Cloud Native Technologies

Like a new universe, the cloud native ecosystem has many technologies and projects quickly spinning off and expanding from the initial core of containers. An especially intense area of innovation is workload deployment along with management technologies. While Kubernetes has become the industry standard general-purpose container orchestrator, other technologies like serverless add abstract complexity associated with managing hardware and operating systems. The differences between these technologies are often small and nuanced, which makes it challenging to understand the benefits and tradeoffs.

As defined by the Cloud Native Computing Foundation's (CNCF) charter, cloud native systems have the following properties:

Container packaged: Running applications and processes in software containers as isolated units of application deployment, and as mechanisms to achieve high levels of resource isolation. Improves overall developer experience, fosters code and component reuse, and simplifies operations for cloud native applications.

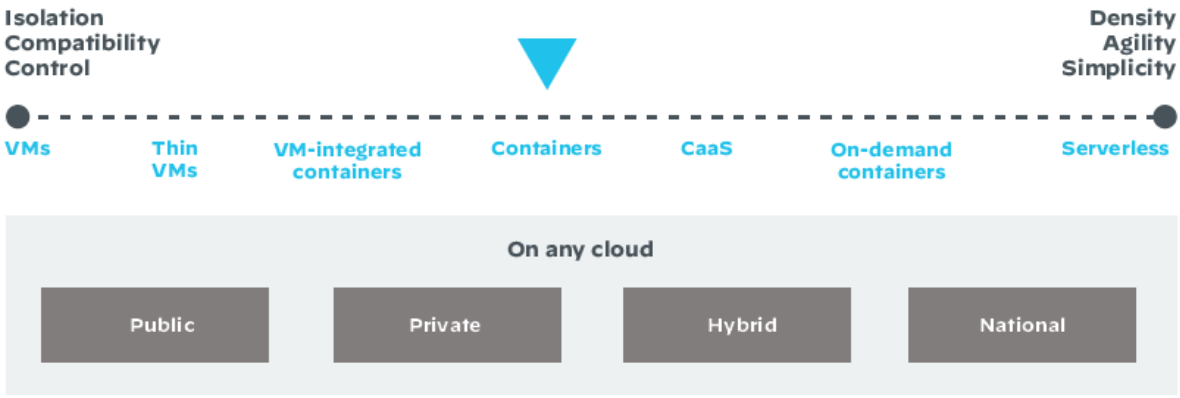
Dynamically managed: Actively scheduled and actively managed by a central orchestrating process. Radically improves machine efficiency and resource utilization while reducing the cost associated with maintenance and operations.

Microservices oriented: Loosely coupled with dependencies explicitly described (for example, through service endpoints). Significantly increases the overall agility and maintainability of applications. The foundation will shape the evolution of the technology to advance the state of the art for application management, and to make the technology ubiquitous and easily available through reliable interfaces.

A useful way to think of cloud native technologies is as a continuum spanning from virtual machines (VMs) to containers to serverless. On one end are traditional VMs operated as stateful entities, as we've done for over a decade now. On the other are completely stateless, serverless apps that are effectively just bundles of app code without any packaged accompanying operating system (OS) dependencies.

In between, there are things like Docker, the new Amazon Web Services (AWS) Fargate service, container-as-a-service (CaaS) platforms, and other technologies that try to provide a different balance between compatibility and isolation on one hand, and agility and density on the other. That balance is the reason for such diversity in the ecosystem. Each technology tries to place the fulcrum at a different point, but the ends of the spectrum are consistent: one end prioritizes familiarity and separation while the other trades off some of those characteristics for increased abstraction and less deployment effort (see Figure 3-2).

Figure 3-2 *The continuum of cloud native technologies*



There's a place for all these technologies—they're different tools with different advantages and tradeoffs, and organizations typically use at least a few of them simultaneously. That heterogeneity is unlikely to change as organizations bring increasingly more-critical workloads into their cloud native stacks, especially those with deep legacy roots.

Virtualization

Virtualization technology emulates real – or physical – computing resources, such as servers (compute), storage, networking, and applications. Virtualization allows multiple applications or server workloads to run independently on one or more physical resources.

