Octopus Deploy

A modern view of multi-tenancy

Steve Fenton

Contents

Introduction

There's no single correct way for organizations to approach multi-tenancy. We've helped customers deploy multi-tenanted applications, built our own Software as a Service (SaaS) offering, and have decades of collective experience as software engineers. This has allowed us to see how companies of diverse shapes and sizes tackle multi-tenancy challenges differently.

While there's no "right" way, there is a body of knowledge and experience about the trade-offs practitioners have discovered when dealing with multi-tenancy in the real world. The goal of this white paper is to present this knowledge in a practical way.

This white paper explains:

- Why existing definitions for multi-tenancy are limiting.
- A new approach to multi-tenancy that looks beyond software architecture.
- How to successfully choose an approach to multi-tenancy.

The origins of multi-tenancy

The software industry is filled with concepts. Some of these were invented, and others were discovered.

You can credit inventions to one or more authors who define the concept with their articles, books, and conference presentations. Eric Evans created Domain Driven Design, Geoffrey Moore gave us the innovation-adoption chasm, and 17 signatories brought us the *Manifesto for Agile Software Development*.

To understand invented concepts, we can return to the original sources and read their definition. With discoveries, no such canonical authority exists. Early descriptions aren't definitive, and the concepts arrive partly formed and need more robust development.

Multi-tenancy is a discovery, not an invention. It's based on the convergence of earlier concepts, such as multi-user applications and time-sharing. Multi-tenancy has existed since the 1950s but gained traction between 2000 and 2010 with the rise of web services and Software as a Service (SaaS).

There's no canonical source for multi-tenancy, and previous attempts to define its meaning have failed to provide helpful language or a framework for practical decision-making. Previous references for multi-tenancy focused on software multi-tenancy at a broad architectural level and ignored crucial business needs and alternative ways to achieve tenant-aware systems.

Architectural concerns

To understand when multi-tenancy becomes an architectural concern, we can use Martin Fowler's popular definition:

> "[Architecture is...] things that people perceive as hard to change"
>
> ***Martin Fowler***[1]

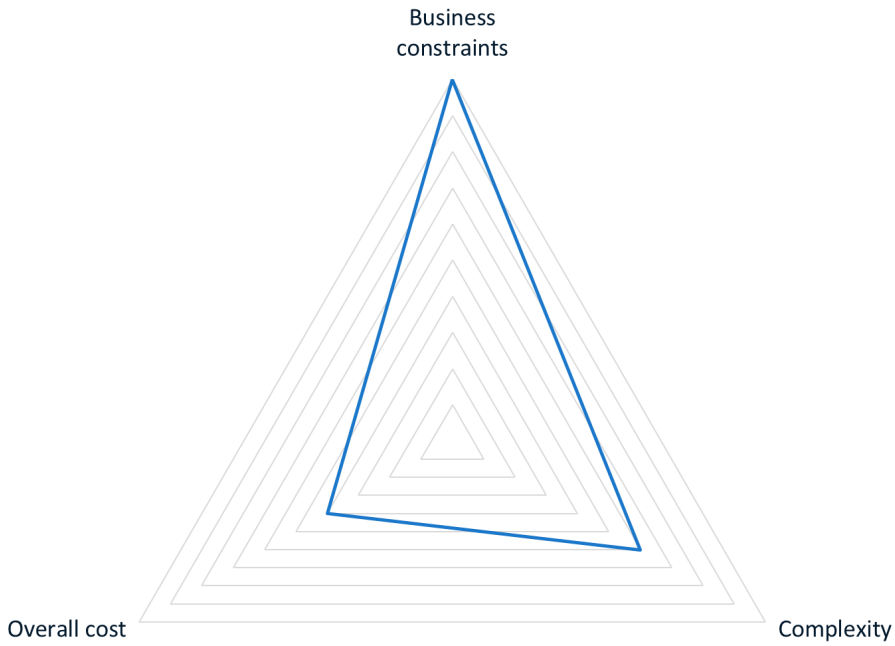
A decision is only architectural if you think it will be hard to change it later. If you can devise a creative way to make a decision reversible, you can avoid making it architectural. The fewer big decisions you must make, the less your development work will stall.

The paradox of architecture is that making something easy to change makes the whole system a little more complex. By trying to make *everything* easy to change, you make the system so complicated it's hard to change anything.

Multi-tenancy is architectural by default. It's usually better to think of the trade-offs early to avoid spending time re-architecting your system later. This white paper has practical advice on choosing a multi-tenancy approach to help you make a better decision for your system.

Multi-tenancy trade-offs

Multi-tenancy is a broad set of trade-offs. The cost savings from sharing parts of your application or infrastructure should be greater than the additional complexity introduced to enable it. You need to determine the optimal balance for your circumstances.



A simplified view of the 3 elements of the trade-off: Business constraints, complexity, and overall cost.

