and a host of other attributes that will have a direct impact on the success of your SaaS business.

In this chapter, I'll be walking through a range of different multi-tenant deployment models, exploring how the footprint of each of these models can be used to address a variety of different technology and business requirements. Along the way, I'll highlight the pros and cons of the various models and give you a good sense of how the model you select can shape the complexity, scalability, performance, and agility of your SaaS offering. Understanding these models and their core values/tradeoffs is essential to arriving at an architecture strategy that balances the realities of your business, customers, time pressures, and long-term SaaS objectives. While there are themes in these models that are common to many SaaS teams, there is no one blueprint that everyone will follow. Instead, it will be your job to navigate these deployment models, weigh the options, and select a model or combination of models that address your current and emerging needs.

We'll also use this chapter to continue to expand our SaaS vocabulary, attaching terminology to these models and their supporting constructs that will be referenced throughout the remainder of this book. These new terms will give you more precise ways to describe the nature of SaaS environments and enable you to be more crisp and granular about how you describe the moving parts of a multi-tenant architecture. These new terms and concepts allow us to describe and classify SaaS architectures in a way that better accommodates the broad range of multi-tenant permutations that we find in the wild.

What's a Deployment Model?

In Chapter 1, I spent a fair amount of time looking at how we needed to expand our definition of SaaS and multi-tenancy. This included looking at how the notion of multi-tenancy needed to embrace a broader range of architecture patterns and strategies. Now, in this chapter, we can move from the conceptual to a more concrete view of how this definition of SaaS influences the options you have when you start defining and choosing your SaaS architecture.

Here, we'll start to identify and name specific multi-tenant architecture patterns, providing a better sense of the different high-level architecture strategies you'll want to consider when designing your own SaaS environment. I refer to these different patterns as deployment models largely because they represent how your tenants will be deployed into the application plane of your SaaS solution. Each deployment model is meant to identify a distinct approach to defining the high-level multi-tenant architectural model that could be employed by your system.

The deployment model you select will have a significant influence on the technical and business profile of your SaaS architecture. Each deployment model comes with its own unique blend of pros and cons that you'll need to weigh to figure out which pattern best aligns with the realities and goals of your solution. Are your tenants sharing some of their infrastructure? Are they sharing all of their infrastructure? Or, is there some mix of shared and dedicated infrastructure in your SaaS environment? What domain and compliance models will you need to support? Are you planning to offer a tiered model? Your answers to these questions (and others) will all be used as data points that will guide your selection of a deployment model. From there, you'll see how the selection of a deployment model will have a cascading impact across all the moving parts of your multi-tenant implementation, fundamentally shaping how your solution is built, managed, operated, configured, and provisioned. Thus, the need to have a firm grasp of these models and their corresponding tradeoffs.

Let's look at a couple of conceptual deployment models to help clarify this concept. Figure 3-1 provides examples of two sample deployment models. On the left, you'll see a deployment model that has all of its tenant resources being shared across the compute layers of our multi-tenant environment. However, the storage resources are dedicated to individual tenants. In contrast, the right-hand side of the diagram shows another variation of a deployment model where all of a tenant's infrastructure (compute, storage, etc.) is deployed in a dedicated model. These are just a small sample of two deployment models, but they give you a more concrete view of what we're talking about when we're describing the fundamental aspects of a SaaS deployment model.

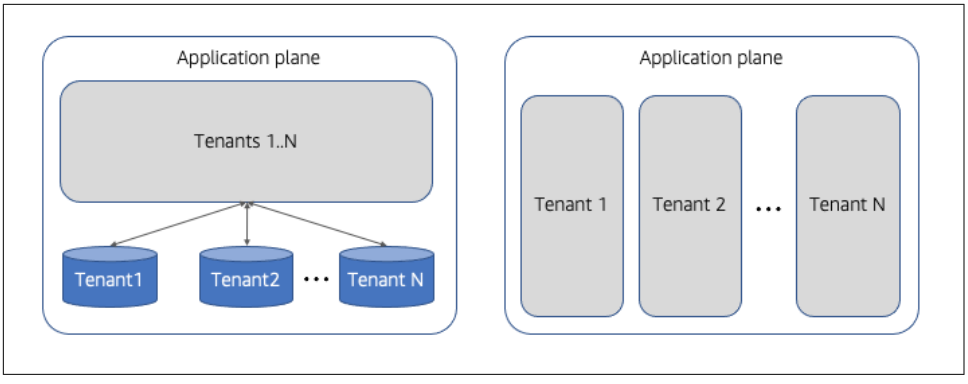


Figure 3-1. Conceptual deployment models

It's important to note that a deployment model represents a pattern that may have multiple implementations that share a common set of values and guiding principles. My goal here is to clearly identify the different categories of deployment models, acknowledging that the attributes and realities of these models will vary based on the technology stack, services, tools, and other environmental factors that are used to move from concept to implementation. Also, these models should not be viewed as being mutually exclusive. As you'll see in the sections that follow, there can be compelling business reasons that may have teams creating solutions that rely on combinations of these models. For our discussion here, you should start by understanding the principles and value proposition of each deployment model. Then, from there, you can consider which flavors/combinations of these models best align with the requirements of your SaaS solution.

Picking a Deployment Model

Understanding the value proposition of each deployment model is helpful. However, selecting a deployment model goes beyond evaluating the characteristics of any one model. When you sit down to figure out which deployment model is going to best for your application and business, you'll often have to weigh a wide spectrum of parameters that will inform your deployment selection process.

In some cases, the state of your current solution (if you're migrating) might have a huge impact on the deployment model you choose. A SaaS migration can often be more about finding a target deployment model that lets you get to SaaS without rebuilding your entire solution. Time to market, competitive pressures, legacy technology considerations, and team makeup are also factors that could represent significant variables in a SaaS migration story. Each of these factors would likely shape the selection of a deployment model.

Obviously, teams that are building a new SaaS solution have more of a blank canvas to work with. Here, the deployment model that you choose is probably more driven by the target personas and experience that you're hoping to achieve with your multi-tenant offering. The challenge here is selecting a deployment model that balances the near- and long-term goals of the business. Selecting a model that is too narrowly focused on a near-term experience could limit growth as your business hits critical mass. At the same time, over-rotating to a deployment model that reaches too far beyond the needs of current customers may represent pre-optimization. It's certainly a tough balancing act to find the right blend of flexibility and focus here (a classic challenge for most architects and builders).

No matter where you start your path to SaaS, there are certainly some broader global factors that will influence your deployment model selection. Packaging, tiering, and pricing goals, for example, often play a key role in determining which deployment model(s) might best fit with your business goals. Cost and operational efficiency are also part of the deployment model puzzle. While every solution would like to be as cost and operationally efficient as possible, each business may be facing realities that can impact their deployment model preferences. If your business has very tight margins, you might lean more toward deployment models that squeeze every last bit of cost efficiency out of your deployment model. Others may be facing challenging compliance and/or performance considerations that might lead to deployment models that strike a balance between cost and customer demands.

These are just some simple examples that are part of the fundamental thought process you'll go through as part of figuring out which deployment model will address the core needs of your business. As I get deeper into the details of multi-tenant architecture patterns, you'll see more and more places where the nuances of multi-tenant architecture strategies will end up adding more dimensions to the deployment model picture. This will also give you a better sense of how the differences in these models might influence the complexity of your underlying solution. The nature of each deployment model can move the complexity from one area of our system to another.

The key here is that you should not be looking for a one-size-fits-all deployment model for your application. Instead, you should start with the needs of your domain, customers, and business and work backward to the combination of requirements that will point you toward the deployment model that fits with your current and aspirational goals.

It's also important to note that the deployment model of your SaaS environment is expected to be evolving. Yes, you'll likely have some core aspects of your architecture that will remain fairly constant. However, you should also expect and be looking for ways to refine your deployment model based on the changing/emerging needs of customers, shifts in the market, and new business strategies. Worry less about getting it right on day one and just expect that you'll be using data from your environment to

find opportunities to refactor your deployment model. A resource that started out as a dedicated resource, might end up switched to a shared resource based on consumption, scaling, and cost considerations. A new tier might have you offering some parts of your system in a dedicated model. Being data driven and adaptable are all part of the multi-tenant technical experience.

Introducing the Silo and Pool Models

As we look at deployment models, we're going to discover that these models will require the introduction of new terminology that can add precision to how we characterize SaaS architecture constructs. This relates to our earlier exploration of how the term multi-tenant had to take on a broader meaning to fit the realities of SaaS businesses. Now, as we start to look at deployment models, you'll notice that we still need terminology that can better capture and accurately convey how the resources in our architecture are consumed by tenants.

There are two terms that I'm going to introduce here to give us a more granular way to think about classifying dedicated and shared resources. Across the rest of this book, you'll see that I will use the term *silo* to refer to any model where a resource is dedicated to a given tenant. I'll use the term *pool* to reference any model where a tenant resource is shared by one or more tenants.

This may seem like a subtle nuance in language. In reality, it has significant implications on how we describe multi-tenant architecture. It allows us to describe the behavior and scope of our SaaS architecture resources without the ambiguity and legacy baggage that comes with labeling resources a multi-tenant. As we look more at deployment models and the full range of SaaS architecture concepts that span this book, I will be using silo and pool as the foundational terms that characterize the usage, isolation, deployment, and consumption of the resources in our multi-tenant architecture.

To help crystallize this concept, let's look at a conceptual architecture that includes resources that are being consumed in a combination of dedicated and shared models. Figure 3-2 provides a view of a series of microservices that have been deployed into a SaaS architecture. In this image, I've created a hypothetical environment where we have a series of microservices that are using different strategies for dedicating and sharing tenant resources.