A *hypervisor* allows multiple, virtual ("guest") operating systems to run concurrently on a single physical host computer. The hypervisor functions between the computer operating system and the hardware kernel. There are two types of hypervisors:

Type 1 (*native* or *bare-metal*). Runs directly on the host computer's hardware

Type 2 (*hosted*). Runs within an operating system environment

> **Key Terms**
>
> A *hypervisor* allows multiple, virtual (or guest) operating systems to run concurrently on a single physical host computer.
>
> A *native* (also known as a *Type 1* or *bare-metal*) hypervisor runs directly on the host computer's hardware.
>
> A *hosted* (also known as a *Type 2*) hypervisor runs within an operating system environment.

Virtualization is a key technology used in data centers and cloud computing to optimize resources. Important security considerations associated with virtualization include:

Dormant virtual machines (VMs). In many data center and cloud environments, inactive VMs are routinely (often automatically) shut down when they are not in use. VMs that are shut down

for extended periods of time (weeks or months) may be inadvertently missed when anti-malware updates and security patches are applied.

Hypervisor vulnerabilities. In addition to vulnerabilities within the hosted applications, VMs, and other resources in a virtual environment, the hypervisor itself may be vulnerable, which can expose hosted resources to attack.

Intra-VM communications. Network traffic between virtual hosts, particularly on a single physical server, may not traverse a physical switch. This lack of visibility increases troubleshooting complexity and can increase security risks because of inadequate monitoring and logging capabilities.

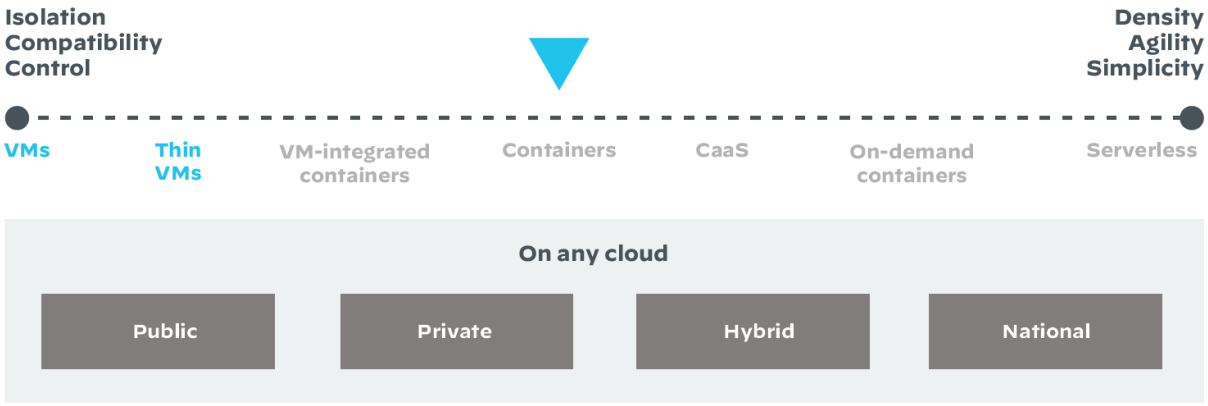
VM sprawl. Virtual environments can grow quickly, leading to a breakdown in change management processes and exacerbating security issues such as dormant VMs, hypervisor vulnerabilities, and intra-VM communications.

Virtual machines

While it may be surprising to see VMs discussed in the context of cloud native, the reality is that the vast majority of the world's workloads today run "directly" (non-containerized) in VMs. Most organizations do not see VMs as a legacy platform to eliminate, nor simply as a dumb host on which to run containers. Rather, they acknowledge that many of their apps have not yet been containerized and that the traditional VM is still a critical deployment model for them. While a VM not hosting containers doesn't meet all three attributes of a cloud native system, it nevertheless can be operated dynamically and run microservices.

VMs provide the greatest levels of isolation, compatibility, and control in the continuum and are suitable for running nearly any type of workload. Examples of VM technologies include VMware vSphere, Microsoft Hyper-V, and the instances provided by virtually every IaaS cloud provider, such as Amazon EC2. VMs are differentiated from "thin VMs" to their right on the continuum (see Figure 3-3) because they're often operated in a stateful manner with little separation between OS, app, and data.