At a high level, there are 3 elements involved in the multi-tenancy trade-off:

- **Business constraints**: Functional and non-functional requirements, including legal and contractual obligations, security, and performance.
- **Complexity**: The complexity of the system to build, maintain, and operate.
- **Overall cost**: The overall cost of the system to build, maintain, and operate.

You'd generally expect all business constraints to be met, as the system is only fit for purpose when this is the case. The overall cost represents the outcome you want to achieve when making your decision. This reveals the weakness in this simplified view of the trade-offs.

If the business constraints are fixed, and the overall cost is the outcome, the definition of complexity must be expanded to reveal where you have control over the result. We need to view complexity along 3 dimensions:

- **Code complexity**: How much code is added to handle multi-tenancy.
- **Pipeline complexity**: The impact multi-tenancy has on builds, tests, artifacts, and deployments.
- **Operational complexity**: The effect of multi-tenancy on infrastructure and operations tasks.

An example of the trade-offs involved in designing a system that handles tenants.

You can now decide how to balance the different kinds of complexity to achieve the business constraints with the best overall cost. This is the model used throughout this white paper, though you should feel free to add dimensions if you think they materially impact your decision.

A new approach to multi-tenancy

Multi-tenancy aims to reduce costs by sharing elements of the software system. This isn't accomplished with a single big architectural decision. You must make many small decisions with different drivers and trade-offs to achieve this goal.

The traditional view of multi-tenancy is to reduce costs by increasing *tenant density*, which means servicing more tenants (groups of users and data) on the same infrastructure. By taking a more general approach, tenant density becomes just one way to achieve the goal of controlling costs. There are also many ways to increase tenant density beyond application architecture, examples of which are discussed later.

Our first challenge to the traditional view of multi-tenancy is to introduce a layered view that extends beyond the application and database. We'll present and group the layers, guiding what to consider in your design. You can use this information to optimize the whole system, not just one part.

Each decision you make represents a set of trade-offs. Understanding the benefits you hope to achieve, and the sacrifices you must make to get them, results in higher quality selections. It also helps to consider how difficult it is to change your mind later so that you can identify the critical decisions.

This leads to our second challenge to multi-tenancy: infrastructure is a crucial part of multi-tenancy design. Traditionally, multi-tenancy has been viewed through the siloed development and operations perspective, but in a DevOps world, it needs interdisciplinary design and strong goals alignment.

Multi-tenancy has always involved a trade-off between code complexity and operational complexity, so it makes sense to take a DevOps approach to its design.

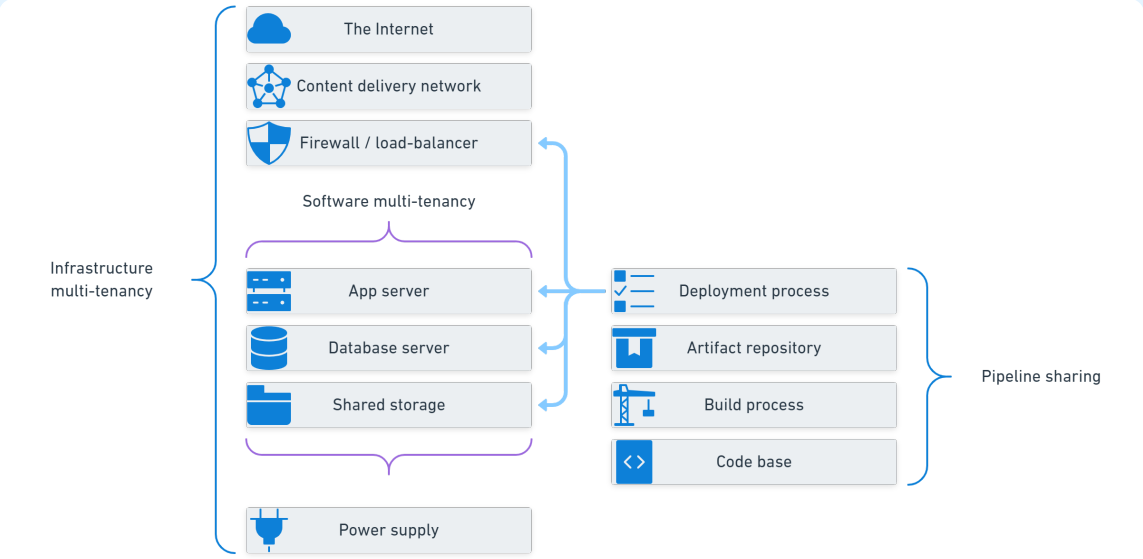
Our final challenge to the traditional view of multi-tenancy is to re-introduce the business constraints that are so often overlooked. As the goal is to reduce costs, the ultimate measure of multi-tenancy success is whether the business can deliver appropriate software at the right price.

Our new approach to multi-tenancy is a cross-functional process for deciding which layers of the software system to share and which to isolate.

Layers of sharing

With multi-tenancy, there are many layers you can either share between tenants or dedicate to a single tenant. For example, you might give each tenant a physical server in your data center, but these servers may share a power supply and network. You can classify all these layers into 3 general approaches.

