Understanding Deployment Choices with Different Models

6.1.1

Introduction to Deployment Choices

At this point in the course, you have learned about the basic concepts behind software development and APIs. Now it is time to look at deploying applications.

Even if you are a solo developer building an application for yourself, when deploying your application you must account for a number of different factors, from creating the appropriate environments, to properly defining the infrastructure, to basic security concepts. This simply means that developers need to do more than deliver application code: they need to concern themselves with how applications are deployed, secured, operated, monitored, scaled, and maintained.

Meanwhile, the physical and virtual infrastructure and platforms on which applications are being developed and deployed are quickly evolving. Part of this rapid evolution is aimed at making life easier for developers and operators. For example, platform paradigms such as Containers as a Service and Serverless Computing are designed to let developers focus on building core application functionality, without having to worry about handling underlying platform configuration, mechanics, scaling, and other operations.

But not all current development takes place on these platforms. Developers are confronted with an expanding "stack" of platform options: bare metal, virtual machines, containers, and others, are all hosted on infrastructures and frameworks of increasing flexibility and complexity.

This module discusses some of the places today's software "lives". It goes on to cover basic techniques for deploying and testing applications, plus workflow techniques and tools for delivering software (to development platforms, test environments, staging, and production) quickly and efficiently. Finally, it covers some networking and security basics. All developers should be familiar with these concepts.

6.1.2

Deployment Environments

Some chaos in the early stages of development is normal, but code should be well tested by the time it gets to users. To make that happen, code needs to go through a series of steps to improve its reliability. Code passes through a number of environments, and as it does, its quality and reliability increases. These environments are self-contained, and intended to mimic the ultimate environment in which the code will 'live'.

Typically, large organizations use a four-tier structure: development, testing, staging, and production.

Development environment

The development environment is where you do your coding. Usually, your development environment bears little resemblance to the final environment. The development environment is typically just enough for you to manage fundamental aspects of your infrastructure, such as containers or cloud networking. You may use an Integrated Development Environment (IDE) or other tool to make deployment easier.

This environment may also include "mock" resources that provide the form of the real resources, but not the content. For example, you might have a database with a minimal number of test records, or an application that mimics the output of a remote service. Each developer typically has their own development environment.

Testing environment

When you believe your code is finished, you may move on to a second environment that has been set aside for testing the code, though when working on small projects, the development and testing environments are often combined. This testing environment should be structurally similar to the final production environment, even if it is on a much smaller scale.

The testing environment often includes automated testing tools such as Jenkins, CircleCI, or Travis CI, as well as integration with a version control system. It should be shared among the entire team. It may also include code review tools such as Gerrit.

Staging environment

After the code has been tested, it moves to the staging environment. Staging should be as close as possible to the actual production environment, so that the code can undergo final acceptance testing in a realistic environment. Instead of maintaining a smaller-scale staging environment, some organizations maintain two matching production environments, one of which hosts the current release of an application, the other standing by to receive a new release. In this case, when a new version is deployed, traffic is shifted (gradually or suddenly, as in "cut over") from the current production environment to the other one. With the next release, the process is done in reverse.

This is, of course, much more affordable in clouds, where an unused, virtualized environment can be torn down and rebuilt automatically when needed.

Production environment

Finally, the code arrives at the production environment, where end users interact with it. At this point it has been tested multiple times, and should be error free. The production environment itself must be sized and constructed to handle expected traffic, including surges that might come seasonally or with a particular event.

Handling those surges is something you can plan for when designing your infrastructure. Before looking at infrastructure, however, you need to know about different models that you can use for deploying your software.