Figure 3-3 *VMs and thin VMs on the continuum of cloud native technologies*



Thin virtual machines

Less a distinct technology than a different operating methodology, “thin” VMs are typically the same underlying technology as VMs but deployed and run in a much more stateless manner. Thin VMs are typically deployed through automation with no human involvement, are operated as fleets rather than individual entities, and prioritize separation of OS, app, and data. Whereas a VM may store app data on the OS volume, a thin VM would store all data on a separate volume that could easily be reattached to another instance. While thin VMs also lack the container attribute of a cloud native system, they typically have a stronger emphasis on dynamic management than traditional VMs. Whereas a VM may be set up and configured by a human operator, a thin VM would typically be deployed from a standard image, using automation tools like Puppet, Chef, or Ansible, with no human involvement.

Thin VMs are differentiated from VMs to their left on the continuum (see Figure 3-3) by the intentional focus on data separation, automation, and disposability of any given instance. They’re differentiated from VM-integrated containers to their right on the continuum by a lack of a container runtime. Thin VMs have apps installed directly on their OS file system and executed directly by the host OS kernel without any intermediary runtime.

Containers and orchestration

Developers have widely embraced containers because they make building and deploying cloud-native applications simpler than ever. Not only do containers eliminate much of the friction typically associated with moving application code from testing through to production, application code packaged up as containers can also run anywhere. All the dependencies associated with any application are included within the containerized application. That makes a containerized application highly portable across virtual machines or bare-metal servers running in a local data center or in a public cloud.

That level of flexibility enables developers to make huge gains in productivity that are too great to ignore. However, as is the case with the rise of any new IT architecture, cloud-native applications still need to be secured. Container environments bring with them a range of cybersecurity issues involving images, containers, hosts, runtimes, registries, and orchestration platforms, all of which need to be secured.

Kubernetes is an open-source orchestration platform that provides an API that enables developers to define container infrastructure in a declarative fashion – that is, infrastructure as code (IaC). Leveraging Kubernetes orchestration and a microservices architecture, organizations can publish, maintain, and update containerized cloud-native applications rapidly and at scale.

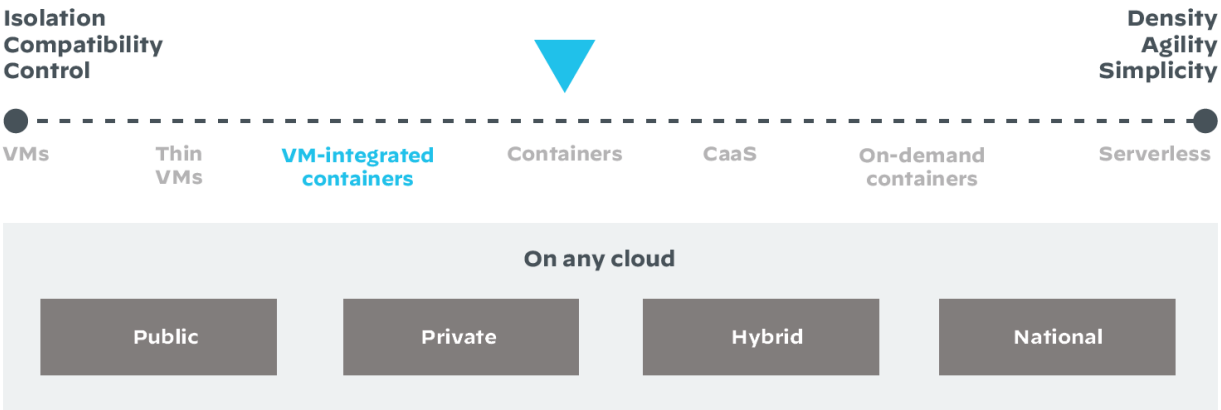
VM-integrated containers

For some organizations, especially large enterprises, containers provide an attractive app deployment and operational approach but lack sufficient isolation to mix workloads of varying sensitivity levels. Recently discovered hardware flaws like Meltdown and Spectre aside, VMs provide a much stronger degree of isolation but at the cost of increased complexity and management burden. VM-integrated containers, like Kata containers and VMware vSphere Integrated Containers, seek to accomplish this by providing a blend of a developer-friendly API and abstraction of app from the OS while hiding the underlying complexities of compatibility and security isolation within the hypervisor.

