Understanding Kubernetes: What, Why, and When to Adopt

Platform9 Kubernetes Journey eBook series

- Understanding Kubernetes ← This eBook
- Architecting Kubernetes Deployments
- Operating Kubernetes
- Scaling Kubernetes
- Securing Kubernetes

Download at: https://www.platform9.com/k8s-journey

Table of Contents

Introduction

If you asked the typical developer or IT engineer to make a list of the most transformative technologies to emerge over the past decade, Kubernetes would almost certainly feature prominently on it. By making it practical to deploy containerized applications at scale, Kubernetes not only allows organizations to take fuller advantage of containers and the cloud, but also to build more flexible and scalable software delivery processes.

None of this is news to anyone who has followed the conversation surrounding Kubernetes over the past several years. Yet, knowing that Kubernetes is an important technology is a far cry from understanding what, exactly, it can do; why so many teams have adopted it; and how your organization can go about leveraging it. Indeed, Kubernetes is a complex platform, and to say simply that it "orchestrates containers" or "manages containers at scale" – two generic phrases that are commonly used to describe Kubernetes's purpose – only hints at its full scope and value within the modern software delivery landscape.

Platform9 has prepared this eBook to provide a fuller understanding of Kubernetes – or K8s, as it is sometimes called by those familiar with it. It is designed to provide a fuller and more detailed understanding of Kubernetes for readers who have a basic familiarity with software delivery, and may even know in basic terms what Kubernetes does, but have not yet themselves deployed Kubernetes, or are still evaluating whether it is the right tool to address the pain points they face in software development and deployment.

In the following chapters, we explain where Kubernetes fits within the modern IT landscape, the various components that comprise the Kubernetes platform and the main reasons to use Kubernetes. We also outline the different approaches to deploying Kubernetes, although we will save specific details on Kubernetes deployment and configuration best practices for future eBooks.

The Changing Development Landscape

Fully understanding the value of Kubernetes requires an appreciation of how software development and delivery strategies have changed over the past decade, and the ways in which Kubernetes empowers software delivery teams to embrace the practices and architectures that have emerged out of those changes.

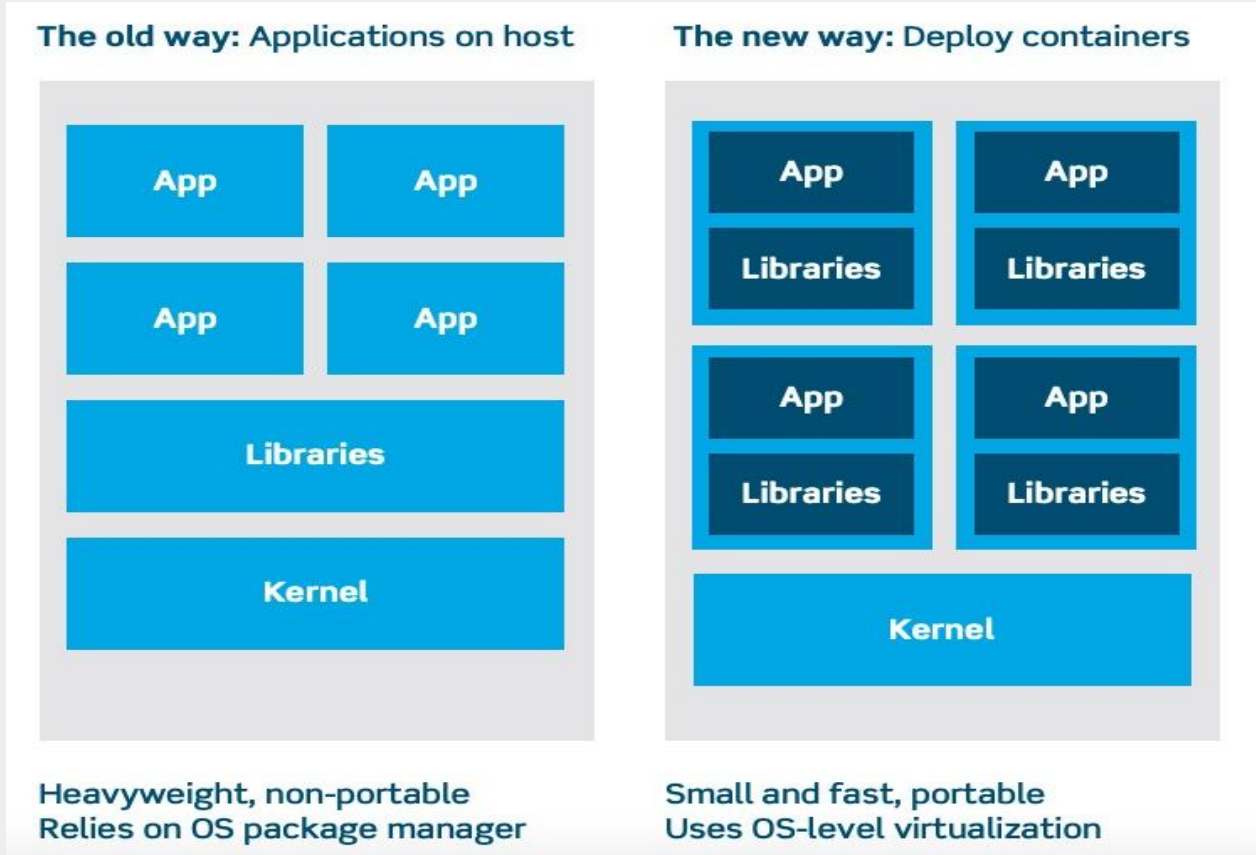
Software delivery has evolved

The way we build and run applications has changed dramatically over the years. Traditionally, apps ran on top of physical machines. Those machines eventually became virtual. In both cases, the application and all its dependencies were installed on top of an OS.

This relationship between OS and applications created a tightly-coupled bundle of everything needed to run that application. Each virtual machine (VM) ran a complete OS, no matter how big or small the VM was, or how demanding the application on top.

Each OS provided a complete execution environment for applications: this included binaries, libraries and services, as well as compute, storage, and networking resources.

Drawbacks of this approach are the inherent size and volume of VMs. Each OS is many gigabytes in size, which not only requires storage space, but also increases the memory footprint.



Virtualization vs. containerization

This size and tight coupling results in a number of complexities in the VM lifecycle and the applications running on top. Without a good way of separating different layers in a VM (OS, libraries, services, application binaries, configuration, and data), swapping out different parts in this layer cake is nearly impossible. For this reason, once a VM is built, configured and running, it usually lives on for months or years. This leads to pollution and irreversible entangling of the VM in terms of OS, data, and configuration.

New versions of the OS, its components, and other software inside the VM are layered on top of the older version. Because of this, each inplace upgrade creates potential version conflicts, stability problems, and ballooning of uncleaned recent versions on disk. Maintaining this ever-increasing complexity is a major operational pain point, and often leads to downtime.

This places an unbalanced operational focus on the OS and underlying layers, instead of the place it should be: the application.

Operational friction, an unnecessarily large and perennial operating environment, and lack of decoupling between layers are all in sharp contrast with how lean and agile software development works. It's no surprise, then, that the traditional approach doesn't work for modern software development.

In the new paradigm, developers actively break down work into smaller chunks, create (single-piece) flow, and take control and ownership over the pipeline that brings code from local testing all the way to production. Containers, microservices and cloud-native application design are facilitating this.

The Benefits of Creating Cloud Native Applications

Let's break down how these technologies enable modern software development methodologies.

Containers

First and foremost, containers package up only the parts of the application unique to that container, like the business logic. Containers share the underlying OS and often common libraries, frameworks, or other pieces of middleware. This results in much lighter packages (containers are usually megabytes, instead of the gigabytes that are typical with VMs), and are clearly decoupled from the layer cake underneath.

Because of this decoupling, a new one-to-one relationship between the container image and the application unlocks the full benefits of containers.

A container can be spun up on different hosts, clusters or clouds without any change to the container or its definition. Decoupling from the OS underneath makes it simpler to maintain those underlying layers. The OS becomes a commodity to developers: a black box layer that just works. Developers no longer have to think about that layer.

This allows easier and automated updating and changing of the layers underneath. Because the layers are decoupled, production systems are rarely patched or updated. The new version of the OS is deployed fresh, and the old system with the old version is discarded.

The same goes for a new version of the application inside the container: instead of updating the container, a new container with the new version is deployed, and traffic is diverted to that new container. The old one is killed as soon as the new container is operating correctly.

This approach is called 'immutable infrastructure,' defined as a clearer separation between the application, operating system, and the underlying infrastructure. This allows easier and more independent changes in each layer. Operationally, this makes a world of difference as different teams can take more ownership and responsibility of each layer.

With this decoupling comes a new interface between the OS and container, giving developers freedom to deploy new versions of their applications without intervention from the teams managing the layers underneath. This gives developers more control over when to deploy what to production. Rolling back a bad release or redirecting more traffic to a new version is a simple task, without friction or dependency on the infrastructure or operations teams.

In turn, the infrastructure and operations teams can take more control over their parts of the layer cake, enabling paradigms like Infrastructure-as-Code that allow treating infrastructure as a software development problem. This enables solutions like creating declarative code that instructs a pipeline of infrastructure automation software how to create and configure infrastructure.

Cloud-Native Services

While containers are a great fit for custom business logic and code, many of the moving parts of an application stack are standard and common components. Instead of re-inventing the wheel, using commercially available and/or open source software for those components makes sense. Other than a few niche and extreme use cases, why build your own database engine, caching layer or web server?

That's why many public cloud providers offer those components and middleware as a service; the goal is to make consumption as frictionless as possible. Developers can simply configure the entire software stack with a few clicks, using databases, proxies, web servers, message queues and much more.

But cloud-native means more than simply consuming existing technology as a service. The Cloud Native Computing Foundation, or CNCF for short, defines "cloud native" as follows:

> *Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.*

> *These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.*

This definition puts the focus on more than just a set of technological tools. It encompasses business outcomes like scalability, dynamic behavior, and resiliency; standards regarding certain patterns of methodology and design like immutability and frequent changes; and a focus on operational excellence with abilities like decoupling, observability, and automation.

It's this comprehensive approach that makes cloud-native so appealing: it's not just about technology, but about how tech is used within organizations, and what outcomes are achieved.

This creates an integrated ecosystem of products that checks all the boxes of CNCF's definition, and which organizations can use to hit the ground running. As such, it eliminates much of the groundwork processes like design, integration, and implementation that otherwise takes a lot of time.

CNCF's biggest and highest-velocity projects are integrated and broad, including:

Kubernetes is a container orchestration platform that helps users build, scale and manage modern applications and their dynamic lifecycles. The cluster scheduler capability lets developers focus on code rather than ops. Kubernetes future-proofs infrastructure management on-premises or in the cloud, without vendor or cloud provider lock-in.

Prometheus delivers real-time monitoring, alerting, and time series database capabilities (including powerful queries and visualizations) for cloud-native applications.It's the de facto standard for monitoring container-based infrastructure. Prometheus provides needed visibility into, and troubleshooting for, cloud-native architectures.

\Envoy is a distributed proxy designed for single services and applications, as well as a universal data plane designed for large microservice service mesh architectures. Envoy runs alongside every application, and abstracts the network by providing common features in a platform-agnostic manner. It's easy to visualize problem areas via consistent observability, tune overall performance, and add substrate features in a single place.

CoreDNS is a DNS server, written in Go. It can be used in a multitude of environments because of its flexibility.