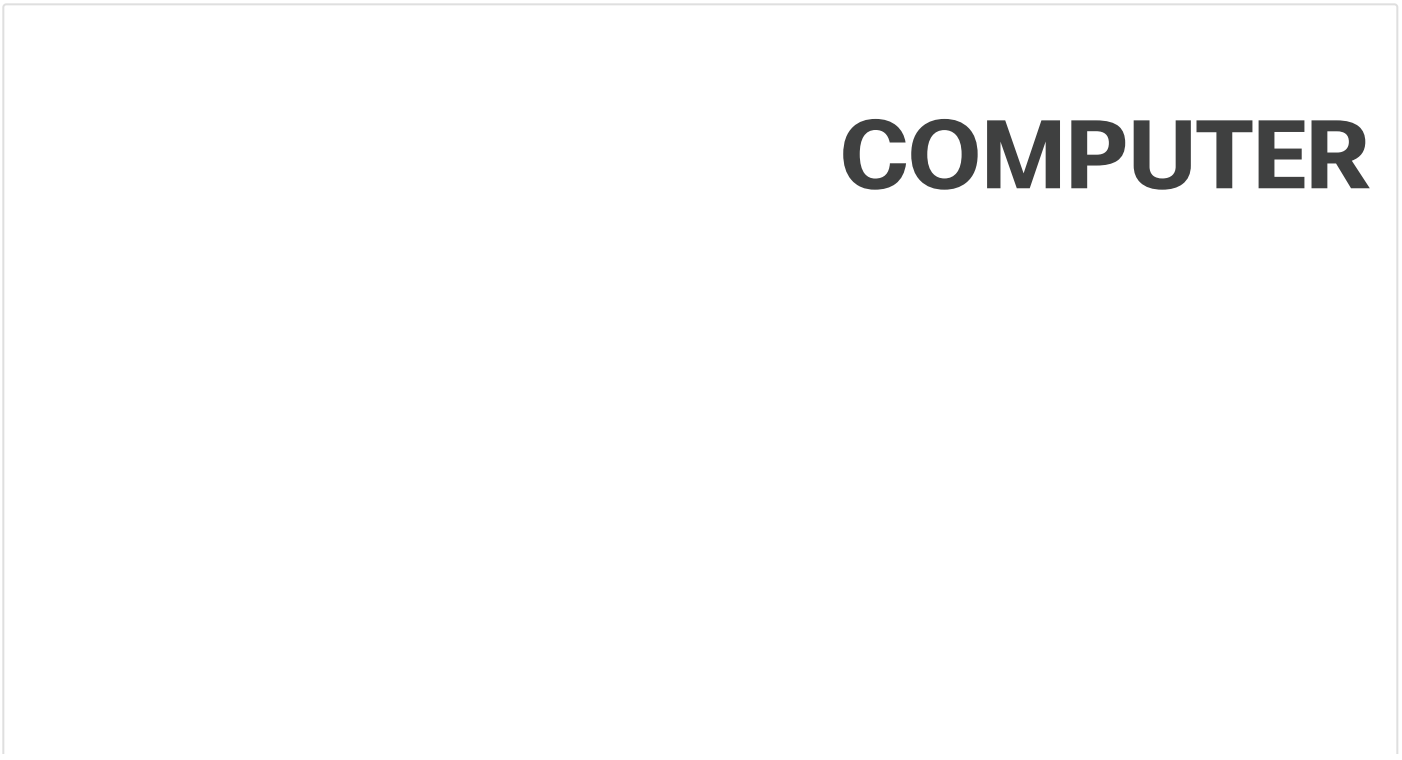
6.1.3

Deployment Models

In the early days of computers, there were no choices regarding how to deploy your software; you simply installed it on the computer itself. Today this model is known as “bare metal,” but it is only one of a variety of options available to you. These options include virtual machines, containers, and newer options such as serverless computing.

Bare metal

The most familiar, and the most basic way to deploy software is by installing it directly on the target computer, or the “bare metal.” In addition to this being the simplest method, bare metal deployment has other advantages, such as the fact that software can access the operating system and hardware directly. This is particularly useful for situations in which you need access to specialized hardware, or for High Performance Computing (HPC) applications in which every bit of speed counts.



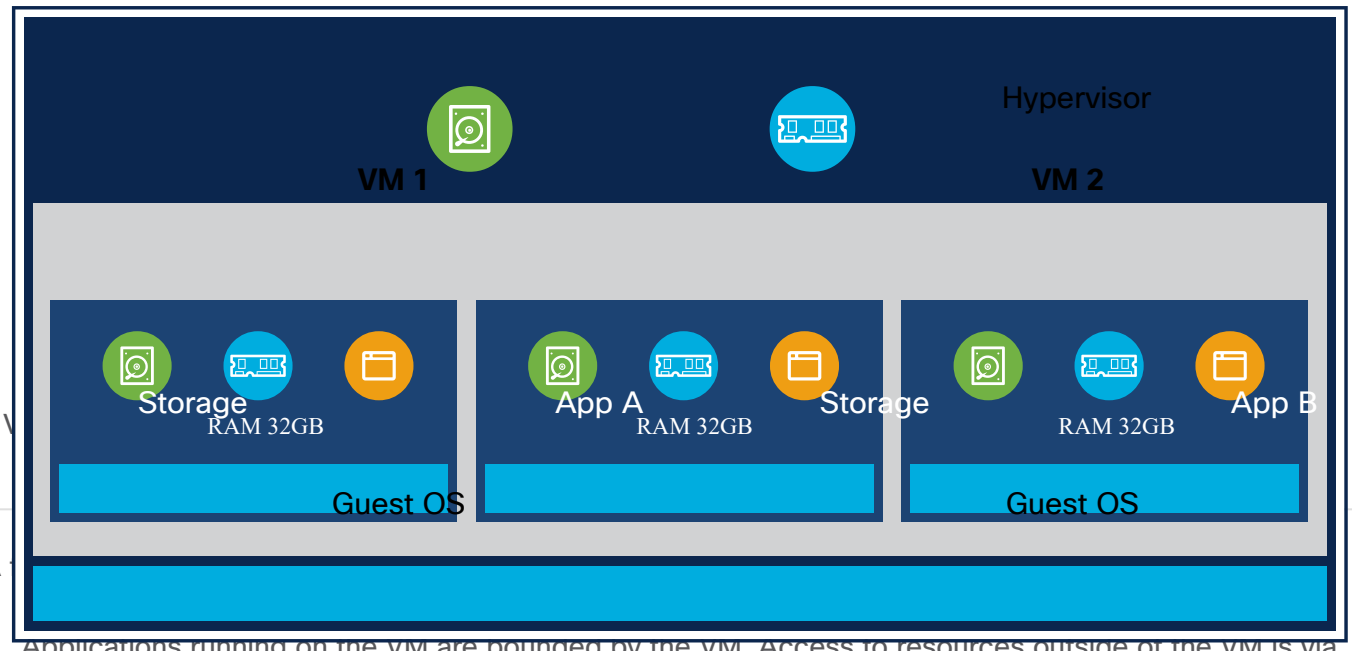
Virtual machines

One way to solve the flexibility and isolation problems is through the use of Virtual Machines, or VMs. A virtual machine is like a computer within your computer; it has its own computing power, network interfaces, and storage.

A hypervisor is software that creates and manages VMs. Hypervisors are available as open-source (OpenStack, Linux KVM, XEN), and also from commercial vendors such as Oracle (VirtualBox), VMware (Horizon, vSphere, Fusion), Microsoft (Hyper-V), and others. Hypervisors are generally classified as either 'Type 1', which run directly on the physical hardware ('bare metal'), and 'Type 2', which run, usually as an application, under an existing operating system.

The use of VMs overcome a number of restrictions. For example, if you had three workloads you wanted to isolate from each other, you could create three separate virtual machines on one bare metal server.

COMPUTER



1. Applications running on the VM are bounded by the VM. Access to resources outside of the VM is via virtual networks.
2. Even though the VMs are running on the same computer, they can run different operating systems from one another (called 'guest operating systems'), and from the bare metal on which VMs are running (called the 'host operating system').
3. The total amount of virtual memory allocated to these three VMs is greater than the amount of RAM available on the host machine. This is called “overcommitting”. This is possible because it is unlikely that all three VMs will need all of their virtual memory at the same time, and the hypervisor can timeshare VMs as needed. Overcommitting can lead to performance issues if resource consumption is too extreme.