Basically, these technologies seek to provide VMs without users having to know they're VMs or manage them. Instead, users execute typical container commands like "docker run," and the underlying platform automatically and invisibly creates a new VM, starts a container runtime within it, and executes the command. The end result is that the user has started a container in a separate operating system instance, isolated from all others by a hypervisor. These VM-integrated containers typically run a single container (or set of closely related containers akin to a pod in Kubernetes) within a single VM. VM-integrated containers possess all three cloud native system attributes and typically don't even provide manual configuration as an optional deployment approach.

VM-integrated containers are differentiated from thin VMs to their left on the continuum (see Figure 3-4) because they're explicitly designed to solely run containers and tightly integrate VM provisioning with container runtime actions. They're differentiated from pure containers to their right on the continuum by the mapping of a single container per OS instance and the integrated workflow used to instantiate a new VM, along with the container it hosts, via a singular, container-centric flow.

Figure 3-4 *VM-integrated containers on the continuum of cloud native technologies*



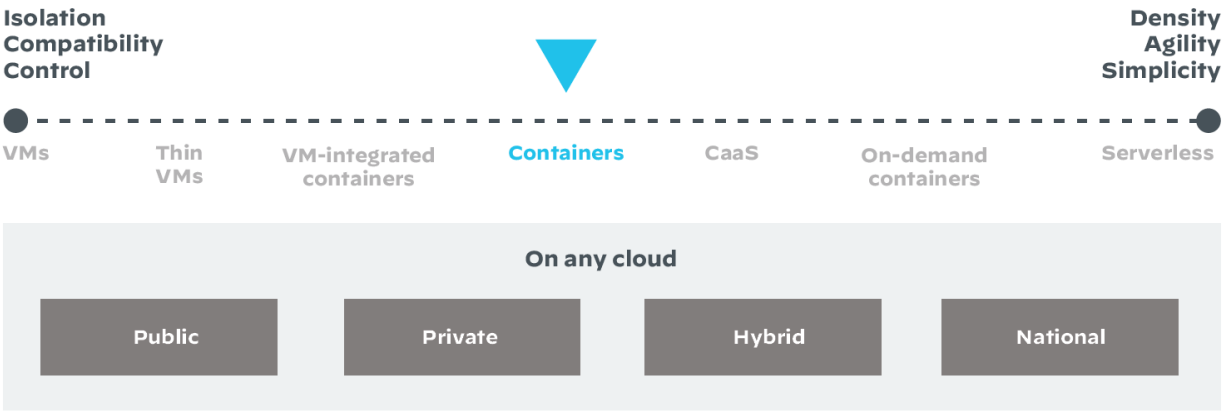
Containers

Containers deliver all three cloud native system characteristics as well as provide a balanced set of capabilities and tradeoffs across the continuum. Popularized and best known by the Docker project, containers have existed in various forms for many years and have their roots in technologies like Solaris Zones and BSD Jails. While Docker is a well-known brand, other vendors are adopting its underlying technologies of runc and containerd to create similar but separate solutions.

Containers balance separation (though not as well as VMs), excellent compatibility with existing apps, and a high degree of operational control with good density potential and easy integration into software development flows. Containers can be complex to operate, primarily due to their broad configurability and the wide variety of choices they present to operational teams. Depending on these choices, containers can be either completely stateless, dynamic, and isolated; highly intermingled with the host operating system and stateful; or anywhere in between. This degree of choice is both the greatest strength and the great weakness of containers. In response, the market has created systems to their right on the continuum—such as serverless—to both make them easier to manage at scale and abstract some of their complexity by reducing some configurability.

Containers are differentiated from VM-integrated containers to their left on the continuum (see Figure 3-5) by neither using a strict 1:1 mapping of container to VM nor wrapping the provisioning of the underlying host operating system into the container deployment flow. They're differentiated from container-as-a-service platforms to their right on the continuum by requiring users to be responsible for deployment and operation of all the underlying infrastructure, including not just hardware and VMs but also the maintenance of the host operating systems within each VM.