Besides these four, there are many additional projects that are relevant to Kubernetes in 2019. The most notable include:

- Fluentd. This is a unified logging tool that helps users better understand what's happening in their environments by providing a unified layer for collecting, filtering, and routing log data.
- NATS. This is a simple, high-performance open source message queueing and publish/subscribe system for cloud-native applications.
- gRPC. This is a high-performance, open source universal RPC framework.
- Containerd. This is an industry-standard container runtime with an emphasis on simplicity,

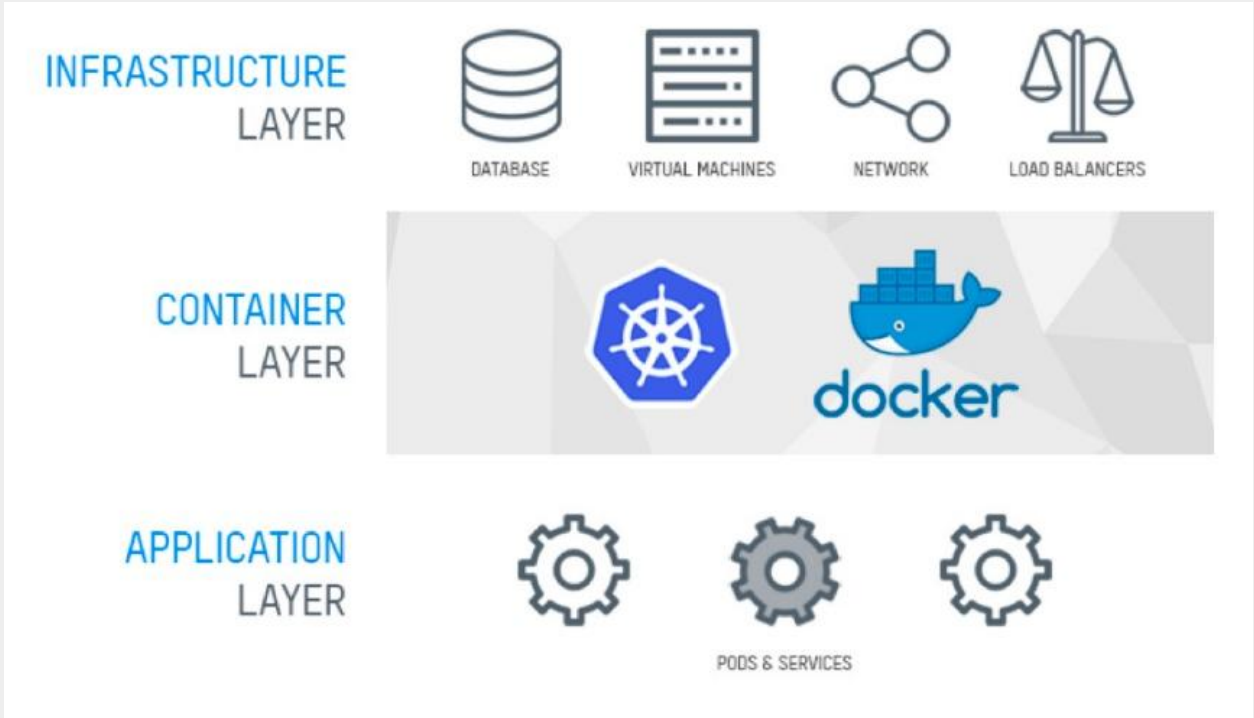
robustness and portability.

- Linkerd. An ultralight service mesh for Kubernetes and beyond, Linkerd provides observability, reliability, and security for microservices, with no code change required.
- CNI. The Container Network Interface provides networking for Linux containers.
- CSI. This stands for Container Storage Interface. It provides storage for Linux containers. See more on Kubernetes Storage and CSI.
- Helm. This is the package manager for Kubernetes. Helm is the best way to find, share, and use software built for Kubernetes.

Of course, there are numerous software projects not part of the CNCF that fit into the ecosystem very well. Examples include Istio, the popular service mesh, and Terraform, the composable infrastructure automation tool.

Why Kubernetes

Let's look at the CNCF's most popular project, Kubernetes.



The Kubernetes layer cake of infrastructure, containers, and applications.

Kubernetes is the orchestration layer that manages containers across a group of physical servers or VMs. Kubernetes is specifically designed to manage the ephemeral nature of thousands of containers spinning up, scaling up, and winding down.

Kubernetes manages versioning of containers, figures out how containers can talk to each other over the network, exposes services running inside containers, and handles storage considerations. It also deals with failed hardware, and maintaining container availability.

Kubernetes makes it easy to quickly ramp up container instances to match spikes in demand. New versions can be put into production in small increments (these are known as canary deployments.)

Kubernetes can be thought of as a container-centric computing platform. It has much of the flexibility of Infrastructure-as-a-Service (in terms of managing compute, storage and networking resources), with the developer-friendly workflows and constructs found in Platforas-a-Service on top. These include deployment, scaling, load balancing, logging, monitoring, and composition of application containers across clusters of container hosts.

Kubernetes is more than just a container orchestrator or resource scheduler. On the infrastructure side, it aims to remove the toil of orchestrating compute, network, and storage resources. It also abstracts those constructs so application developers and operators can focus entirely on container-centric workflows and self-service operation.

On the container side, Kubernetes provides a platform for building customized workflows and higher-level automation. It integrates into the continuous integration/continuous delivery (CI/CD) pipelines developers use to bring code into production in a controlled, tested and automated fashion.

The platform brings together infrastructure operations and software development by design. It uses declarative, infrastructure-agnostic constructs to describe applications and how they interact, without the traditional close ties into the underlying infrastructure.

Kubernetes runs just as well on traditional on-premises infrastructure stacks as it does for third-party service providers and public cloud environments.

Developer Agility

We've seen that containers unlock the full benefits of agile software development and operations. Creating smaller, portable container images that contain only the application increases developer velocity and the speed through the pipeline into production, which massively reduces the inertia of each release.

Creating "flow" is one of the core principles of agile software development, and reducing the size of the piece of code moving through the developer's delivery pipeline without being blocked is critical.

Containers are a major reduction in size compared to VMs, and help developers push code to production in smaller increments, and more often. This limits the impact of mistakes, as any changes causing the mistake will be small; this makes them quick and easy to roll back, due to image immutability. Developers can simply roll back to a previous version, without having to worry very much about data consistency or data loss.

> *A major cause of mistakes in production is the lack of environmental consistency across development and production environments. With containers, the image is identical and immutable, no matter where it runs; this is true even if the underlying resources differ massively. So, if it runs on the developer's laptop, it will run in production.*

A common blocker of the pipeline is the separation of concerns between development and operations. This typically leads to a dependency of the developer on the Ops team to install the new application version during deployment, often by using configuration management tooling like Chef or a package manager.

With containers, images are built automatically at build/release time and deployed as an atomic unit. This allows Ops to influence how the images are built asynchronous to the deployment, while developers have full control during deployment. In a container configuration, dependencies are added as lines of code and either specify a specific version of that dependency, or depend on the latest version at build time. This helps in managing security breaches and keeping code secure (and lean), as dependencies are updated automatically and often.

While Kubernetes and the common underlying container runtime themselves don't deploy source code or build your application, they're easily integrated into CI/CD workflows and pipelines.

Cost Management

Similar to the move from physical to virtual servers, moving to containers optimizes resource usage. This lowers the cost of each application, as it runs more efficiently. As discussed before, a major difference between a VM and a container is its relative size: a container is magnitudes smaller than a VM. This makes it nimbler and more flexible, especially from a cost perspective. This allows the container to run where it's cheaper, an important consideration in ephemeral compute instances where the application is non-production or resilient itself.

Secondly, more but smaller containers are more easily scheduled across multiple hosts as compared to fewer but bigger VMs. This is called the "bin-packing problem."

The dynamic nature of containers in a Kubernetes cluster, utilizing the Horizontal Pod Autoscaler, means that application cost goes hand-in-hand with application demand. While this is fantastic for scalability, it can sometimes have unintended consequences on the budget. The plethora of options muddies the waters pretty quickly. Even with the relatively simple cost model of physical servers, assigning a fraction of cost to a certain team, department, or application is difficult. Add in the complex offering of public cloud instance types, and it becomes near impossible to assign cost.

There are some solutions for cost control, like CloudHealth, CoreOS Operator Framework, and Platform9's Arbitrage that help assign cost across the multitude of layers in Kubernetes and the underlying public cloud or on-premises platform. These solutions figure out the charges for consumed infrastructure cost and assign them to clusters, namespaces, and pods inside Kubernetes. Besides the pods that run the actual applications, these solutions also split pods into administrative, monitoring, logging, and idle resources.

But in reality, many people apply the 'guesstimate' method, especially in the early phases of containerization projects. And however unscientific it is, this method does fit in with the reasoning behind the move toward containers and developer agility: create flow, increase velocity, and remove hurdles in their pipeline to production.

> *Only after implementation does cost control start to matter. The tangible benefits of the system have started to manifest in day-to-day operations; after that, the downsides, including cost sprawl, need to be reined in, but only after it's proven successful.*

And here lies the true cost/benefit analysis: it's not just about controlling infrastructure costs, but developer costs, too: how much quicker can they move to production or roll back a faulty release, for instance, and what financial consequences, good or bad, does that have?

This brings us to the fundamental value of agility: smaller iterations of work. This means going through the "discover-plan-build-review" cycle much, much more often. Optimizing the developer's flow makes them more efficient, which in turn makes them less expensive. As more and more companies invest in software development, the cost balance is shifting from infrastructure to developer; given this, it makes more sense to optimize the higher-cost items.

Accelerate Project Timelines

For many developers, Kubernetes means 'less friction.' A production-grade Kubernetes platform usually includes monitoring, logging, tracing, release management for blue/green or canary deployments, automated testing in the pipeline, and automated deployment.

All of these reduce friction, making it cheap and easy to deploy software to production. This means less management overhead and associated processes, including approvals, change advisory boards, and release/deployment managers.

This is especially true for development in microservices environments, where boundaries between teams are carried forward in the services and products they deliver. These microservices are loosely-coupled, small and independent pieces of a larger network of services that make up an application. All these services can be deployed and managed independently and dynamically, making it easier for a team to put a new piece of code into

production without dependence on another team.

This gives teams the freedom to decide if they want to bring in an existing (paid-for) solution, or if they'll build it themselves. While existing solutions may be more expensive up front, the delivery timeframe is usually compressed significantly.

A Note on Kubernetes for Stateful Applications:

Yes, Containers Can Be Stateful, too.

In the earlier stages of Kubernetes and container maturity, it was often believed that containers were only suitable for stateless workloads, and that storing any data or state in a container was impossible. This belief is wrong; both the underlying container runtime (which is often Docker) and Kubernetes fully support a diverse variety of workloads, including stateful applications. Containers themselves are ephemeral and immutable, meaning that

any file system changes are lost after the container shuts down. But there are plenty of options for adding stateful storage to a container, ranging from NFS network shares to S3 object stores and full-fledged data center storage options like a SAN. Many organizations deploying Kubernetes actually use existing storage assets for stateful storage. Another popular storage option is a hyperconverged storage deployment pattern like the open source CEPH or VMware's VSAN.

Learn more about the key concepts and components in our blog post on the Kubernetes storage architecture, and in our webinar on using Kubernetes for stateful applications.

Kubernetes and the Cloud-Native Transformation

The previous chapter discussed how software delivery practices and architectures have changed, and how those changes intersect with Kubernetes's functionality.

