PLATFORM9 PRESENTS

THE GORILLA GUIDE TO...®

Kubernetes in the Enterprise

Joep Piscaer

INSIDE THE GUIDE:

- Why the Industry Has Embraced Containers
- Keys to Smoothing Your Kubernetes On-ramp
- Managed Kubernetes Provides the Optimal Experience

HELPING YOU NAVIGATE THE TECHNOLOGY JUNGLE!

ActualTech Media
www.actualtechmedia.com

In Partnership With
PLATFORM9

THE GORILLA GUIDE TO...

Kubernetes in the Enterprise

AUTHOR
Joep Piscaer

EDITOR
Keith Ward, ActualTech Media

LAYOUT AND DESIGN
Olivia Thomson, ActualTech Media

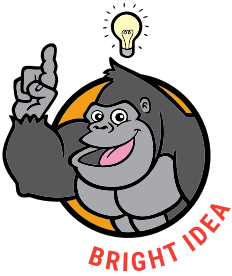
ACTUALTECH MEDIA
Okatie Village Ste 103-157
Bluffton, SC 29909
www.actualtechmedia.com

ENTERING THE JUNGLE

CALLOUTS USED IN THIS BOOK

The Gorilla is the professorial sort that enjoys helping people learn. In the School House callout, you'll gain insight into topics that may be outside the main subject but are still important.

This is a special place where you can learn a bit more about ancillary topics presented in the book.



When we have a great thought, we express them through a series of grunts in the Bright Idea section.

Takes you into the deep, dark depths of a particular topic.

Discusses items of strategic interest to business leaders.

ICONS USED IN THIS BOOK

DEFINITION

Defines a word, phrase, or concept.

KNOWLEDGE CHECK

Tests your knowledge of what you've read.

PAY ATTENTION

We want to make sure you see this!

GPS

We'll help you navigate your knowledge to the right place.

WATCH OUT!

Make sure you read this so you don't make a critical error!

CHAPTER 1

The Changing Development Landscape

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. See **Figure 1**.

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.

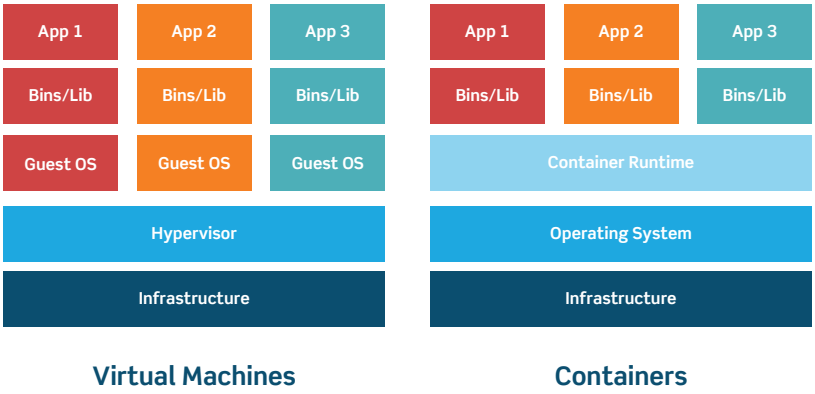


Figure 1: Virtualization vs. containerization

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 in-place 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:

1. **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.
2. **NATS.** This is a simple, high-performance open source message queueing and publish/subscribe system for cloud-native applications.
3. **gRPC.** This is a high-performance, open source universal RPC framework.
4. **Containerd.** This is an industry-standard container runtime with an emphasis on simplicity, robustness and portability.
5. **Linkerd.** An ultralight service mesh for Kubernetes and beyond, Linkerd provides observability, reliability, and security for microservices, with no code change required.
6. **CNI.** The Container Network Interface provides networking for Linux containers.
7. **CSI.** This stands for Container Storage Interface. It provides storage for Linux containers.
8. **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 (**Figure 2**).