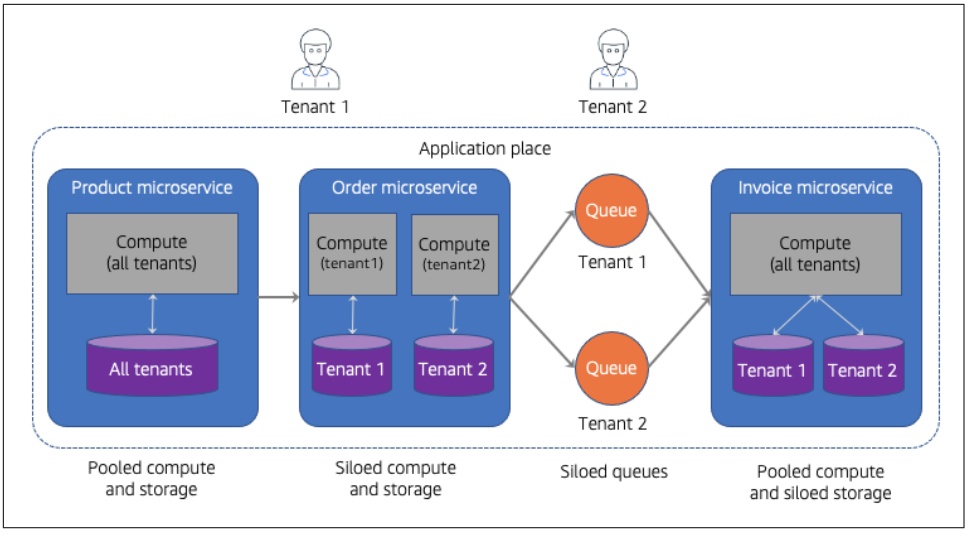


Figure 3-2. Silo and pooled resource models

At the top of the diagram, you'll see that I've put two tenants here to illustrate how the tenants in our environment are landing in and consuming resources. These tenants are running an ecommerce application that is implemented via Product, Order, and Invoice microservices. Now, if we follow a path through these microservices from left to right, you'll see how we've applied different deployment strategies for each of these microservices.

Let's start with the Product microservice. Here, I've chosen a strategy where the compute and the storage for all of our tenants will be deployed in a pooled model. For this scenario, I have decided that the isolation and performance profile of this service fits best with the values of a pooled approach. As we move to the Order microservice, you'll see that I've chosen a very different model here. In this case, the service has siloed compute and storage for every tenant. Again, this was done based on the specific needs of my environment. This could have been driven by some SLA requirement or, perhaps, a compliance need.

From the Order service, you'll then see that our system sends a message to a queue that prepares these orders for billing. This scenario is included to highlight the fact that our siloed and pooled concepts are extended beyond our microservices and applied to any resource that might be part of our environment. For this solution, I've opted to have a siloed queues for each tenant. Finally, I have an Invoice service on the right-hand side that pulls messages from these queues and generates invoices. To meet the requirements of our solution, I've used a mix of siloed and pooled models in this microservice. Here, the compute is pooled and the storage is siloed.

The key takeaway here is that the terms silo and pool are used to generally characterize the architecture footprint of one or more resources. These terms can be applied in a very granular fashion, highlighting how tenancy is mapped to very specific elements of your architecture. These same terms can also be used more broadly to describe how a collection of resources are deployed for a tenant. So, don't try to map silo and pool to specific constructs. Instead, think of them as describing the tenancy of a single resource or a group of resources.

This caveat will be especially important as we look at deployment models throughout this chapter, allowing us to apply to silo and pool concepts and varying scopes across our multi-tenant architecture.

Full Stack Silo Deployment

Now that youhave a high-level sense of the scope and role of deployment models, it's time to dig in a bit more and start looking at defining specific types of deployment models. Let's start by looking at what I'll label as a full stack silo deployment model.

As its name suggests, the full stack silo model places each tenant into a fully siloed environment where all of the resources of a tenant are completely siloed. The diagram in Figure 3-3 provides an example of a full stack silo environment. Here you'll see that we have an environment where our application plane is running workloads for two tenants. These two tenants are running in siloes where the compute, storage, and every resource that's needed for the tenant is deployed into some logical construct that creates a clear boundary between our tenants.

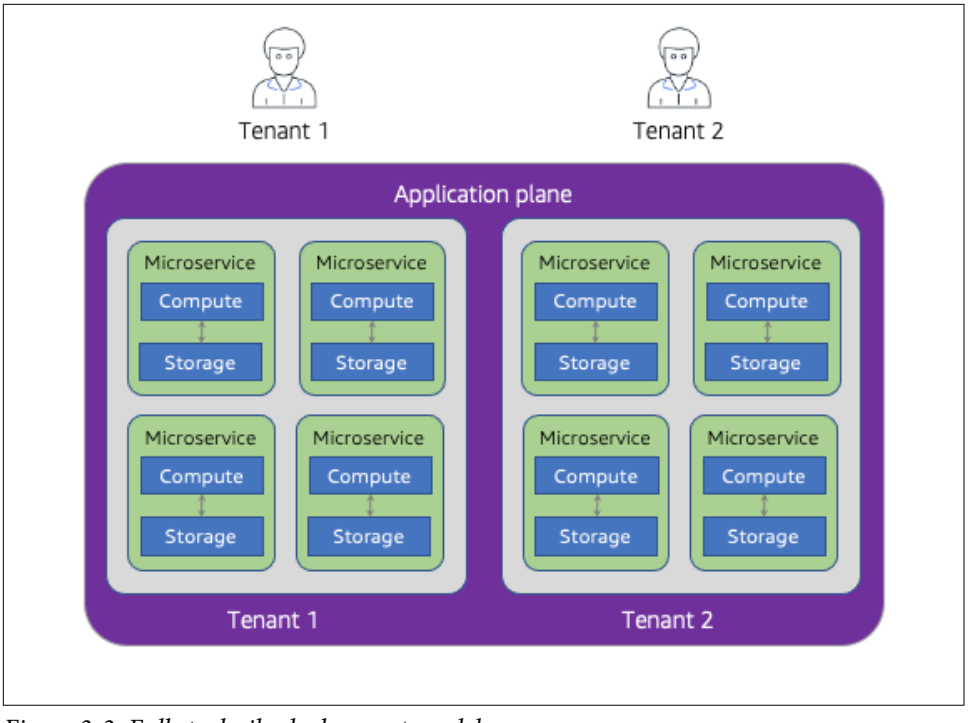


Figure 3-3. Full stack silo deployment model

In this particular example, I simplified the contents of the slilo, showing a range of microservices that are running in each tenant environment. In reality, what's in the silo and how your application is represented could be represented by any number of different technologies and design strategies. This could have been an n-tier environment with separate web, application, and storage tiers. I could have included any number of different compute models and other services here as well (queues, object storage, messaging, and so on). The emphasis, at this stage, is less about what's in each of these tenant environments and more on the nature of how they are deployed.

Where Full Stack Silo Fits

The full stack silo model can feel like a bit of a SaaS anti-pattern. After all, so much of our discussion of SaaS is centered around agility and efficiency. Here, where we have fully siloed tenant resources, it can appear as though we've compromised on some of the fundamental goals of SaaS. However, if you think back to the definition of SaaS in Chapter 1, you'll recall that SaaS isn't exclusively about sharing infrastructure for economies of scale. SaaS is about operating in a model where all of our tenants are operated, managed, and deployed collectively. This is the key thing to keep in mind when you're looking at the full stack silo model. Yes, it has efficiency challenges. We'll get into those. At the same time, as long as every one of these environments is the

same and as long as these are running the same version of our application, then we can still realize much of the value proposition of SaaS.

So, knowing that full stack silo meets our criteria for SaaS, the real question here is more about when it might make sense for you to employ this model. Which factors typically steer organizations toward a full stack silo experience? When might it be a fit for the business and technology realities of your environment? While, there are no absolutes here, there are common themes and environmental factors that have teams selecting the full stack silo model. Compliance and legacy considerations are two of the typical reasons teams will end up opting for full stack silo footprint. In some heavily regulated domains, teams may choose a full stack silo model to simplify their architecture and make it easier for them to address specific compliance criteria. Customers in these domains might also have some influence on the adoption of a full stack silo, insisting on having siloed resources as part of selecting a SaaS solution.

The full stack silo model can also represent a good fit for organizations that are migrating a legacy solution to SaaS. The fully siloed nature of this model allows these organizations to move their existing code into a SaaS model without major refactoring. This gets them to SaaS faster and reduces their need to more immediately take on adding tenancy to all the moving parts of their architecture. Migrating teams will still be required to retrofit your legacy environment to align with the SaaS control plane, its identity model, and a host of other multi-tenant considerations. However, the scope and reach of these impacts can be less pronounced if your solution is moving into a full stack silo environment that doesn't need to consider scenarios where any of a tenant's resources are pooled.

Full stack silo can also be a tiering strategy. For example, some organizations may offer a premium version of their solution that, for the right price, will offer tenants a fully dedicated experience. It's important to note that this dedicated experience is not created as a one-off environment for these tenants. It's still running the same version of the application and is centrally managed alongside all the other tiers of the system.

In some cases, the full stack model simply represents a lower barrier of entry for teams–especially those that may not be targeting a large number of tenants. For these organizations, full stack silo allows them to get to SaaS without tackling some of the added complexities that come with building, isolating, and operating a pooled environment. Of course, these teams also have to consider how adoption of a full stack silo model might impact their ability to rapidly scale the business. In this case, the advantages of starting with a full stack could be offset by the inefficiencies and margin impacts of being in a full stack silo model.