Now, let's explore Kubernetes's importance from a slightly different, but related, angle: The shift toward cloud-native computing. We touched on cloud-native architectures above, but in this chapter, we'll dive deeper into how Kubernetes helps organizations work past the common pitfalls that arise during migrations to a cloud-native strategy.

What is cloud-native?

First, though, a word about what *cloud-native* actually means.

The term *cloud-native* is a bit hard to define. Some people use it to mean any type of technology or strategy that they consider modern. From that perspective, cloud-native ends up as just another relatively meaningless buzzword.

On the other hand, when invested with specific and limited meaning, cloud-native is a useful term and concept. A good point of reference is the CNCF's definition, which emphasizes "loosely coupled systems" and resilience as characteristics of cloud-native computing. The CNCF definition also points to a specific and limited list of technologies and architectures — "containers, service meshes, microservices, immutable infrastructure, and declarative APIs" — as examples of cloud-native technologies.

For the purposes of this chapter, we'll be sticking to the CNCF's definition of cloud-native. I want to discuss the specific challenges that arise when you use technologies and strategies like those described above.

Now, let's discuss those challenges…

Persistent data storage

One common challenge with many cloud-native technologies is storing data persistently. Containers, serverless functions and applications deployed using an immutable infrastructure model do not typically have a way to store data permanently inside themselves because all internal data is destroyed when the application shuts down.

Solving this challenge requires rethinking approaches to data storage by decoupling it from applications and host environments. Instead of storing data within the application environment, cloud-native workflows store it externally, and expose the data as a service. Then, workloads that need to access the data, connect to it just as they would connect to any other service.

This approach — which is enabled by various tools, like volumes in Kubernetes — has two benefits. In addition to enabling persistent data storage for applications that are not themselves designed to be persistent, it also makes it easy to share a single storage pool among multiple applications or services.

Service integration

Cloud-native applications are typically composed of a set of disparate services. This distributed nature is what helps make them scalable and flexible, as compared to monoliths.

But it also means that cloud-native workloads have many more moving pieces that need to be connected together seamlessly in order to achieve success.

In part, service integration is an issue for developers to address as they build cloud-native apps. They must ensure that each service is properly sized; a best practice is to create a distinct service for each type of functionality within a workload, rather than trying to make a single service do multiple things. It's also important to avoid adding services just because you can. Before you introduce more complexity to your app in the form of another service, make sure that the service advances a particular goal.

Beyond the architecture of the application itself, effective service integration also depends on choosing the right deployment techniques. Containers are probably the most obvious way to deploy multiple services and unify them into a single workload, but in some cases, serverless functions, or non-containerized apps connected by APIs, might be better methods of service deployment.

Management and monitoring

The more services you have running as part of an application, the more difficult it becomes to monitor and manage them. This is true, due not just to the sheer number of services that you have to track, but also because guaranteeing application health requires monitoring relationships between services, not just services themselves.

Successfully monitoring and managing services in a cloud-native environment, then, requires an approach that can predict how a failure in one service will impact others, as well as understand which failures are most critical. Dynamic baselining, which means replacing static thresholds with ones that continually reassess applications environments in order to determine what is normal and what is an anomaly, is also critical.

Avoiding cloud lock-in

Lock-in risks are not unique to the cloud; they can arise from almost any type of technology, and they have been a threat to agility for decades. However, in the case of cloud-native applications or architectures, the threat of becoming too dependent on a particular cloud provider or service can be especially great, due to the ease with which workloads can be deployed in such a way that they require a particular service from a particular cloud.

Fortunately, mitigating this cloud lock-in risk is easy enough as long as you plan ahead. Sticking to community-based standards (like those promoted by the OCCI) will do much to ensure that you can move your workloads easily from one cloud to another. Likewise, as you plan which cloud services you will use to go cloud-native, consider whether any of the services you are considering have features that are truly unique and not available from other clouds. If they do, avoid those features, because they can lock you in.

For example, the specific languages and frameworks supported by the serverless computing platforms of various public clouds vary somewhat. AWS Lambda supports Go, for example, but Azure does not. For that reason, you'd be wise to avoid writing your serverless functions in Go. Even if you plan to use AWS to host them initially, this dependency would make it difficult to migrate to a different cloud in the future. Stick with a language like Java, which you can safely bet will be supported everywhere.

Building cloud-native delivery pipelines apps

By definition, cloud-native apps run in the cloud. But many application delivery pipelines still run largely on-premises: Code is often written on local machines, integrated by on-premises CI servers and tested in local environments before it is deployed to the cloud.

This creates a challenge in several respects. One is that deploying code from a local environment to an on-premises can introduce delays. Another is that developing and testing locally makes it harder to emulate production conditions, which can lead to unexpected application behavior, post-deployment.

The most effective way to overcome these hurdles is simply to move your CI/CD pipeline into the cloud. That way, code is written in the same place where it is deployed, making deployment faster. It also becomes easier to spin up test environments that are identical to production ones.

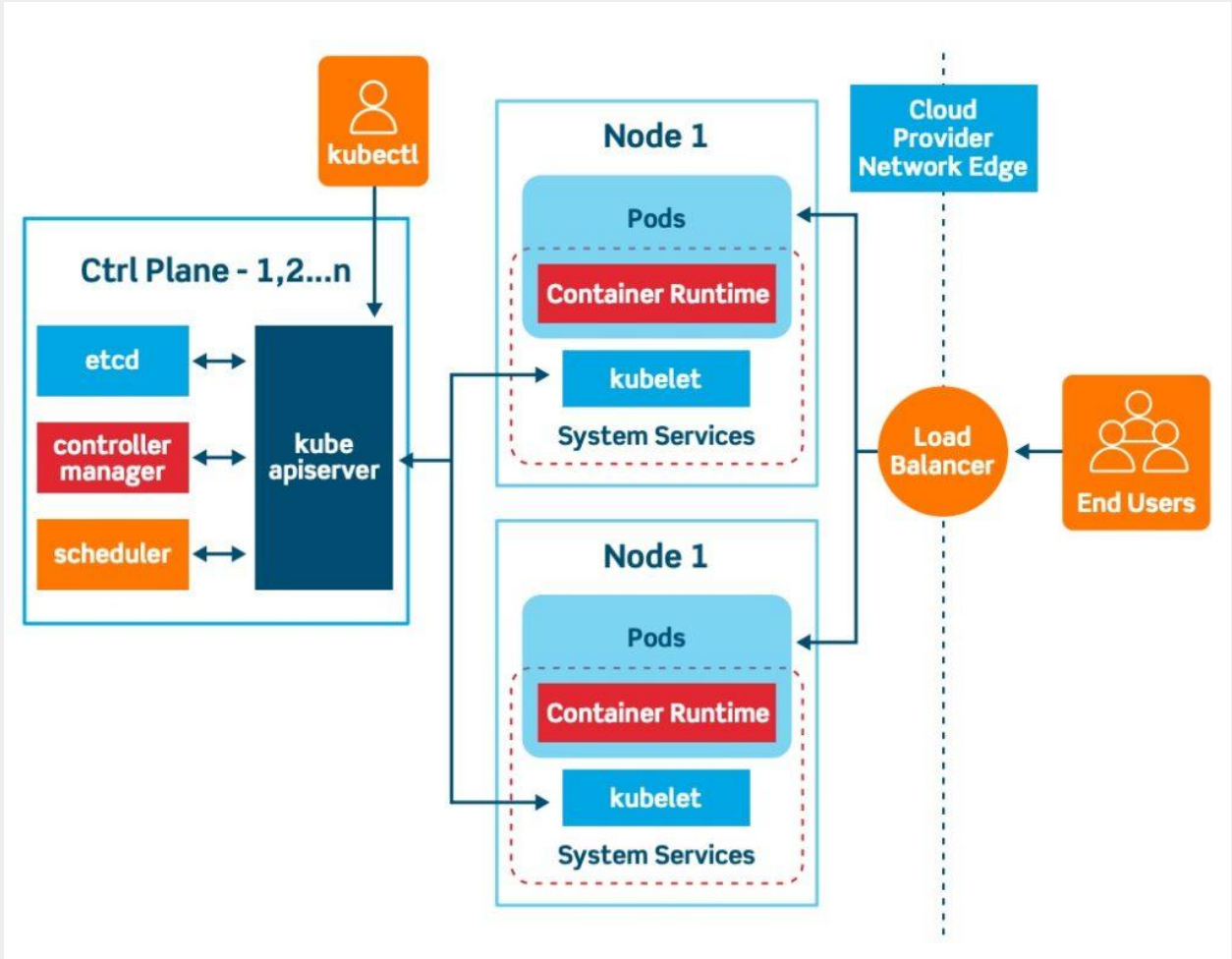
On the other hand, development that is completely cloud-based is not for everyone. Some developers prefer the familiarity and responsiveness of local IDEs over cloud-based ones. For that reason, you might consider keeping your IDEs, at least, on local machines. But there is typically not a good reason to keep the CI/CD pipeline itself on-premises if you are building a cloud-native app.

> *The bottom line: No matter how you spin it, going cloud-native is difficult. Compared to legacy applications, cloud-native applications are more complex and have many more places where things can go wrong. That said, cloud-native computing challenges can be overcome — and implementing strategies that can address the challenges is key to unlocking the agility, reliability and scalability that only cloud-native architectures can deliver.*

Kubernetes Architecture: The Core Concepts

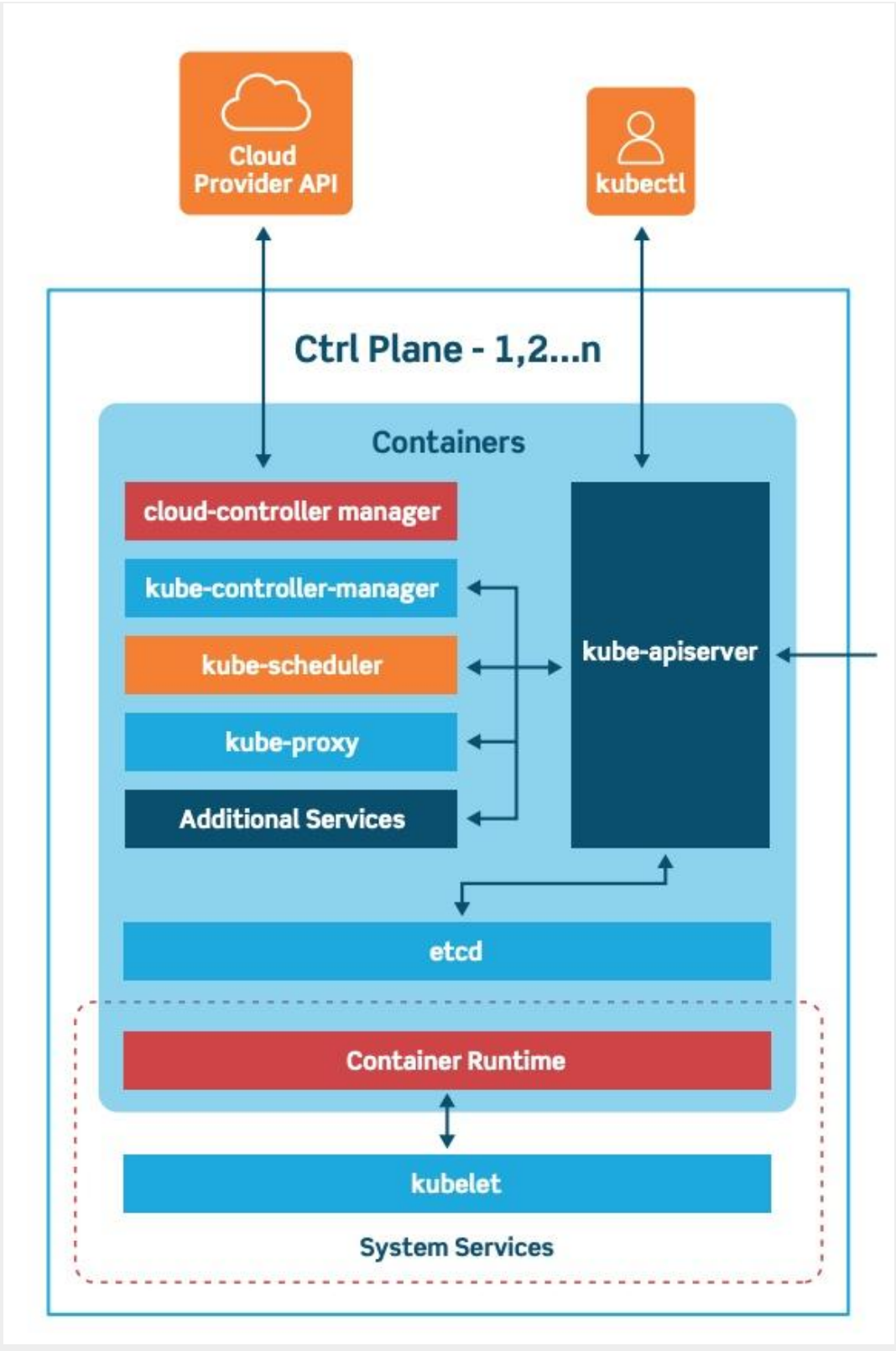
Now that you know what Kubernetes does to improve software delivery and support cloud-native development, let's look at what actually makes Kubernetes tick. As we explained, Kubernetes is not a single tool, but rather a broad platform comprised of a number of tools and services.

