Building multi-tenant architectures for agentic AI on AWS

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Building multi-tenant architectures for agentic AI on AWS

Table of Contents

Building multi-tenant architectures for agentic AI on AWS

Aaron Sempf and Tod Golding, Amazon Web Services

July 2025 (document history)

Agentic AI represents a disruptive paradigm shift that requires organizations to rethink how to build, deliver, and operate their systems. The agentic model has teams exploring new ways to decompose systems into one or more agents that create new paths, possibilities, and values.

Much of the agentic discussion centers around the tools, frameworks, and patterns that are used to build and implement agents. We must not only adopt good tools to create agents but also new integration protocols, authentication strategies, and discovery mechanisms that can serve as the basis of agentic architectures.

While the number of agentic tools grows, teams must also consider how their agents address more traditional architecture challenges. Scale, noisy neighbor, resilience, cost, and operational efficiency are fundamental topics that must be evaluated when you're designing, building, and deploying agents. Regardless of how autonomous and smart agents might be, we must also ensure that they achieve economies of scale, efficiency, and agility that align with business needs.

This guide's goal is to explore various dimensions of agentic footprints. This includes reviewing various agent deployment and consumption patterns and highlighting different strategies for creating agents that address architectural goals. It also means looking at how agents might be consumed in a multi-tenant environment by introducing internal constructs that are typically required in a multi-tenant setting.

Intended audience

This guide is for architects, developers, and technology leaders who want to build AI-driven multi-tenant systems.

Objectives

This guide helps you do the following:

- Understand multi-tenant agent deployments, exploring both siloed and pooled models, and how tenant context affects agent implementation
- Explore agent management, including onboarding, tenant isolation, and resource management across single- and multi-provider environments
- Evaluate aspects of multi-tenant agents, including data ownership, monitoring, and testing

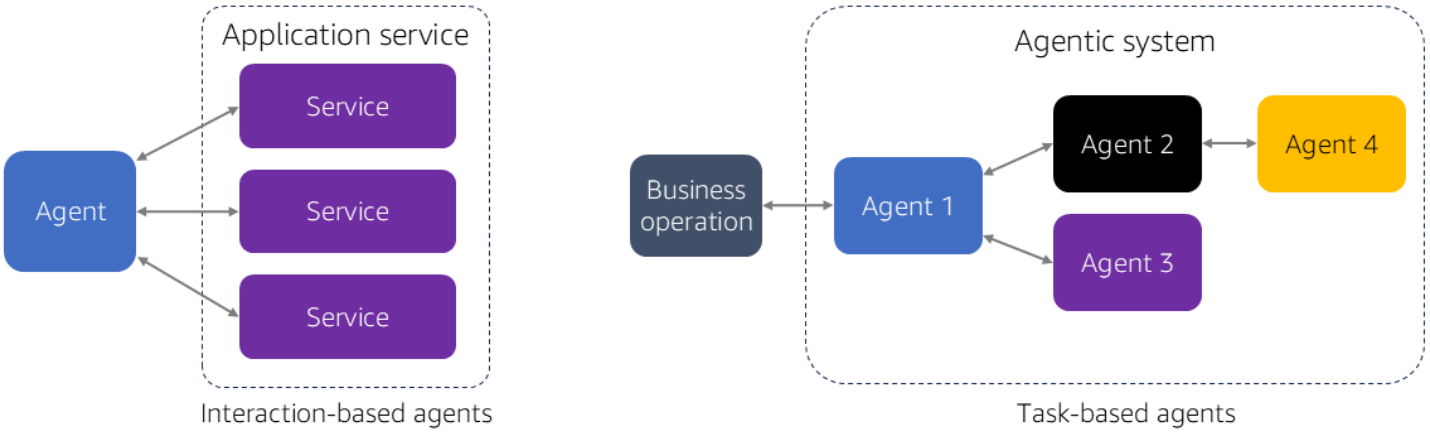
About this content series

This guide is part of a series about agentic AI on AWS. For more information and to view the other guides in this series, see Agentic AI on the AWS Prescriptive Guidance website.

Agent fundamentals

Before we discuss architectural details, we should outline the different roles that agents play because "agent" is an overloaded term that can be applied to many use cases. Let's start with some broad terms that can help categorize them.

At the outermost level, we need to start by classifying the role and nature of agents. This is challenging because there's a wide range of scenarios where agents can be applied to any number of problems. For this discussion, though, we focus on what it means to introduce an agent into an application or system. In this model, we emphasize how and where agents can best enrich your system's experience. The options you choose influence how your agents are built, integrated, and applied to different domains and use cases. The following diagram shows two agentic patterns that builders use.

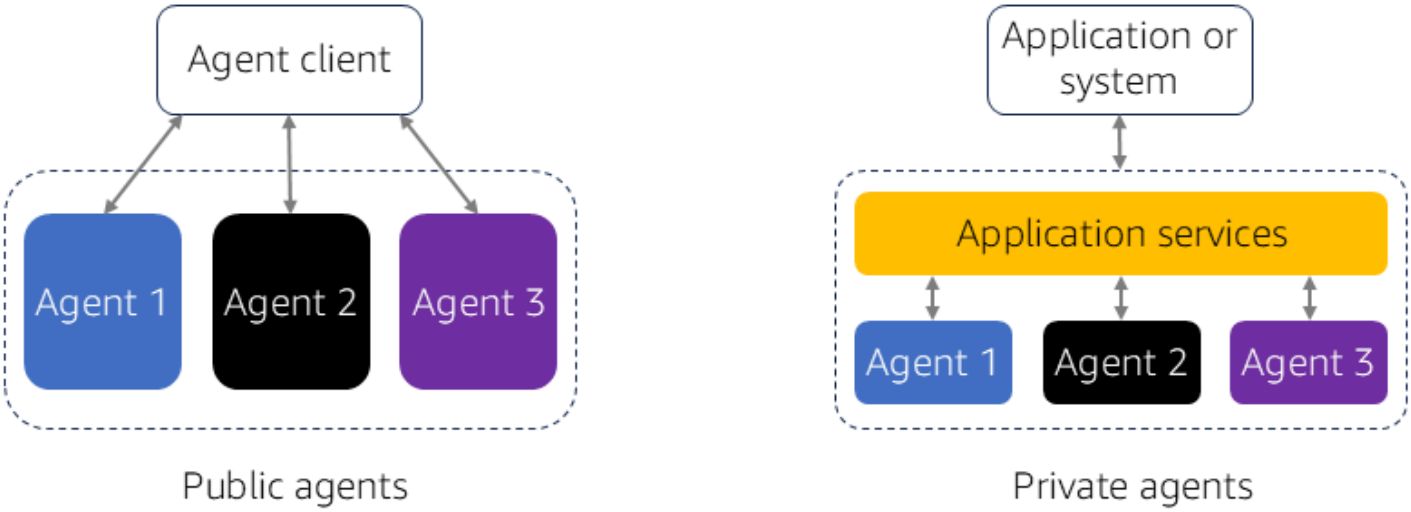


On the left-hand side of the diagram is an interaction-based agent. In this mode, an agent creates a view into an existing system to orchestrate interactions with the underlying services to achieve a goal or outcome. The key is that the agent is added to a system as an alternate approach to driving the system's features and capabilities. Imagine, for example, that an independent software vendor (ISV) has an accounting system with a UX that is used to perform operations. The interaction-based agent simplifies the interaction with these existing capabilities. It's less about learning how to reach a loosely defined goal and more about providing a way of orchestrating known pathways.

In contrast, the task-based system on the right-hand side of the diagram represents a different approach. The agents in that system use their knowledge and abilities to learn to complete tasks and drive business outcomes. You could argue that both models achieve business outcomes, but a task-based model relies on the agents themselves to determine how to achieve an outcome. Such agents are less deterministic and instead rely on their ability to learn and evolve. In contrast,

interaction-based agents are mostly designed to orchestrate a set of known capabilities. These differences affect how you build, scope, and integrate agents to support your business.

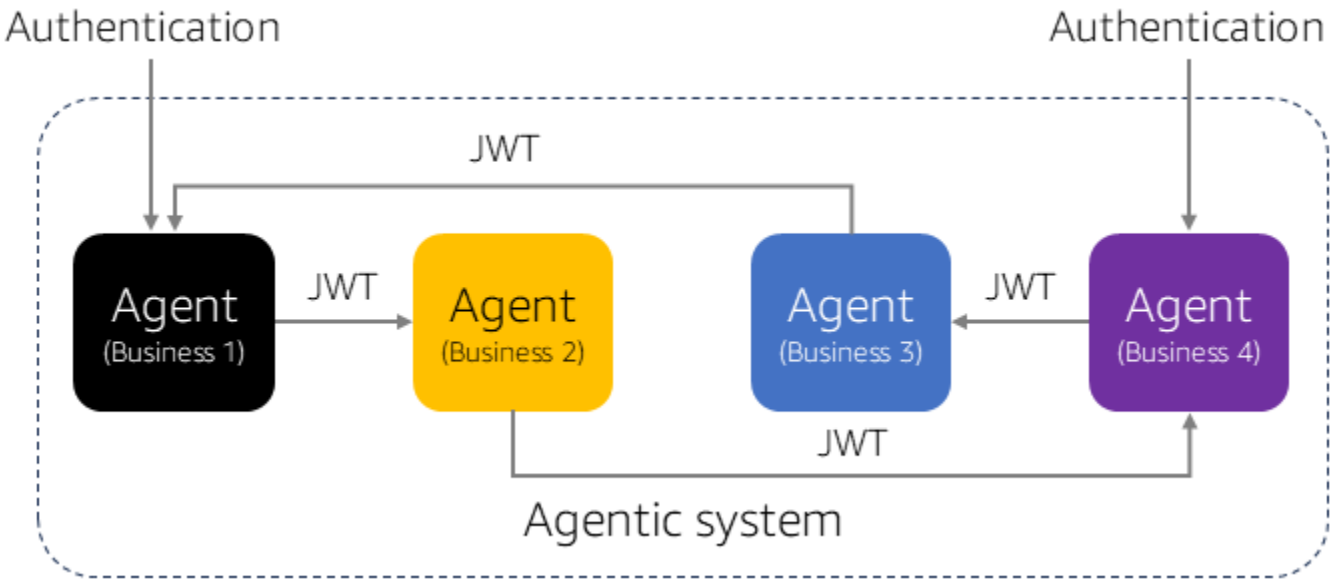
We also need terms that characterize how and where we deploy agents. Where an agent lives within your system's footprint can influence how it's built, scoped, and secured. The following diagram outlines two distinct models that could be applied to agents.



On the left-hand side of the diagram is a deployment system with three different agents. The agents are exposed to external clients that may be other agents or applications. For this model, agents are referred to as public agents.