- *Pipeline sharing*: Tenants share the same code base, builds, and deployment pipeline.
- *Infrastructure multi-tenancy*: Tenants share physical or virtual infrastructure, also known as infrastructure sharing.
- *Software multi-tenancy*: Tenants share the same application instance.



Infrastructure and software multi-tenancy overlap, while pipeline sharing remains separate.

The relationship between these approaches is shown above, where the application itself could use software or infrastructure multi-tenancy to achieve tenant density.

The most common layer of multi-tenancy is pipeline sharing. The main trade-off at this level is that you must support different tenant configurations, which increases code complexity. If you need to target different runtime environments, your build process needs to handle this, and you'll need some way of testing that the artifacts you generate work correctly.

When it comes to software multi-tenancy and infrastructure sharing, you face a balancing act between code complexity and operational complexity. Where the 2 overlap, usually the application servers, database servers, and shared storage, you should avoid mixing both patterns unless it gains you phenomenal tenant density. Applying both increases the total cost of complexity.

Let's explore the 3 approaches in more detail.

Pipeline sharing

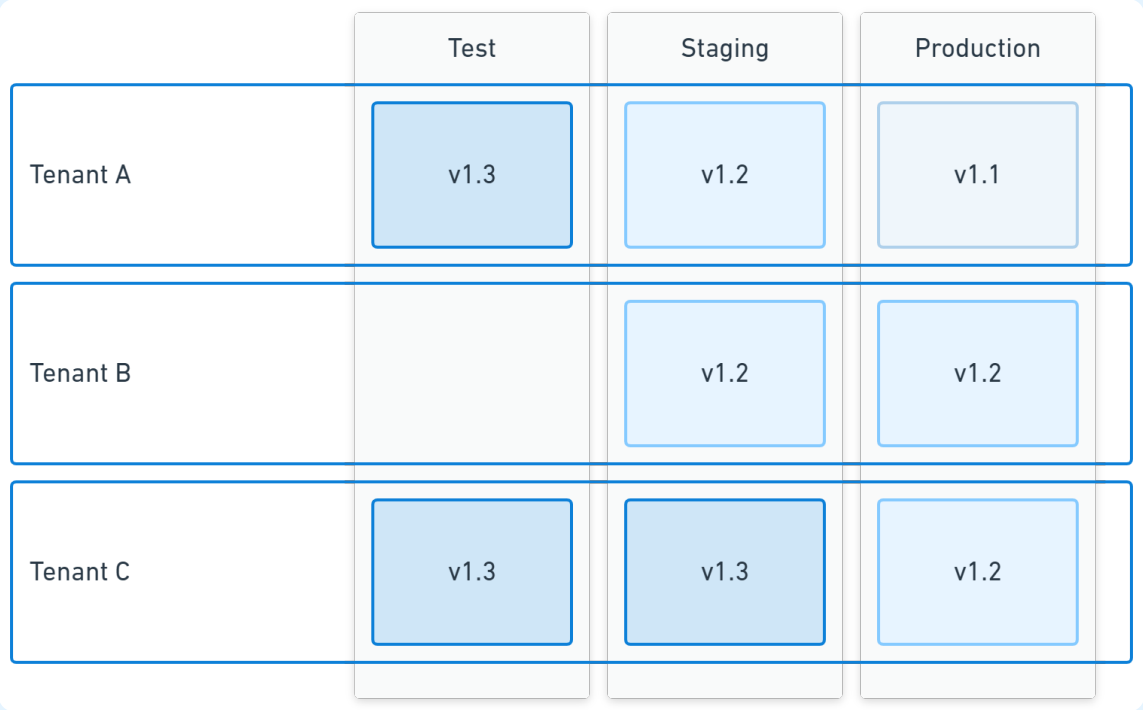
Tenants often share a code base, build process, tests, and deployment process, whether you deploy the software to isolated instances or shared infrastructure. We call this pipeline sharing.

Sharing at the pipeline layer adds some complexity, but compared to other approaches, this is minimal. Some examples of pipeline-sharing complexity include:

- The application must be configurable to adapt to different tenant needs.
- The testing is more complex when runtime configuration can modify the application behavior.
- Deployments may need to be made to more deployment targets.

The alternative to this complexity is duplicating the process so tenants have a tailored version. For pipelines, duplication almost always costs more than the complexity of sharing. Where builds and deployments are automated, they are usually far cheaper than maintaining multiple copies of the process - even a few copies.

Unless you write bespoke software, you likely already share the code base with many tenants. Pipeline sharing encourages you to extend this to the whole deployment pipeline. Deploying to multiple tenants should be as easy as deploying to multiple environments.



Environments provide columns, with versions progressing from left to right. Each tenant is a horizontal swimlane and may have a subset of environments.

In the above example, different tenants have installed different versions at various lifecycle stages. Not all tenants use the complete set of environments.