Full Stack Silo Considerations

Teams that opt for a full stack silo model, will need to consider some of the nuances that come with this model. There are definitely pros and cons to this approach that

you'll want to add to your mental model when selecting this deployment model. The sections that follow provide a breakdown of some of the key design, build, and deployment considerations that are associated with the full stack silo deployment model.

Control Plane Complexity

As you may recall, I have described all SaaS architectures as having control and application planes where our tenant environments live in the application plane and are centrally managed by the control plane. Now, with the full stack silo model, you have to consider how the distributed nature of the full stack model will influence the complexity of your control plane.

In Figure 3-4, you can see an example of a full stack silo deployment that highlights some of the elements that come with building and managing this model. Since our solution is running in a silo per tenant model, the application plane must support completely separate environments for each tenant. Instead of interacting with a single, shared resource, our control plane must have some awareness of each of these tenant siloes. This inherently adds complexity to our control plane, which now must be able to operate each of these separate environments.

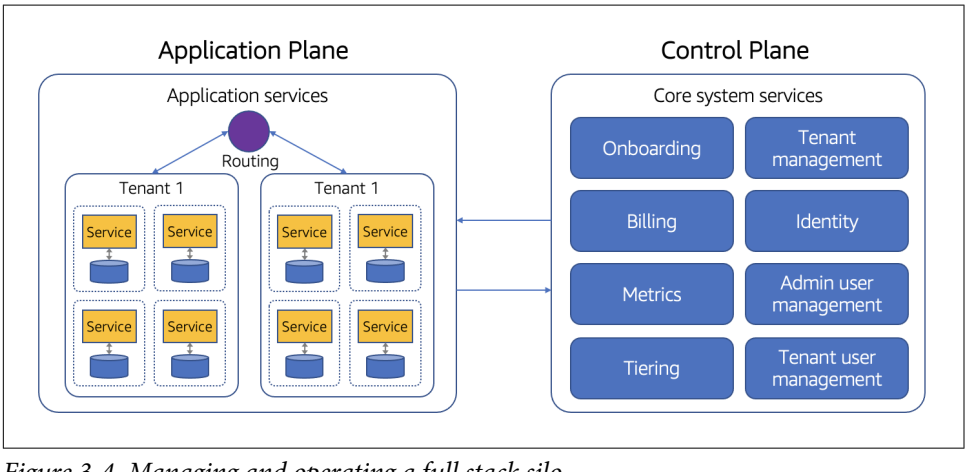


Figure 3-4. Managing and operating a full stack silo

Imagine implementing tenant onboarding in this example. The addition of each new tenant to this environment must fully provision and configure each siloed tenant environment. This also complicates any tooling that may need to monitor and manage the health of tenant environments. Your control plane code must know where each tenant environment can be found and be authorized to access each tenant silo. Any operational tooling that you have that is used to manage infrastructure resources will also need to deal with the larger, more distributed footprint of these tenant environments, which could make troubleshooting and managing infrastructure resources

more unwieldy. Any tooling you've created to centralized metrics, logs, and analytics for your tenants will also need to be able to aggregate the data from these separate tenant environments. Deployment of application updates is also more complicated in this model. Your DevOps code will have to roll out updates to each tenant silo.

There are likely more complexities to cover here. However, the theme is that the distributed nature of the full stack silo model touches many of the moving parts of your control plane experience, adding complexity that may not be as pronounced in other deployment models. While the challenges here are all manageable, it definitely will take extra effort to create a fully unified view of management and operations for any full stack silo environment.

Scaling Impacts

Scale is another important consideration for the full stack silo deployment model. Whenever you're provisioning separate infrastructure for each tenant, you need to think about how this model will scale as you add more tenants. While the full stack silo model can be appealing when you have 10 tenants, its value can begin to erode as you consider supporting hundreds or thousands of tenants. The full stack silo model, for example, would not be sustainable in many classic BC2 environments where the scale and number of tenants would be massive. Naturally, the nature of your architecture would also have some influence on this. If you're running Kubernetes, for example, it might come down to how effectively the silo constructs of Kubernetes would scale here (clusters, namespaces, etc.). If you're using separate cloud networking or account constructs for each siloed tenant, you'd have to consider any limits and constraints that might be applied by your cloud provider.

The broader theme here is that full stack siloed deployments are not for everyone. As I get into specific full stack silo architectures, you'll see how this model can run into important scaling limits. More importantly, even if your environment can scale in a full stack silo model, you may find that there's a point at which the full stack silo can become difficult to manage. This could undermine your broader agility and innovation goals.

Cost Considerations

Costs are also a key area to explore if you're looking at using a full stack silo model. While there are measures you can take to limit over-provisioning of siloed environments, this model does put limits on your ability to maximize the economies of scale of your SaaS environment. Typically, these environments will require lots of dedicated infrastructure to support each tenant and, in some cases, this infrastructure may not have an idle state where it's not incurring costs. For each tenant, then, you will have some baseline set of costs for tenants that you'll incur–even if there is no load on the system. Also, because these environments aren't shared, we don't get the efficiencies that would come with distributing the load of many tenants across shared infrastruc-

ture that scales based on the load of all tenants. Compute, for example, can scale dynamically in a silo, but it will only do so based on the load and activity of a single tenant. This may lead to some over-provisioning within each silo to prepare for the spikes that may come from individual tenants.

Generally, organizations offering full stack siloed models are required to create cost models that help overcome the added infrastructure costs that come with this model. That can be a mix of consumption and some additional fixed fees. It could just be a higher subscription price. The key here is that, while the full stack silo may be the right fit for some tiers or business scenarios, you'll still need to consider how the siloed nature of this model will influence the pricing model of your SaaS environment.

As part of the cost formulas, we must also consider how the full stack silo model impacts the operational efficiency of your organization. If you've built a robust control plane and you've automated all the bits of your onboarding, deployment, and so on, you can still surround your full stack silo model with a rich operational experience. However, there is some inherent complexity that comes with this model that will likely add some overhead to your operational experience. This might mean that you will be required to invest more in the staff and tooling that's needed to support this model, which will add additional costs to your SaaS business.

Routing Considerations

In Figure 3-4, I also included a conceptual placeholder for the routing of traffic within our application plane. With a full stack silo, you'll need to consider how the traffic will be routed to each silo based on tenant context. While there are any number of different networking constructs that we can use here to route this load, you'll still need to consider how this will be configured. Are you using subdomains for each tenant? Will you have a shared domain with the tenant context embedded in each request? Each strategy you choose here will require some way for your system to extract that context and route your tenants to the appropriate silo.

The configuration of this routing construct must be entirely dynamic. As each new tenant is onboarded to your system, you'll need to update the routing configuration to support routing this new tenant to its corresponding silo. None of this is wildly hard, but you'll see that this is an area that will need careful consideration as you design your full stack siloed environment. Each technology stack will bring its own set of considerations to the routing problem.

Availability and Blast Radius

The full stack silo model does offer some advantages when it comes to the overall availability and durability of your solution. Here, with each tenant in its own environment, there is potential to limit the blast radius of any potential operational issue. The

dedicated nature of the silo model gives you the opportunity to contain some issues to individual tenant environments. This can certainly have an overall positive effect on the availability profiles of your service.

Rolling out new releases also behaves a bit differently in siloed environments. Instead of having your release pushed to all customers at the same time, the full stack silo model may release to customers in waves. This can allow you to detect and recover from issues related to a deployment before it is released to the entire population. It, of course, also complicates the availability profile. Having to deploy to each silo separately requires you to have a more complicated rollout process that can, in some cases, undermine the availability of your solution.

Simpler Cost Attribution

One significant upside to the full stack silo model is its ability to attribute costs to individual tenants. Calculating cost-per-tenant for multi-tenant environments, as you'll see in Chapter 14, can be tricky in SaaS environments where some or all of a tenant's resources may be shared. Knowing just how much of a shared database or compute resource was consumed by a given tenant is not so easy to infer in pooled environments. However, in a full stack silo model, you won't face these complexities. Since each tenant has its own dedicated infrastructure, it becomes relatively easy to aggregate and map costs to individual tenants. Cloud providers and third-party tools are generally good at mapping costs to individual infrastructure resources and calculating a cost for each tenant.

Full Stack Silo in Action

Now that we have a good sense of the full stack silo model, let's look at some working examples of how this model is brought to life in real-world architecture. As you can imagine, there are any number of ways to implement this model across the various cloud providers, technology stacks, and so on. The nuances of each technology stack adds its own set of considerations to your design and implementation.

The technology and strategy you use to implement your full stack silo model will likely be influenced by some of the factors that were outlined above. They might also be shaped attributes of your technology stack and your domain realities.

The examples here are pulled from my experience building SaaS solutions at Amazon Web Services (AWS). While these are certainly specific to AWS, these patterns have corresponding constructs that have mappings to other cloud providers. And, in some instances, these full stack silo models could also be built in an on-premises model.

The Account Per Tenant Model

If you're running in a cloud environment–which is where many SaaS applications often land–you'll find that these cloud providers have some notion of an account.

These accounts represent a binding between an entity (an organization or individual) and the infrastructure that they are consuming. And, while there's a billing and security dimension to these accounts, our focus is on how these accounts are used to group infrastructure resources.

In this model, accounts are often viewed as the strictest of boundaries that can be created between tenants. This, for some, makes an account a natural home for each tenant in your full stack silo model. The account allows each silo of your tenant environments to be surrounded and protected by all the isolation mechanisms that cloud providers use to isolate their customer accounts. This limits the effort and energy you'll have to expend to implement tenant isolation in your SaaS environment. Here, it's almost a natural side effect of using an account-per-tenant in your full stack silo model.