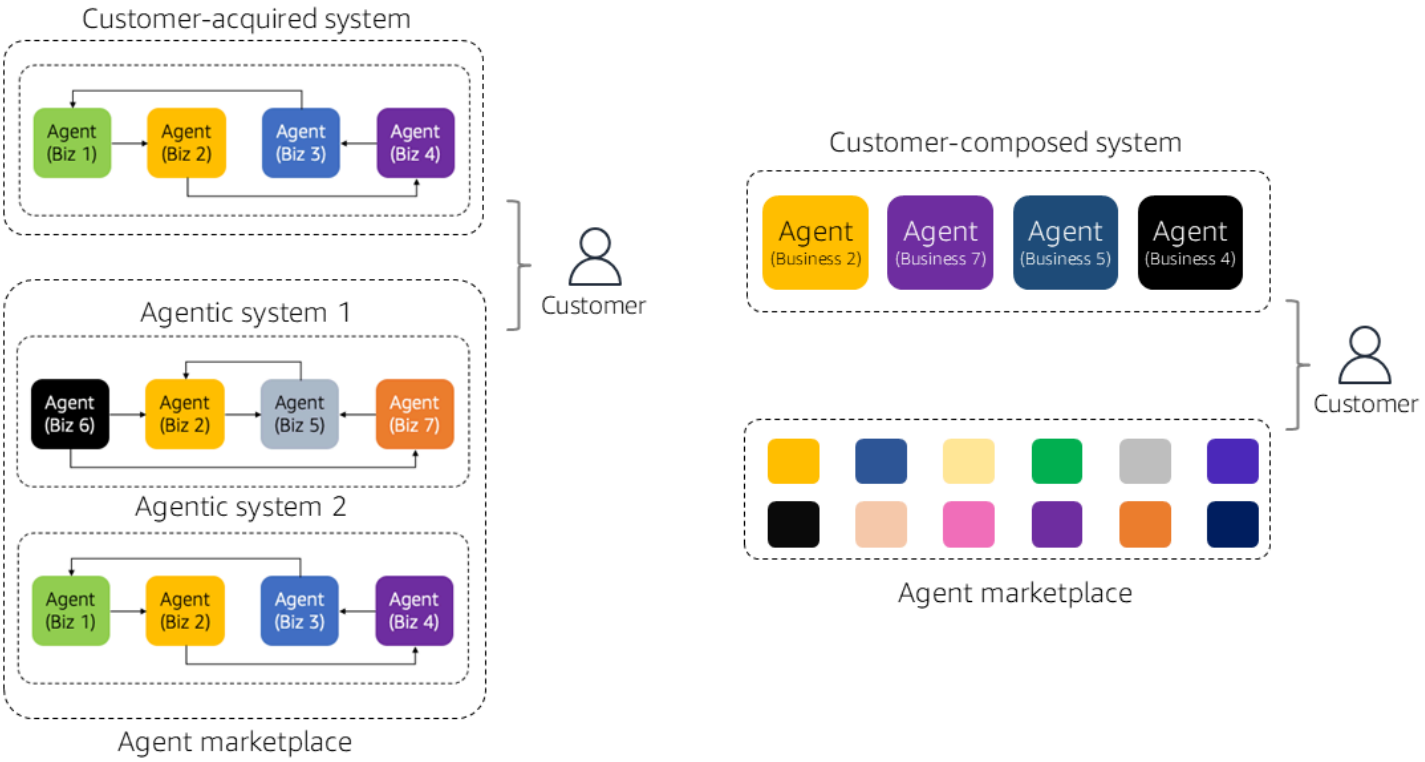
In contrast, the diagram on the right-hand side shows agents within the solution's implementation. In this case, there are a series of application services that are consumed by users or systems. These users interact with the application with no awareness of the fact that agents are part of the experience. The agents are then invoked and orchestrated by the underlying system's services. Agents deployed in this manner are referred to as private agents.

Much of an agent's value focuses on the public model where providers may be publishing their agents with the intention of integrating them with other third-party agents. The agents would then be part of a mesh or web of interconnected services that, collectively, are able to address many use cases. While these agents could be used in many domains, the business-to-business use case is a natural fit. The following diagram provides a conceptualized view of what it might look like to assemble a collection agent that solves a specific problem.



The diagram shows four business agents that work together to achieve a set of objectives. When agents are composed this way, they represent an agentic system, and there're many flavors of such systems. They could be a prepackaged set of collaborating agents that are commonly consumed as a single unit. Or the system could be dynamically assembled by customers who want to pick and choose a combination of agents that best meet their needs.

Both approaches offer viable pathways for agent integration. Some agents are built with the expectation that they will be integrated into specific systems where they can maximize their value, reach, and impact. This notion of agentic systems also raises questions about how agents are acquired, and there could be many ways to address this. The following diagram provides examples of how these agents and systems can be created through transactional experiences.

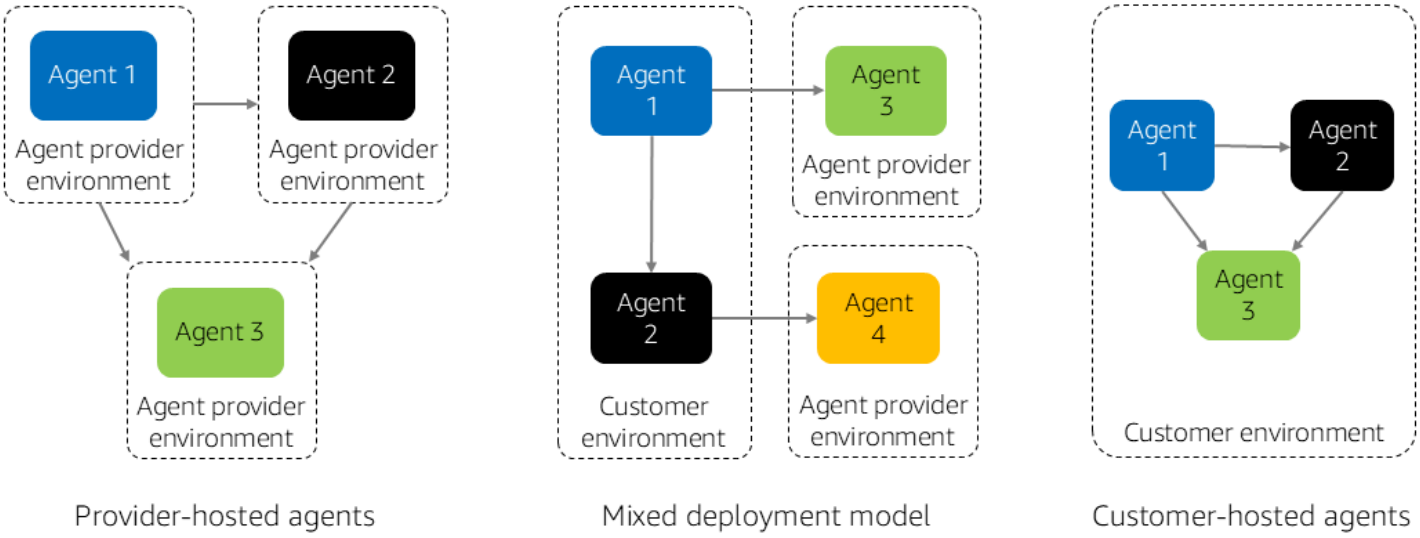


Two examples of marketplace experiences are shown. On the left-hand side, a marketplace is used to acquire prepackaged systems. In this scenario, the marketplace discovers and onboards systems that address broader objectives that require the integration and orchestration of multiple agents.

The example on the right-hand side shows a marketplace where agents are discovered and composed into agentic systems. In this scenario, customers can build any system of compatible, integrated agents to meet their needs. The ability to assemble agents in this manner depends on the compatibility model and integration requirements of individual agents.

Agent hosting considerations

Now that you have a sense of the broader agentic concepts, let's discuss what it means to host and run these agents. We must think about how and where computations run, how they scale, how they operate, and how they're managed. At the same time, some patterns that we expect to see as agents are more widely applied and adopted. The following diagram shows an example of likely permutations.



Three distinct strategies are represented here. On the left-hand side of the diagram, you see a model where our agents are hosted, scaled, and managed within the environments of each agent provider. These agents are published and consumed as services, operating in what is labeled as an agent as a service (AaaS) model. On the right-hand side is a model where a provider's agents are all hosted in a dedicated customer environment.

In the middle of the diagram is a mixed deployment model that combines these two strategies, hosting some agents locally in the customer's environment and interacting with some agents that are hosted remotely in a provider's environment.

A fourth option (not shown) could be where agents are built as low or no-code services that are scaled and managed by agent infrastructure services. We won't cover these in detail because the architecture and hosting of managed agents is dictated primarily by the organization that owns the services.

You can imagine the range of factors that might influence the adoption of one of these models. Compliance, regulatory, and security constraints, for example, could push someone toward

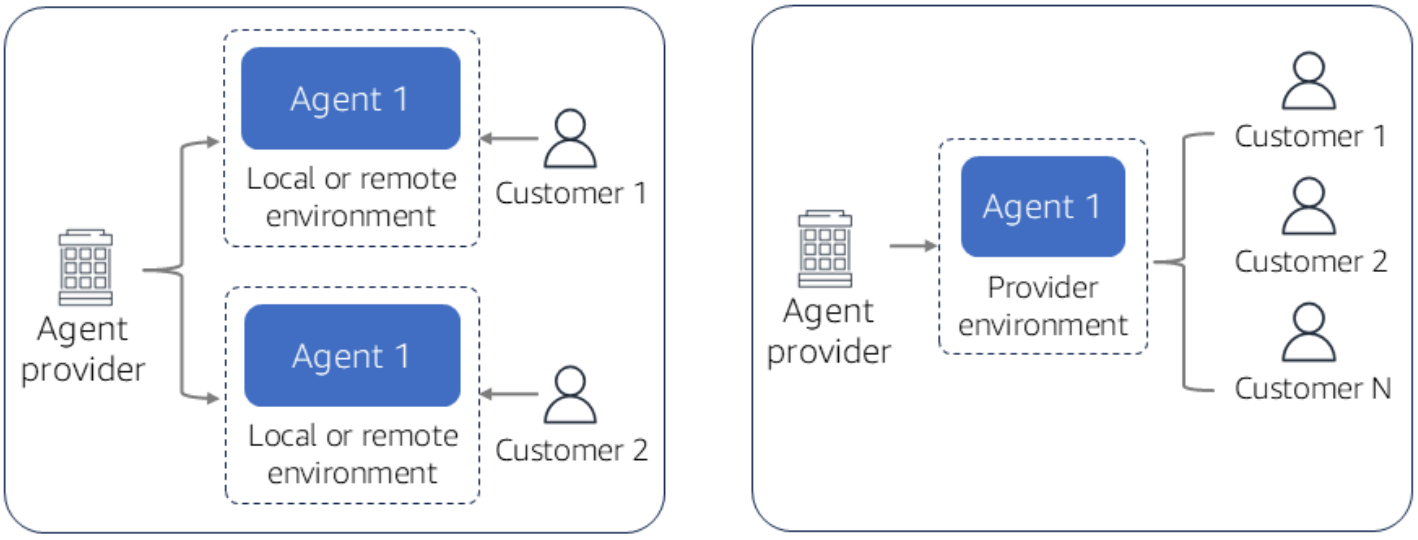
customer-hosted agents. Scale, agility, and efficiency could push organizations more toward the AaaS model.