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 Platform-

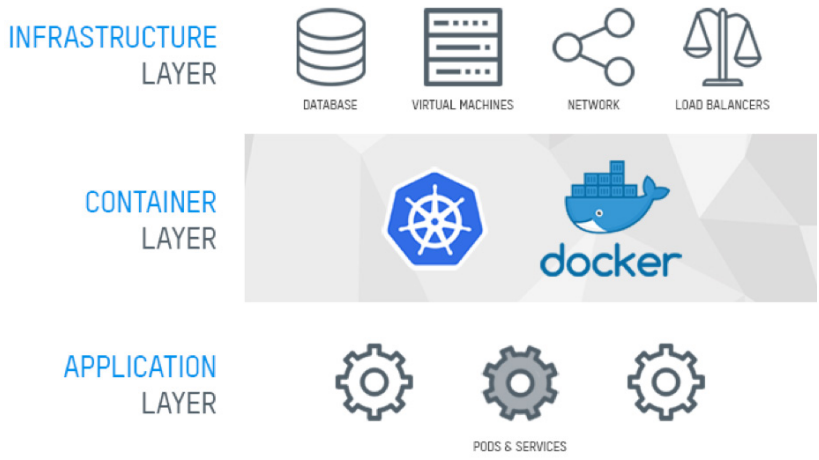


Figure 2: The Kubernetes layer cake of infrastructure, containers and applications.

as-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.

Yes, Containers Can Be Stateful

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.

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,[1] 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

[1] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

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.

CHAPTER 2

Kubernetes Concepts and Architecture

As stated before (but is worth stating again), Kubernetes is an open-source platform for deploying and managing containers. It provides a container runtime, container orchestration, container-centric infrastructure orchestration, self-healing mechanisms, service discovery and load balancing. It's used for the deployment, scaling, management, and composition of application containers across clusters of hosts.

But Kubernetes is more than just a container orchestrator. It could be thought of as the operating system for cloud-native applications in the sense that it's the platform that applications run on, just as desktop applications run on MacOS, Windows, or Linux.

It aims to reduce the burden of orchestrating underlying compute, network, and storage infrastructure, and enable application operators and developers to focus entirely on container-centric workflows for self-service operation. It allows developers to build customized workflows and higher-level automation to deploy and manage applications composed of multiple containers.

While Kubernetes runs all major categories of workloads, such as monoliths, stateless or stateful applications, microservices, services, batch jobs and everything in between, it's commonly used for the microservices category of workloads.

In the early years of the project, it mostly ran stateless applications, but as the platform has gained popularity, more and more storage

integrations have been developed to natively support stateful applications (see "Yes, Containers Can Be Stateful" on **page 16**.)

Kubernetes is a very flexible and extensible platform. It allows you to consume its functionality a-la-carte, or use your own solution in lieu of built-in functionality. On the other hand, you can also integrate Kubernetes into your environment and add additional capabilities.

Kubernetes Constructs and Concepts

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*). See **Figure 3**.