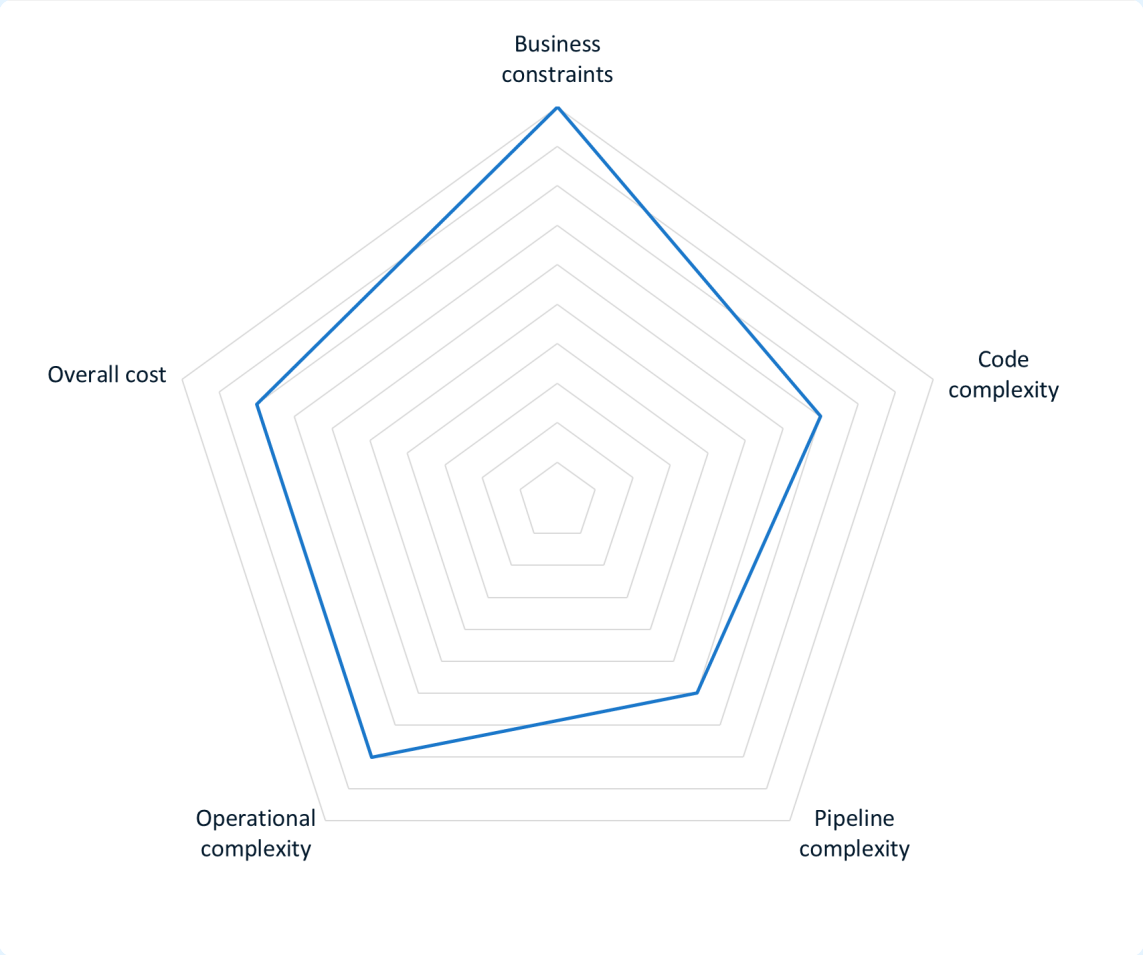
Infrastructure multi-tenancy and sharing

Infrastructure multi-tenancy seeks to achieve high tenant density with minimal code complexity. Rather than setting up physical machines each time you add a tenant, you can use virtualization (virtual machines or containers) to share the infrastructure between many tenants while keeping a high isolation level.

Infrastructure multi-tenancy reduces costs and provides an easy way to offer different service tiers, such as a more powerful infrastructure configuration for an enterprise customer.

Even if each tenant is allocated a physical machine, cost savings are made at the data center level by sharing power supply, networking, air-conditioning, and security. There is a sweet spot for the cost and complexity trade-offs, though it can change over time.

We assume the software is not multi-tenanted when we explore the options below. An organization that designs a multi-tenanted software system and then ends up deploying isolated instances pays twice as it must manage the code complexity and the operational complexity.



Attempting to use both infrastructure and code complexity typically costs more than favoring one, as you miss out on the upside of exploiting a trade-off, though operating at scale makes it more likely you'll mix multiple approaches.

When reviewing the various infrastructure multi-tenancy options, consider the following criteria as part of the selection process.

- The cost per tenant and tenant density.
- The number and types of shared physical resources.
- The robustness of service levels based on tenant usage patterns.