The key concept here is that agents can and are deployed and hosted in many ways. It's your job to determine how agents can best be applied. The footprint, security, and deployment, among other factors, significantly affects how you approach building and operating agents. Private and public agents, for example, may have different designs and release lifecycles.

Agents meet multi-tenancy

It's easy to think of agents as building blocks where agents are viewed as a series of autonomous components that are assembled to support the needs of a specific domain or business problem. Where it gets more interesting is when we start to think about how these agents are packaged and consumed by providers. In many respects, an agent becomes a source of cost and revenue for a business. Agent providers must consider the different personas that consume their services, the consumption profile of the personas, and the monetization strategies that allow agent providers to create pricing and tiering models that align with consumers.

Agent providers could support multiple models for deploying their agents to meet customer needs. The following diagram shows a conceptual view of the two main agent deployment models.



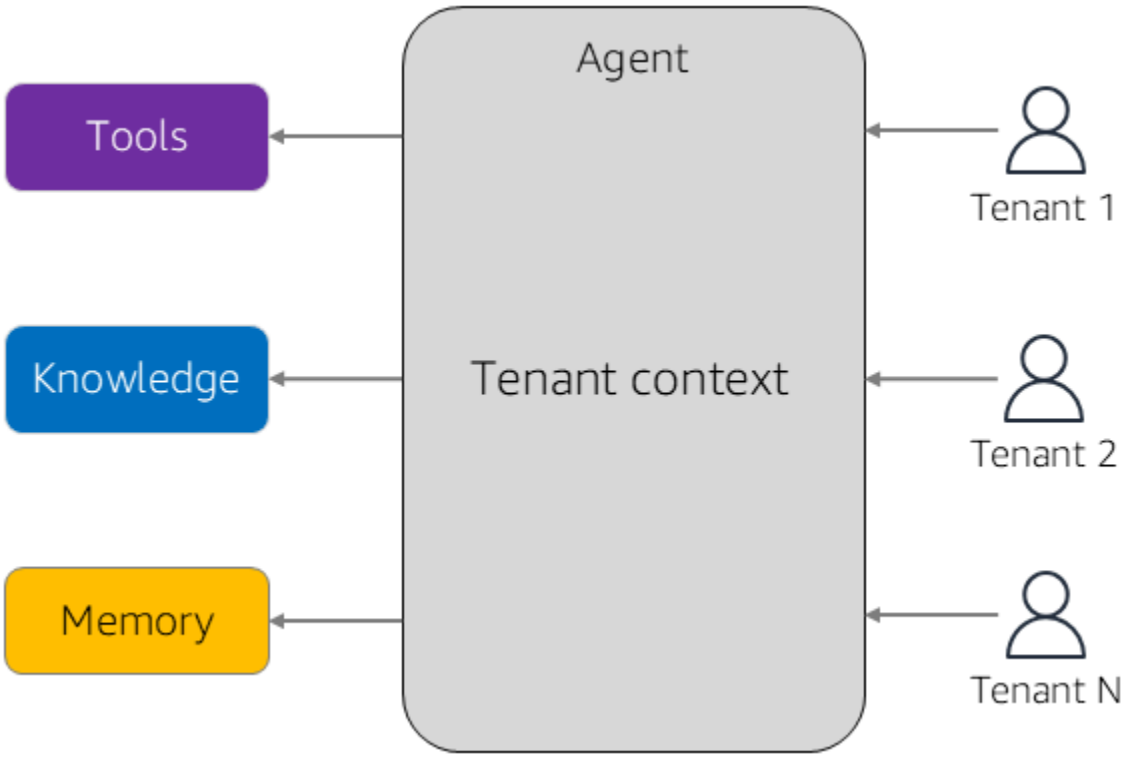
The left-hand side of the diagram shows the customer-dedicated agent model. An agent provider builds an agent by deploying a separate agent instance for each onboarded customer. With this approach, the capabilities of the agent and its ability to acquire knowledge would be limited to the scope of a given customer's environment. This ends up representing a per-customer experience that inherits some of the complexities and advantages of supporting dedicated customer environments.

In contrast, the diagram on the right-hand side of the diagram has a single agent that is deployed in the provider's environment. The agent processes requests from multiple customers, evolving and learning based on the collective experience of all customers. Each new customer that's added would simply represent another valid client of the agent. The agent runs like an agent as a

service (AaaS) model, using shared constructs to support a client's needs. In both instances, agent consumers can be applications, systems, or even other agents.

There are two ways you can look at the AaaS model. The model above delivers the same experience to all customers. This means the internals of the agent will not include any level of specialization that considers the context of the requesting client. Generally, for this mode, the assumption is that the nature of an agent's scope, goals, and value centers around a shared set of resources, knowledge, and outcomes that are applied universally to all clients.

The alternative approach to AaaS is where the context of clients influences the agent's experience and implementation. The following diagram provides a conceptual view of an AaaS agent footprint in this context.

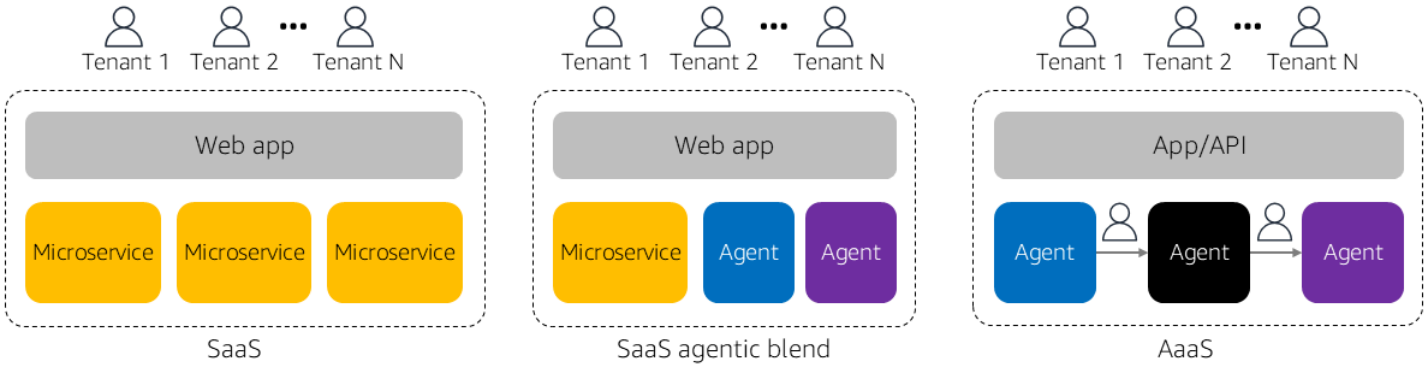


In this AaaS view, the origin and context of the incoming requests significantly affect the agent's footprint. The resources, actions, and tools that are part of the agent's underlying implementation may vary for each incoming tenant request. The value of an agent is connected to its ability to use tenant context to arrive at actions and outcomes that are influenced by tenant state, knowledge, and other factors. Some requests may yield a unique tenant outcome, and others may lead to more tailored per-tenant outcomes. This adds a new dimension to the agent's ability to learn, which could include being more contextual and acquiring and applying knowledge that enhances targeted outcomes.

For providers, the AaaS model offers many advantages. With multiple customers consuming a single agent, the provider has a better opportunity to achieve economies of scale, drive operational efficiency, control costs, and create a unified management experience. This has the potential for greater agility, innovation, and growth for the agent business.

These qualities overlap with the same principles that drive the adoption of the software as a service (SaaS) model. Essentially, the AaaS model is built as a multi-tenant service that inherits many of the same scale, resilience, isolation, onboarding, and operational attributes that are found in a SaaS environment. In many respects, the AaaS experience borrows heavily from the strategies and practices used by SaaS providers, but it's reasonable to separate these terms. For our purposes, the emphasis is mostly on the implications that come with building and operating agents that require multi-tenant support.

For a system that can treat all users equally and doesn't require the management of persistent, sensitive, or customer-specific data, the notion of tenancy would minimally affect their agents. For systems that are expected to serve multiple customers while preserving data isolation, customization, and context awareness, supporting multiple tenants could be an essential element of an agent's design, strategy, and goal. The following diagram shows how multi-tenancy can be used in agentic environments.



On the left-hand side of this diagram is a classic multi-tenant architecture. It includes a web application and a series of microservices that implement business logic. Multiple tenants consume the shared infrastructure of this environment, scaling to meet the shifting workloads of a tenant population that evolves. The environment is operated and managed through a single pane of glass for all tenants.

Imagine how this mental model maps to the agent on the right-hand side of this diagram. One agent runs an AaaS model that's consumed by one or more tenants. The agents could be from multiple providers with tenant context flowing between them because a single instance of one agent must process requests from multiple tenants.

The example in the middle of this diagram is a hybrid model where agents are part of the overall SaaS experience. Some parts of the system are implemented in a more traditional model and other parts of the system rely on agents. This pattern is likely to be common for many SaaS offerings—especially for organizations that are transitioning to an agentic experience. It's common for this model to persist because not all systems are delivered as pure AaaS. Also note that multi-tenancy still applies to the model's agents. While the agents may be embedded within a system, they may still process requests from multiple tenants.

It's natural to ask whether multi-tenancy really matters. You could argue that an agent processes requests, so supporting tenancy may have little effect. But as we dig deeper into multi-tenant agentic implications, tenancy may directly affect how agents influence how tools, memory, data, and other agent parts are accessed, deployed, and configured to support individual tenants. Tenancy also influences how scaling, throttling, pricing, tiering, and other business aspects apply to your agent's architecture.

One takeaway from this is that there are agentic use cases that require multi-tenancy support. The challenge is to determine how multi-tenancy shapes the overall design and architecture of your agentic experience. For some agents, multi-tenant support represents a differentiating capability, allowing agents to apply tenant-specific context to agents that deliver targeted outcomes.