VMs run on top of a hypervisor, such as KVM, QEMU, or VMware, which provides them with simulated hardware, or with controlled access to underlying physical hardware. The hypervisor sits on top of the operating system and manages the VMs.

VMs can be convenient for several reasons, not the least of which is that a VM image can be saved for future use, or so that others can instantiate and use. This enables you to distribute a VM, or at least, the means to use it. Applications that run VMs, such as VirtualBox and VMware, can also take snapshots, or backups, of a VM so that you can return it to its previous state, if necessary.

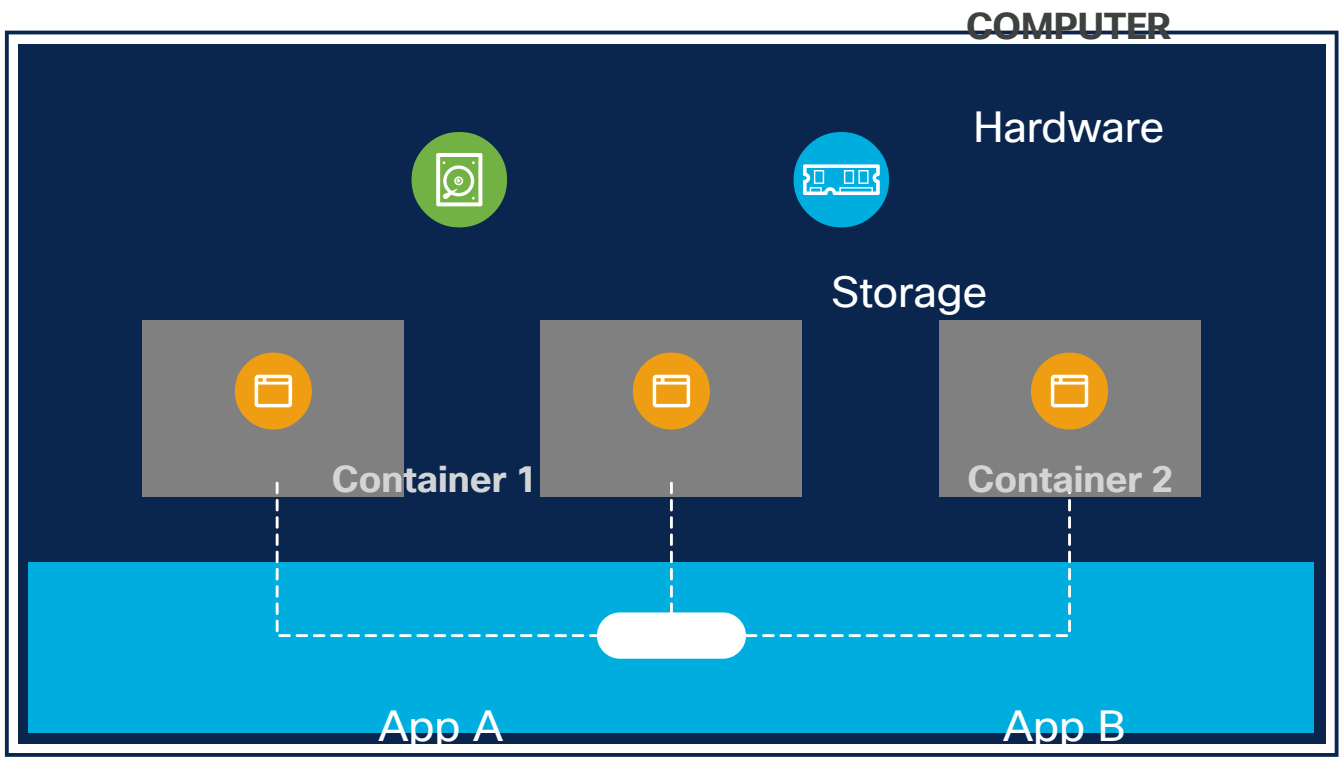
Because they are so much like physical machines, VMs can host a wide range of software, even legacy software. Newer application environments, like containers, may not be "real machine-like" enough to host applications that are not written with their limitations in mind.

Container-based infrastructure

Moving up the abstraction ladder from VMs, you will find containers. Software to create and manage or orchestrate containers is available from Docker, AWS (Elasticized Container Service), Microsoft (Azure Container Service), and others.

Containers were designed to provide many of the same benefits as VMs, such as workload isolation and the ability to run multiple workloads on a single machine, but they are architected a bit differently.

For one thing, containers are designed to start up quickly, and as such, they do not include as much underlying software infrastructure. A VM contains an entire guest operating system, but a container shares the operating system of the host machine and uses container-specific binaries and libraries.



Containers share resources of the host including the kernel

Where VMs emulate an entire computer, a container typically represents just an application or a group of applications. The value of using containers is that all of the libraries and binaries you need to run the application are included, so the user does not have to take that additional installation step.

Kernel

Operating system

An important distinction between a Docker container and a VM is that each VM has its own complete operating system. Containers only contain part of the operating system. For example, you may have an Ubuntu Linux host computer running a CentOS Linux VM, an Ubuntu Linux VM, and a Windows 10 VM. Each of these VMs has its own complete OS. This can be very resource intensive for the host computer.

With Docker, containers share the same kernel of their host computer. For example, on the Ubuntu Linux host computer you may have an Ubuntu Linux container and a Centos Linux container. Both of these containers are sharing the same Linux kernel. However, you could not have a container running Windows 10 on this same Ubuntu Linux host computer, because Windows uses a different kernel. Sharing the same kernel requires far fewer resources than using separate VMs, each with its own kernel.

Containers also solve the problem that arises when multiple applications need different versions of the same library in order to run. Because each application is in its own container, it is isolated from any conflicting libraries and binaries.

Containers are also useful because of the ecosystem of tools around them. Tools such as Kubernetes make fairly sophisticated orchestration of containers possible, and the fact that containers are often designed to be stateless and to start up quickly means that you can save resources by not running them unless you need to.

Containers are also the foundation of cloud native computing, in which applications are generally stateless. This statelessness makes it possible for any instance of a particular container to handle a request. When you add this to another aspect of cloud computing that emphasizes services, serverless computing becomes possible.