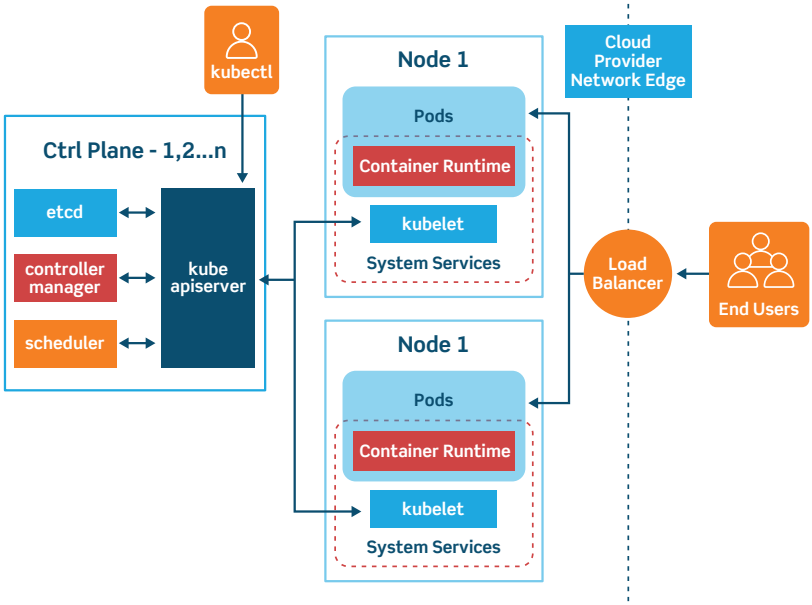


Figure 3: Architectural overview of Kubernetes.

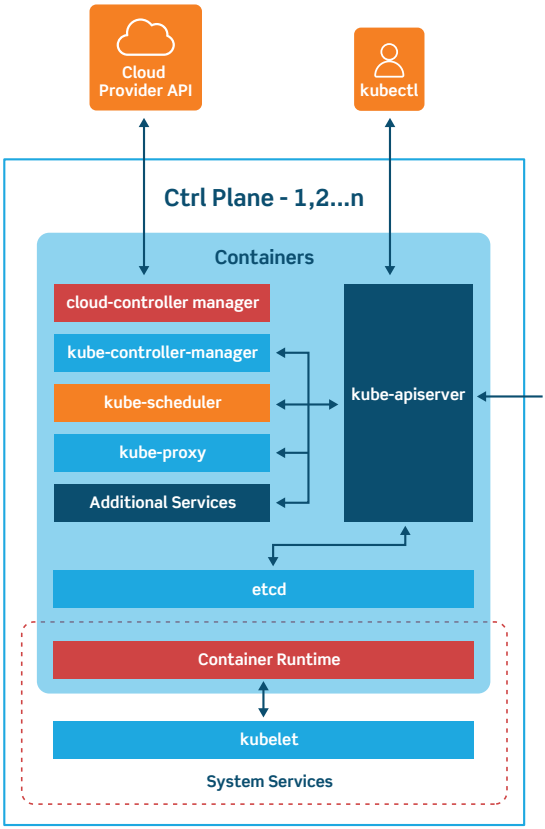


Figure 4: Kubernetes' control plane taxonomy.

Control Plane

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 **Figure 4** 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

Cluster nodes are machines that run containers and are managed by the master nodes (**Figure 5**). 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.

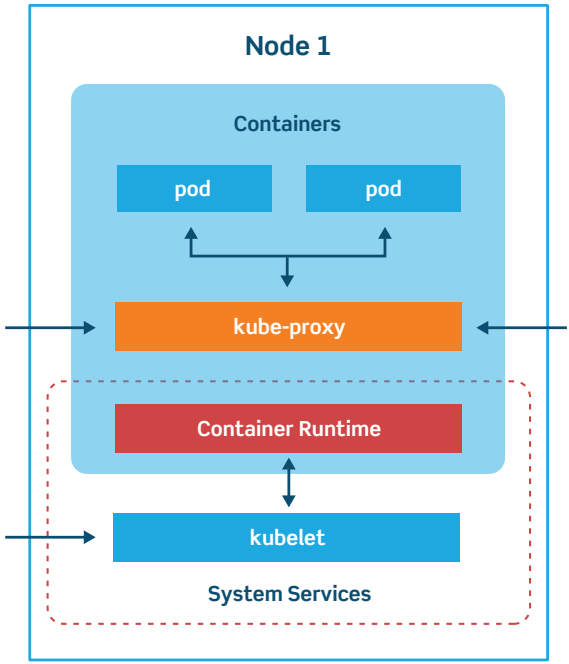


Figure 5: Kubernetes node taxonomy.

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. **Figure 6** shows the 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.