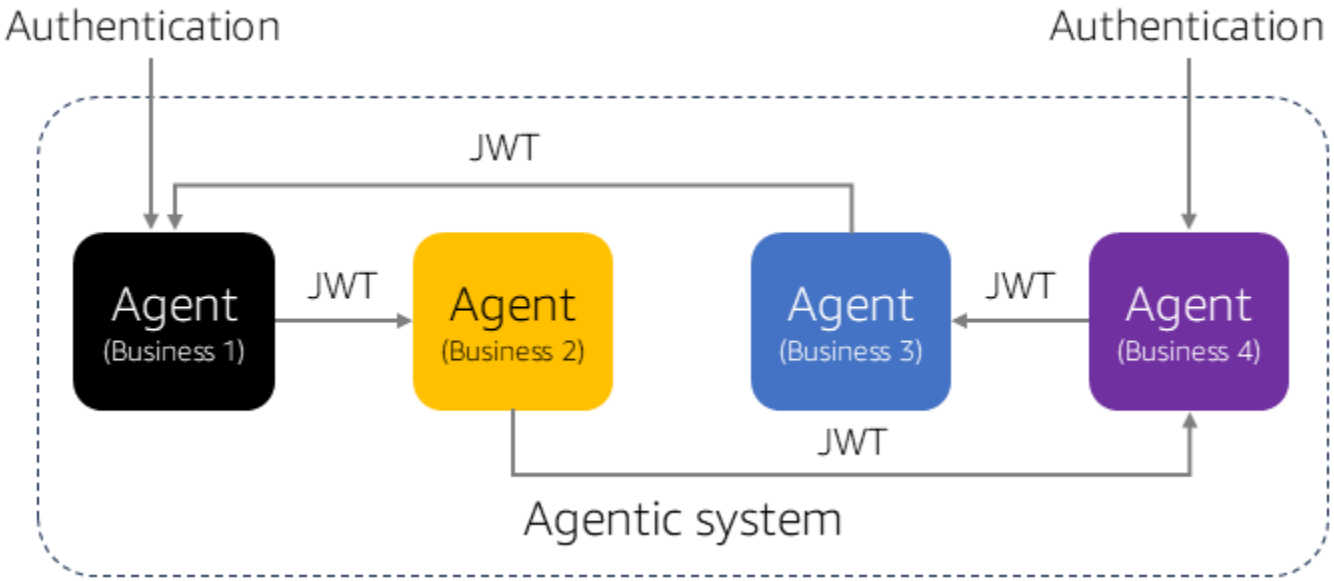
In subsequent sections, you'll see how the terminology and design patterns that we create to describe multi-tenant SaaS architectures will be useful. These concepts can be adopted by the AaaS model by borrowing useful aspects, which introduces new agent-specific concepts where they're needed.

Identity, tenant context, and agentic systems

Adding tenant context to individual agents isn't particularly challenging. In many instances, teams can rely on typical mechanisms that bind users and systems to tenants and pass tenant-aware tokens to agents. This is relevant when we consider how tenant context and identity supports multiple agents. In this model, tenants must be bound to an identity that spans all collaborating agents.

In general, the agentic domain requires a more cross-cutting identity model that aligns with the current and emerging needs of agentic systems. Agent providers require identity mechanisms that support unique security, compliance, and authorization models that come with operating agentic systems. This is especially challenging in environments where systems are composed by customers or other agents. Each onboarded agent must connect its identity and tenant context to agent

interactions. The following diagram highlights the potential identity and tenant context challenges that are part of agent-to-agent (a2a) interactions.



This diagram shows a series of provider-built agents interacting as part of the agentic system we covered. It's now retrofitted with identity and tenant context. This scenario is an example of an agentic system that supports multiple entry points. We assume that each agent in this system requires its own authentication mechanism to resolve the system or user to a given tenant. As these agents interact, tenant context is passed to a JSON web token (JWT) that will be used to authorize access and inject tenant context into the agent.

Conceptually, the main difference with this scenario is that agents deploy and operate independently, which means that each agent must be able to resolve its identity and authorize access. The key is that its identity must have some distributed ability to handle the needs of the broader agentic system. There must also be alignment on how agents share tenant context.

Applying SaaS business value to AaaS

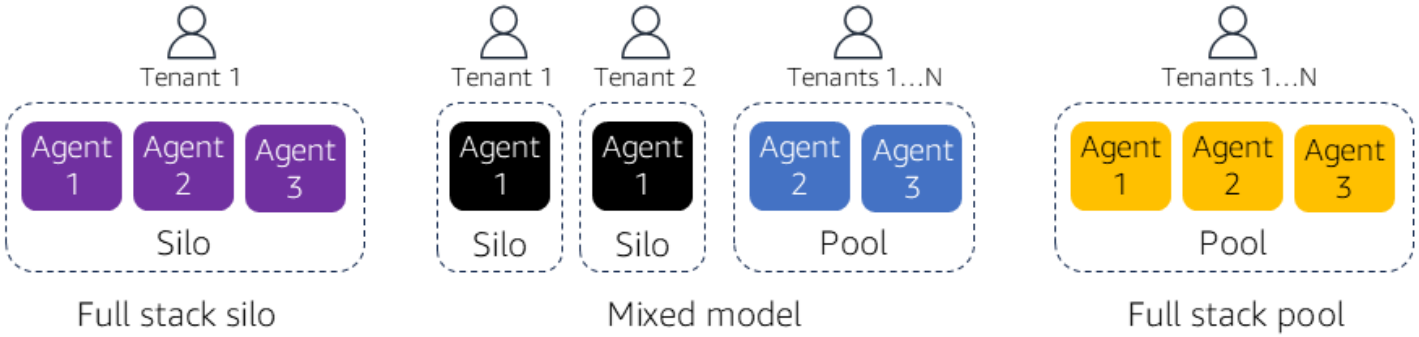
Generally, when we look at running any system in an as-a-service model, we consider the nature of the experience and how its technical and operational footprint drives business outcomes. When adopting SaaS, for example, organizations use economies of scale, operational efficiencies, cost profiles, and agility to drive growth, margins, and innovation.

Agents delivered as AaaS are likely to target similar business outcomes. By supporting multiple tenants, an agent can align resource consumption with tenant activities. This yields economies of scale that come with traditional SaaS environments. AaaS also allows organizations to manage,

operate, and deploy agents in a way that enables frequent releases and drives agility for agent providers. The key is that the AaaS model doesn't depend on technology. It creates and drives business strategies that promote growth, streamline adoption, and simplify operations.

Agent deployment models

In a basic AaaS experience, a provider may deploy agents using various patterns. There are myriad factors that influence how agents are deployed to meet customer, performance, compliance, geography, and security needs. Different deployment strategies affect how an agent is designed, implemented, and consumed. It's here that we can introduce classic multi-tenant terms to label different deployment strategies. The following diagram shows different permutations for deploying agents in an AaaS environment.

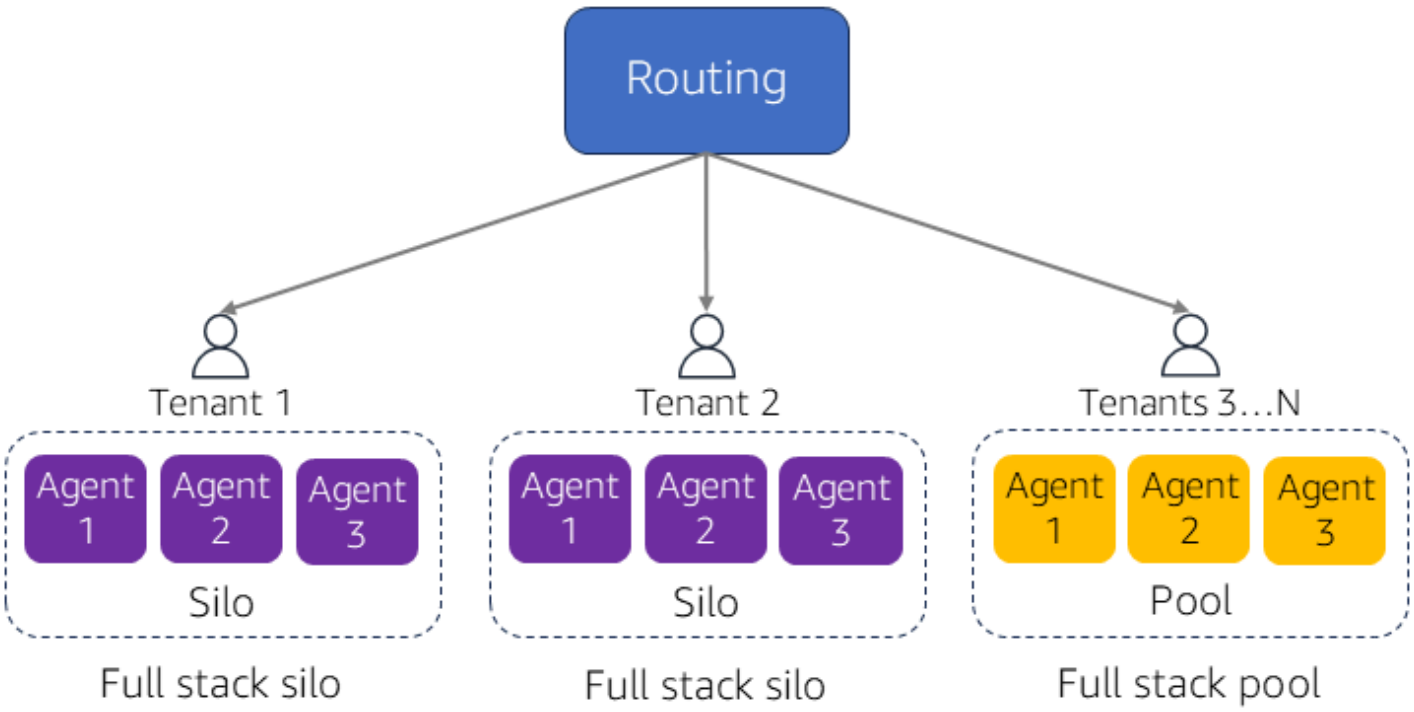


This diagram represents three modes of agent deployment. On the left-hand side is a siloed model, where each tenant is provided with a fully isolated experience and a dedicated set of agents. In this scenario, agents do not share compute, resources, or execution environments across tenants.

The middle example illustrates a hybrid model, where tenants use a combination of siloed and pooled agents. For instance, Agent 1 is deployed in siloed mode—each tenant receives a dedicated instance—while agents 2 and 3 operate in a pooled model, sharing resources across tenants.

On the right-hand side is a fully pooled model, where all agents are shared across tenants, offering a classic multi-tenant deployment. In this scenario, tenants leverage common compute, memory, and service infrastructure for agent execution.

The idea is that agents can operate in different deployment models, with compute and dependent resources either dedicated (siloed) or shared (pooled) across tenants. These deployment strategies are not mutually exclusive. Agent services often support a spectrum of customer needs, combining both models to balance performance, isolation, cost, and scalability. The following diagram shows an agentic system that supports multiple deployment configurations within the same operational environment.