Serverless computing

Let's start with this important point: to say that applications are "serverless" is great for marketing, but it is not technically true. Of course your application is running on a server. It is just running on a server that you do not control, and do not have to think about. Hence the name "serverless".

Serverless computing takes advantage of a modern trend towards applications that are built around services. That is, the application makes a call to another program or workload to accomplish a particular task, to create an environment where applications are made available on an "as needed" basis.

It works like this:

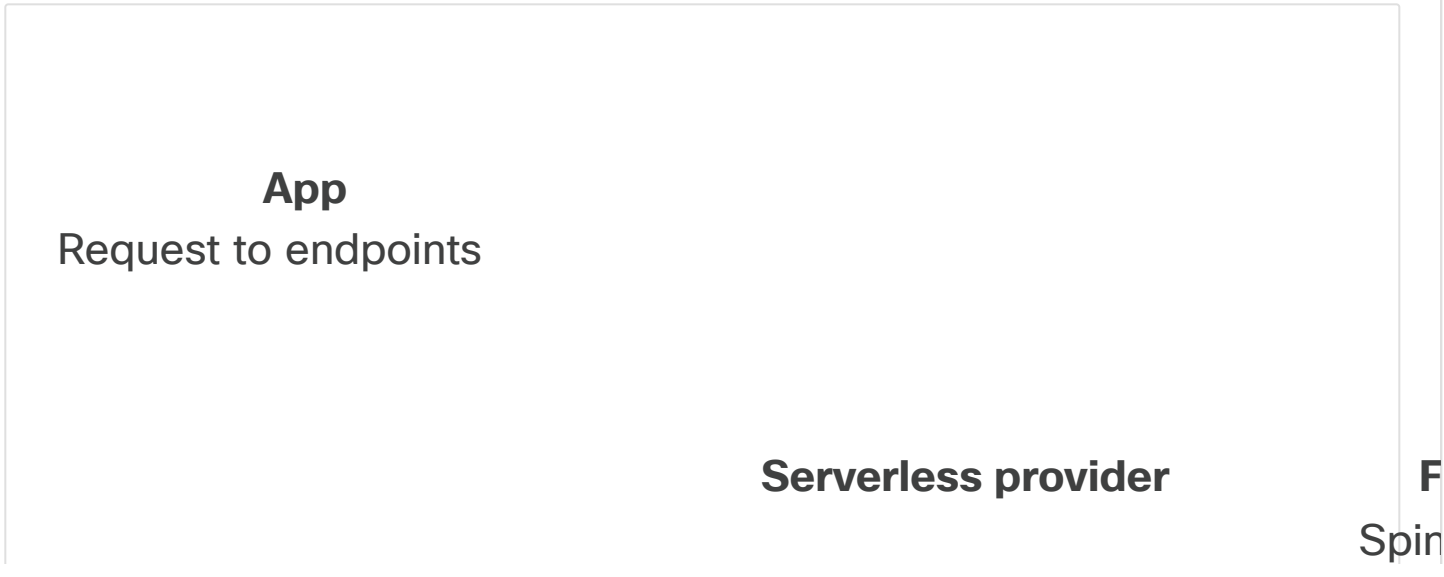
Step 1. You create your application.

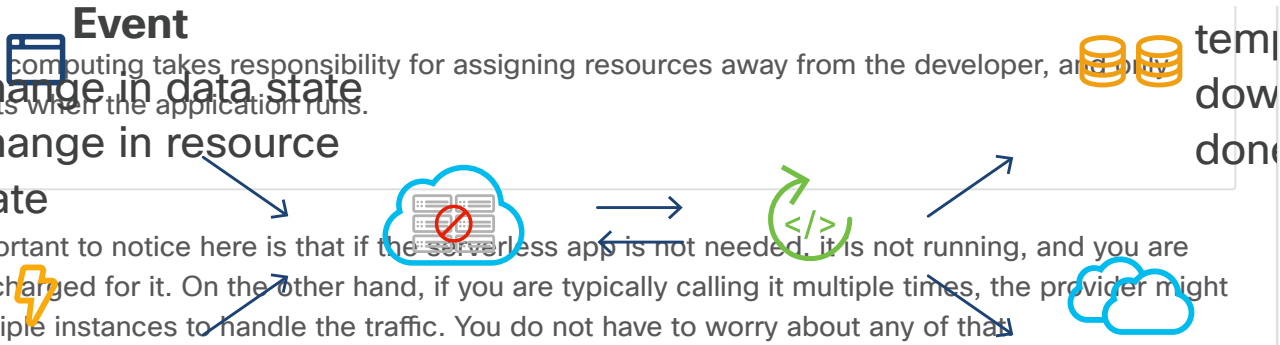
Step 2. You deploy your application as a container, so that it can run easily in any appropriate environment.

Step 3. You deploy that container to a serverless computing provider, such as AWS Lambda, Google Cloud functions, or even an internal Function as a Service infrastructure. This deployment includes a specification of how long the function should remain inactive before it is spun down.

Step 4. When necessary, your application calls the function.

Step 5. The provider spins up an instance of the container, performs the needed task, and returns the result.





Serverless computing takes responsibility for assigning resources away from the developer, and only incurs costs when the application runs.

What is important to notice here is that if the serverless app is not needed, it is not running, and you are not getting charged for it. On the other hand, if you are typically calling it multiple times, the provider might spin up multiple instances to handle the traffic. You do not have to worry about any of that.

Because the capacity goes up and down with need, it is generally referred to as “elastic” rather than “scalable.”

While there is a huge advantage in the fact that you are only paying for the resources that are actually in use, as opposed to a virtual machine that may be running all the time, even when its capacity is not needed. The serverless computing model means that you have zero control over the host machine, so it may not be appropriate from a security perspective.

6.1.4

Types of Infrastructure

In the early days of computers, infrastructure was pretty straightforward. The software you so carefully wrote ran on a single computer. Eventually, you had a network that could link multiple computers together. From there, things just got more and more complicated. Let’s look at the various options for designing your infrastructure, such as different types of clouds, and what each does and does not do well.