Figure 3-5 *Containers on the continuum of cloud native technologies*



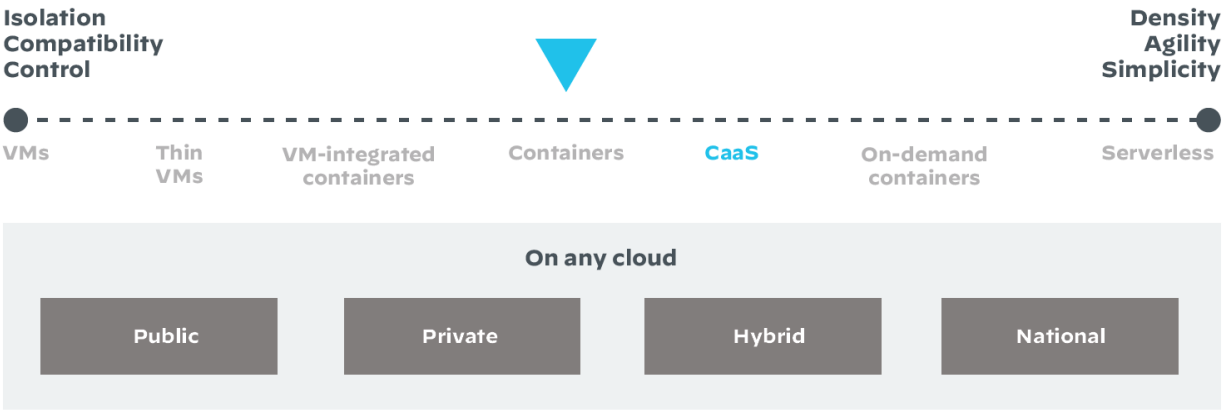
Containers as a Service

As containers grew in popularity and use diversified, orchestrators like Kubernetes (and its derivatives like OpenShift), Mesos, and Docker Swarm became increasingly important to deploy and operate containers at scale. While abstracting much of the complexity required to deploy and operate large numbers of microservices composed of many containers and running across many hosts, these orchestrators themselves can be complex to set up and maintain. Additionally, these orchestrators are focused on the container runtime and do little to assist with the deployment and management of underlying hosts. While sophisticated organizations often use technologies like thin VMs wrapped in automation tooling to address this, even these approaches do not fully unburden the organization from managing the underlying compute, storage, and network hardware. Containers as a Service (CaaS) platforms provide all three cloud native characteristics by default and, while assembled from many more generic components, are highly optimized for container workloads.

Since major public cloud IaaS providers already have extensive investments in lower-level automation and deployment, many have chosen to leverage this advantage to build complete platforms for running containers that strive to eliminate management of the underlying hardware and VMs from users. These CaaS platforms include Google Kubernetes Engine, Azure Kubernetes Service, and Amazon EC2 Container Service. These solutions combine the container deployment and management capabilities of an orchestrator with their own platform-specific APIs to create and manage VMs. This integration allows users to more easily provision capacity without the need to manage the underlying hardware or virtualization layer. Some of these platforms, such as Google Kubernetes Engine, even use thin VMs running container-focused operating systems, like Container-Optimized OS or CoreOS, to further reduce the need to manage the host operating system.

CaaS platforms are differentiated from containers on their left on the continuum (see Figure 3-6) by providing a more comprehensive set of capabilities that abstract the complexities involved with hardware and VM provisioning. They're differentiated from on-demand containers to their right on the continuum by typically still enabling users to directly manage the underlying VMs and host OS. For example, in most CaaS deployments, users can SSH directly to a node and run arbitrary tools as a root user to aid in diagnostics or customize the host OS.