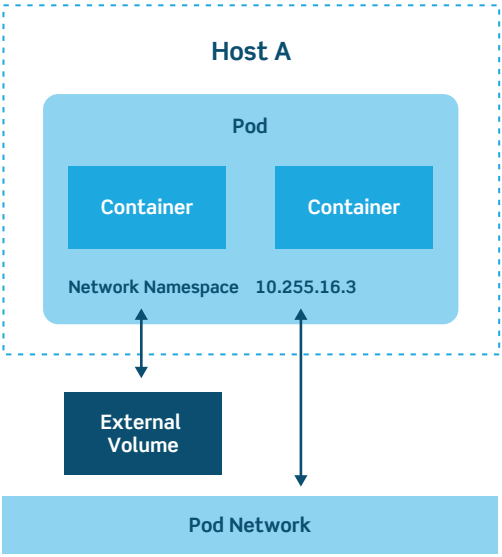


Figure 6: Pod architecture.

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.

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.

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

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 **Figure 7**). 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

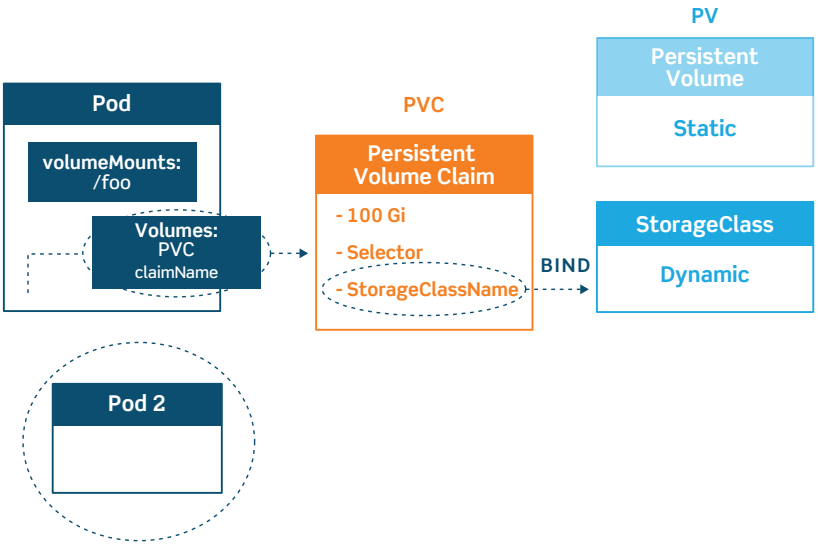


Figure 7: Persistent volumes, claims and storage classes.

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.

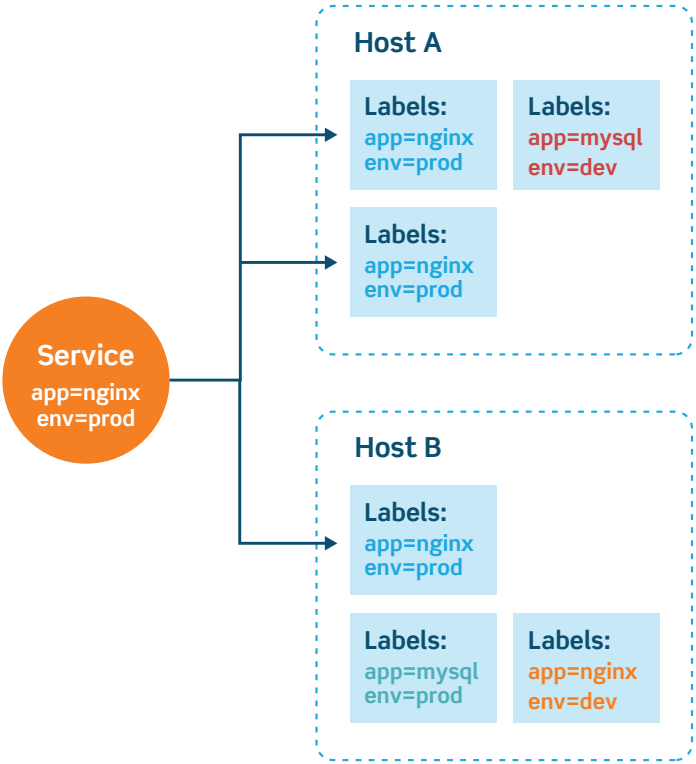


Figure 8: The Kubernetes service taxonomy.