6.1.5

On-Premises

On-premises

Technically speaking, “on-premises” means any system that is literally within the confines of your building. In this case we are talking about traditional data centers that house individual machines which are provisioned for applications, rather than clouds, external or otherwise.

These traditional infrastructures are data centers with servers dedicated to individual applications, or to VMs, which essentially enable a single computer to act like multiple computers.

Operating a traditional on-premises data center requires servers, storage devices, and network equipment to be ordered, received, assembled in racks ("racked and stacked"), moved to a location, cabled for power and data. This equipment must be provided with environmental services such as power protection, cooling, and fire prevention. Servers then need to be logically configured for their roles, operating systems and software must be installed, and all of it needs to be maintained and monitored.

All of this infrastructure work takes time and effort. Requests for resources need to go through the operations team, which can lead to delays of days, weeks, or even months while new hardware is obtained, prepared, and provisioned.

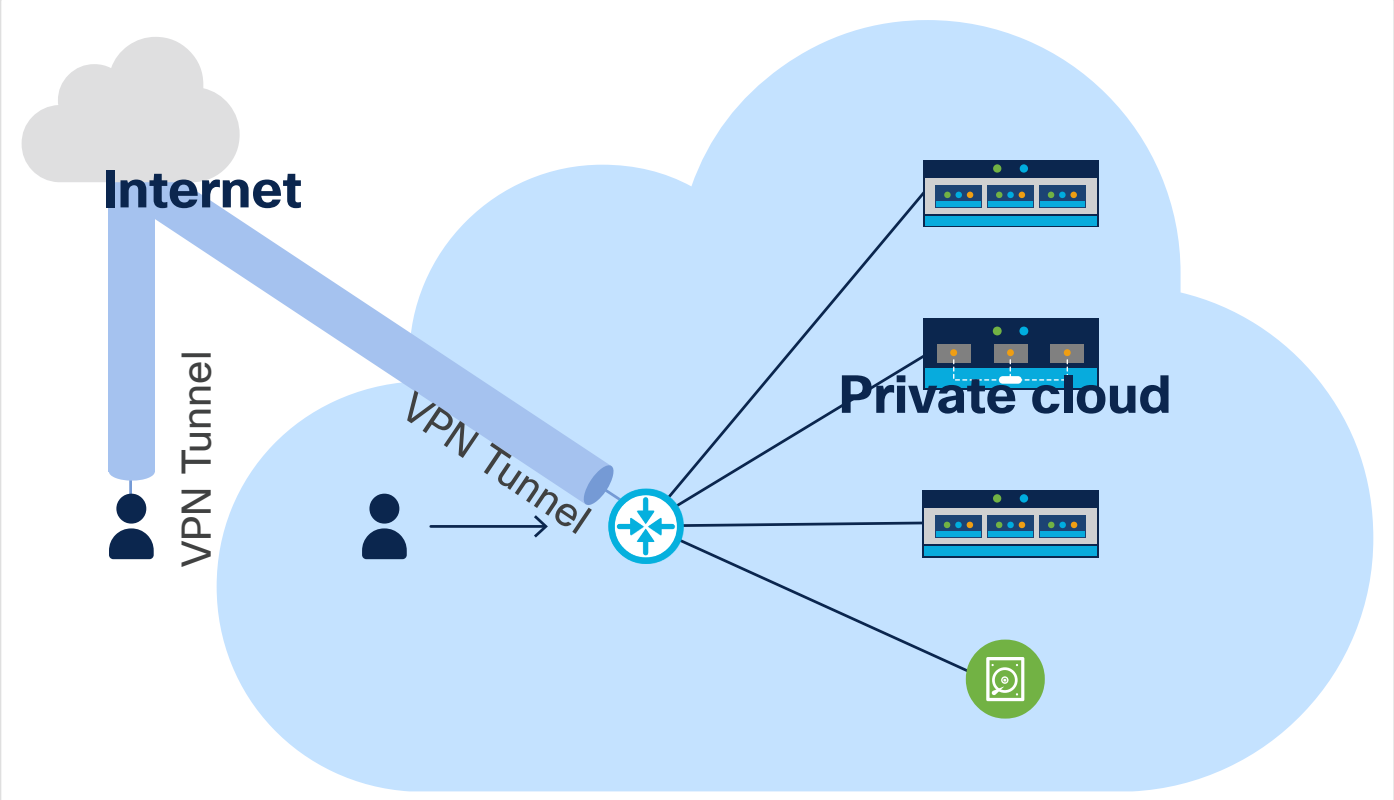
In addition, scaling an application typically means moving it to a larger server, which makes scaling up or down a major event. That means an application is almost always either wasting money with excess capacity that is not being used, or underperforming because it does not have enough resources.

These problems can be solved by moving to a cloud-based solution.

6.1.6

Private Cloud

The downside of on-premises infrastructure can be easily solved by cloud computing. A cloud is a system that provides self-service provisioning for compute resources, networking, and storage.



In a private cloud infrastructure, the organization controls all of the resources.

External user Internal user Internal network

A cloud consists of a control plane, which enables you to perform requests. You can create a new VM, attach a storage volume, even create a new network and compute resources.

Clouds provide self-service access to computing resources, such as VMs, containers, and even bare metal. This means that users can log into a dashboard or use the command line to spin up new resources themselves, rather than waiting for IT to resolve a ticket. These platforms are sometimes referred to as Infrastructure-as-a-Service (IaaS). Common private cloud platforms include VMware (proprietary), OpenStack (open source), and Kubernetes (a container orchestration framework). Underlying hardware infrastructure for clouds may be provided by conventional networked bare-metal servers, or by more advanced, managed bare-metal or "hyperconverged" physical infrastructure solutions, such as Cisco UCS and Cisco HyperFlex, respectively.

What distinguishes a private cloud from other types of clouds is that all resources within the cloud are under the control of your organization. In most cases, a private cloud will be located in your data center, but that is not technically a requirement to be called “private.” The important part is that all resources that run on the hardware belong to the owner organization.