Figure 3-6 *CaaS platform on the continuum of cloud native technologies*



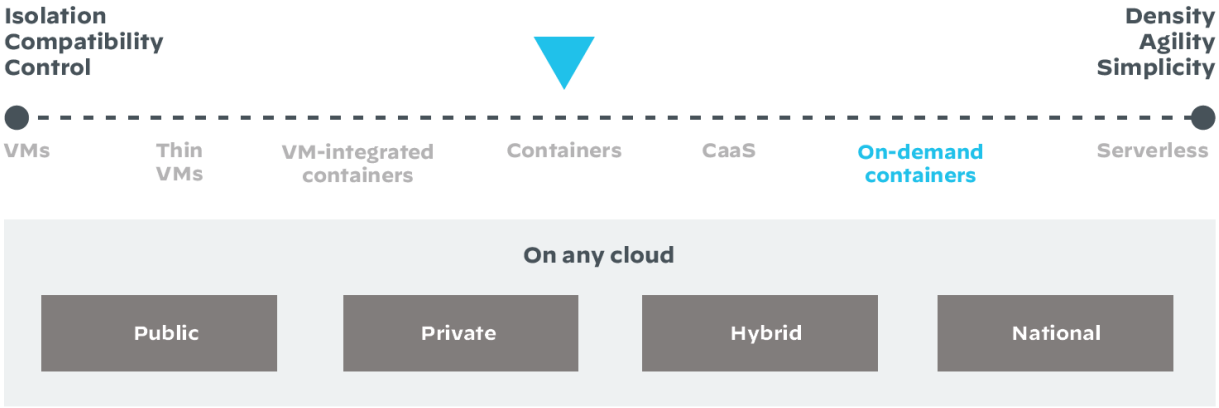
On-demand containers

While CaaS platforms simplify the deployment and operation of containers at scale, they still provide users with the ability to manage the underlying host OS and VMs. For some organizations, this flexibility is highly desirable, but in other use cases it can be an unneeded distraction. Especially for developers, the ability to simply run a container, without any knowledge or configuration of the underlying hosts or VMs can increase development efficiency and agility.

On-demand containers are a set of technologies designed to trade off some of the compatibility and control of CaaS platforms for lessened complexity and ease of deployment. On-demand container platforms include AWS Fargate and Azure Container Instances. On these platforms, users may not have any ability to directly access the host OS and must exclusively use the platform interfaces to deploy and manage their container workloads. These platforms provide all three cloud native attributes and arguably even require them; it's typically impractical not to build apps for them as microservices, and the environment can only be managed dynamically and deployed as containers.

On-demand containers are differentiated from CaaS platforms to their left on the continuum (see Figure 3-7) by the lack of support for direct control of the host OS and VMs, along with the requirement that typical management occurs through platform-specific interfaces. They're differentiated from serverless on their right on the continuum because on-demand containers still run normal container images that could be executed on any other container platform. For example, the same image that a user may run directly in a container on their desktop can be run unchanged on a CaaS platform or in an on-demand container. The consistency of an image format as a globally portable package for apps, including all their underlying OS-level dependencies, is a key difference from serverless environments.

Figure 3-7 *On-demand containers on the continuum of cloud native technologies*



Serverless computing

Serverless architectures (also referred to as function as a service, or FaaS) enable organizations to build and deploy software and services without maintaining or provisioning any physical or virtual servers. Applications made using serverless architectures are suitable for a wide range of services and can scale elastically as cloud workloads grow.