From a high level, a Kubernetes environment consists of a control plane (master), a distributed storage system for keeping the cluster state consistent (etcd), and a number of cluster nodes (Kubelets).



Architectural overview of Kubernetes

Kubernetes Control Plane



Kubernetes control plane taxonomy

The control plane is the system that maintains a record of all Kubernetes objects. It continuously manages object states, responding to changes in the cluster; it also works to make the actual state of system objects match the

desired state. As the above illustration shows, the control plane is made up of three major components: kube-apiserver, kube-controller-manager and kube-scheduler. These can all run on a single master node, or can be replicated across multiple master nodes for high availability.

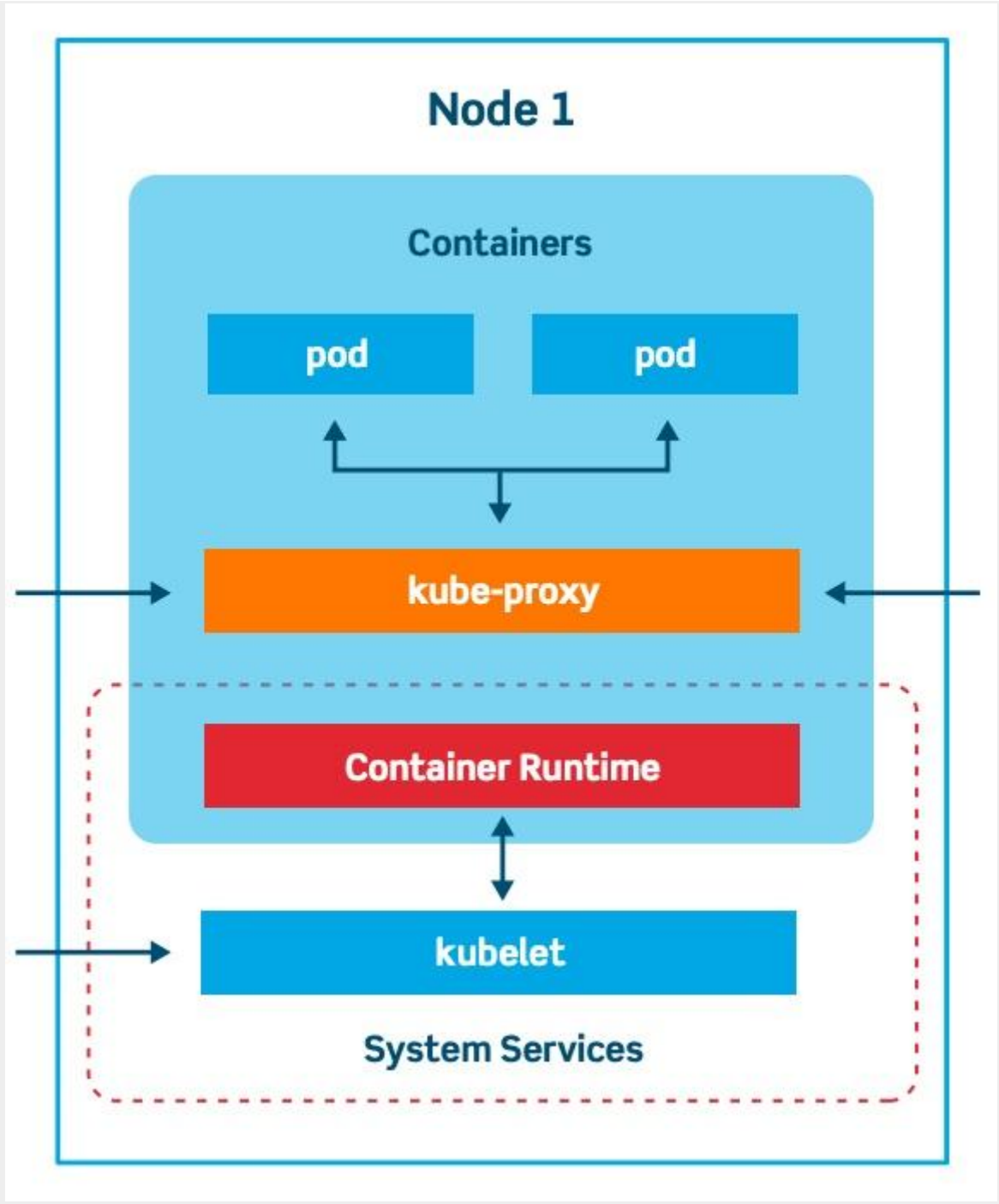
The API Server provides APIs to support lifecycle orchestration (scaling, updates, and so on) for different types of applications. It also acts as the gateway to the cluster, so the API server must be accessible by clients from outside the cluster. Clients authenticate via the API Server, and also use it as a proxy/tunnel to nodes and pods (and services).

Most resources contain metadata, such as labels and annotations, desired state (specification) and observed state (current status). Controllers work to drive the actual state toward the desired state.

There are various controllers to drive state for nodes, replication (autoscaling), endpoints (services and pods), service accounts and tokens (namespaces). The Controller Manager is a daemon that runs the core control loops, watches the state of the cluster, and makes changes to drive status toward the desired state. The Cloud Controller Manager integrates into each public cloud for optimal support of availability zones, VM instances, storage services, and network services for DNS, routing and load balancing.

The Scheduler is responsible for the scheduling of containers across the nodes in the cluster; it takes various constraints into account, such as resource limitations or guarantees, and affinity and anti-affinity specifications.

Cluster Nodes



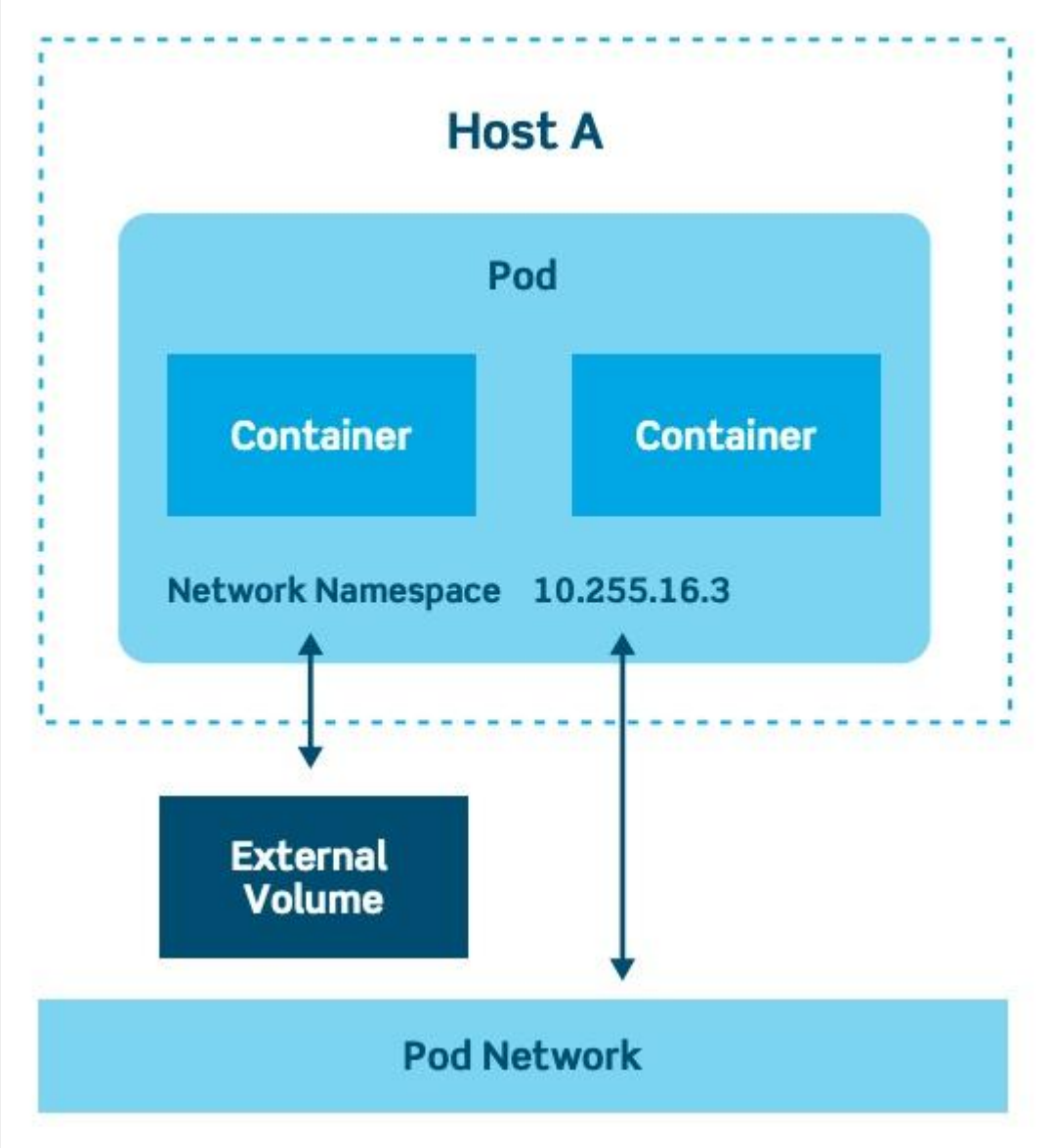
Kubernetes node taxonomy

Cluster nodes are machines that run containers and are managed by the master nodes. The Kubelet is the primary and most important controller in Kubernetes. It's responsible for driving the container execution layer, typically Docker.

Pods and Services

Pods are one of the crucial concepts in Kubernetes, as they are the key construct that developers interact with. The previous concepts are infrastructure-focused and internal architecture.