Dedicated physical machines

Let's start with dedicated physical machines, which have the lowest tenant density. A single tenant is served by one or more machines reserved for their sole use. While this is rarely an option for business applications, some applications are designed to control specific machines, so they must be installed directly on them. Examples include:

- Retail point-of-sale devices
- Automated machinery
- Self-service kiosks
- Medical devices

Application pools

Application pools offer process isolation on shared machines. You can set the maximum number of processes for each pool to limit the resources a single tenant can consume. You can configure each application pool, so you can sell different service tiers.

Shared-hosting providers and resellers use this approach. It means they can run many websites on a single machine, splitting the costs between many tenants.

You'll find application pools in Apache, Nginx, and Microsoft Internet Information Services (IIS). The limit to the number of sites you can run will vary based on server resources (CPU, memory, and disk space) and configuration. For example, it's possible to **run 100,000 websites within IIS**[2] on a server with sufficient resources, as long as you configure centralized logging.

Virtualization: Virtual machines

Virtualization provides a hardware emulation layer on top of the actual hardware, allowing many virtual instances to run on a single machine. Each instance has an allocation of disk space, CPU, and memory, so you can easily apply service tiers.

You can increase tenant density by allocating more than the physically available resources (over-provisioning), allowing tenants to use more than a simple share of resources. This works as long as they don't all need to use the extra capacity at the same time.

It's possible to run virtual machines across a pool of physical machines. The virtual machines can be automatically migrated to different hosts to smooth resource use and handle faults.

Each tenant can have an isolated virtual machine with a dedicated application instance. This means the database, CPU, memory, and disk are all isolated with hard walls, though they share the same physical host, network, and data center. Licenses for operating systems and databases aren't shared.

Virtualization: Containers

Containers take things a step further than virtual machines. Where a virtual machine needs to run a guest operating system, containers don't. This reduces the disk and memory requirements as only the host operating system needs to be installed.

It's possible to create tenant-specific containers to partition the application and data. Because containers are lightweight, creating a container per component is common rather than using a single container to host multiple components, as often happens with virtual machines.

Containers provide high tenant density and medium to high levels of isolation.

Infrastructure sharing summary

Suppose a physical server is like running a train for each tenant. In that case, virtual machines give each tenant their own train carriage, and containerization provides each tenant with their own compartment in a carriage.

It's relatively easy to sell different classes of service using virtualization by providing different resource allocations at appropriate price points.

With software multi-tenancy, all tenants share the whole train. They can take up any free seat in any carriage. More thought is required to provide different service tiers and to prevent tenants from affecting the service levels of other tenants sharing the train.

The economics of infrastructure sharing have changed over time and will likely continue to change. These days there are fewer reasons to take on the code complexity associated with software multi-tenancy compared to the early days of SaaS when it was considered the default option.