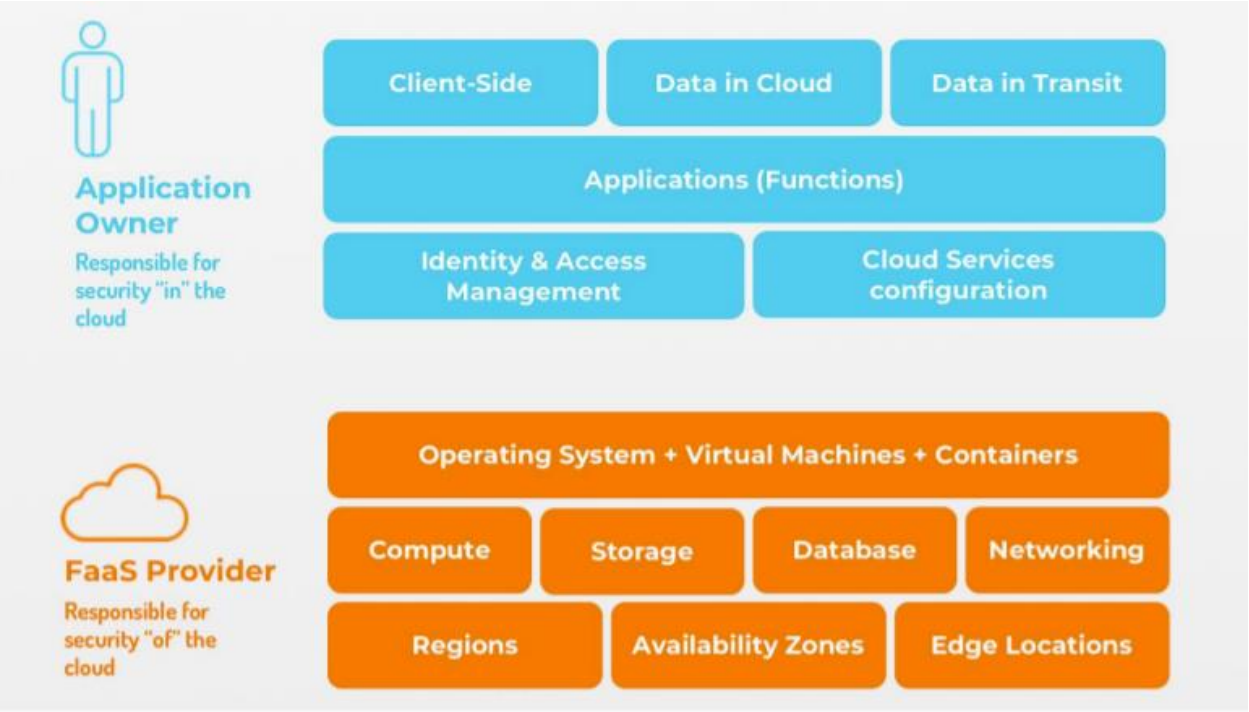
From a software-development perspective, organizations adopting serverless architectures can focus on core product functionality and completely disregard the underlying operating system, application server, or software runtime environment.

By developing applications using serverless architectures, users relieve themselves of the daunting task of continually applying security patches for the underlying operating system and application servers. Instead, these tasks are now the responsibility of the serverless architecture provider.

In serverless architectures, the serverless provider is responsible for securing the data center, network, servers, operating systems, and their configurations. However, application logic, code, data, and application-layer configurations still need to be robust and resilient to attacks. These are the responsibility of application owners (see Figure 3-8).

Figure 3-8

Serverless architectures and the shared-responsibility model



Adopting a serverless model can impact application development in several ways:

- **Reduced operational overhead.** With no servers to manage, developers and DevOps don't need to worry about scaling infrastructure, installing and maintaining agents, or other infrastructure-related operations.
- **Increased agility.** Because serverless applications rely heavily on managed services for things like databases and authentication, developers are free to focus on the business logic of the application, which will typically runs on an FaaS, such as AWS Lambda or Google Cloud Functions.
- **Reduced costs.** With most services used in serverless applications, the customer pays only for usage. For example, with AWS Lambda, customers pay for the executions of their functions. This pricing model typically has a significant impact on cost because customers don't have to pay for unused capacity as they would with virtual machines.

While on-demand containers greatly reduce the "surface area" exposed to end users and, thus, the complexity associated with managing them, some users prefer an even simpler way to deploy their apps. Serverless is a class of technologies designed to allow developers to provide only their app code to a service, which then instantiates the rest of the stack below it automatically.

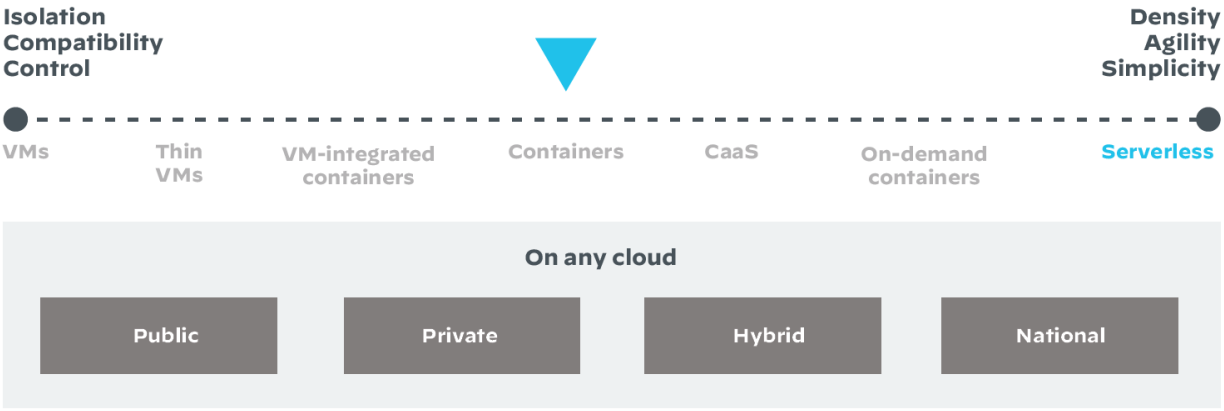
In serverless apps, the developer only uploads the app package itself, without a full container image or any OS components. The platform dynamically packages it into an image, runs the image in a container, and (if needed) instantiates the underlying host OS and VM as well as the hardware required to run them. In a serverless model, users make the most dramatic trade-offs of compatibility and control for the simplest, most efficient deployment and management experience.