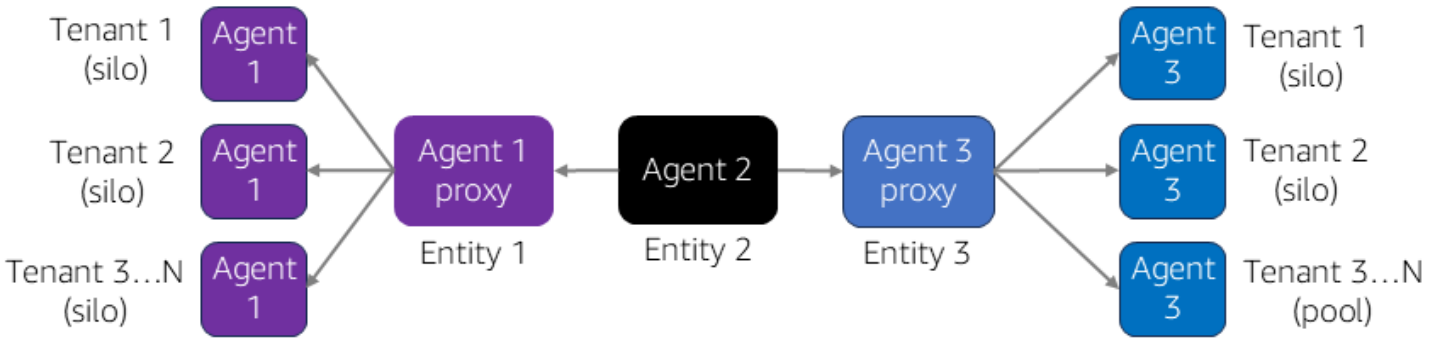


In this diagram, an agent provider has three agents that are deployed through agent as a service (AaaS). They support two types of tenants. On the left-hand side, two tenants have compliance and performance requirements that they address through a full-stack silo model. The remaining tenant on the right-hand side runs in a pooled model where tenants share resources.

If the goal is agility and operational efficiency, try to limit the effects associated with supporting per-tenant deployment models. This means putting routing and other experience mechanisms in place that allow agents to be managed, operated, and deployed through a single pane of glass.

If you build an agent in a low- or no-code environment, there will be no notion of siloed or pooled agents. Instead, agents may be fully managed by another agent. Siloed and pooled models apply more to environments where an organization controls the agent's construction and footprint. In this case, teams should consider which deployment model to support.

On the surface, these deployment models don't directly affect how an agent functions in a broader system. An agent may have no direct awareness of other agents that are deployed in a silo or pooled model. Instead, these deployment strategies can be implemented as part of a routing construct within an environment. The following diagram shows an example of how siloed and pooled models can be implemented using a routing strategy.



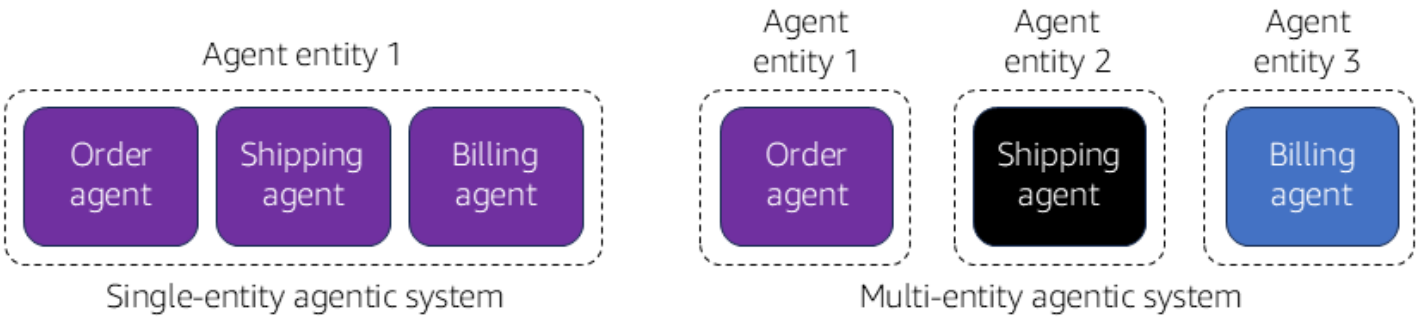
This example includes three agents from three different providers. Each agent provider has the option to implement its own deployment strategy. For example, agent 1 uses a proxy to distribute inbound requests to a set of siloed tenant agents. Agent 2 requires no routing and supports all tenant requests through one pooled agent. Agent 3 is a hybrid-model deployment where some tenants are siloed and others are pooled.

If and how you choose to support these deployment models depends on the nature of your solution. You may have no need to support either model. You may, however, have instances where you must consider supporting this strategy, such as with compliance, noisy neighbor, performance, or tiering.

Introducing and applying tenant context

If we build agents that support multi-tenancy, we must start by considering how to set up tenant context, which will be used to apply tenant-specific policies, strategies, and mechanisms within the agent's implementation.

At the most basic level, you can introduce tenant context into agents though the common tools and mechanisms that we use in classic multi-tenant architectures. This could be through an API key, OAuth, or various other validation mechanisms. Many examples of this focus on resolving an authenticated system or user to a JSON web token (JWT) key that holds tenant context. The JWT is then propagated through the system. This gets more interesting when we consider how to compose agentic systems. The following diagram shows an example of two varieties of agentic environments.



In this diagram, the model on the left-hand side represents an agentic system where all of the agents are owned, managed, and hosted by a single entity. When you have full control of the entire experience, you can use typical strategies to pass tenants through each agent.

The model on the right-hand side, which may be more common, represents a system of agents that span multiple entities. The agents are independently built, managed, and operated, so they each have their own authentication and authorization schemes. The challenge here is that we need a universal way to resolve and share tenant context among these agents. This relies on a more distributed model where each agent must be able to authenticate systems or users and resolve them to a tenant according to applied mechanisms.

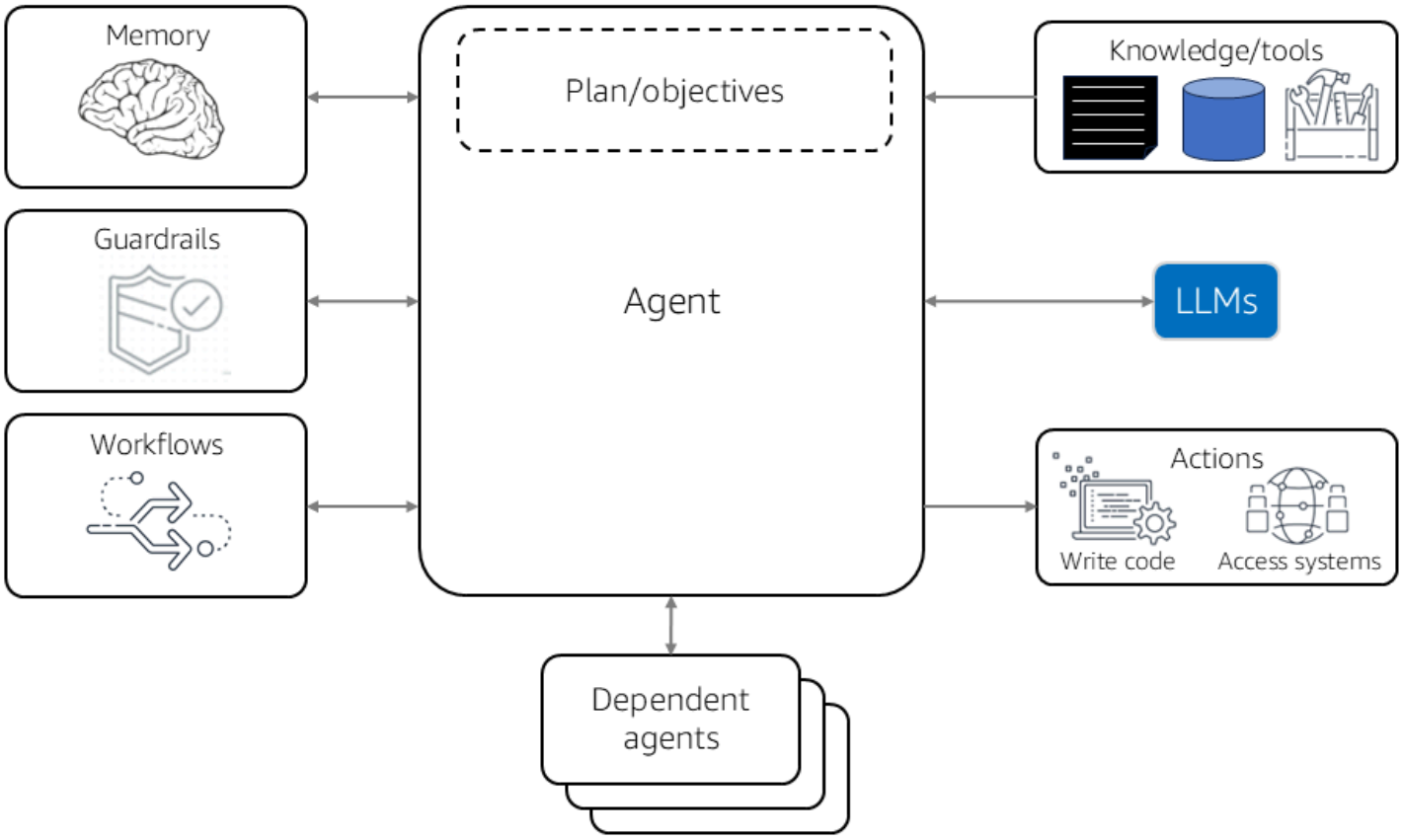
Building tenant-aware agents

Multi-tenancy influences how we implement individual agents. As an agent processes requests, consider how tenant context affects how an agent accesses data, makes decisions, and invokes

actions. To better understand how and where multi-tenancy affects your agent's profile, first determine how constructs can be part of any agent.

The challenge is that the scope, nature, and design of agents is anything but concrete because providers make their own choices about the design of an agent experience. Ultimately, the point of an agent is that it's an autonomous learning service that can access a range of tools, data sources, and memory to determine how best to solve a task.

It's less important to know exactly which strategies and patterns an agent uses. In a multi-tenant model, it's more important to identify how various parts of an agent are configured, accessed, and applied. Consider a potential agent environment that relies on a series of resources and mechanisms to achieve its goals. The following diagram shows an example of such an agent.