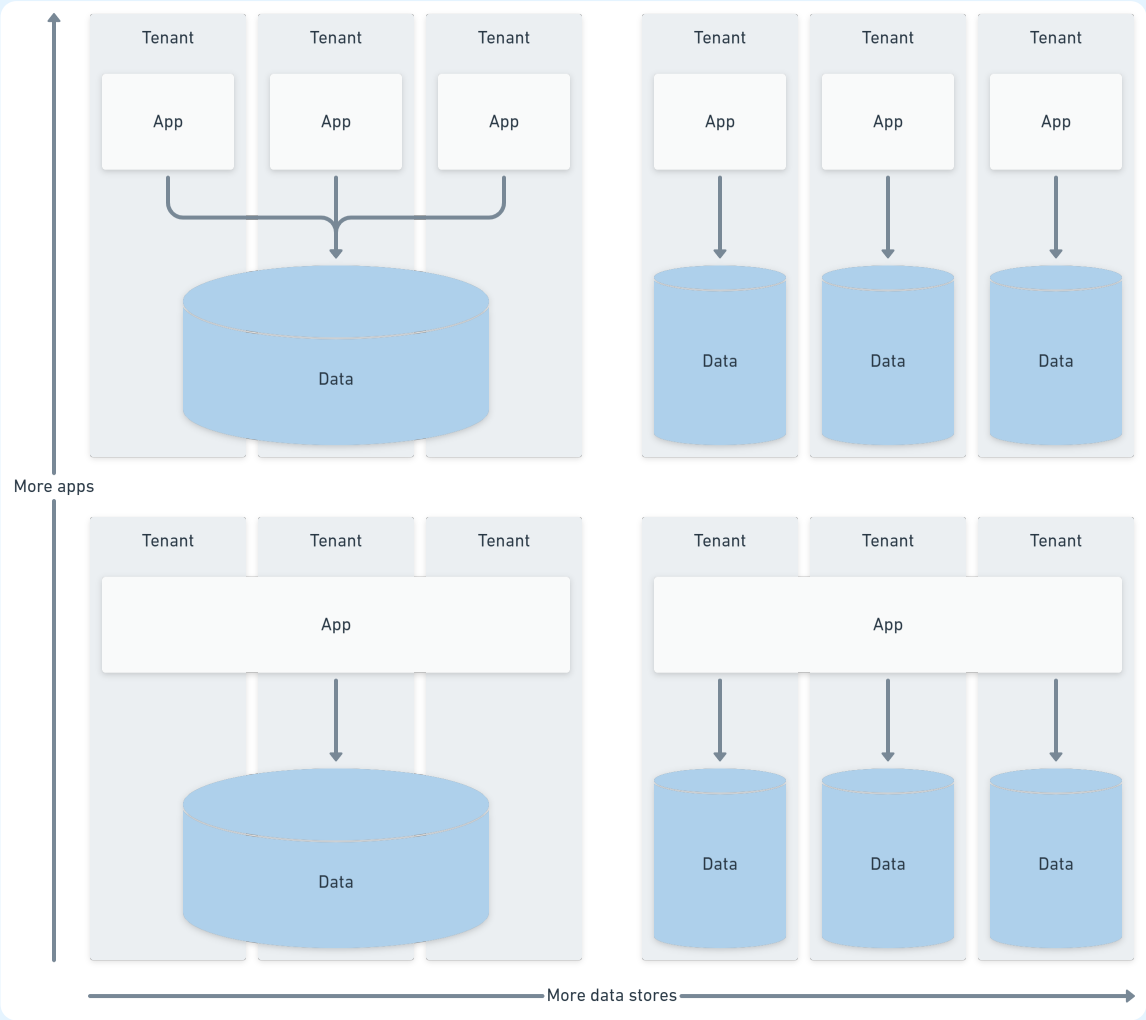
Software multi-tenancy

With software multi-tenancy, a single application instance handles requests for all tenants. The software instance is responsible for partitioning the configuration and data for each tenant so the software users get the expected behavior and their data is secure.

Tenants share the application instance and use the same servers, CPUs, memory, disks, networks, and power supply. This results in high tenant density for the software, infrastructure, and licenses (such as the operating system license).

You need a strong architectural design to ensure your tenant concept isn't scattered throughout the application. The consensus is to push tenant handling away from the middle of the application, handling high up in authentication and low down in the database connector. Avoid passing the tenant identity around within the core business logic.

You can further split your software multi-tenancy by considering the application and database instances separately. This results in 4 quadrants that describe the possible designs. Application instances are shown along the first axis, and database instances along the second.



*The *software multi-tenancy quadrants*, based on whether the application instance or database instance is shared.*

Full software multi-tenancy is shown in the bottom-left quadrant. A single application and database instance serves all tenants.

The top-left quadrant shows a variation for applications with high compute load. To monitor or limit tenant use, provide service tiers, and reduce noisy neighbors, you may provide a dedicated application instance for each tenant over a single shared database.

The bottom-right quadrant has a single shared application instance with tenant-specific databases. This provides high isolation of data and simplifies operational activities for data retention, deletion, and recovery for a single tenant.

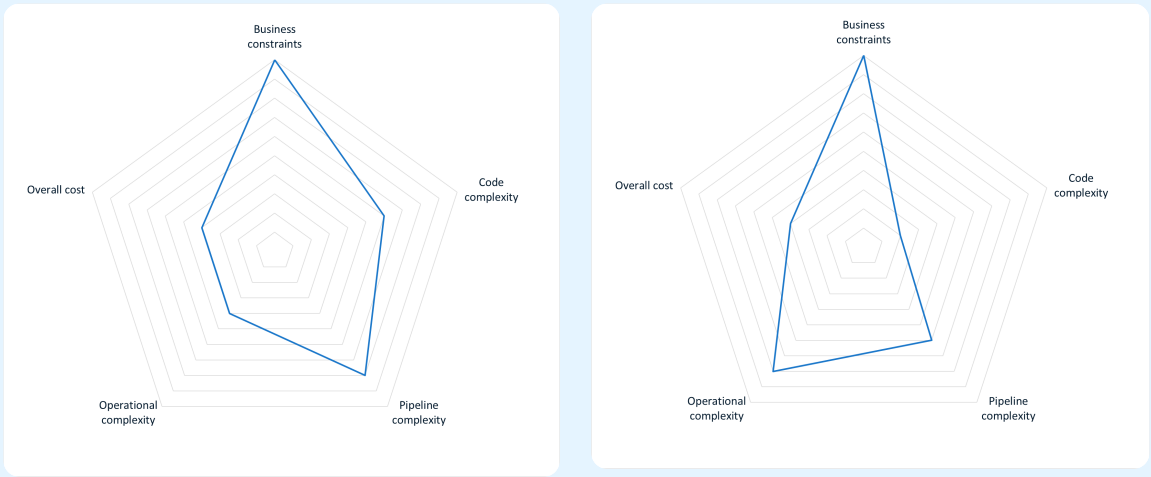
The top-right quadrant shows dedicated tenant instances of the application and database. Software multi-tenancy is not used in this zone, though pipeline and infrastructure sharing might still be in play.

Choosing an approach to tenancy

Multi-tenancy is a set of trade-offs. You must diminish some qualities of your system to gain others. Most trade-offs relate to cost and complexity along one axis or another.