This logical construct packages up a single application, which can consist of multiple containers and storage volumes. Usually, a single container (sometimes with some helper program in an additional container) runs in this configuration – as shown in the diagram below.



Kubernetes Pod Architecture

Alternatively, pods can be used to host vertically-integrated application stacks, like a WordPress LAMP (Linux, Apache, MySQL, PHP) application. A pod represents a running process on a cluster.

Pods are ephemeral, with a limited lifespan. When scaling back down or upgrading to a new version, for instance, pods eventually die. Pods can do horizontal autoscaling (i.e., grow or shrink the number of instances), and perform rolling updates and canary deployments.

There are various types of pods:

- ReplicaSet, the default, is a relatively simple type. It ensures the specified number of pods are running
- Deployment is a declarative way of managing pods via ReplicaSets. Includes rollback and rolling

update mechanisms
- Daemonset is a way of ensuring each node will run an instance of a pod. Used for cluster services, like health monitoring and log forwarding
- StatefulSet is tailored to managing pods that must persist or maintain state
- Job and CronJob run short-lived jobs as a one-off or on a schedule.

This inherent transience creates the problem of how to keep track of which pods are available and running a specific app. This is where Services come in.

Kubernetes Services

Services are the Kubernetes way of configuring a proxy to forward traffic to a set of pods. Instead of static IP address-based assignments, Services use selectors (or labels) to define which pods uses which service. These dynamic assignments make releasing new versions or adding pods to a service really easy. Anytime a Pod with the same labels as a service is spun up, it's assigned to the service.

By default, services are only reachable inside the cluster using the clusterIP service type. Other service types do allow external access; the LoadBalancer type is the most common in cloud deployments. It will spin up a load balancer per service on the cloud environment, which can be expensive. With many services, it can also become very complex.

To solve that complexity and cost, Kubernetes supports Ingress, a high-level abstraction governing how external users access services running in a Kubernetes cluster using host- or URL-based HTTP routing rules.

There are many different Ingress controllers (Nginx, Ambassador), and there's support for cloud-native load balancers (from Google, Amazon, and Microsoft). Ingress controllers allow you to expose multiple services under the same IP address, using the same load balancers.

Ingress functionality goes beyond simple routing rules, too. Ingress enables configuration of resilience (time-outs, rate limiting), content-based routing, authentication and much more.

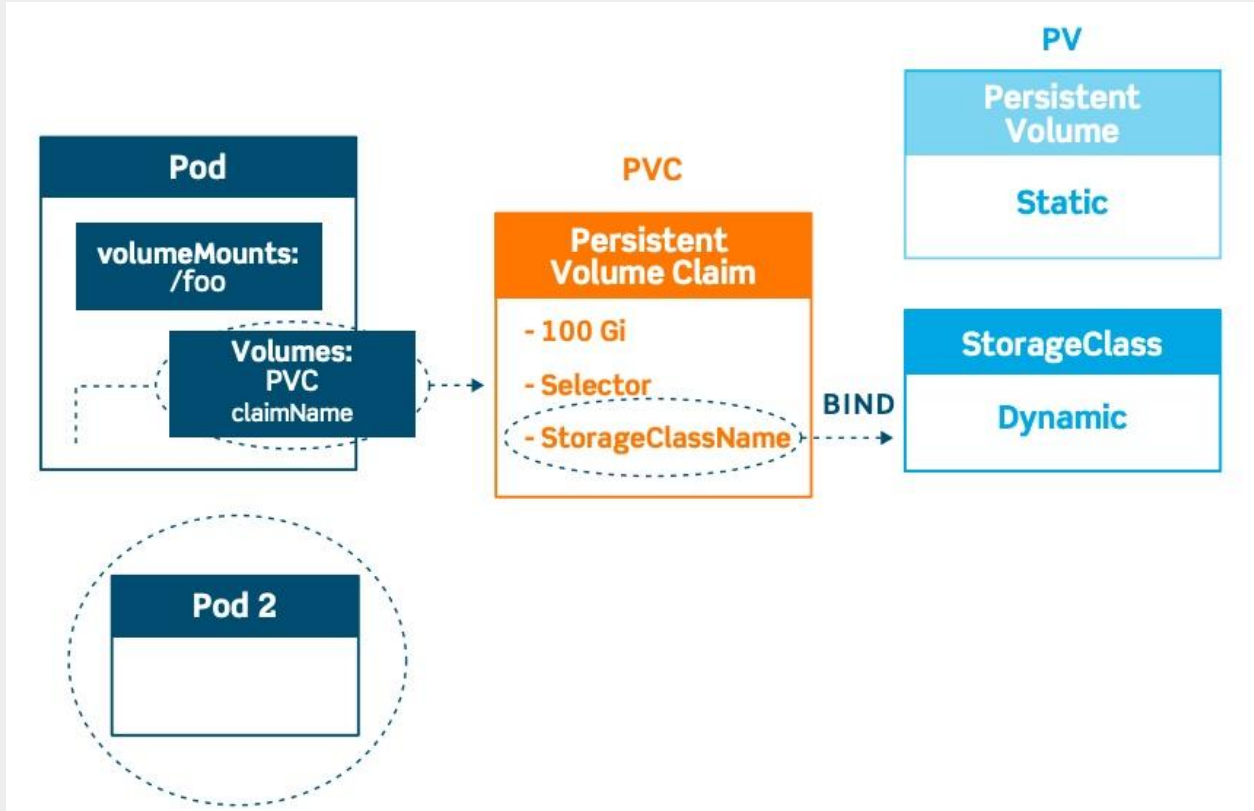
Kubernetes Networking

Networking Kubernetes has a distinctive networking model for cluster-wide, pod-to-pod networking. In most cases, the Container Network Interface (CNI) uses a simple overlay network (like Flannel) to obscure the underlying network from the pod by using traffic encapsulation (like VXLAN); it can also use a fully-routed solution like Calico. In both cases, pods communicate over a cluster-wide pod network, managed by a CNI provider like Flannel or Calico.

Within a pod, containers can communicate without any restrictions. Containers within a pod exist within the same network namespace and share an IP. This means containers can communicate over localhost. Pods can communicate with each other using the pod IP address, which is reachable across the cluster.

Moving from pods to services, or from external sources to services, requires going through kube-proxy.

Persistent Storage in Kubernetes



Kubernetes Persistent Volumes, Claims and Storage Classes

Kubernetes uses the concept of volumes. At its core, a volume is just a directory, possibly with some data in it, which is accessible to a pod. How that directory comes to be, the medium that backs it, and its contents are determined by the particular volume type used.

Kubernetes has a number of storage types, and these can be mixed and matched within a pod (see above illustration). Storage in a pod can be consumed by any containers in the pod. Storage survives pod restarts, but what happens after pod deletion is dependent on the specific storage type.

There are many options for mounting both file and block storage to a pod. The most common ones are public cloud storage services, like AWS EBS and gcePersistentDisk, or types that hook into a physical storage infrastructure, like CephFS, Fibre Channel, iSCSI, NFS, Flocker or glusterFS.

There are a few special kinds, like configMap and Secrets, used for injecting information stored within Kubernetes into the pod or emptyDir, commonly used as scratch space.

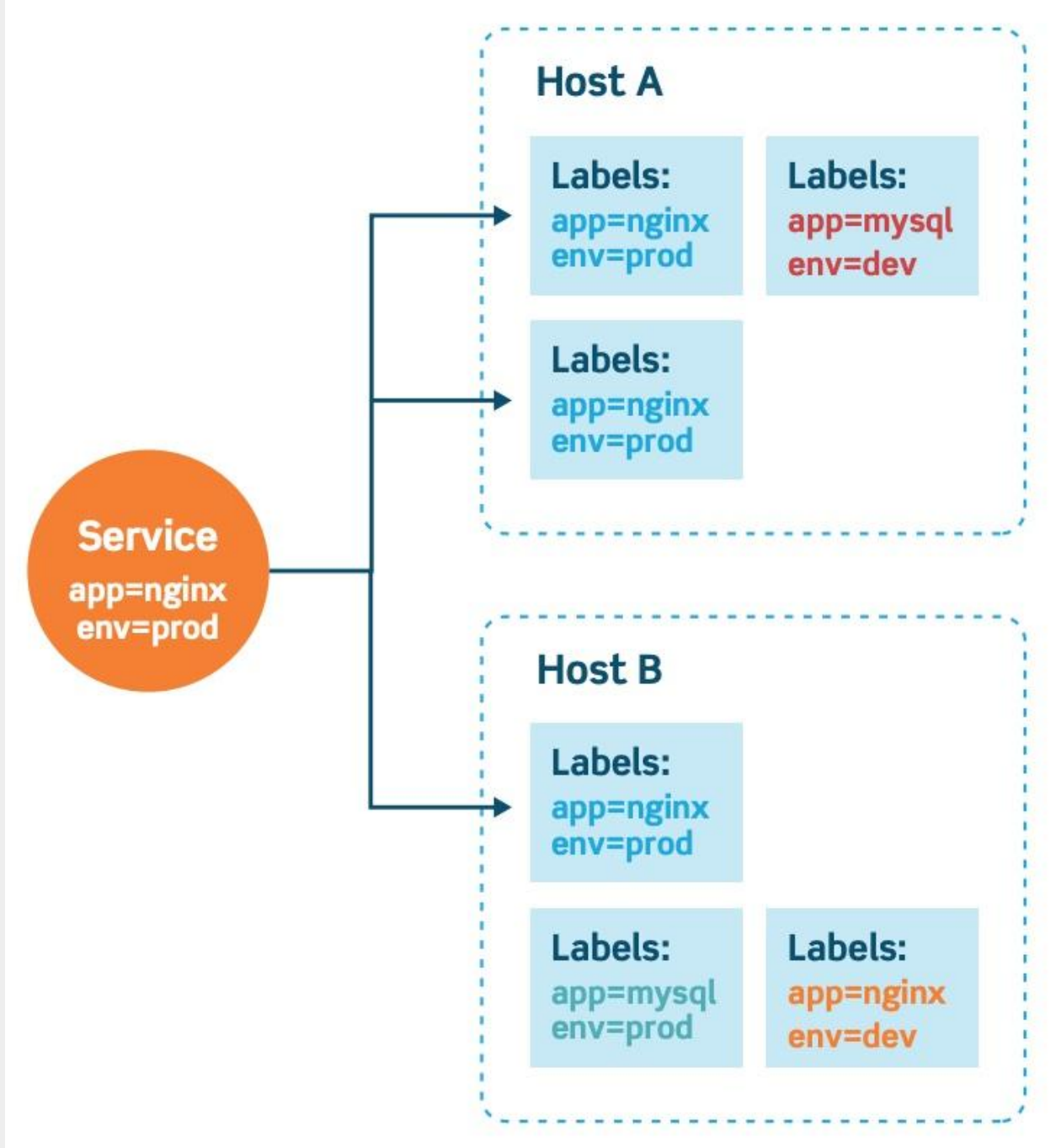
PersistentVolumes (PVs) tie into an existing storage resource, and are generally provisioned by an administrator. They're cluster-wide objects linked to the backing storage provider that make these resources available for consumption.

For each pod, a PersistentVolumeClaim makes a storage consumption request within a namespace. Depending on the current usage of the PV, it can have different phases or states: available, bound (unavailable to others), released (needs manual intervention) and failed (Kubernetes could not reclaim the PV).

Finally, StorageClasses are an abstraction layer to differentiate the quality of underlying storage. They can be used to separate out different characteristics, such as performance. StorageClasses are not unlike labels; operators use them to describe different types of storage, so that storage can be dynamically be provisioned based on incoming

claims from pods. They're used in conjunction with PersistentVolumeClaims, which is how pods dynamically request new storage. This type of dynamic storage allocation is commonly used where storage is a service, as in public cloud providers or storage systems like CEPH.