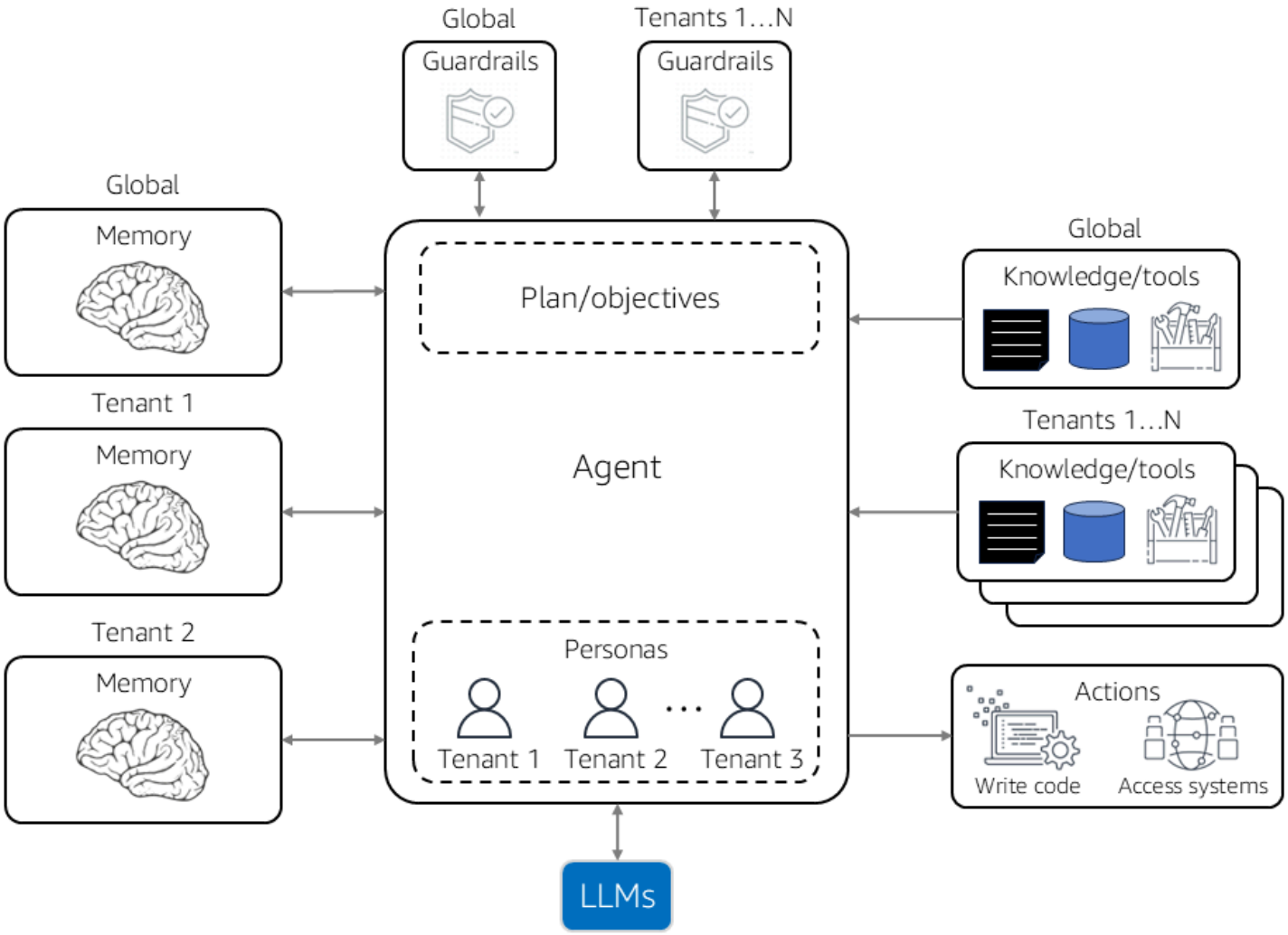


This diagram represents a comprehensive range of agentic possibilities, showcasing various tools and mechanisms that could be combined to accomplish a goal. On the left-hand side of the diagram, note how an agent depends on memory as part of its context, guardrails for defining the policies that guide its activities, and workflows that are directed at specific tasks. Some might argue that workflows shouldn't be included in this context, but there may be scenarios where workflows are integral to an agentic experience.

The right side of the diagram shows how inputs like knowledge and tools can supply additional insights and context that enhance the agent's capabilities. The agent then outputs actions, such as writing code or accessing systems. The bottom of the diagram shows how agents depend on one or more internal or third-party agents that can be orchestrated as part of a broader system.

We can now think about what it means to introduce multi-tenancy. Tenancy forces us to consider how and where an agent introduces strategies and mechanisms that dictate behaviors and actions. This adds another dimension to how we think about agents in terms of their knowledge, learning, tools, and memory.

Let's now consider how to modify this model to support multi-tenancy. The following diagram shows an example of a multi-agent model.



In this diagram, we introduce tenant personas that are intended to shape how an agent integrates tenant context. For example, on the left-hand side of the diagram, agent memory is altered to support tenant-specific memory. The same is true on the right-hand side of the diagram where

the agent supports tenant-specific knowledge and tools. The same support is also applied to guardrails.

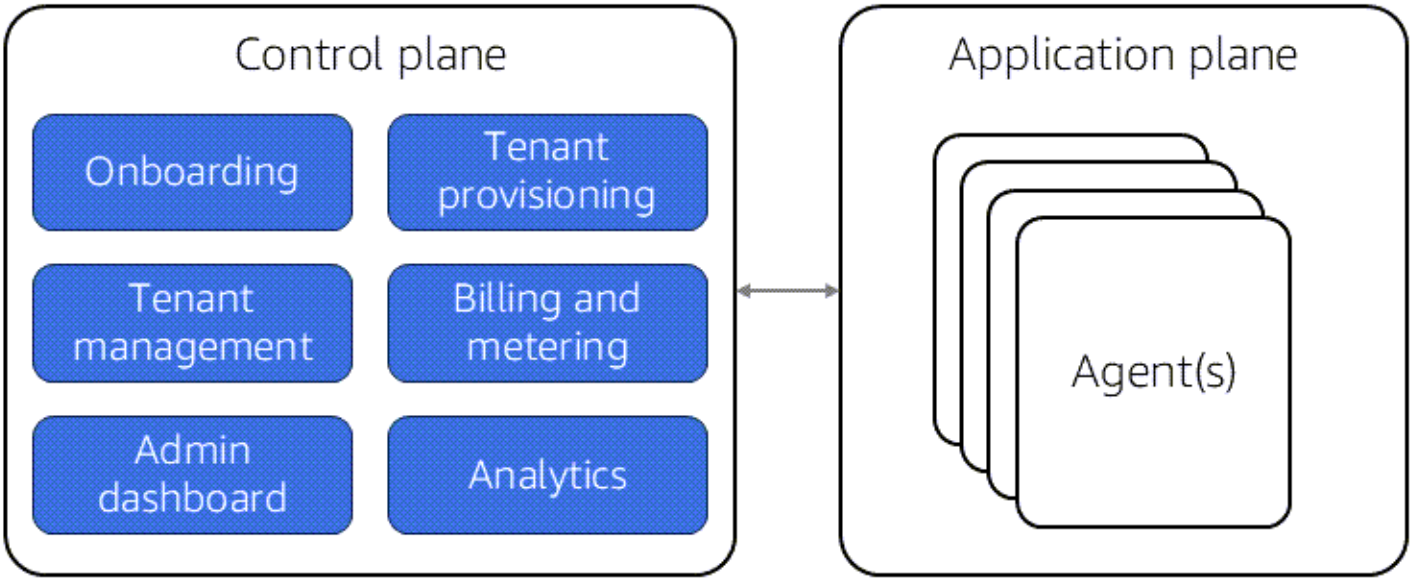
This may be an extreme example because not all aspects of a multi-tenant agent require per-tenant resources. The point is that you should consider how tailoring your agent for specific tenants can enhance its effectiveness. This approach allows your agent to increase its impact and value, provide more relevant context in its responses, and develop specialized capabilities. The agent will then be able to learn, adapt, and perform tasks that are uniquely suited to different personas.

The main idea is that tenant context directly affects how you build agents. It can also shape tenant interactions with external entities, including other agents. Building a multi-tenant agent introduces traditional challenges such as noisy neighbors, tenant isolation, tiering, throttling, and cost management. Your agent's design and architecture must address these foundational multi-tenant concepts, which we'll explore in the next section.

Employing control planes in agentic environments

Multi-tenant best practices often divide implementations into two distinct parts: a control plane and an application plane. The control plane provides a single pane of glass to access operational, management, and orchestration mechanisms that span the environment's tenants. The application plane is where the business logic, features, and functional capabilities reside.

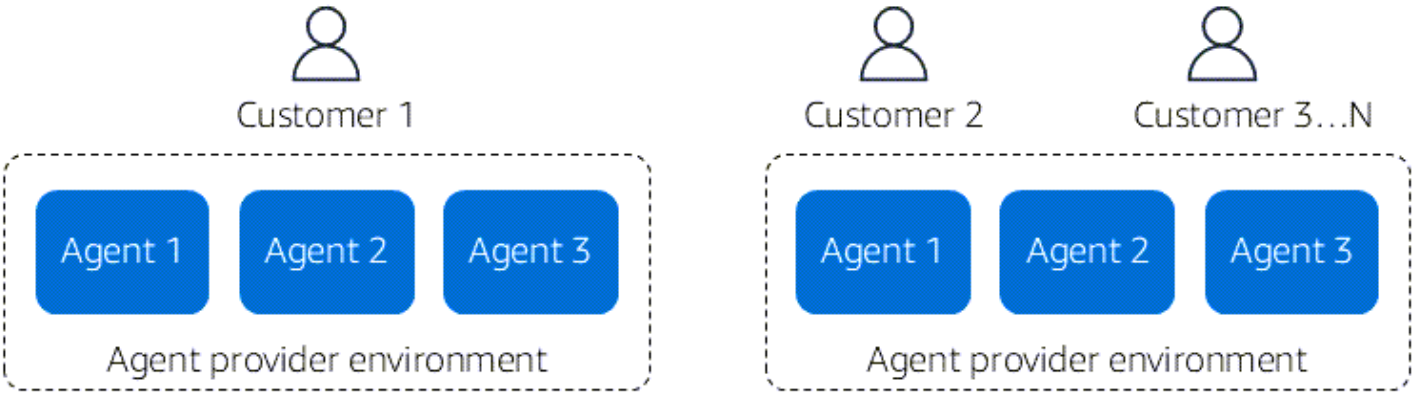
This division of responsibility also applies to agentic models. A multi-tenant agent requires a degree of centralized management, operation, and insights, and it makes sense to continually address these needs through a control plane. The following diagram shows a conceptual view of how these planes are divided within an agent as a service (AaaS) environment.



This diagram shows the traditional separation of control and application planes. What's new is that the control plane now manages the agents that make up an AaaS environment. The control plane interacts with all agents because we presume that the agents are built, managed, and deployed by one provider.

This model introduces additional layers of complexity, especially in agent lifecycle and third-party coordination, but retains the foundational separation of concerns. The control plane still provides the same core capabilities by orchestrating the configuration of agents, providing tenant and agent observability, collecting consumption and metering data for billing, and managing tenant policies.

This scenario becomes more complex if you consider a multi-agent system that incorporates agents from various providers. The following diagram shows an example of such a model.