For example, you can reduce operational complexity by adding code complexity, but the business constraints are less flexible to trades (except when the economics prove a business model isn't viable). To determine the correct answer, you need to work out the total cost for the whole system for each potential design.

The radar chart from the introduction illustrates these interactions. You can compare designs by translating the trade-offs into a tangible overall cost.



Two approaches to tenancy. The left side prefers code complexity, while the right side uses operational complexity. Both satisfy the business objectives and achieve a similar overall cost.

It's tempting to limit your options based on your team's skills. Instead, focus on the product and workloads when you search for options and identify skill gaps you may need to fill. You may have team members interested in learning a new technology that you need to enable your design.

You need to identify legal and contractual requirements early. For example, you'll take a different path if you need to satisfy data residency or data sovereignty needs. Solving these requirements with virtualization means your software doesn't need multi-tenanted instances. Avoid creating a complex software architecture when other conditions require an instance per tenant.

Organizations with automated build and deployment pipelines will find it easy to use pipeline sharing to maximize the return on investment in the code base, build process, and deployment automation. If the code base is shared, it makes sense for the rest of the pipeline to be shared, too.

Other complexity drivers

There are other complexity drivers you need to consider alongside multi-tenancy. They aren't directly related, but multi-tenancy can increase their complexity.

- Monitoring and observability
- Scaling options
- Load balancing
- Failover mechanisms
- Backups
- Data retention and deletion
- Usage limits and monitoring of fair usage

Each of these may become more difficult depending on how you approach multi-tenancy. For example, using infrastructure multi-tenancy, you may need your load-balancing algorithm to be tenant aware, so it can handle stateful operations or optimize cache hits.

Alternatively, you can keep load balancing simple with software multi-tenancy and stateless operations. However, your data management activities become more challenging as they must be tenant-aware. You may need to restore data for a single tenant, which represents a significant risk to the data held for other tenants.

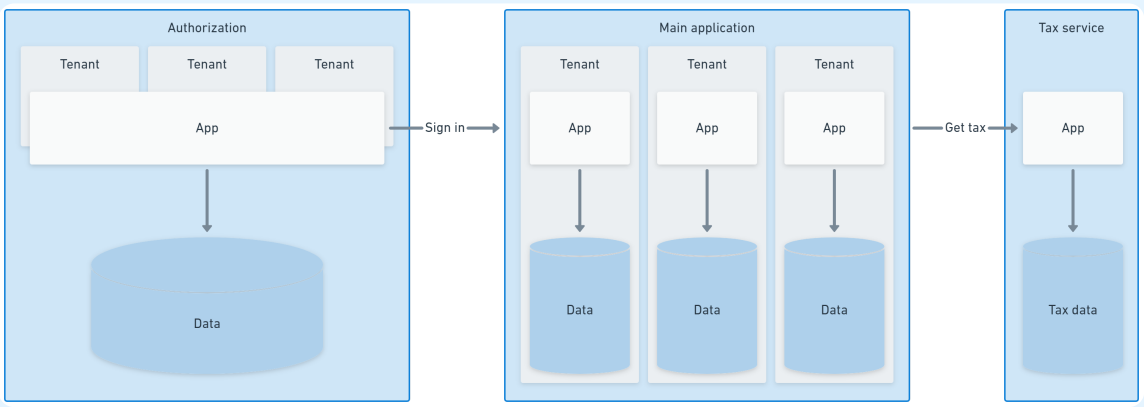
Load balancing affinity may become more complex if you need to keep tenant data in a legal jurisdiction. Instead of being non-affine (where any request can be routed to any server), tenant traffic may need to be cluster affine (always sent to a specific data center) or inter-cluster affine (always sent to data centers in a particular region).

Decide at a granular level

You should consider each component individually rather than applying a decision at the system level. You can reduce the sum of all complexity by making many smaller decisions instead of picking one over-arching answer.

You may have components that don't need partitioning because there's no stateful relationship with a tenant. In these cases, the component doesn't need tenant awareness, and it could be exposed as a single shared instance that all tenants use. This achieves high tenant density with no additional code or operational complexity.

Where you need to partition a component, you can consider the trade-offs and choose between software multi-tenancy and infrastructure sharing.



A multi-tenanted authorization app controls access to tenant-isolated application core application instances. The tax service is tenant unaware and can easily be horizontally scaled. You should maximize offloading work to stateless tenant-unaware services that are easy to scale.

For example:

- Authorization: A software multi-tenanted component with a single application and database instance.
- Core application: Tenant-specific instances running on virtualized infrastructure with hard walls around the configuration and data.
- Tax calculation service: A stateless and tenant-unaware calculation service instance shared by all tenants.

By repeating this process for each component, you optimize the trade-offs. Choosing a system-wide strategy guarantees a sub-optimal balance between tenant density, code complexity, and operational complexity.

Hybrid hosting

The tenant-unaware components you identify can also be useful in hybrid hosting scenarios.