Discovering and Publishing Services in Kubernetes



The Kubernetes Service Taxonomy

Discovering services is a crucial part of a healthy Kubernetes environment, and Kubernetes heavily relies on its integrated DNS service (either Kube-DNS or CoreDNS, depending on the cluster version) to do this. Kube-DNS and CoreDNS create, update and delete DNS records for services and associated pods, as shown in the above illustration. This allows applications to target other services or pods in the cluster via a simple and consistent

naming scheme.

An example of a DNS record for a Kubernetes service:

`service.namespace.svc.cluster.local`
A pod would have a DNS record such as:

`10.32.0.125.namespace.pod.cluster.local`

There are four different service types, each with different behaviors:

- ClusterIP exposes the service on an internal IP only. This makes the service reachable only from within the cluster. This is the default type.
- NodePort exposes the service on each node's IP at a specific port. This gives the developers the freedom to set up their own load balancers, for example, or configure environments not fully supported by Kubernetes.
- LoadBalancer exposes the service externally using a cloud provider's load balancer. This is often used when the cloud provider's load balancer is supported by Kubernetes, as it automates their configuration.
- ExternalName will just map a CNAME record in DNS. No proxying of any kind is established. This is commonly used to create a service within Kubernetes to represent an external datastore like a database that runs externally to Kubernetes. One potential use case would be using AWS RDS as the production database, and a MySQL container for the testing environment.

Namespaces, Labels, and Annotations

Namespaces are virtual clusters within a physical cluster. They're meant to give multiple teams, users, and projects a virtually separated environment to work on, and prevent teams from getting in each other's way by limiting what Kubernetes objects teams can see and access.

Labels distinguish resources within a single namespace. They are key/value pairs that describe attributes, and can be used to organize and select subsets of objects. Labels allow for efficient queries and watches, and are ideal for use in user-oriented interfaces to map organization structures onto Kubernetes objects.

Labels are often used to describe release state (stable, canary), environment (development, testing, production), app tier (frontend, backend) or customer identification. Selectors use labels to filter or select objects, and are used throughout Kubernetes. This prevents objects from being hard linked.

Annotations, on the other hand, are a way to add arbitrary non-identifying metadata, or baggage, to objects. Annotations are often used for declarative configuration tooling; build, release or image information; or contact information for people responsible.

Kubernetes Tooling and Clients:

Here are the basic tools you should know:

- Kubeadm bootstraps a cluster. It's designed to be a simple way for new users to build clusters (more detail on this is in a later chapter).
- Kubectl is a tool for interacting with your existing cluster.
- Minikube is a tool that makes it easy to run Kubernetes locally. For Mac users, HomeBrew makes using Minikube even simpler.

There's also a graphical dashboard, Kube Dashboard, which runs as a pod on the cluster itself. The dashboard is meant as a general-purpose web frontend to quickly get an impression of a given cluster.

Kubernetes Deployment Options

Just as there are many parts that comprise Kubernetes, there are multiple ways to go about deploying Kubernetes. The best approach for your needs depends on your team's technical expertise, your infrastructure availability (or lack thereof), your capital expenditure and ROI goals, and more.

This chapter explains the main Kubernetes deployment options, and provides an overview of how to use each one.

Kubernetes Deployment from Scratch

Deploying a Kubernetes cluster from scratch can be a daunting task. It requires knowledge of its core concepts, the ability to make architecture choices, and expertise on the deployment tools and knowledge of the underlying infrastructure, be it on-premises or in the cloud.

Selecting and configuring the right infrastructure is the first challenge. Both on-premises and public cloud infrastructure have their own difficulties, and it's important to take the Kubernetes architecture into account. You can choose to not run any pods on master nodes, which changes the requirements for those machines. Dedicated master nodes have smaller minimum hardware requirements.

Big clusters put a higher burden on the master nodes, and they need to be sized appropriately. It's recommended to run at least three nodes for etcd, which allows a single node failure. While it may be tempting to run etcd on the same nodes as the Kubernetes master nodes, it's recommended to create and run the etcd as a separate cluster.

Adding more nodes will protect against multiple node failures simultaneously (5 nodes/2 failures and 7 nodes/4 failures), but each node added can decrease Kubernetes' performance. For master nodes, running two protects against failure of any one node.

For both the etcd cluster and Kubernetes master nodes, designing for availability across multiple physical locations (such as Availability Zones in AWS) protects the Kubernetes environment against physical and geographical failure scenarios.

On-Premises Implementations

Many on-premises environments are repurposed to integrate with Kubernetes (like creating clusters on top of VMs). In some cases, a new infrastructure is created for the cluster. In any case, integrating servers, storage and networking into a smoothly-running environment is still highly-skilled work.

For Kubernetes, planning for the right storage and networking equipment is especially important, as it has the ability to interact with these resources to provision storage, load balancers and the like. Being able to automate storage and networking components is a critical part of Kubernetes' value proposition.

Public Cloud

This is why many, for their first foray into Kubernetes, spin up clusters in public cloud environments. Kubernetes deployment tools integrate natively with public cloud environments, and are able to spin up the required compute instances, as well as configure storage and networking services for day-to-day operations.

> *For cloud instances, it's critically important to select the right instance type. While some*

> *instance types are explicitly a bad idea (for example, VMs with partial physical CPUs assigned or with CPU oversubscription), others might be too expensive. An advantage of public clouds is their consumption-based billing, which provides the opportunity to re-evaluate consumption periodically.*

Hybrid implementations: On-Premises + Public Clouds and at the Edge

These are the most complex environments of all. For these, you may want to look into a Managed Kubernetes solution, so that you do not have to do the heavy lifting yourself.

See a comparison of leading Enterprise Kubernetes solutions here.

Networking Concerns

The slightly-different-than-usual networking model of Kubernetes requires some planning. The most basic networking pieces are the addresses for the nodes and public-facing Kubernetes parts. These are part of the regular, existing network. Kubernetes allocates an IP block for pods, as well as a block for services. Of course, these ranges should not collide with existing ranges on the physical network.

Depending on the pod network type – overlay or routed – additional steps have to be taken to advertise these IP blocks to the network or publish services to the network.

Lifecycle Management

There are various tools to manage the lifecycle of a Kubernetes cluster. Broadly speaking, there are tools for the deployment and lifecycle management of clusters, and there are tools for interacting with a cluster for day-to-day operations.

Let's walk through a couple of the more popular tools:

Kubeadm

Kubeadm is the default way of bootstrapping a best-practice-compliant cluster on existing infrastructure. Add-ons and networking setup are both out of scope for Kubeadm, as well as provisioning the underlying infrastructure.

Kubespray

Kubespray takes the configuration management approach, based on Ansible playbooks. This is ideal for those already using Ansible and who are comfortable with configuration management tooling. Kubespray uses kubeadm under the hood.

MiniKube

MiniKube is one of the more popular ways of trying out Kubernetes locally. The tool is a good starting point for taking first steps with Kubernetes. It launches a single-node cluster inside a VM on your local laptop. It runs on Windows, Linux and MacOS, and has a dashboard.

Kops

Kops allows you to control the full Kubernetes cluster lifecycle, from infrastructure provisioning to cluster deletion. It's mainly for deploying on AWS, but support for GCE and VMware vSphere is coming.

Various cloud vendors use their own proprietary tools for deploying and managing cluster lifecycle. These are delivered as part of the managed Kubernetes service, and usually not surfaced up to the user.

Configuration and Add-ons

Add-ons extend the functionality of Kubernetes. They fall into three main categories:

Networking and network policy

These include addons that create and manage the lifecycle of networks, such as Calico (routed) and Flannel (VXLAN overlay)

Service discovery

While Kube-DNS is still the default, CoreDNS will replace it, starting with version 1.13, to do service discovery.

User interface

The Kubernetes dashboard is an addon.

While the name add-on suggests some of these are optional, in reality many are required for a production-grade Kubernetes environment. Choosing the most suitable network provider, like Flannel or Calico, is crucial for integrating the cluster into the existing environment, be it on-premises or in the cloud.

Kubernetes Helm

Although technically not an addon, Helm is considered a vital part of a well-functioning Kubernetes cluster. Helm is the package manager for Kubernetes. Helm Charts define, install and upgrade applications. These application packages (Charts) package up the configuration of containers, pods, and anything else for easy deployment on Kubernetes.

Helm is used to find and install popular software packages and manage the lifecycle of those applications. It's somewhat comparable to a Docker Registry, only Helm charts might contain multiple containers and Kubernetes-specific information, like pod specifications. Helm includes a default repository for many software packages:

- Monitoring: SignalFX, NewRelic, DataDog, Sysdig, ELK-stack (elasticsearch, logstash, kibana), Jaeger, Grafana, Fluentd, Prometheus, Sensu
- Databases: CockroachDB, MariaDB, CouchDB, InfluxDB, MongoDB, Percona, Microsoft SQL on Linux, PostgreSQL, MySQL
- Key/Value: etcd, Memcached, NATS, Redis
- Message Systems: Kafka, RabbitMQ
- CI/CD: Concourse CI, Artifactory, Jenkins, GitLab, Selenium, SonarQube, Spinnaker
- Ingress and API Gateways: Istio, Traefik, Envoy, Kong
- Application and Web Servers: Nginx, Tomcat
- Content Management: WordPress, Joomla, Ghost, Media-Wiki, Moodle, OwnCloud
- Storage: OpenEBS, Minio

There are unique Charts available for things like the games Minecraft and Factorio, the smart home automation system Home Assistant, and Ubiquiti's wireless access point controller, UniFi SDN. Helm makes the life of application developers easier by eliminating the toil of finding, installing and managing the lifecycle of many popular software packages.

Deeper Dive on Kubernetes Deployment: Multiple Clusters, Multiple Tenants and Federation

Not only are there multiple Kubernetes deployment models depending on which infrastructure and management tools you use, but there are also different deployment patterns based on how many Kubernetes clusters you implement, and how "tenants" -- meaning individual workloads -- are managed within them.

Summary of Deployment Models

The deployment models for Kubernetes range from single-server acting as master and worker with a single tenant, all the way to multiple multi-node clusters across multiple datacenters, with federation enabled for some applications.

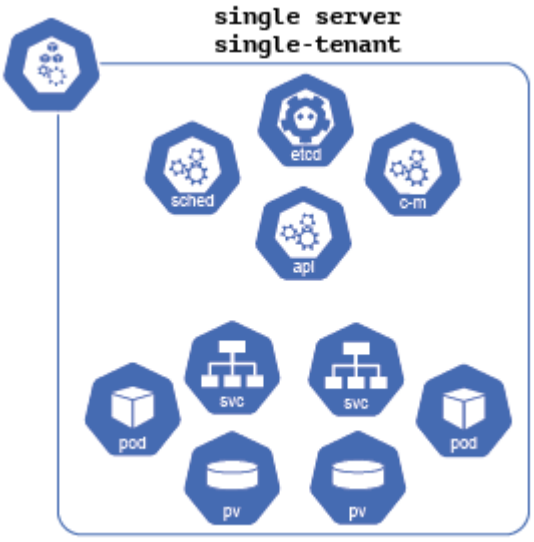
In this chapter, we focus on the more common deployment models and how they are different, as opposed to how they are similar. Some key differences include single-server models which are typically optimized for the use of developers, and not meant for production. Whereas production systems recommend a minimum of multi-master (with an odd number of nodes) and often end up with multi-cluster, and Federation becomes a very real consideration when the deployments need to scale to thousands of nodes and across regions.