Discovering and Publishing Services

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 **Figure 8**. 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.

1. **ClusterIP** exposes the service on an internal IP only. This makes the service reachable only from within the cluster. This is the default type.
2. **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.
3. **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.
4. **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.

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.

CHAPTER 3

Deploying Kubernetes

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.

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 *addon* 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.

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.

CHAPTER 4

Putting Kubernetes To Work

Giving Users Access

Kubernetes uses Role-based Access Control (RBAC) to regulate user access to its resources by assigning roles to users (**Figure 9**). While it's possible to let all users log in using full administrator credentials, most organizations will want to limit who has full access for security, compliance and risk management reasons.

Kubernetes' approach allows administrators to limit the number of operations a user is allowed, as well as limit the scope of said operations. In practical terms, this means users can be allowed or disallowed access to resources in a namespace, as well as granular control over who can change, delete or create resources.

RBAC in Kubernetes is based on three key concepts:

1. Verbs. This is a set of operations that can be executed on resources. There are many verbs, but they're all Create, Read, Update, or Delete (also known as CRUD).
2. Resources. These are the objects available on the clusters. They are the pods, services, nodes, PersistentVolumes and other things that make up Kubernetes.
3. Subjects: These are the objects (users, groups, processes) allowed access to the API, based on Verbs and Resources.

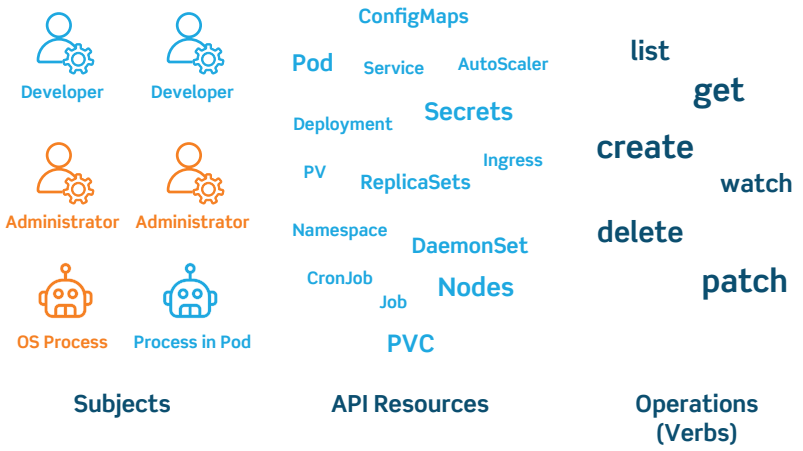


Figure 9: The type of Role Based Access Control used by Kubernetes.

These three concepts combine into giving a user permission to execute certain operations on a set of resources by using Roles (which connects API Resources and Verbs) and RoleBindings (connecting subjects like users, groups and service accounts to Roles).

Users are authenticated using one or more authentication modes. These include client certificates, passwords, and various tokens. After this, each user action or request on the cluster is authorized against the rules assigned to a user through roles.

There are two kinds of users: service accounts managed by Kubernetes, and normal users. These normal users come from an identity store outside Kubernetes.

This means that accessing Kubernetes with multiple users, or even multiple roles, is something that needs to be carefully thought out. Which identity source will you use? Which access control mode most suits you? Which attributes or roles should you define?

For larger deployments, it's become standard to give each app a dedicated service account and launch the app with it. Ideally, each app would run in a dedicated namespace, as it's fairly easy to assign roles to namespaces.