Examples of serverless environments include Amazon Lambda and Azure Functions. Arguably, many PaaS offerings, such as Pivotal Cloud Foundry, are also effectively serverless even if they have not historically been marketed as such. While on the surface, serverless may appear to lack the container-specific, cloud-native attribute, containers are extensively used in the underlying implementations, even if those implementations are not exposed to end users directly.

Serverless is differentiated from on-demand containers to the left on the continuum (see Figure 3-9) by the complete inability to interact with the underlying host and container runtime, often to the extent of not even having visibility into the software that the host runs.

Figure 3-9

Serverless on the continuum of cloud-native technologies



Serverless architectures introduce a new set of issues that must be considered when securing such applications, including:

- **Increased attack surface:** Serverless functions consume data from a wide range of event sources, such as HyperText Transfer Protocol (HTTP) application program interfaces (APIs), message queues, cloud storage, Internet of Things (IoT) device communications, and so forth. This diversity increases the potential attack surface dramatically, especially when messages use protocols and complex message structures. Many of these messages cannot be inspected by standard application-layer protections, such as web application firewalls (WAFs).
- **Attack surface complexity:** The attack surface in serverless architectures can be difficult for some to understand, given that such architectures are still somewhat new. Many software developers and architects have yet to gain enough experience with the security risks and appropriate security protections required to secure such applications.
- **Overall system complexity:** Visualizing and monitoring serverless architectures is still more complicated than standard software environments.

- **Inadequate security testing:** Performing security testing for serverless architectures is more complex than testing standard applications, especially when such applications interact with remote third-party services or with backend cloud services, such as Non-Structured Query Language (NoSQL) databases, cloud storage, or stream processing services. Additionally, automated scanning tools are currently not adapted to examining serverless applications. Common scanning tools currently include the following:

    - **Dynamic application security testing (DAST)** tools will only provide testing coverage for HTTP interfaces. This limited capability poses a problem when testing serverless applications that consume input from non-HTTP sources or interact with backend cloud services. Also, many DAST tools inadequately test web services—for example, Representational State Transfer (RESTful)—that don't follow the classic HyperText Markup Language (HTML)/HTTP request/response model and request format.

    - **Static application security testing (SAST)** tools rely on data-flow analysis, control flow, and semantic analysis to detect vulnerabilities in software. Since serverless applications contain multiple distinct functions that are stitched together using event triggers and cloud services (for example, message queues, cloud storage, or NoSQL databases), statically analyzing data flow in such scenarios is highly prone to false positives. Conversely, SAST tools will suffer from false negatives as well, since source/sink rules in many tools do not consider FaaS constructs. These rule sets will need to evolve to provide proper support for serverless applications.

    - **Interactive application security testing (IAST) tools** have better odds at accurately detecting vulnerabilities in serverless applications when compared to both DAST and SAST. However, similar to DAST tools, their security coverage is impaired when serverless applications use non-HTTP interfaces to consume input. Furthermore, IAST solutions require that the tester deploy an instrumentation agent on the local machine, which is not an option in serverless environments.

    - **Traditional security protections (firewall, web application firewall (WAF), intrusion prevention system (IPS)/intrusion detection system (IDS)):** Since organizations that use serverless architectures do not have access to the physical (or virtual) server or its operating system, they cannot deploy traditional security layers, such as endpoint protection, host-based intrusion prevention, WAFs, and so forth. Additionally, existing detection logic and rules have yet to be "translated" to support serverless environments.