Attributing infrastructure costs to individual tenants also becomes a much simpler process in an account-per-tenant model. Generally, your cloud provider already has all the built-in mechanisms needed to track costs at the account level. So, with an account-per-tenant model, you can just rely on these ready made solutions to attribute infrastructure costs to each of your tenants. You might have to do a bit of extra work to aggregate these costs into a unified experience, but the effort to assemble this cost data should be relatively straightforward.

In Figure 3-5, I've provided a view of an account-per-tenant architecture. Here, you'll see that I've shown two full stack siloed tenant environments. These environments are mirror images, configured as clones that are running the exact same infrastructure and application services. When any updates are applied, they are applied universally to all tenant accounts.

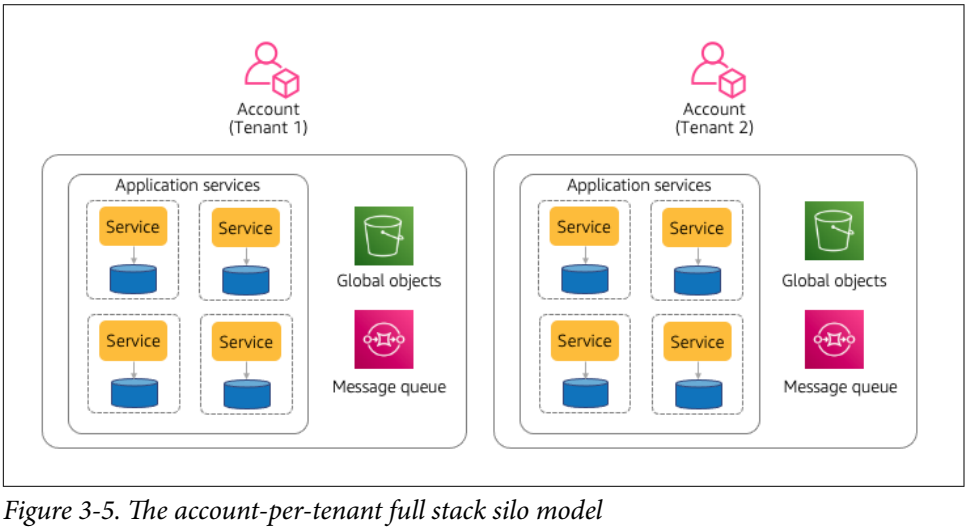


Figure 3-5. The account-per-tenant full stack silo model

Within each account, you'll see examples of the infrastructure and services that might be deployed to support the needs of your SaaS application. There are placeholders here to represent the services that support the functionality of your solution. To the right of these services, I also included some additional infrastructure resources that are used within our tenant environments. Specifically, I put an object store (Amazon Simple Storage Service) and a managed queue service (Amazon's Simple Queue Services). The object store might hold some global assets and the queue is here to support asynchronous messaging between our services. I included these to drive home the point that our account-per-tenant silo model will typically encapsulate all of the infrastructure that is needed to support the needs of a given tenant.

Now, the question is: does this model mean that infrastructure resources cannot be shared between our tenant accounts? For example, could these two tenants be running all of their microservices in separate accounts and share access to a centralized identity provider? This wouldn't exactly be unnatural. The choices you make here are more driven by a combination of business/tenant requirements as well as the complexities associated with accessing resources that are outside the scope of a given account.

Let's be clear. full stack silo, I'm still saying that the application functionality of your solution is running completely in its own account. The only area here where we might allow something to be outside of the account is when it plays some more global role in our system. Here, let's imagine the object store represented a globally managed construct that held information that was centrally managed for all tenants. In some cases, you may find one-off reasons to have some bits of your infrastructure running in some shared model. However, anything that is shared cannot have an impact on the performance, compliance, and isolation requirements of our full stack silo experience. Essentially, if you create some centralized, shared resource that impacts the rationale for adopting a full stack silo model, then you've probably violated the spirit of using this model.

The choices you make here should start with assessing the intent of your full stack silo model. Did you choose this model based on an expectation that customers would want *all* of their infrastructure to be completely separated from other tenants? Or, was it more based on a desire to avoid noisy neighbor and data isolation requirements? Your answers to these questions will have a significant influence on how you choose to share parts of your infrastructure in this model.

If your code needs to access any resources that are outside of your account, this can also introduce new challenges. Any externally accessed resource would need to be running within the scope of some other account. And, as a rule of thumb, accounts have very intentional and hard boundaries to secure the resources in each account. So, then, you'd have to wander into the universe of authorizing cross-account access

to enable your system to interact with any resource that lives outside of a tenant account.

Generally, I would stick with the assumption that, in a full stack silo model, your goal is to have all tenant's resources in the same account. Then, only when there's a compelling reason that still meets the spirit of your full stack silo, consider how/if you might support any centralized resources.

Onboarding Automation. The account-per-tenant silo model adds some additional twists to the onboarding of new tenants. As each new tenant is onboarded (as we'll see in Chapter 4), you will have to consider how you'll automate all the provisioning and configuration that comes with introducing a new tenant. For the account-per-tenant model, our provisioning goes beyond the creation of tenant infrastructure–it also includes the creation of new accounts.

While there are definitely ways to automate the creation of accounts, there are aspects of the account creation that can't always be fully automated. In cloud environments, however, there are some intentional constraints here that may restrict your ability to automate the configuration or provisioning of resources that may exceed the default limits for those resources. For example, your system may rely on a certain number of load balancers for each new tenant account. However, the number you require for each tenant may exceed the default limits of your cloud provider. Now, you'll need to go through the processes, some of which may not be automated, to increase the limits to meet the requirements of each new tenant account. This is where your onboarding process may not be able to fully automate every step in a tenant onboarding. Instead, you may need to absorb some of the friction that comes with using the processes that are supported by your cloud provider.

While teams do their best to create clear mechanisms to create each new tenant account, you may just need to allow for the fact that, as part of adopting an account-per-tenant model, you'll need to consider how these potential limit issues might influence your onboarding experience. This might mean creating different expectations around onboarding SLAs and better managing tenant expectations around this process.

Scaling Consideration. I've already highlighted some of the scaling challenges that are typically associated with the full stack silo model. However, with the account-per-tenant model, there's another layer to the full stack silo scaling story.

Generally speaking, mapping accounts to tenants could be viewed as a bit of an anti-pattern. Accounts, for many cloud providers, were not necessarily intended to be used as the home for tenants in multi-tenant SaaS environments. Instead, SaaS providers just gravitated toward them because they seemed to align well with their goals. And, to a degree, this makes perfect sense.

Now, if you have an environment with 10s of tenants, you may not feel much of the pain as part of your account-per-tenant model. However, if you have plans to scale to a large number of tenants, this is where you may begin to hit a wall with the account-per-tenant model. The most basic issue you can face here is that you may exceed the maximum number of accounts supported by your cloud provider. The more subtle challenge here shows up over time. The proliferation of accounts can end up undermining the agility and efficiency of your SaaS business. Imagine having hundreds or thousands of tenants running in this model. This will translate into a massive footprint of infrastructure that you'll need to manage. While you can take measures to try to streamline and automate your management and operation of all these accounts, there could be points at which this may no longer be practical.

So, where is the point of no return? I can't say there's an absolute data point at which the diminishing returns kicks in. So much depends on the nature of your tenant infrastructure footprint. I mention this mostly to ensure that you're factoring this into your thinking when you take on an account-per-tenant model.

The VPC-Per-Tenant Model

The account-per-tenant model relies on a pretty coarse-grained boundary. Let's shift our focus to constructs that realize a full stack silo within the scope of a single account. This will allow us to overcome some of the challenges of creating accounts for individual tenants. The model we'll look at now, the Virtual Private Cloud (VPC)-Per-Tenant Model, is one that relies more on networking constructs to house the infrastructure that belongs to each of our siloed tenants.

Within most cloud environments you're given access to a fairly rich collection of virtualized networking constructs that can be used to construct, control, and secure the footprint of your application environments. These networking constructs provide natural mechanisms for implementing a full stack siloed implementation. The very nature of networks and their ability to describe and control access to their resources provides SaaS builders with a powerful collection of tools that can be used to silo tenant resources.

Let's look at an example of how a sample networking construct can be used to realize a full stack silo model. Figure 3-6 provides a look at a sample network environment that uses Amazon's VPC to silo tenant environments.

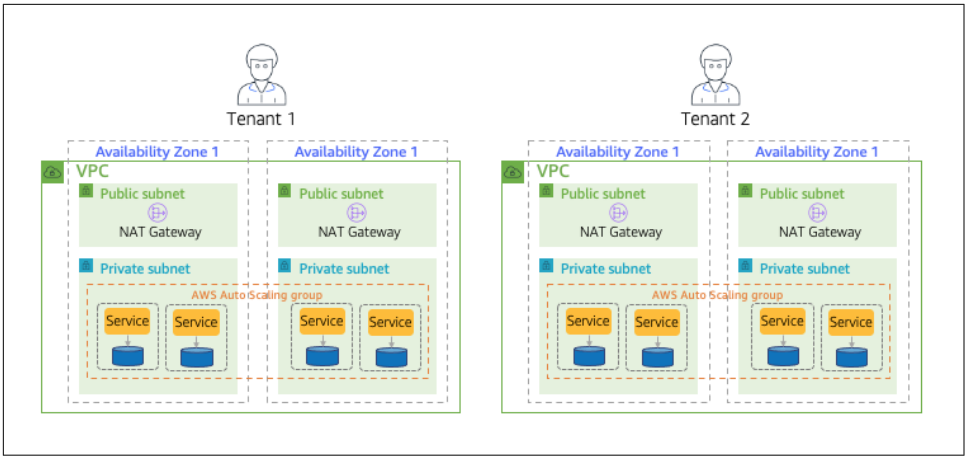


Figure 3-6. The VPC-per-tenant full stack silo model

At first glance, there appears to be a fair amount of moving parts in this diagram. While it's a tad busy, I wanted to bring in enough of the networking infrastructure to give you a better sense of the elements that are part of this model.