The advantage of a private cloud is that you have complete control over where it is located, which is important in situations where there are specific compliance regulations, and that you do not typically have to worry about other workloads on the system.

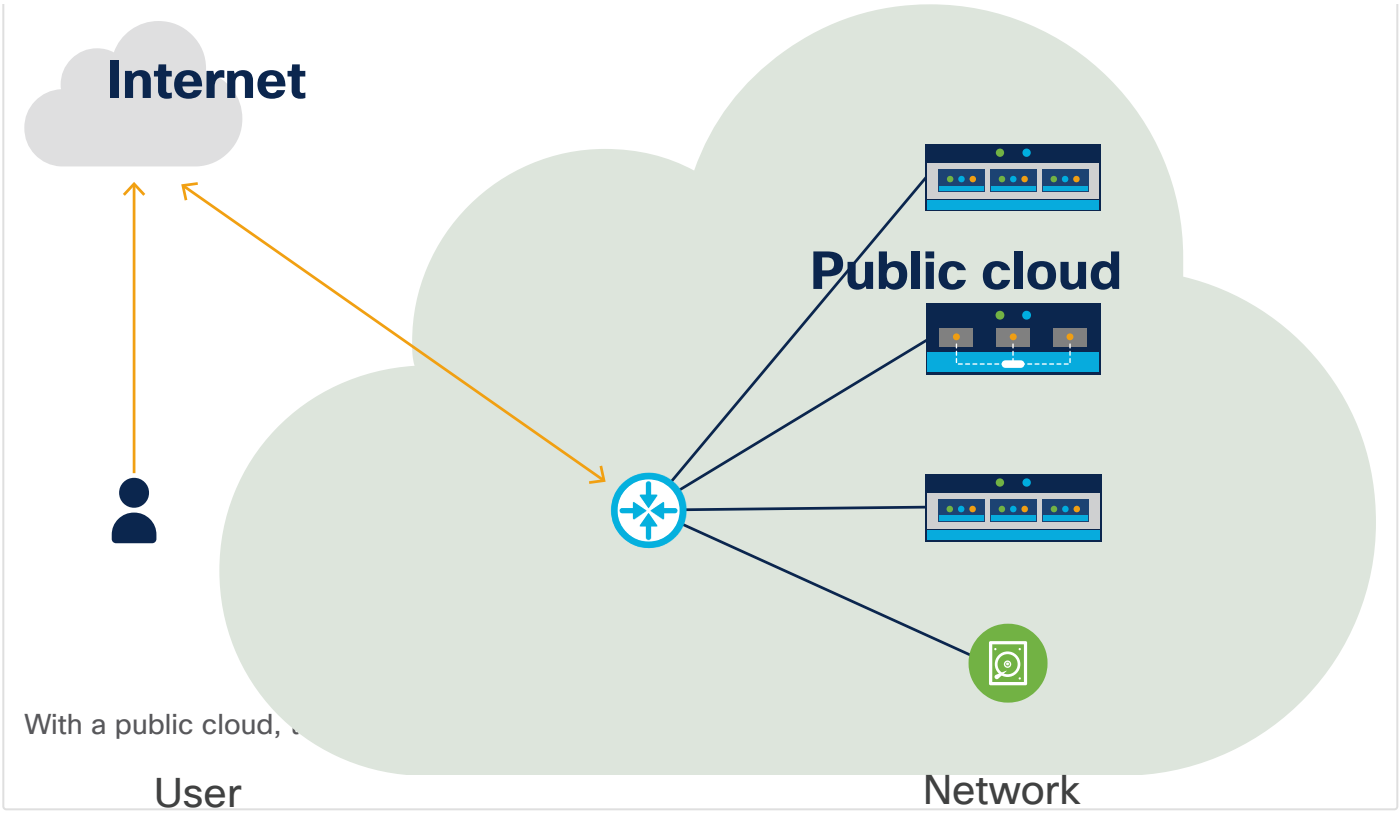
On the downside, you do have to have an operations team that can manage the cloud and keep it running.

6.1.7

Public Cloud

A public cloud is essentially the same as a private cloud, but it is managed by a public cloud provider. Public clouds can also run systems such as OpenStack or Kubernetes, or they can be specific proprietary clouds such as Amazon Web Services or Azure.

Public cloud customers may share resources with other organizations: your VM may run on the same host as a VM belonging to someone else. Alternatively, public cloud providers may provide customers with dedicated infrastructure. Most provide several geographically-separate cloud 'regions' in which workloads can be hosted. This lets workloads be placed close to users (minimizing latency), supporting geographic redundancy (the East Coast and West Coast regions are unlikely to be offline at the same time), and enabling jurisdictional control over where data is stored.



Public clouds can be useful because you do not have to pay for hardware you are not going to use, so you can scale up virtually indefinitely as long as the load requires it, then scale down when traffic is slow. Because you only pay for the resources you are actually using, this solution can be most economical because your application never runs out of resources, and you do not pay for resources you are not using. You also do not have to worry about maintaining or operating the hardware; the public cloud provider handles that. However, in practice, when your cloud gets to be a certain size, the cost advantages tend to disappear, and you are better off with a private cloud.