Type	Highly Available	Replicated Config	NameSpace Isolation
single-tenant	no	no	no
multi-tenant	N/A	N/A	yes
multi-master	yes	yes	N/A
multi-cluster	yes	only within each cluster	N/A
Federation	N/A	yes	recommended

Single-tenant

A simple deployment, often using a single server, is perfect for a developer or QA team member who is trying to ensure containers are being built properly, and all scripts are Kubernetes compatible. These single-server deployments use a mini distribution such as minikube, or a limited deployment of a full suite like **Platform9 Managed Kubernetes Free Tier** (PMKFT).

These single-server single-tenant deployments run everything, not only on the same host but in the same name space. This is NOT a good practice for production, as it exposes application, deployment, and pod-specific configuration items – from secrets, to resource items like storage volumes, to all containers on the node. In addition, the network is wide open, so there are no restrictions on container interactions, which is against both network and security best practices.



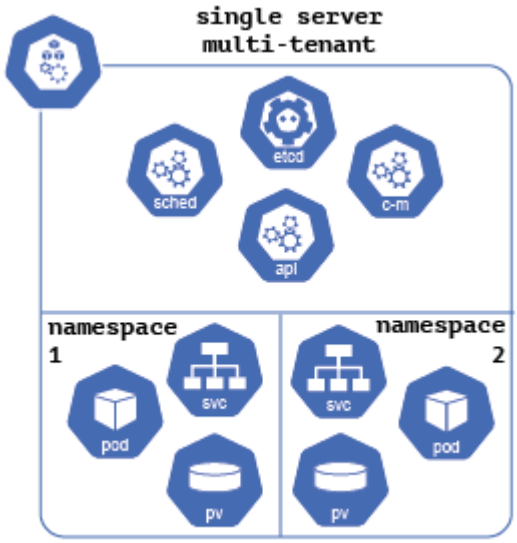
Multi-tenant

Once you passed the basic stage of just having Kubernetes running – whether it was a mini distribution or a full Kubernetes install with only the basic default networking enabled – you have, no doubt, run into the issue where services can access whatever they want, with no control. This makes enforcing any kind of security model or application flow nearly impossible, as developers can simply elect to ignore it and call anything they want.

Multi-tenant is the term for when a program, like a Kubernetes cluster, is configured to allow multiple areas within its purview to operate in isolation. This is critical when any kind of data is used, especially if it falls under financial or privacy regulations. The additional benefits are that multiple development teams, QA staff, or other entities can work in parallel within the same cluster with their own quotas and security profiles, without having any negative impact on the other teams also working in the cluster.

Within Kubernetes, this is primarily accomplished at the network layer for deployed pods that can only see pods within their name space by default, and can be granted access via network policies to get access to other namespaces and the applications they are running.

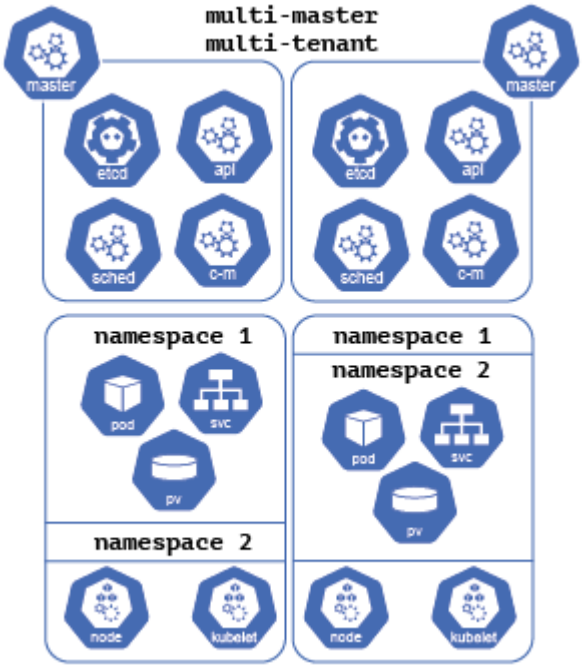
Additional technology, called Service Meshes, can be added to environments that are multi-tenant. The purpose is to increase network resilience between pods, enforce authentication and authorization policies, and to increase compliance to defined network policies. Service Meshes can also be leveraged for intercluster communications, but that is beyond what most deployments currently use them for.



Multi-Master

When Kubernetes is going to be used to support any kind of real workload, it is best to start down the path of having clusters be multi-master. By having more than one master server, the Kubernetes cluster can continue to be administered even in the event that not only a container like the api server fails, but if an entire server crashes taking out instances of all the operators and other controllers that are deployed and required to maintain a steady state across all the worker nodes.

> *Special consideration needs to be paid to etcd, which is the most commonly-used datastore for Kubernetes clusters. As etcd requires a quorum to support read-write operations, it needs to have a minimum of three nodes to be highly available and recommends having an odd number that will increase based on the number of worker nodes in the cluster. By the time you get to about 100 worker nodes, you'll be having five to seven etcd servers to handle the load. Some Kubernetes distributions have etcd share the same servers as the master control plane containers, which works great as long as the individual nodes are sized appropriately.*

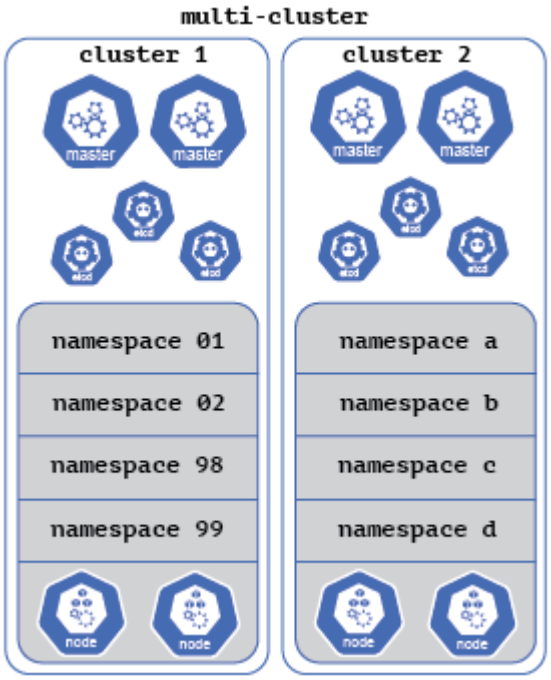


Multi-Cluster

Multi-cluster environments are typically configured as multi-master and multi-tenant, as well. If you are big enough to need multiple clusters, there are probably high-availability and isolation requirements driven by at least a few of the groups that are actively building and deploying containers within your hybrid cloud infrastructure.

A multi-cluster deployment is simply deploying multiple clusters across one or more data centers, ideally with a single management interface to simplify life; although, that is not always the case. Companies like Platform9 offer products with multi-cluster management capabilities that can work with infrastructure provided by all the biggest cloud providers, in addition to on-premises infrastructure like OpenStack and VMware.

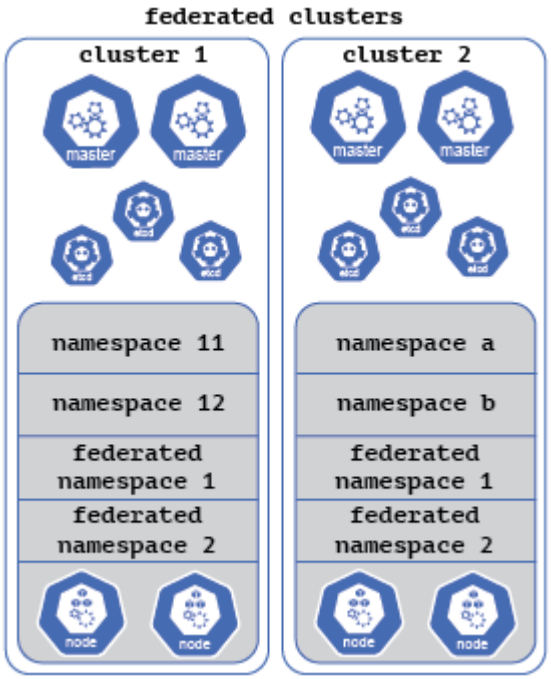
> *The reasons why you might set up multiple clusters center on separating development from production. In addition, since it is not recommended to have a Kubernetes cluster that spans more than a single datacenter, having one or more regions in use across single- or multiple-cloud providers is another reason to adopt a multi-cluster model.*



Federation

Kubernetes Federation is the cherry on top of a multi-cluster deployment as it allows parts of the configuration to be synchronized across specific namespaces in the clusters that are federated. Where this becomes invaluable is when you have the same application and services spread across multiple datacenters or cloud regions, for reasons like capacity and avoiding a single point of failure.

By having these applications managed through Federation, the CI/CD or another deployment process can issue a single Kubernetes Deployment. It can roll out an application update across a truly global infrastructure while coordinating between the member clusters to ensure the chosen deployment model is being adhered to regardless of what cloud provider or region it is located in. This could be a hard cutover to the new application, canary deployments, or even just a gradual cutover leveraging a service mesh so there are no interrupted user sessions.


federated clusters
cluster 1
master
master
etcd
etcd
etcd
namespace 11
namespace 12
federated namespace 1
federated namespace 2
node
node
cluster 2
master
master
etcd
etcd
etcd
namespace a
namespace b
federated namespace 1
federated namespace 2
node
node

Top Use Cases for Kubernetes

So far, we've covered why Kubernetes is valuable for modern software delivery teams, how Kubernetes works as a platform and how to go about deploying Kubernetes. Now, let's take a look at the main reasons why you would use Kubernetes.

The list of Kubernetes use cases discussed in this chapter isn't exhaustive; there are a variety of good reasons to adopt Kubernetes beyond what is discussed here. Below are simply some of the most common deployment patterns.

Kubernetes has gained popularity for a number of use cases, given its unique features. It's a suitable platform to run stateless, 12-factor apps, and is easy to integrate into CI/CD pipelines due to its open API. Let's review some of the key use cases that are primed for taking advantage of Kubernetes-based applications.

(Note also that there are many use cases and tutorials beyond the ones detailed here, available at the Kubernetes website.)

Simple Deployment of Stateless Applications

A very popular stateless app to run on top of Kubernetes is nginx, the open-source web server. Running nginx on Kubernetes requires a deployment YAML file that describes the pod and underlying containers. Here's an example of a deployment.yaml for nginx:

```
apiVersion: apps/v1 # for versions before 1.9.0 use
apps/v1beta2
kind: Deployment
metadata:
 name: nginx-deployment
spec:
 selector:
 matchLabels:
 app: nginx
 replicas: 2 # tells deployment to run 2 pods
matching the template
 template:
 metadata:
 labels:
 app: nginx
 spec:
 containers:
 - name: nginx
 image: nginx:1.7.9
```

```
 ports:
 - containerPort: 80
```

Create this deployment on a cluster based on the YAML file:

```
kubectl apply -f https://k8s.io/examples/application/
deployment.yaml
```

The code creates two pods, each running a single container:

```
NAME READY STATUS RESTARTS AGE
nginx-deployment-1771418926-7o5ns 1/1 Running 0 16h
nginx-deployment-1771418926-r18az 1/1 Running 0 16h
```

While manually creating these deployment files and creating pods is helpful for learning Kubernetes, it isn't the easiest way.