You'll notice that, at the top of this image, we have two tenants. These tenants are accessing siloed application services that are running in separate VPCs. The VPC is the green box that is at the outer edge of our tenant environments. I also wanted to illustrate the high availability footprint of our VPC by having it include two separate availability zones (AZs). We won't get into AZs, but just know that AZs represent distinct locations within an AWS Region that are engineered to be isolated from failures in other AZs. We also have separate subnets here to separate the public and private subnets of our solution. Finally, you'll see the application services of our solution deployed into private subnets of our two AZs. These are surrounded by what AWS labels as an Auto Scaling Group, which allows our services to dynamically scale based on tenant load.

I've included all these network details to highlight the idea that we're running our tenants in the network siloes that offer each of our tenants a very isolated and resilient networking environment that leans on all the virtualized networking goodness that comes with building and deploying your solution in a VPC-pert-tenant siloed model.

While this model may seem less rigid than the account-per-tenant model, it actually provides you with a solid set of constructs for preventing any cross-tenant access. You can imagine how, as part of their very nature, these networking tools allow you to create very carefully controlled ingress and egress for your tenant environments. We won't get into the specifics, but the list of access and flow control mechanisms that are available here is extensive. More details can be found here.

Another model that shows up here, occasionally, is the subnet-per-tenant model. While I rarely see this model, there are some instances where teams will put each tenant silo in a given subnet. This, of course, can also become unwieldy and difficult to manage as you scale.

Onboarding Automation. With the account-per-tenant model, I dug into some of the challenges that it could create as part of automating your onboarding experience. With the VPC-per-tenant model, the onboarding experience changes some. The good news here is that, since you're not provisioning individual accounts, you won't run into the same account limits automation issues. Instead, the assumption is that the single account that is running our VPCs will be sized to handle the addition of new tenants. This may still require some specialized processes, but they can be applied outside the scope of onboarding.

In the VPC-per-tenant model, our focus is more on provisioning your VPC constructs and deploying your application services. That will likely still be a heavy process, but most of what you need to create and configure can be achieved through a fully automated process.

Scaling Considerations. As with accounts, VPCs also face some scaling considerations. Just as there are limits on the number of accounts you can have, there can also be limits on the number of VPCs that you can have. The management and operation of VPCs can also get complicated as you begin to scale this model. Having tenant infrastructure sprawling across hundreds of VPCs may impact the agility and efficiency of your SaaS experience. So, while VPC has some upsides, you'll want to think about how many tenants you'll be supporting and how/if the VPC-per-tenant model is practical for your environment.

Remaining Aligned on Full Stack Silo Mindset

Before I move on to any new deployment models, it's essential that we align on some key principles in the full stack silo. For some, the allure of the full stack silo model can be appealing because it can feel like it opens (or re-opens) the door for SaaS providers to offer one-off customization to their tenants. While it's true that the full stack silo model offers dedicated resources, this should never be viewed as an opportunity to fall back to the world of per-tenant customization. The full stack silo only exists to accommodate domain, compliance, tiering, and any other business realities that might warrant the use of a full stack silo model.

In all respects, a full stack silo environment is treated the same as a pooled environment. Whenever new features are released, they are deployed to *all* customers. If your infrastructure configuration needs to be changed, that change should be applied to *all* of your siloed environments. If you have policies for scaling or other run-time behaviors, they are applied based on tenant tiers. You should never have a policy which is

applied to an individual tenant. The whole point of SaaS is that we are trying to achieve agility, innovation, scale, and efficiency through our ability to manage and operate our tenants collectively. Any drift toward a one-off model will slowly take you away from those SaaS goals. In some cases, organizations that moved to SaaS to maximize efficiency will end up regressing through one-off customizations that undermine much of the value they hoped to get out of a true SaaS model.

The guidance I always offer to drive this point home centers around how you arrive at a full stack silo model. I tell teams that–even if you're targeting a full stack silo as your starting point–you should build your solution as if it were going to be a full stack pooled model. Then, treat each full stack silo as an instance of your pooled environment that happens to have a single tenant. This serves as a forcing function that allows the full stack siloed environments to inherit the same values that are applied to a full stack pool (which we're covering next).

The Full Stack Pool Model

The full stack pool model, as its name suggests, represents a complete shift from the full stack silo mindset and mechanisms we've been exploring. With the full stack pool model, we'll now look at SaaS environments where all of the resources for our tenants are running in a shared infrastructure model.

For many, the profile of a fully pooled environment maps to their classic notion of multi-tenancy. It's here where the focus is squarely on achieving economies of scale, operational efficiencies, cost benefits, and a simpler management profile that are the natural byproducts of a shared infrastructure model. The more we are able to share infrastructure resources, the more opportunities we have to align the consumption of those resources with the activity of our tenants. At the same time, these added efficiencies also introduce a range of new challenges.

Figure 3-7 provides a conceptual view of the full stack silo model. You'll see that I've still included that control plane here just to make it clear that the control plane is a constant across any SaaS model. On the left of the diagram is the application plane, which now has a collection of application services that are shared by all tenants. The tenants shown at the top of the application plane are all accessing and invoking operations on the application microservices and infrastructure.

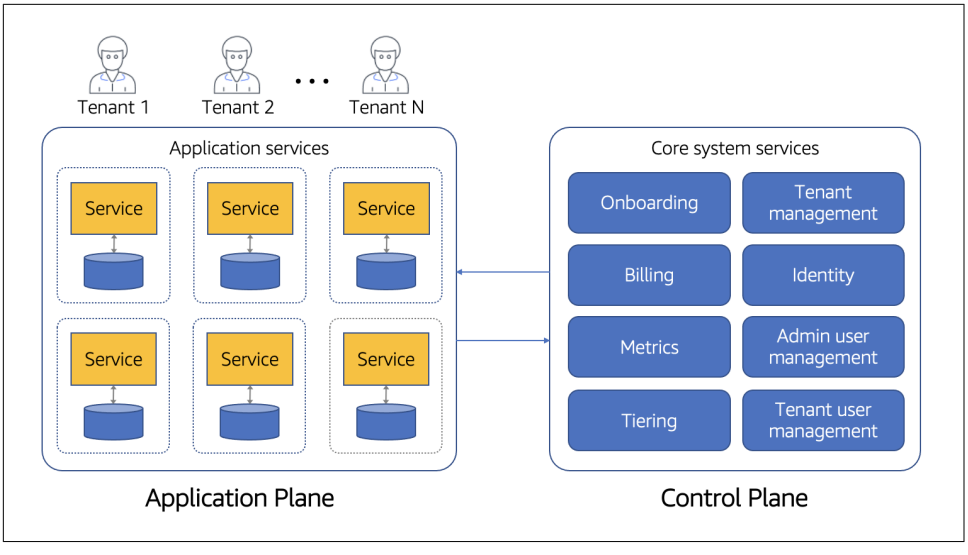


Figure 3-7. A full stack pooled model

Now, within this pool model, tenant context plays a much bigger role. In the full stack silo model, tenant context was primarily used to route tenants to their dedicated stack. Once a tenant lands in a silo, that silo knows that all operations within that silo are associated with a single tenant. With our full stack pool, however, this context is essential to every operation that is performed. Accessing data, logging messages, recording metrics–all of these operations will need to resolve the current tenant context at run-time to successfully complete their task.

Figure 3-8 gives you a better sense of how tenant context touches every dimension of our infrastructure, operations, and implementation are influenced by this tenant context in a pooled model. This conceptual diagram highlights how each microservice must acquire tenant context and apply it as part of its interactions with data, the control plane, and other microservices. You'll see tenant context being acquired and applied as we send billing events and metrics data to the control plane. You'll see it injected in your call to downstream microservices. It also shows up in our interaction with data.

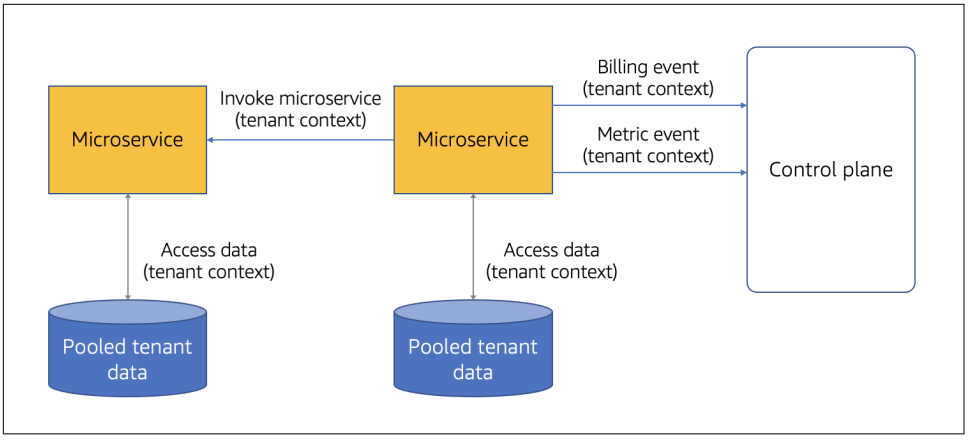


Figure 3-8. Tenant context in the full stack pooled environment

The fundamental idea here is that, when we have a pooled resource, that resource belongs to multiple tenants. As a result, tenant context is needed to apply scope and context to each operation at run-time.