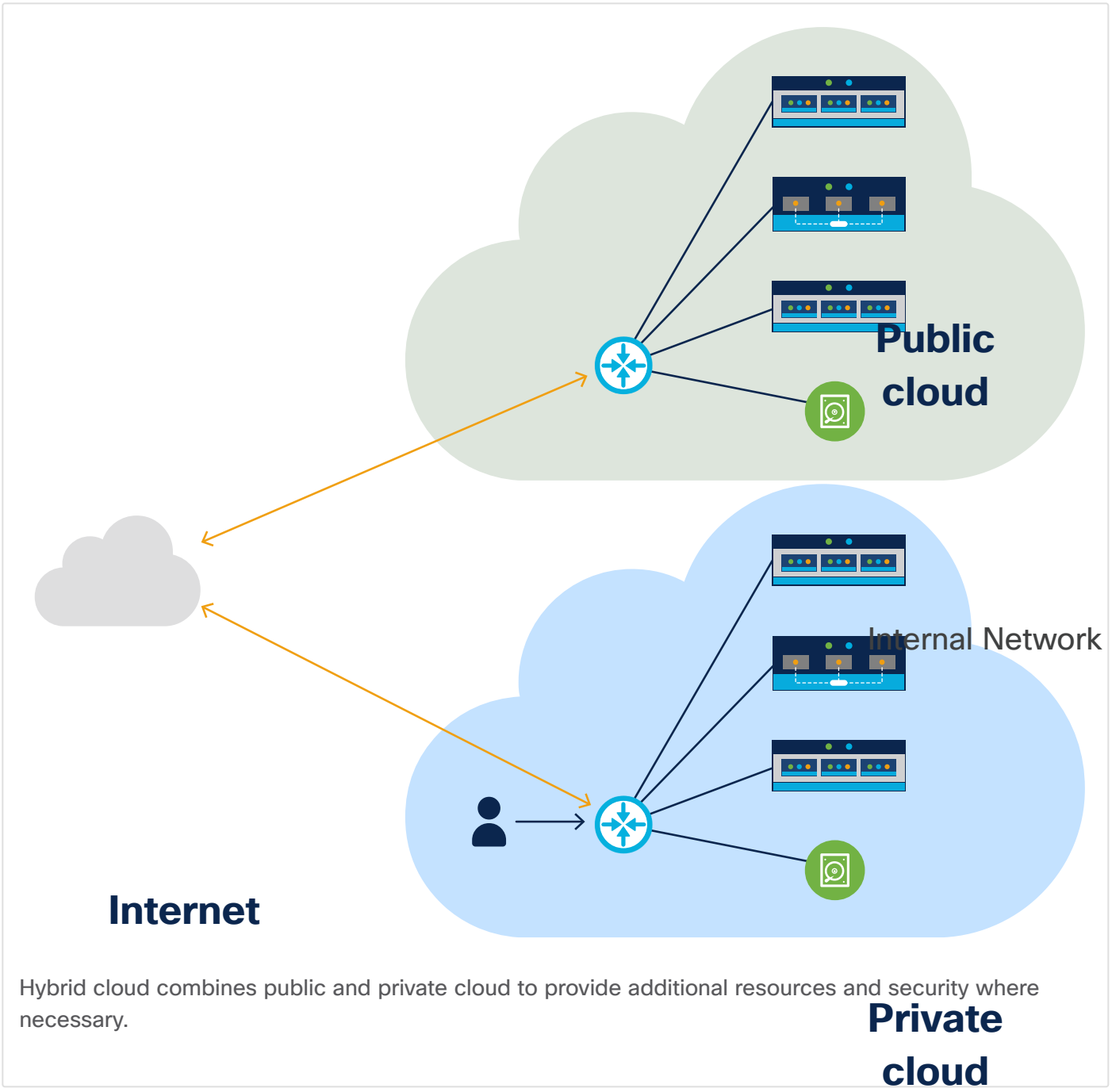
There is one disadvantage of public cloud. Because you are sharing the cloud with other users, you may have to contend with situations in which other workloads take up more than their share of resources.

This problem is worse when the cloud provider is overcommitting. The provider assumes not all resources will be in use at the same time, and allocates more "virtual" resources than "physical" resources. For example, it is not unusual to see an overcommit ratio of 16:1 for CPUs, which means that for every physical CPU, there may be 16 virtual CPUs allocated to VMs. Memory can be overcommitted as well. With a ratio of 2:1 for memory, a server with 128GB of RAM might be hosting 256GB of workloads. With a public cloud you have no control over that (save for paying more for dedicated instances or other services that help guarantee service levels).

6.1.8

Hybrid Cloud

As you might guess, hybrid cloud is the combination of two different types of clouds. Typically, hybrid cloud is used to bridge a private cloud and a public cloud within a single application.



Hybrid cloud combines public and private cloud to provide additional resources and security where necessary.

For example, you might have an application that runs on your private cloud, but “bursts” to public cloud if it runs out of resources. In this way, you can save money by not overbuying for your private cloud, but still have the resources when you need them.

You might also go in the other direction, and have an application that primarily runs on the public cloud, but uses resources in the private cloud for security or control. For example, you might have a web application that serves most of its content from the public cloud, but stores user information in a database within the private cloud.

Hybrid cloud is often confused with multi-cloud, in which an organization uses multiple clouds for different purposes. What distinguishes hybrid cloud is the use of more than one cloud within a single application. As such, a hybrid cloud application has to be much more aware of its environment than an application that lives in a single cloud.

A non-hybrid cloud application and its cloud are like a fish and the ocean; the fish does not need to be aware of the ocean because the ocean is just there, all around the fish. When you start adding hybrid cloud capabilities to an application, that application has to be aware of what resources are available and from where.

It is best if the application itself does not have to handle these things directly. It is a better practice to have some sort of interface that the application can call when it needs more resources, and that interface makes the decision regarding where to run those resources and passes them back to the application. This way the resource mapping logic can be controlled independently of the application itself, and you can adjust it for different situations. For example, you may keep all resources internal during the testing and debugging phase, then slowly ramp up public cloud use.

One way to accomplish this is through a tool such as Cisco Hybrid Cloud Platform for Google Cloud, which manages networking, security, management, data center, open-source and API software and tools. This provides you with a single, consistent, secure environment for your application, enabling it to work across both on-premises data centers and the Google Cloud.

In addition, container orchestrators have become very popular with companies employing hybrid-cloud deployments. The orchestrators provide a cloud-vendor agnostic layer which the application can consume to request necessary resources, reducing the environmental awareness needed in the application itself.