Many organizations are finding it more cost-effective to run their own servers rather than use the cloud. The cloud provides an incredibly flexible way to build and operate a software product when you don't know what resources it needs. After you establish the resources required to run your software, moving it in-house (either on-premises or in a data center) can help you achieve a lower, more predictable, month-on-month cost.

While you can achieve high performance at a lower cost with your own infrastructure, you can't provide elasticity. You must choose to over-provision to handle peak load or face slow-downs or faults when things get busy.

An alternative approach is to run typical loads on your infrastructure and use the cloud to gain additional capacity for short periods of high load. This gives you a predictable cost for normal operations and extra capacity when needed.

If you have a tenant-unaware component that all tenants can share, this is an ideal candidate for this kind of elastic scaling. The tax calculator in the example can scale out to handle month-end and year-end workloads. Identifying expensive operations such components can perform could give you great hybrid scaling ability.

You can read more about the **repatriation and consolidation trend in our post on cloud-nomad architecture**[3].

There isn't always a choice

When you have free rein over how you host your software, you can continuously review and balance the cost and resource use. In some cases, other constraints limit your available options.

There are many situations where you don't get to choose the model if you want to do business with organizations in particular industries. They may need to run the software at a physical location like a retail store or hospital. There may be legal requirements to keep data in a specific region or on their private network.

These demands are usually offset by a higher cost, which customers are happy to pay if they get what they need. It may even involve running the software on customer-operated infrastructure, which simplifies the operational complexity for the supplier.

Your existing software may also have insurmountable architectural issues preventing software multi-tenancy. For example:

- The cost of introducing a tenant concept.
- Dependence on the application or database server clock.
- Lack of globalization/localization support.

Infrastructure multi-tenancy and pipeline sharing can help achieve a successful result even if software multi-tenancy isn't possible.

Scaling affects multi-tenancy

When you scale your software, it affects the complexity dimensions in the trade-off. Running at scale often increases code and operational complexity. This amplifies the effect of decisions you made for multi-tenancy.

Scaling is already represented by the 3 kinds of complexity in the trade-off chart, but if it's helpful in your situation, you could make it an explicit dimension.

For example, suppose you introduced a tenant concept into your database as a tenant table. Operating it at scale will require new techniques, such as sharding, replication, and a more sophisticated maintenance plan. This means your plan to favor code complexity no longer limits operational complexity in the same way it did before scaling.

The likely outcome from scaling is introducing more approaches to multi-tenancy to re-balance the complexity. You might deploy multiple instances of your software to reduce the scale per instance or move away from shared databases.

Automation is key

Even for teams developing and deploying a single instance of an application, automation of the deployment pipeline increases their software delivery performance. Using trunk-based development, automated builds with tests that provide fast feedback, and deployment automation to reliably and repeatably deliver their software means they can:

- Create higher-quality software
- Shorten lead time
- Resolve incidents faster

If you have an automated deployment pipeline, you should also be able to handle multi-tenancy where needed, such as deploying tenant-specific instances of your system's components. Just as automation helps you deploy multiple instances to a web farm, it can deploy instances to tenant-specific virtual infrastructure or even to machines running in customer data centers or physical locations.

We have more information on Continuous Delivery and DevOps in our **resource center**[4] and in **the DevOps engineer's handbook**[5].

Summary

Because multi-tenancy wasn't invented, it lacks a definitive description. This led to a narrow view of the problem and its available solutions. The DevOps movement provides an opportunity to re-integrate infrastructure and pipeline sharing into the multi-tenancy discussion, removing arbitrary limitations to your options for cost optimization.

Taking a fine-grained layered approach reduces the size of the problem and surfaces the different trade-offs that exist when you look at various parts of the software system. Some of these trade-offs present straightforward decisions or are easy to change later. You can focus most attention where changing your mind later costs you more.

While layers help you focus, expanding multi-tenancy from an architectural concern to a cross-functional design process ensures you optimize decisions across development, operations, and for the benefit of the business. Although multi-tenancy is often measured in terms of tenant density, the true measure is the overall cost.

No matter how you handle multi-tenancy, an automated deployment pipeline is essential. You need to quickly and safely deploy your software using a repeatable and reliable process. Automated deployments assist horizontal scaling and multi-tenancy, and tenanted deployments can transform multi-tenancy economics.

References

1. https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
2. https://social.msdn.microsoft.com/Forums/en-US/6bca735e-7c63-4a81-887c-7fc09ce325c4/iis-website-limit
3. https://octopus.com/blog/cloud-nomad-architectures
4. https://octopus.com/resource-center
5. https://octopus.com/devops/

Further reading

- Learn more about tenanted deployments: **https://octopus.com/use-case/tenanted-deployments**
- Read the Octopus multi-tenant deployment guides: **https://octopus.com/docs/tenants/guides**
- Find more white papers in our resource center: **https://octopus.com/resource-center**

Octopus Deploy

Octopus Deploy
Level 4, 199 Grey St
South Brisbane, QLD 4101, Australia

Email: sales@octopus.com

Phone: +1 512-823-0256

octopus.com