This diagram depicts four agents from different providers that are part of multi-agentic system. Third-party providers still operate and deploy each agent, which are configured to enable authorized access from one or more providers. The agents, however, remain under the provider's control, so each agent maintains its own control plane.

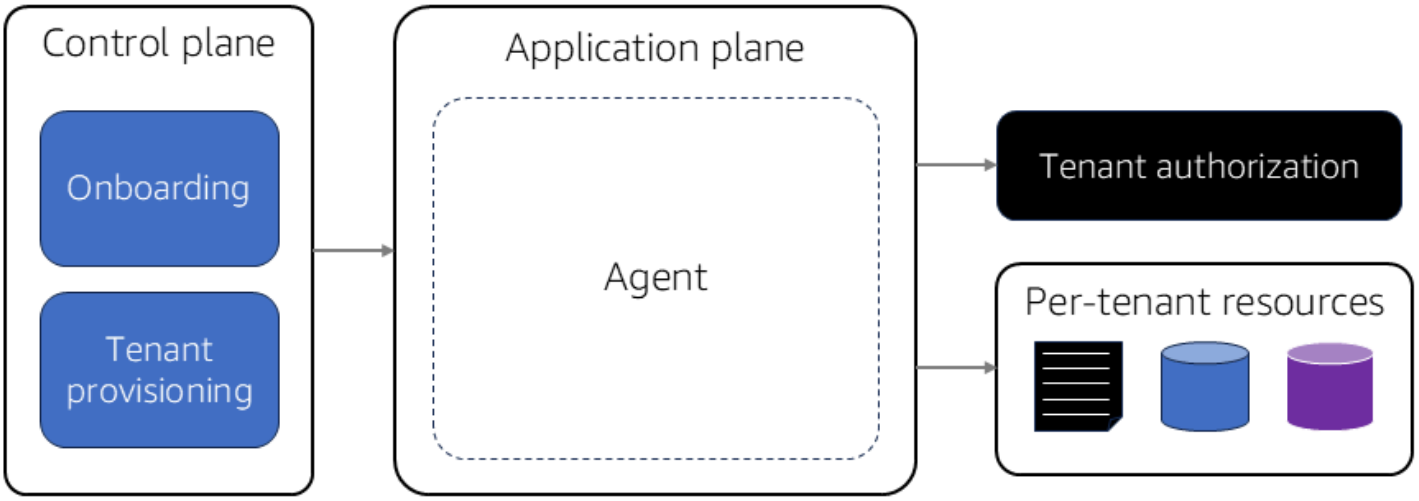
Essentially, these multi-tenant agents behave as third-party services that integrate with other agents. As such, they must have their own control plane to provide the centralized operation, configuration, and management of an agent's capabilities.

We assume that agents are independent services that run in a provider-hosted experience. But this may be unclear in a scenario where an agent consumer imposes more constraints on how and where to host an agent.

Onboarding tenants to agents

Onboarding is typically a vital part of any AaaS environment. How you create, configure, and provision tenants often involves many moving parts, integrations, and tools. The agent onboarding experience may require the same services that are found in an AaaS control plane, which includes tenant identity, tiering, provisioning per-tenant resources, and configuring tenant policies.

Your approach to agent onboarding is influenced by the footprint and tenancy model of your agentic environment. Siloed and pooled agents each have their own nuances, and the choice of using either a single agent or multiple agents also affects the onboarding process. The following diagram shows a conceptual view of how onboarding affects an agent's configuration.



Each time you onboard an agent, the control plane must take the necessary steps to enable the tenant to access the agent. How to introduce tenants varies based on the agent authorization model, but assume that you'll create a tenant identity that associates agent requests with individual tenants. This tenant context dictates the agent experience by applying it to routes, scopes, and control access.

Onboarding may also require you to configure any per-tenant resources that an agent uses. It's here that the control plane's tenant provisioning service connects your agent to tenant-specific data and resources that the agent consults.

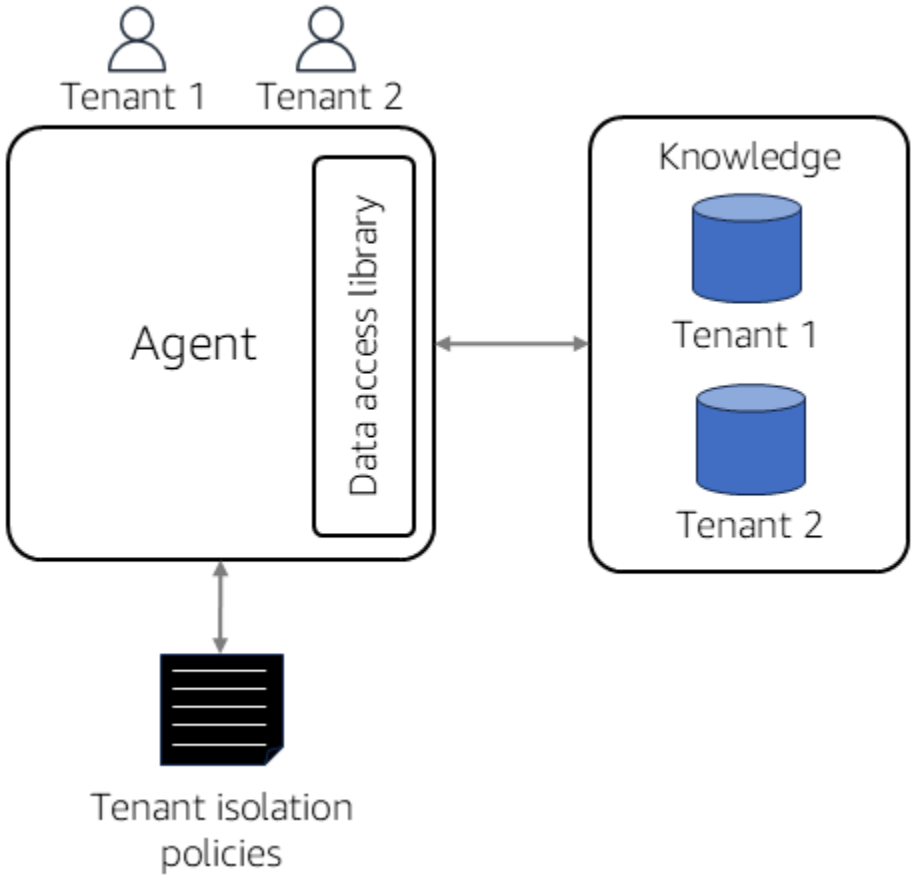
If your system relies on integrating third-party agents, you must also address the needs of those agents during the onboarding process. How this works depends on the security and integration mechanisms for authorizing access between agents. Ideally, the required steps to orchestrate and configure agent-to-agent authentication and authorization are addressed through automated onboarding.

Enforcing tenant isolation

Tenant isolation is a concept that applies to all multi-tenant settings. It means that your policies and strategies ensure that one tenant can't access other tenant resources. For multi-tenant agents, you may need to introduce constructs and mechanisms that help enforce and agent's tenant isolation requirements.

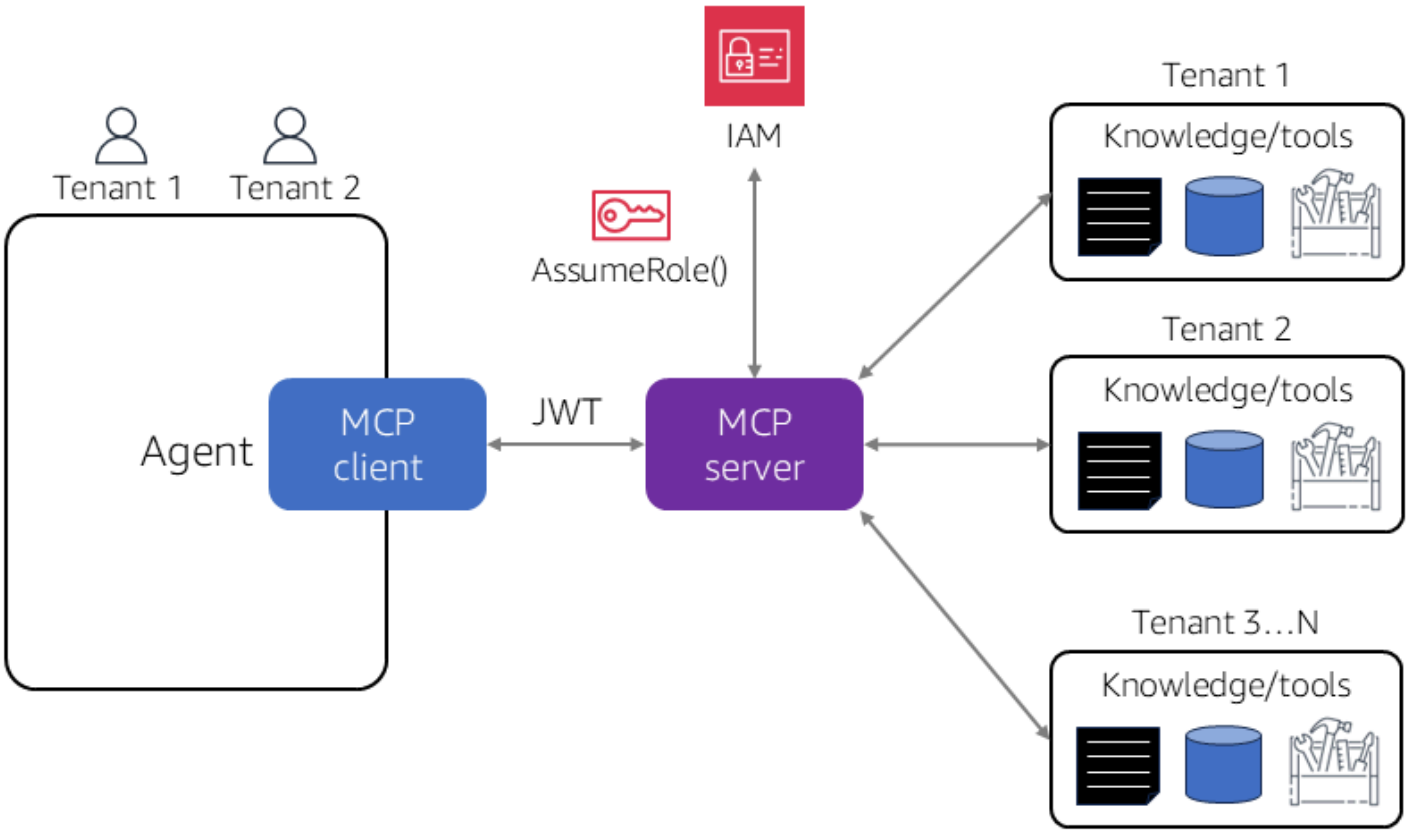
Applying tenant isolation is like other strategies that use traditional multi-tenant systems. Generally, when you construct an AaaS architecture, identify any area in your system where a request or action can access resources to determine if the request crosses any tenant boundaries. For example, microservices may have dependencies on per-tenant, dedicated Amazon DynamoDB tables. This requires you to introduce policies that ensure that one tenant's table can't be accessed by another tenant.