An easier way to deploy the same nginx application is to use the Kubernetes package manager, Helm. Using Helm, the deployment looks like this:

```
helm install docs/examples/nginx
```

Deploying an app like this is easy; in the example, we skipped over the harder parts, like exposing the web server outside of the cluster, and adding storage to the pod.

And this is where Kubernetes is both a little complex to get started, as well as explicit about separation of services and functionality. For example, if we were to add a database connection to this nginx pod to store data for a WordPress-based website, here's how that would work.

First, we'd need to add a service, to make the database (running outside of the Kubernetes cluster for now) available for consumption in pods:

```
kind: Service
apiVersion: v1
metadata:
 name: external-mysql-service
Spec:
 type: ClusterIP
 ports:
 - port: 3306
```

```
 targetPort: 3306
selector: {}
```

Since the database doesn't run on the cluster, we don't need to add any pod selectors. Kubernetes doesn't know where to route the traffic, so we need to create an endpoint to instruct Kubernetes where to send traffic from this service.

```
kind: Endpoints
apiVersion: v1
metadata:
 name: external-mysql-service
subsets:
 - addresses:
 - ip: 10.240.0.4
 ports:
 - port: 3306
```

Within the WordPress configuration, you can now add the MySQL database using the metadata name from the example above,

```
external-mysql-service.
```

Determining what components your app uses before deployment makes it easier to separate them into separate containers in a single pod, or different pods; even different services, external or on Kubernetes. This separation helps in fully using all of Kubernetes' features like horizontal auto-scaling and self-healing, which only works well if the pods adhere to the rules and assumptions Kubernetes makes about data persistence.

Deploy Stateful Data Services

In the previous example, we assumed that the MySQL instance ran outside of Kubernetes. What if we want to put it under Kubernetes' control and run it as a pod?

The StatefulSet pod type and a PersistentVolume help with this. Let's look at an example:

```
apiVersion: v1
kind: Service
metadata:
 name: mysql-on-kubernetes
spec:
 ports:
 - port: 3306
 selector:
 app: mysql
```

```
 clusterIP: None
---
apiVersion: apps/v1 # for versions before 1.9.0 use
apps/v1beta2
kind: Deployment
metadata:
 name: mysql-on-kubernetes
spec:
 selector:
 matchLabels:
 app: mysql
 strategy:
 type: Recreate
 template:
 metadata:
 labels:
 app: mysql
 spec:
 containers:
 - image: mysql:5.6
 name: mysql
 env:
 # Use secret in real usage
 - name: MYSQL_ROOT_PASSWORD
 value: password
 ports:
 - containerPort: 3306
 name: mysql
 volumeMounts:
 - name: mysql-persistent-storage
 mountPath: /var/lib/mysql
 volumes:
 - name: mysql-persistent-storage
 persistentVolumeClaim:
 claimName: mysql-pv-claim
```

The file defines a volume mount for /var/lib/mysql, and then creates a PersistentVolumeClaim that looks for a 20GB volume. This claim is satisfied by any existing volume that meets the requirements, or by a dynamic provisioner.

This means we need to create a PersistentVolume that satisfies the claim:

```
kind: PersistentVolume
apiVersion: v1
metadata:
```

```
 name: mysql-pv-volume
 labels:
 type: local
spec:
 storageClassName: manual
 capacity:
 storage: 20Gi
 accessModes:
 - ReadWriteOnce
 hostPath:
 path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: mysql-pv-claim
spec:
 storageClassName: manual
 accessModes:
 - ReadWriteOnce
 resources:
 requests:
 storage: 20Gi
```

Using the same data structure and the same tools, it's possible to create services on the platform with persistent storage. While it's a little extra work to re-platform data services like databases onto the Kubernetes platform, it does make sense.

With such a clear separation between application binaries, configuration and data in the Kubernetes layers, and a distinct separation between a container and the underlying OS, many of the lifecycle issues of the VM world disappear. Managing the lifecycle for these applications is traditionally very hard, a lot of work, and, not unusually, requires downtime.

> *With Kubernetes, by contrast, a new version of the application can be deployed by deploying a new pod (with the new container version) to production. As this only switches the container, and doesn't need to include the underlying OS and higher-level configuration and data, this is a very fast and lightweight operation.*

CI/CD Platform with Kubernetes

Kubernetes' open API brings many advantages to developers. The level of control means developers can integrate Kubernetes into their automated CI/CD workflow effortlessly. So even while Kubernetes doesn't provide any CI/CD features out of the box, it's very easy to add Kubernetes to a CI/CD pipeline.

Let's look at Jenkins, the popular CI solution. Running Jenkins as a pod is easy, by deploying it via the Kubernetes package manager, Helm.

```
$ helm install --name my-release stable/jenkins
```

The more interesting part is driving Kubernetes from within Jenkins. This uses a plugin in Jenkins that allows Jenkins to dynamically provision a Jenkins Agent on Kubernetes to run a single build. These agents even have a ready-to-go Jenkins-slave Docker image.

This setup allows developers to automate their pipeline: each new code commit on git triggers a container build (which is built using the Jenkins Agent) and subsequently pushed to Kubernetes to replace the old version of the app in question for a rolling upgrade. (see more detailed explanation here).

Kubernetes and Multi-Cloud Management

Beyond the basic use cases described in the preceding chapter, another common reason to adopt Kubernetes today is to simplify the considerable challenge of managing the multi-cloud architectures that are increasingly common within IT strategies today.

A multi-cloud strategy is hard. Fortunately, Kubernetes makes it easier. In several respects, Kubernetes helps to solve or work around the most pressing pain points within a multi-cloud architecture.

Let's explore how.

Multi-cloud challenges

Multi-cloud architectures can deliver a number of benefits, including better cost-efficiency, reliability and scalability. Yet those benefits come with tradeoffs.

The reason why is simple: The more clouds you have, the more challenging it is to configure and manage them. Multi-cloud models involve more integrations between different clouds and different services, more accounts to oversee, more vendor-specific tools and processes to worry about, and so on.

And if your multi-cloud strategy includes a hybrid cloud (which it does if you have on-premises infrastructures or private clouds running alongside public clouds), integrating and managing all of the complexity becomes even more challenging.

Using Kubernetes to address multi-cloud challenges

Kubernetes can't magically solve every multi-cloud challenge. But it can make a number of multi-cloud pain points easier to manage. In particular, consider the following ways in which Kubernetes can help reduce the complexity and risk of a multi-cloud strategy.

Provisioning multi-cloud with Kubernetes

One of the first challenges you will face as you attempt to roll out a multi-cloud infrastructure is provisioning all of your clouds.

When you have a single cloud, you can use vendor-supplied provisioning tools (like AWS CloudFormation) to set up your workloads in an efficient and automated way. But those tools typically don't work with third-party clouds.

You could also use provisioning tools that are not tied to any specific cloud vendor, such as Terraform or Ansible. That approach would help, but you will still likely find that you need to customize or tweak your configurations for the different clouds you are targeting.

> *With the help of Kubernetes, however, you can avoid all of this hassle. Kubernetes doesn't care which cloud it is running on. Thus, if you host your workloads in Kubernetes, you can use the same configurations on any or all of the clouds that comprise your architecture. And because Kubernetes defines its configurations as code, you get the same level of automation and efficiency in provisioning as you would from a traditional provisioning tool.*

Multi-cloud monitoring challenges

For similar reasons to those described above, monitoring multiple clouds is difficult. You can't rely on monitoring tools from individual vendors. And if you use third-party APM tools, you still typically have to tailor them to each of the clouds within your infrastructure.

When your workloads run in Kubernetes, however, the only thing your monitoring tools need to be configured to monitor, is Kubernetes. (Well, maybe some basic monitoring of your cloud infrastructure would be wise, too, but Kubernetes becomes the central focus of your monitoring efforts.) Kubernetes supplies a rich set of metrics that can be used to track workload availability and health, no matter which cloud or clouds are hosting your Kubernetes clusters.

Varying skillsets for multi-cloud administration

Another common multicloud challenge is the need for developers and IT teams to learn all of the different clouds within your architecture. Engineers who know the ins-and-outs of AWS may not be as skilled when it comes to Azure and GCP, for example. And if you have private cloud platforms like OpenStack in the mix, the skills required to administer your multi-cloud architecture become even more diverse.

> *A Kubernetes-based strategy significantly simplifies these requirements. It allows your engineers to focus primarily on Kubernetes. They may still need to have some basic ability to work with whichever clouds (or on-premises infrastructure) you are using to host Kubernetes, but their main focus shifts to Kubernetes.*

Securing multi-cloud

Much has been written about the challenges of multi-cloud security. The more clouds you have, the larger your attack surface, and the more potential security vulnerabilities you are exposed to.

Here again, however, relying on Kubernetes to host your workloads helps to simplify security configurations and reduce your attack surface. Although Kubernetes doesn't completely eliminate the need to secure underlying cloud infrastructures using each cloud's IAM framework, or to monitor the infrastructure as a whole for security incidents, Kubernetes does help to standardize your configurations in order to reduce the risk of oversights that could create security vulnerabilities.

At the same time, Kubernetes-native features, like pod security policies, network policies and RBAC, provide an additional set of tools to help protect workloads. These would not be available in a multi-cloud architecture that did not incorporate Kubernetes.

Redundancy and availability

One of the most common reasons to adopt multi-cloud is to increase workload availability by hosting redundant instances of the same workload.

You can certainly achieve redundancy without Kubernetes. But it's harder to do, because your redundant workloads would in most cases not be identical. Instead, you might set up a VM on AWS, and another one on Azure. Even if each of these instances hosts the same workload, each one would need to be configured somewhat differently, according to the needs of the cloud that hosts it.

If you containerize your applications and host them on Kubernetes, however, redundancy is much easier to achieve. You can configure the workload once and then reproduce it across as many clouds as you like – because, again, Kubernetes doesn't care which cloud it is hosted on.

Controlling multi-cloud costs

Multi-cloud is also often touted as a way to reduce costs, because it lets you pick and choose the most cost-

efficient solutions from among multiple cloud vendors.

That's true. However, it's also true that the more clouds you have, the easier it is to spin up unnecessary workloads, or otherwise waste cloud resources. If you do this, you undercut the cost-efficiency goals of your multi-cloud strategy.

> *While it's certainly possible to use Kubernetes in a cost-inefficient way, it is arguably harder to make cost-related mistakes with Kubernetes. That is due, in part, to the fact that Kubernetes standardizes workloads and configurations across all clouds. That makes it easier to set up environments in a consistently cost-efficient way, without worrying about the nuances of each cloud provider. At the same time, pricing for managed Kubernetes services tends to be more clear-cut than do pricing schemes for many other types of cloud services. There are fewer variables, which makes costs easier to predict.*

Conclusion

If you've read this far, you know that Kubernetes is a powerful, but complex, platform. Not only does it involve numerous different parts, but there are also a variety of different ways and reasons to use it. And, we've covered only the essentials of these topics in this eBook.

For further guidance on how to make the most of Kubernetes, follow the Platform9 blog for subsequent eBooks that dive deeper into topics like how to architect a Kubernetes cluster and how specific teams within an IT organization can best leverage Kubernetes.

PLATFORM9

Freedom in Cloud Computing

Instantly Deploy Open Source Kubernetes for Free
On-premises, AWS, Azure

Sign up today: platform9.com/signup

About Platform9

Platform9 enables freedom in cloud computing for enterprises that need the ability to run private, edge or hybrid clouds. Our SaaS-managed cloud platform makes it easy to operate and scale clouds based on open-source standards such as Kubernetes and OpenStack; while supporting any infrastructure running on-premises or at the edge.