Now, to be fair, tenant context is valid across all SaaS deployment models. Silo still needs tenant context as well. What's different here is that the silo model knows its binding to the tenant at the moment it's provisioned and deployed. So, for example, I could associate an environment variable as the tenant context for a siloed resource (since its relationship to the tenant does not change at run-time). However, a pooled resource is provisioned and deployed for all tenants and, as such, it must resolve its tenant context based on the nature of each request it processes.

As we dig deeper into more multi-tenant implementation details, we'll discover that these differences between silo and pool models can have a profound impact on how we architect, deploy, manage, and build the elements of our SaaS environment.

Full Stack Pool Considerations

The full stack pool model also comes with a set of considerations that might influence how/if you choose to adopt this model. In many respects, the considerations for the full stack pool model are the natural inverse of the full stack silo model. Full stack pool certainly has strengths that are appealing to many SaaS providers. It also presents a set of challenges that come with having shared infrastructure. The sections that follow highlight these considerations.

Scale

Our goal in multi-tenant environments is to do everything we can to align infrastructure consumption with tenant activity. In an ideal scenario, your system would, at a

given moment in time, only have enough resources allocated to accommodate the current load being imposed by tenants. There would be zero over-provisioned resources. This would let the business optimize margins and ensure that the addition of new tenants would not drive a spike in costs that could undermine the bottom line of the business.

This is the dream of the full stack pooled model. If your design was somehow able to fully optimize the scaling policies of your underlying infrastructure in a full stack pool, you would have achieved multi-tenant nirvana. This is not practical or realistic, but it is the mindset that often surrounds the full stack pooled model. The reality, however, is that creating a solid scaling strategy for a full stack pooled environment is very challenging. The loads of tenants are often constantly changing and new tenants may be arriving every day. So, the scaling strategy that worked yesterday, may not work today. What typically happens here is teams will accept some degree of over-provisioning to account for this continually shifting target.

The technology stack you choose here can also have a significant impact on the scaling dynamics of your full stack pool environment. In Chapter 12 we'll look at a serverless SaaS architecture and get a closer look at how using serverless technologies can simplify your scale story and achieve better alignment between infrastructure consumption and tenant activity.

The key theme here is that, while there are significant scaling advantages to be had in a full stack pooled model, the effort to make this scaling a reality can be challenging to fully realize. You'll definitely need to work hard to craft a scaling strategy that can optimize resource utilization without impacting tenant experience.

Isolation

In a full stack siloed model, isolation is a very straightforward process. When resources run in a dedicated model, you have a natural set of constructs that allow you to ensure that one tenant cannot access the resources of another tenant. However, when you start using pooled resources, your isolation story tends to get more complicated. How do you isolate a resource that is shared by multiple tenants? How is isolation realized and applied across all the different resource types and infrastructure services that are part of your multi-tenant architecture? In Chapter 10, we'll dig into the strategies that are used to address these isolation nuances. However, it's important to note that, as part of adopting a full stack pool model, you will be faced with a range of new isolation considerations that may influence your design and architecture. The assumption here is that the economies of scale and efficiencies of the pooled model offset any of the added overhead and complexity associated with isolating pooled resources.

Availability and Blast Radius

In many respects, a full stack pool model represents an all-in commitment to a model that places all the tenants of your business into a shared experience. Any outage or issues that were to show up in a full stack pool environment are likely to impact *all* of your customers and could potentially damage the reputation of your SaaS business. There are examples across the industry of SaaS organizations that have had service outages that created a flurry of social media outcry and negative press that had a lasting impact on these businesses.

As you consider adopting a full-stack pool model, you need to understand that you're committing to a higher DevOps, testing, and availability bar that makes every effort to ensure that your system can prevent, detect, and rapidly recover from any potential outage. It's true that every team should have a high bar for availability. However, the risk and impact of any outage in a full stack pool environment demands a greater focus on ensuring that your team can deliver a zero downtime experience. This includes adopting best-of-breed CI/CD strategies that allow you to release and roll-back new features on a regular basis without impacting the stability of your solution.

Generally, you'll see full stack pool teams leaning into fault tolerant strategies that allow their microservices and components to limit the blast radius of localized issues. Here, you'll see greater application of asynchronous interactions between services, fallback strategies, and bulkhead patterns being used to localize and manage potential microservice outages. Operational tooling that can proactively identify and apply policies here is also essential in a full stack pool environment.

It's worth noting that these strategies apply to any and all SaaS deployment models. However, the impact of getting this wrong in a full stack pool environment can be much more significant for a SaaS business.

Noisy Neighbor

Full stack pooled environments rely on carefully orchestrated scaling policies that ensure that your system will effectively add and remove capacity based on the consumption activity of your tenants. The shifting needs of tenants along with the potential influx of new tenants means that the scaling policies you have today may not apply tomorrow. While teams can take measures to try and anticipate these tenant activity trends, many teams find themselves over-provisioning resources that create the cushion needed to handle the spikes that may not be effectively addressed through your scaling strategies.

Every multi-tenant system must employ strategies that will allow them to anticipate spikes and address what is referred to as noisy neighbor conditions. However, noisy neighbor takes on added weight in full stack pooled environments. Here, where essentially everything is shared, the potential for noisy neighbor conditions is much higher. You must be especially careful with the sizing and scaling profile of your

resources since everything must be able to react successfully to shifts in tenant consumption activity. This means accounting for and building defensive tactics to ensure that one tenant isn't saturating your system and impacting the experience of other tenants.

Cost Attribution

Associating and tracking costs at the tenant level is a much more challenging proposition in a full stack pooled environment. While many environments give you tools to map tenants to specific infrastructure resources, they don't typically support mechanisms that allow you to attribute consumption to the individual tenants that are consuming a shared resource. For example, if three tenants are consuming a compute resource in a multi-tenant setting, I won't typically have access to tools or mechanisms that would let me determine what percentage of that resource was consumed by each tenant at a given moment in time. We'll get into this challenge in more detail in Chapter 14. The main point here is that, with the efficiency of a full stack pooled model also comes new challenges around understanding the cost footprint of individual tenants.

Operational Simplification

I've talked about this need for a single pane of glass that provides a unified operational and management view of your multi-tenant environment. Building this operational experience requires teams to ingest metrics, logs, and other data that can be surfaced in this centralized experience. Creating these operational experiences in a full stack pooled environment tends to be a simpler experience. Here, where all tenants are running in shared infrastructure, I can more easily assemble an aggregate view of my multi-tenant environment. There's no need to connect with one-off tenant infrastructure and create paths for each of those tenant-specific resources to publish data to some aggregation mechanism.

Deployment is also simpler in the full stack pooled environment. Releasing a new version of a microservice simply means deploying one instance of that service to the pooled environment. Once it's deployed all tenants are now running on the new version.

A Sample Architecture

As you can imagine, the architecture of a full stack pool environment is pretty straightforward. In fact, on the surface, it doesn't look all that unlike any classic application architecture. Figure 3-9 provides an example of a fully pooled architecture deployed in an AWS architecture.

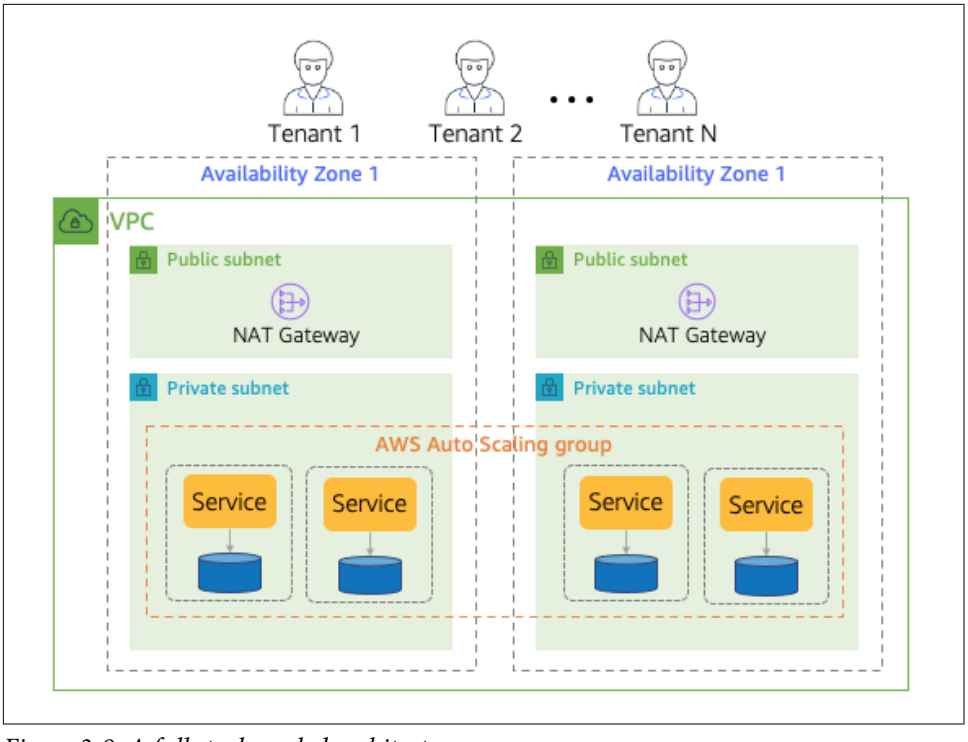


Figure 3-9. A full stack pooled architecture

Here you'll see I've included many of the same constructs that were part of our full stack silo environment. There's a VPC for the network of our environment and it includes two Availability Zones for high availability. WIthin the VPC there are separate private and public subnets that separate the external and internal view of our resources. And, finally, within the private subnet you'll see the placeholders for the various microservices that deliver the server side functionality of our application. These services have storage that is deployed in a pooled model and their compute is scaled horizontally using an auto-scaling group. At the top, of course, we also illustrate that this environment is being consumed by multiple tenants.