In this instance, consider tenant isolation through an agent lens and its interactions with any of its per-tenant resources. The following diagram shows a conceptual example of how agents apply tenant isolation policies to control access to tenant resources.



On the right-hand side of this diagram, the agent has per-tenant knowledge that's stored in separate vector databases. As the agent processes a request, it examines the context of the tenant making the request. Based on this, the agent applies an appropriate isolation policy to ensure that tenants are restricted from accessing data or resources outside of their designated boundaries.

If your agent uses a Model Context Protocol (MCP), it can also implement your tenant isolation model. The following diagram shows an example of how to introduce MCP and apply isolation policies.



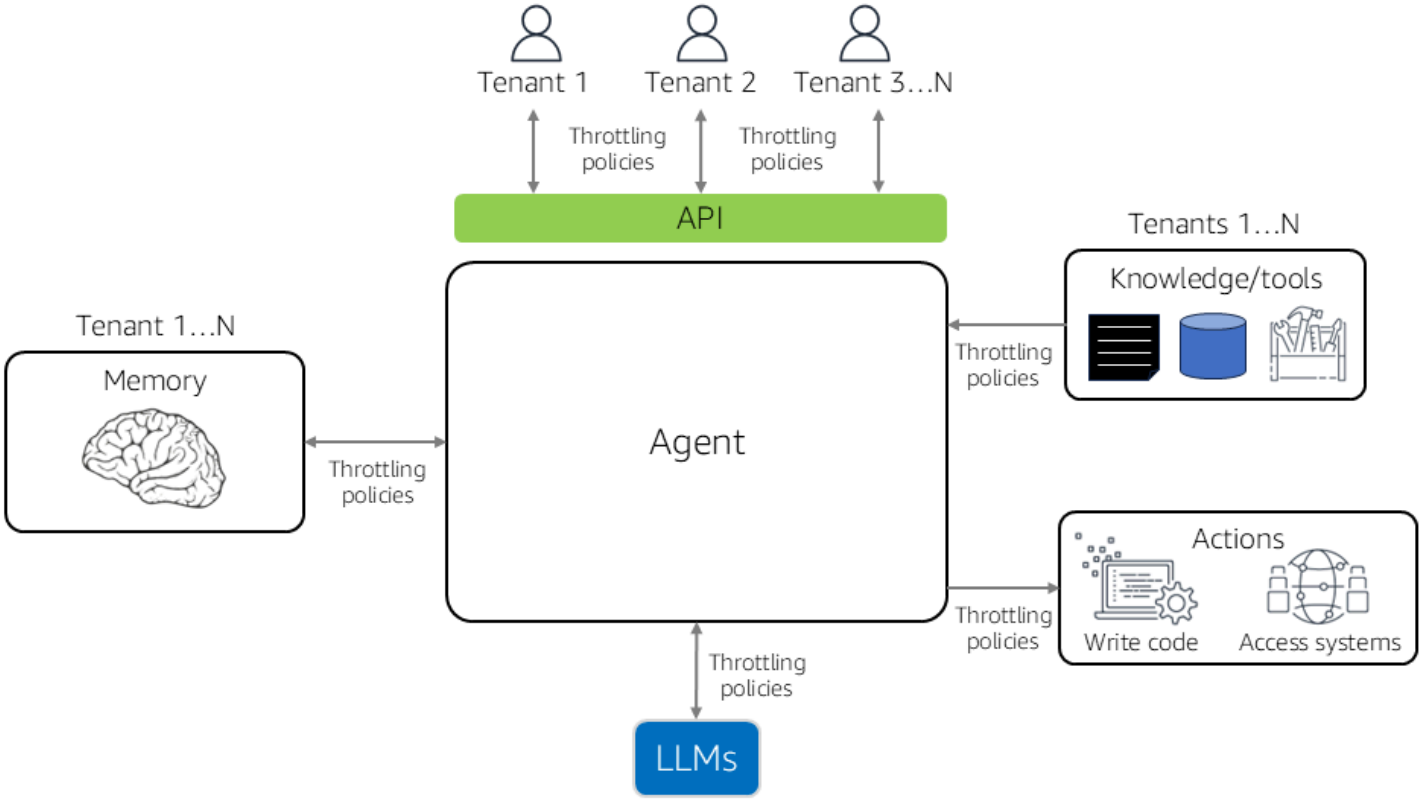
MCP is a standardized protocol that an agent uses to integrate with any tools, data, and resources. In this example, an MCP client and MCP server interact with the tenant-specific knowledge and tools shown on the right side of the diagram. The tenant context flows from client to server, and the server uses this context to acquire tenant-scoped credentials from the AWS Identity and Access Management (IAM) service. The credentials control access to each tenant's resources, ensuring that one tenant can access another tenant's resources.

As agents incorporate multi-tenancy, they must introduce mechanisms that apply tenant isolation policies as they process requests. In some instances, IAM can help limit access to tenant resources. In other instances, you may need to introduce other tools or frameworks to apply tenant isolation policies.

Noisy neighbor and agents

In a multi-tenant AaaS environment where multiple tenants share an agent, think about where and how to introduce policies that prevent noisy neighbor conditions. Policies can introduce general-purpose throttling that applies to all consumption, or you can have tenant or tier-based policies that apply throttling based on a given persona. You might place greater consumption restrictions on basic-tier tenants than you would on premium-tier tenants.

This notion of throttling can be applied at multiple architecture points. The following diagram shows an example of some possible areas to introduce noisy neighbor policies.



In our prior review of multi-agent implementation, we examined different resources that your agent can use, highlighting the potential for per-tenant resources within an agent. Each touchpoint is a potential area to introduce throttling policies, which helps to ensure that tenants don't exceed the consumption limits of your system or a tenant's tiering policies.

The best places to introduce noisy neighbor protections are at points in the architecture where tenants share resources. These shared or pooled components, such as compute, memory, APIs, and large language models, are the most susceptible to performance degradation if a single tenant consumes disproportionately.

One natural place to apply throttling is at the agent's entry point, sometimes called the "outer edge." Here, you can introduce global or tenant-tier-based rate limits before the agent begins processing the request. Throttling can also be applied deeper in the execution path, such as when the agent calls an LLM, accesses memory, or invokes shared tools.

These policies help you enforce fair usage, maintain agent resilience under load, and preserve a consistent experience across tenants. Depending on your goals, you might focus on general system protection (resilience) or on granularly managing the tenant experience (for example, with tier-based entitlements).

Data, operations, and testing

Agents and data ownership

A review of agent implementation highlights scenarios where an agent relies on a given tenant's data. In this instance, consider the data lifecycle and, more importantly, where it's stored. This is especially important for industries and use cases where the data's nature influences how an agent accesses it.

AaaS providers must evaluate how to resolve data issues in a multi-tenant environment, which can affect an agent's onboarding, isolation, and operations. Applicable nuances and strategies vary according to the tools, technologies, and data that you consume. You can approach this in many ways, which is something to be aware of as you create any AaaS offering.

Multi-tenant agent operations

As you construct agent environments, think about how to operate and manage your agents. As a provider, you need metrics, data, insights, and logs that allow you to monitor an agent's health, scale, and activity. This is more pronounced in a multi-tenant agentic environment where you'll want to understand how individual tenants consume agent resources.

This is even more significant in multi-agent settings when you need insights into agent interactions. Being able to profile and track activities between agents may be essential to troubleshooting issues that affect your system's scale, accuracy, and efficacy.

Operations teams may also profile LLM interactions to derive a better sense of the loads that agents place on LLMs. This data is essential for refining agent implementation. It can also give operational teams a view of how agents and tenancy affect the overall cost profile of a system.

Training and testing multi-tenant agents

One challenge associated with building agents is that they're expected to learn and evolve. It also means that we must test our agent, refine it, and improve its accuracy in advance of moving it into production. There are many areas where you can inspect and assess if your agent is correctly assessing and categorizing intent or choosing and invoking appropriate tools and actions. The list of variables is extensive, but this is ultimately about ensuring that your agent finds outcomes that achieve your goals.

Examining all the moving parts and principles associated with testing agents is beyond this document's scope but note that testing strategies add complexity to multi-tenant AaaS environments. For example, if an agent has data, memory, and other constructs that are contextually applied to each tenant, then an agent's outcomes can be shaped by per-tenant resources.

If you use an agent to simulate a scenario, you may need to expand your simulation for tenant-specific use cases. Correspondingly, you must refine validation procedures to allow for instances where validation criteria differ for each tenant.

Considerations and discussion

Where does SaaS fit?

Industry experts actively debate on how agents influence the software as a service (SaaS) landscape. While it's true that agents are changing software for many systems, it's a stretch to suggest that agents make delivery models obsolete. Some SaaS providers will likely be disrupted by adopting agents, and some may entirely rethink their value proposition, by leaning into an agent as a service (AaaS) model. Others may strike a balance by selectively introducing agents to address specific needs.

This topic is interesting because adopting the best SaaS principles may represent the next evolution of SaaS. This might mean that SaaS is going way, or it might mean that the foundational principles of SaaS are being packaged and realized in an agent-based model. It's probably less important to decide where the terminology ultimately lands, but it seems unlikely that SaaS as a concept will disappear. It's more likely that agents will shape the SaaS footprint.

Ultimately, we must decide which strategies can be applied to AaaS, which means enabling organizations to adopt agentic architectures and business strategies so that providers can maximize the efficiency, value, and impact of their agentic systems. Agents aren't black boxes. Agents consume resources, scale operations, depend on data, and generate costs—all factors that providers must address. Agent providers must evaluate how multi-tenant principles can shape service offerings and optimize operational models.

Discussion

The agentic landscape continues to evolve with designs varying based on domains, intended use cases, and target industries. Part of this evolution includes further refining our view of strategies, patterns, and tradeoffs that architects consider as they design and build agents.

A comprehensive agent strategy must align with both business and technical objectives. This includes defining target markets and personas, establishing pricing and resource management strategies, and determining how agents fit into larger systems. These considerations are particularly important when delivering AaaS, where scale, cost efficiency, and innovation are primary goals.

Operational capabilities are equally important. The environment must support monitoring of agent activity, health metrics, and usage patterns. This becomes more complex in multi-agent systems, where operations must be coordinated across independent agents.

Overall, this discussion of agents only scratches the surface of the various architectural considerations that could be part of agentic systems. Beyond selecting appropriate tools, frameworks, and LLMs, success depends on creating an architecture that meets business requirements for scalability, efficiency, deployment, and multi-tenancy.

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
Initial publication	—	July 14, 2025