6.1.9

Edge Cloud

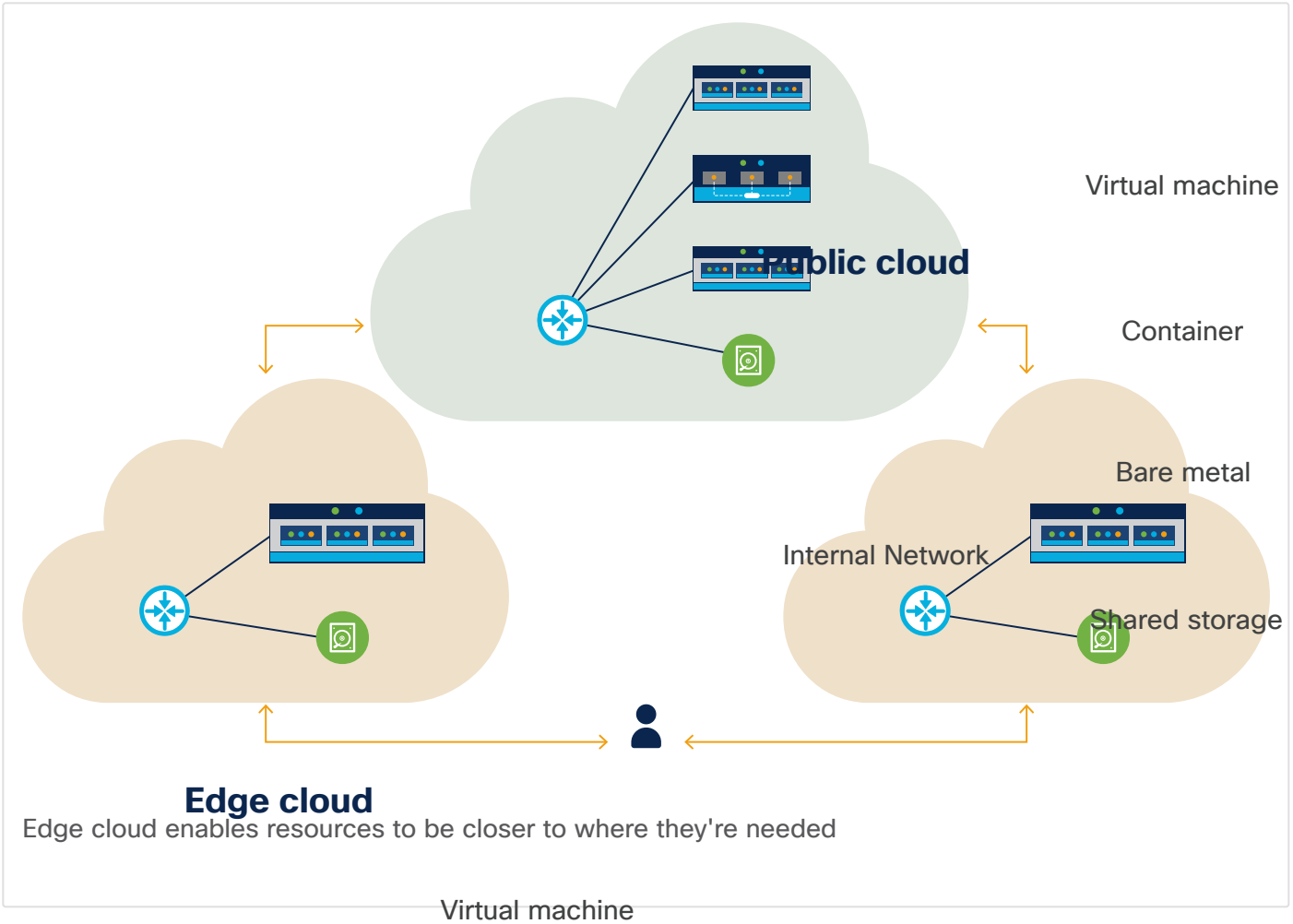
The newest type of cloud is edge cloud. Edge cloud is gaining popularity because of the growth of the Internet of Things (IoT). These connected devices, such as connected cameras, autonomous vehicles, and even smartphones, increasingly benefit from computing power that exists closer to them on the network.

The two primary reasons that closer computing power helps IoT devices are speed and bandwidth. For example, if you are playing a first person shooter game, even half a second of latency between when you pull the trigger and when the shot registers is unacceptable. Another instance where latency may be fatal, literally is with self-driving vehicles. At 55 miles per hour, a car travels more than 40 feet in just 500ms. If a pedestrian steps off the curb, the car cannot wait for instructions on what to do.

There is a second issue. Typically the self-driving car prevents the latency problem by making its own decisions, but that leads to its own problems. These vehicles use machine learning, which requires enormous amounts of data to be passed to and from the vehicle. It is estimated that these vehicles

generate more than 4 TB of data every hour, and most networks cannot handle that kind of traffic (especially with the anticipated growth of these vehicles in the market).

To solve both of these problems, an edge cloud moves computing closer to where it is needed. Instead of transactions making their way from an end user in Cleveland, to the main cloud in Oregon, there may be an intermediary cloud, an edge cloud, in Cleveland. The edge cloud processes the data or transaction. It then either sends a response back to the client, or does preliminary analysis of the data and sends the results on to a regional cloud that may be farther away.



Edge cloud computing comprises one or more central clouds that act as a hub for the edge clouds themselves. Hardware for the edge clouds is located as close as possible to the user. For example, you might have edge hardware on the actual cell tower handling the signals to and from a user's mobile phone.

Another area where you may see edge computing is in retail, where you have multiple stores. Each store might have its own internal cloud. This is an edge cloud which feeds into the regional cloud, which in turn might feed into a central cloud. This architecture gives local offices the benefits of having their own cloud (such as consistent deployment of APIs to ensure each store can be managed, updated, and monitored efficiently).

There is nothing "special" about edge clouds. They are just typical clouds. What makes them "edge" is where they are, and that they are connected to each other. There is one more thing about edge clouds, however. Because they often run on much smaller hardware than "typical" clouds, they may be more resource-constrained. In addition, edge cloud hardware must be reliable, efficient in terms of power usage,

and preferably remotely manageable, because it may be located in a remote area, such as a cell tower in the middle of the desert, where servicing the hardware is difficult.

6.0
Introduction to Application Deployment and ...

6.2
Creating and Deploying a Sample Application