Now, in looking at this at this level of detail, you'd be hard-pressed to find anything distinctly multi-tenant about this architecture. In reality, this could be the architecture of almost any flavor of application. Multi-tenancy doesn't really show up in a full stack pooled model as some concrete construct. The multi-tenancy of a pooled model is only seen if you look inside the run-time activity that's happening within this environment. Every request that is being sent through this architecture is accompanied by tenant context. The infrastructure and the services must acquire and apply this context as part of every request that is sent through this experience.

Imagine, for example, a scenario where Tenant 1 makes a request to fetch an item from storage. To process that request, your multi-tenant services will need to extract the tenant context and use it to determine which items within the pooled storage are associated with Tenant 1. As I move through the upcoming chapters, you'll see how this context ends up having a profound influence on the implementation and deployment of these services. For now, though, the key here is to understand that a full stack pooled model relies more on its run-time ability to share resources and apply tenant context where needed.

This architecture represents just one flavor of a full stack pooled model. Each technology stack (containers, serverless, relational storage, NoSQL storage, queues) can influence the footprint of the full stack pooled environment. The spirit of full stack pool remains the same across most of these experiences. Whether you're in a Kubernetes cluster or a VPC, the basic idea here is that the resources in that environment will be pooled and will need to scale based on the collective load of all tenants.

A Hybrid Full Stack Deployment Model

So far, I've mostly presented full stack silo and full stack pool deployment models as two separate approaches to the full stack problem. It's fair to think of these two models as addressing a somewhat opposing set of needs and almost view them as being mutually exclusive. However, if you step back and overlay market and business realities on this problem, you'll see how some organizations may see value in supporting both of these models.

Figure 3-10 provides a view of a sample hybrid full stack deployment model. Here we have the same concepts we covered with full stack silo and pool deployment models sitting side-by-side.

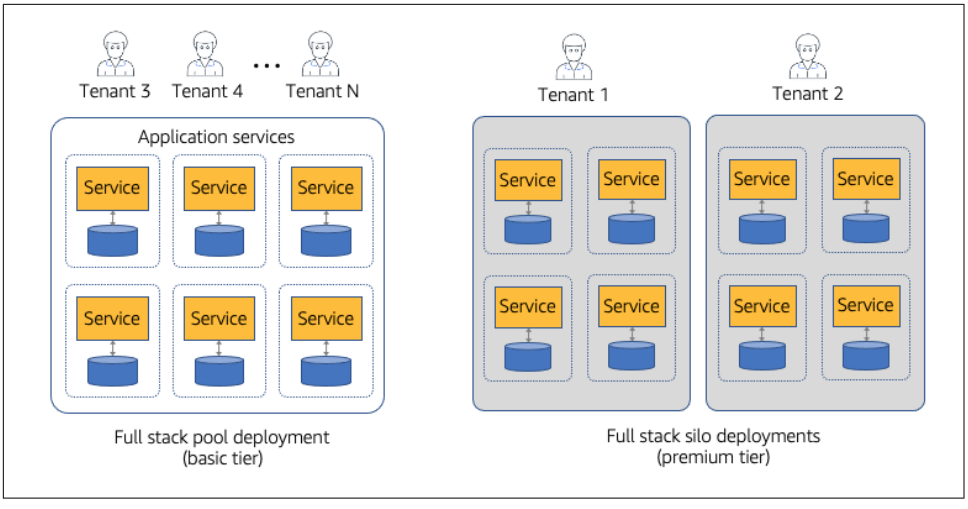


Figure 3-10. A hybrid deployment model

So, why both models? What would motivate adopting this approach? Well, imagine you've built your SaaS business and you started out offering all customers a full stack pooled model (shown on the left here). Then, somewhere along the way, you ran into a customer that was uncomfortable running in a pooled model. They may have noisy neighbor concerns. They may be worried about some compliance issues. Now, you're not necessarily going to cave to every customer that has this pushback. That would undermine much of what you're trying to achieve as a SaaS business. Instead, you're going to make efforts to help customers understand the security, isolation, and strategies you've adopted to address their needs. This is always part of the job of selling a SaaS solution. At the same time, there may be rare conditions when you might be open to offering a customer their own full stack siloed environment. This could be driven by a strategic opportunity or it may be that some customer is willing to write a large check that could justify offering a full stack silo.

In Figure 3-10, you can see how the hybrid full stack deployment model lets you create a blended approach to this problem. On the left-hand side of this diagram is an instance of a full stack pooled environment. This environment supports that bulk of your customers and we label these tenants, in this example, as belonging to your basic tier experience.

Now, for the tenants that demanded a more siloed experience, I have created a new premium tier that allows tenants to have a full stack silo environment. Here we have two full stack siloed tenants that are running their own stacks. The assumption here (for this example) is that these tenants are connected to a premium tier strategy that has a separate pricing model.

For this model to be viable, you must apply constraints to the number of tenants that are allowed to operate in a full stack silo model. If the ratio of siloed tenants becomes too high, this can undermine your entire SaaS experience.

The Mixed-Mode Deployment Model

To this point, I've focused heavily on the full stack models. While it's tempting to view multi-tenant deployments through these more coarse-grained models, the reality is that many systems rely on a much more fine-grained approach to multi-tenancy, making silo and pool choices across the entire surface of their SaaS environment. This is where we look more at what I refer to as a mixed mode deployment model.

With mixed mode deployments, you're not dealing with the heavy absolutes that come with full stack models. Instead, mixed mode allows us to look at the workloads within our SaaS environment and determine how each of the different services and resources within your solution should be deployed to meet the specific requirements of a given use case.

Let's take a simple example. Imagine I have two services in my e-commerce solution. I have an order service that has challenging throughput requirements that are prone to noisy neighbor problems. This same service also stores data that is going to grow significantly and has strict compliance requirements that are hard to support in a pooled model. I also have a ratings service that is used to manage product ratings. It doesn't really face any significant throughput challenges and can easily scale to handle the needs of tenants–even when a single single tenant might be putting a disproportionate load on the service. Its storage is also relatively small and contains data that isn't part of the system's compliance profile.

In this scenario, I can step back and consider these specific parameters to arrive at a deployment strategy that best serves the needs of these services. Here, I might choose to make both the compute and the storage of my order service siloed and the compute and storage of my rating service pooled. There might even be cases where the individual layers of a service could have separate silo/pool strategies. This is the basic point I was making when I was first introducing the notion of silo and pool at the outset of this chapter.

Equipped with this more granular approach to silo and pool strategies, you can now imagine how this might yield much more diverse deployment models. Consider a scenario where you might use this strategy in combination with a tiering model to define your multi-tenant deployment footprint.

The image in Figure 3-11 provides a conceptual view of how you might employ a mixed mode deployment model in your SaaS environment. I've shown a variety of different deployment experiences spanning the basic and premium tier tenants.

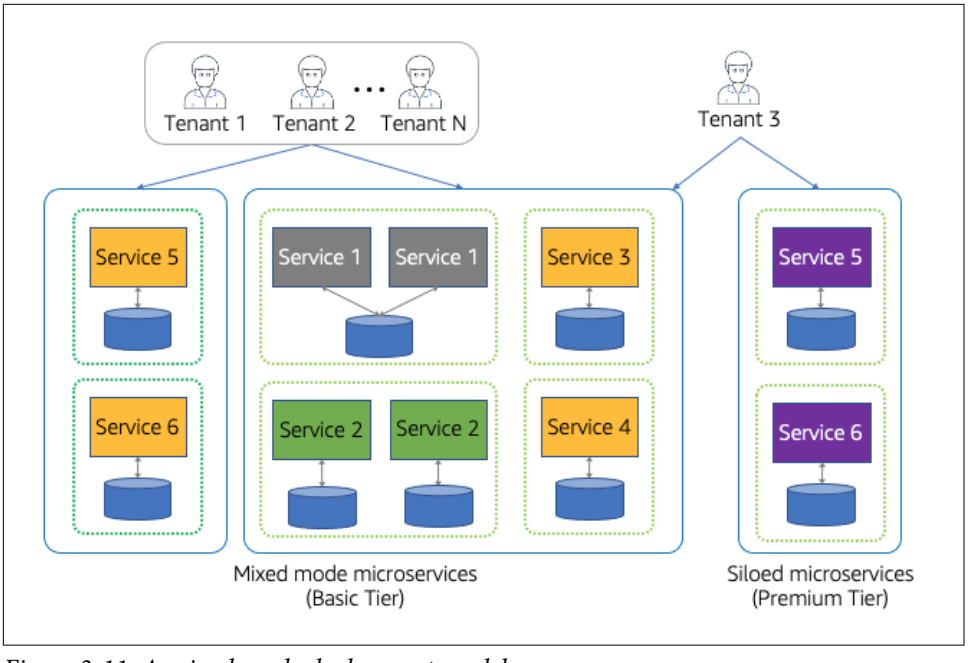


Figure 3-11. A mixed mode deployment model

On the left-hand side of this image, we have our basic tier services. These services cover all the functionality that is needed for our SaaS environment. However, you'll note that they are deployed in different silo/pool configurations. Service 1, for example, has siloed compute and pooled storage. Meanwhile, Service 2 has siloed compute and siloed storage. Services 3-6 are all pooled compute and pooled storage. The idea here is that I've looked across the needs of my pooled tenants and identified, on a service-by-service basis, which silo/pool strategy will best fit the needs of that service. The optimizations that have been introduced here were created as baseline strategies that were core to the experience of *any* tenant using the system.

Now, where tiers do come into play is when you look at what I've done with the premium tier tenants. Here, you'll notice that Services 5 and 6 are deployed in the basic tier and they're also deployed separately for a premium tier tenant. The thought was that, for these services, the business determined that offering these services in a dedicated model would represent value that could distinguish the experience of the system's premium tier. So, for each premium tier tenant, we'll create new deployments of Service 5 and 6 to support the tiering requirements of our tenants. In this particular example, Tenant 3 is a premium tier tenant that consumes a mix of the services on the left and these dedicated instances of Services 5 and 6 on the right.

Approaching your deployment model in this more granular fashion provides a much higher degree of flexibility to you as the architect and to the business. By supporting

silo and pool models at all layers, you have the option to compose the right blend of experiences to meet the tenant, operational, and other factors that might emerge throughout the life of your solution. If you have a pooled microservice with performance issues that are creating noisy neighbor challenges, you could silo the compute and/or storage of the service to address this problem. If your business wants to offer some parts of your system in a dedicated model to enable new tiering strategies, you are better positioned to make this shift.

This mixed mode deployment model, in my opinion, often represents a compelling option for many multi-tenant builders. It allows them to move away from having to approach problems purely through the lens of full stack solutions that don't always align with the needs of the business. Yes, there will always be solutions that use the full stack model. For some SaaS providers, this will be the only way to meet the demands of their market/customers. However, there are also cases where you can use the strengths of the mixed mode deployment model to address this need without moving everything into a full stack silo. If you can just move specific services into the silo and keep some lower profile services in the pool, that could still represent a solid win for the business.

The Pod Deployment Model

So far, I've mostly looked at deployment models through the lens of how you can represent the application of siloed and pooled concepts. We explored coarse- and fine-grained ways to apply the silo and pool model across your SaaS environment. To be complete, I also need to step out of the silo/pool focus and think about how an application might need to support a variation of deployment that might be shaped more by where it needs to land, how it deals with environmental constraints, and how it might need to morph to support the scale and reach of your SaaS business. This is where the pod deployment model comes into the picture.

When I talk about pods here, I'm talking about how you might group a collection of tenants into some unit of deployment. The idea here is that I may have some technical, operational, compliance, scale, or business motivation that pushes me toward a model where I put tenants into individual pods and these pods become a unit of deployment, management, and operation for my SaaS business. Figure 3-12 provides a conceptual view of a pod deployment.

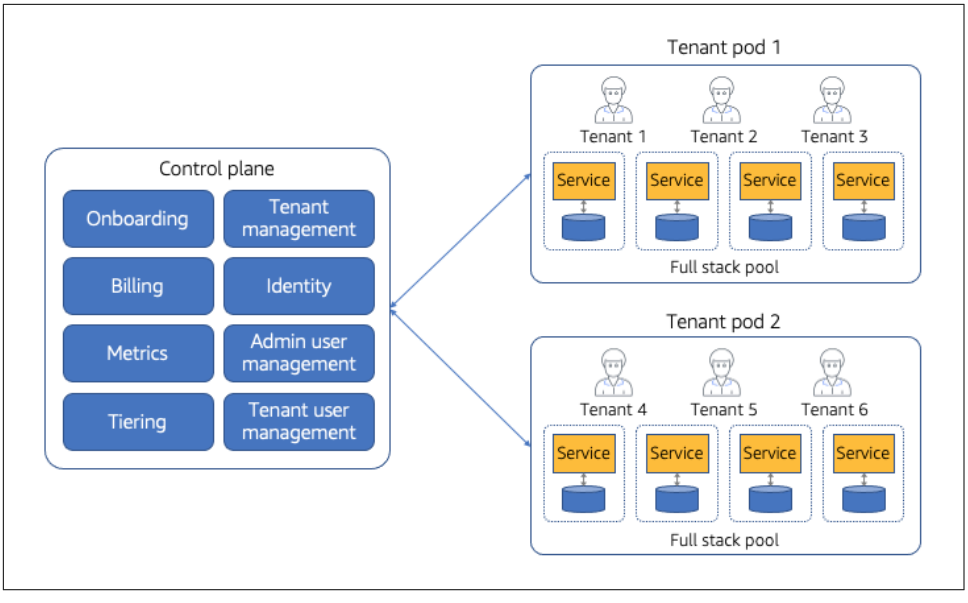


Figure 3-12. A pod deployment model

In this pod deployment model, you'll notice that we have the same centralized control plane on the left-hand side of this experience. Now, however, on the right-hand side, I have included individual pods that are used to represent self-contained environments that support the workload of one or more tenants. In this example, I have Tenants 1-3 in pod 1 and Tenants 4-6 in pod 2.

These separate pods bring a degree of complexity to a SaaS environment, requiring your control to build in the mechanisms to support this distribution model. How tenants are onboarded, for example, must consider which pod a given tenant will land in. Your management and operations must also become pod aware, providing insights into the health and activity of each pod.

There are a number of factors that could drive the adoption of a pod-based delivery model. Imagine, for example, having a full stack pooled model running in the cloud that, at a certain number of tenants, begins to exceed infrastructure limits of specific services. In this scenario, your only option might be to create separate cloud accounts that host different groups of tenants to work around these constraints. This could also be driven by a need for deploying a SaaS product into multiple geographies where the requirements of that geography or performance considerations could tip you toward a pod-based deployment model where different geographies might be running different pods.

Some teams may also use pods as an isolation strategy where there's an effort to reduce cross-tenant impacts. This can be motivated by a need for greater protections

from noisy neighbor conditions. Or, it might play a role in the security and availability story of a SaaS provider.

If you choose to adopt a pod model, you'll want to consider how this will influence the agility of your business. Adopting a pod model means committing to absorbing the extra complexity and automation that allows you to support and manage pods without having any one-off mechanisms for individual pods. To scale successfully, the configuration and deployment of these pods must all be automated through your control plane. If some change is required to pods, that change is applied universally to all pods. This is the mirror of the mindset I outlined with full stack silo environments. The pod cannot be viewed as an opportunity to enable targeted customization for individual tenants.

One dynamic that comes with pods is this idea of placing tenants into pods and potentially viewing membership within a pod as something can be shifted during the life of a tenant. Some organizations may have distinct pod configurations that might be optimized around the profile of a tenant. So, if a tenant's profile somehow changes and their sizing or consumption patterns aren't aligned with those of a given pod, you could consider moving that tenant to another pod. However, this would come with some heavy lifting to get the entire footprint of the tenant transferred to another pod. Certainly this would not be a daily exercise, but is something that some SaaS teams support–especially those that have pods that are tuned to a specific experience.

While pods have a clear place in the deployment model discussion, it's important not to see pods as a shortcut for dealing with multi-tenant challenges. Yes, the pod model can simplify some aspects of scale, deployment, and isolation. At the same time, pods also add complexity and inefficiencies that can undermine the broader value proposition of SaaS. You may not, for example, be able to maximize the alignment between tenant consumption and infrastructure resources in this model. Instead, you may end up with more instances of idle or overprovisioned resources distributed across the collection of pods that your system supports. Imagine an environment where you had 20 pods. This could have a significant impact on the overall infrastructure cost profile and margins of your SaaS business.

Conclusion

This chapter focused on identifying the range of SaaS deployment models that architects must consider when designing a multi-tenant architecture. While some of these models have very different footprints, they all fit within the definition of what it means to be SaaS. This aligns with the fundamental mindset I outlined in Chapter 1, identifying SaaS as a business model that can be realized through multiple architecture models. Here, you should see that–even though I outlined multiple deployment models–they all shared the idea of having a single control plane that enables each environment and its tenant to be deployed, managed, operated, onboarded, and billed

through a unified experience. Full stack silo, full stack pool, mixed mode–they all conform with the notion of having all tenants running the same version of a solution and being operated through a single pane of glass.

From looking at these deployment models, it should be clear that there are a number of factors that might push you toward one model or another. Legacy, domain, compliance, scale, cost efficiency, and a host of other business and technical parameters are used to find the deployment model (or combination of deployment models) that best align with the needs of your team and business. It's important to note that the models I covered here represent the core themes while still allowing for the fact that you might adopt some variation of one of these models based on the needs of your organization. As you saw with the hybrid full stack model, it's also possible that your tiering or other considerations might have you supporting multiple models based on the profile of your tenants.

Now that you have a better sense of these foundational models, we can start to dig into the more detailed aspects of building a multi-tenant SaaS solution. I'll start covering the under-the-hood moving parts of the application and control planes, highlighting the services and code that is needed to bring these concepts to life. The first step in that process is to look at multi-tenant identity and onboarding. Identity and onboarding often represent the starting point of any SaaS architecture discussion. They lay the foundation for how we associate tenancy with users and how that tenancy flows through the moving parts of your multi-tenant architecture. As part of looking at identity, I'll also explore tenant onboarding which is directly connected to this identity concept. As each new tenant is onboarded to a system, you must consider how that tenant will be configured and connected to its corresponding identity. Starting here will allow us to explore the path to a SaaS architecture from the outside in.

About the Author

Tod Golding is a cloud applications architect who has spent the last seven years immersed in cloud-optimized application design and architecture. As a global SaaS lead within AWS, Tod has been a SaaS technology thought leader, publishing and providing SaaS best practices guidance through a broad set of channels (speaking, writing, and working directly with a wide range of SaaS companies). Tod has over 20 years of experience as an architect and developer, including time at both startups and tech giants (AWS, eBay, Microsoft). In addition to speaking at technical conferences, Tod also authored *Professional .NET Generics*, was coauthor on another book, and was a columnist for *Better Software* magazine.