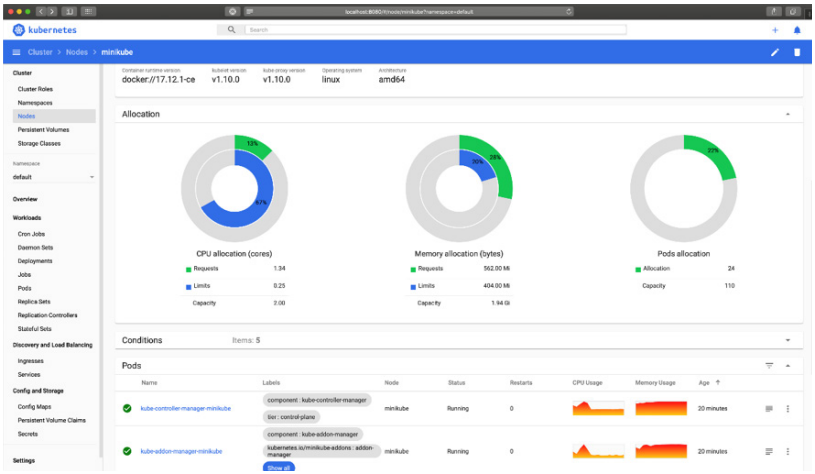


Figure 10: The Kubernetes Dashboard.

Kubernetes does lend itself to securing namespaces, granting only permissions where needed so users don't see resources in their authorized namespace for isolation. It also limits resource creation to specific namespaces, and applies quotas.

Many organizations take this one step further and lock down access even more, so only tooling in their CI/CD pipeline can access Kubernetes, via service accounts. This locks out real, actual humans, as they're expected to interact with Kubernetes clusters only indirectly.

Monitoring and Ensuring Cluster Health

The easiest way of manually checking your cluster after deployment is via the Kubernetes Dashboard,[2] shown in **Figure 10**. This is the default dashboard and is usually included in new clusters.

The dashboard gives a graphical overview of resource usage, namespaces, nodes, volumes, and pods. The dashboard provides a quick and

[2] https://github.com/kubernetes/dashboard

easy way to display information about the cluster. Because of its ease of use, it's usually the first step in a health check.

You can also use the dashboard to deploy applications, troubleshoot deployments and manage cluster resources. It can fetch an overview of applications running on your cluster, as well as create or modify individual Kubernetes resources. For example, you can scale a deployment, initiate a rolling update, restart a pod, or deploy new applications using a wizard.

The dashboard also provides information on the state of Kubernetes resources in your cluster, and any errors that may have occurred.

After deployment, it's wise to run standard conformance tests to make sure your cluster has been deployed and configured properly. The standard tool for these tests is Sonobuoy.

Clusters running as part of a service in the public cloud, like Amazon EKS, Google GKE or Azure AKS, will benefit from the managed service aspect: the cloud provider takes care of the monitoring and issue mitigation within the Kubernetes cluster. An example is Google's Cloud Monitoring service.

Monitoring and Ensuring Application Health

Most real-world Kubernetes deployments feature native, full metrics solutions. Generally speaking, there are two main categories to monitor: the cluster itself, and the pods running on top.

For cluster monitoring, the goal is to monitor the health of the Kubernetes cluster, nodes, and resource utilization across the cluster. Because the performance of this infrastructure dictates your application performance, it's a critical area.

Monitoring tools look at infrastructure telemetry: compute, storage, and network. They look at (potential) bottlenecks in the infrastructure,

such as processor and memory usage, or disk and network I/O. These resources are an important part of your monitoring strategy, as they're limited to the capacity procured, and costly to expand.

There's another important reason to study these metrics: they define the behavior of the infrastructure on which the applications run, and they can serve as an early warning sign of potential issues. Should issues be identified, you can mitigate the issue before applications dependent on that infrastructure are impacted.

Pod monitoring is slightly more complex. Not only do you want to correlate metrics from Kubernetes with container metrics, you also want application metrics. This requires a metrics and monitoring solution that hooks into all layers, and possibly into layers outside of the Kubernetes cluster.

As applications become more complex and distributed across multiple services, pods and containers, monitoring tools need to be aware of the taxonomy of applications, and understand dependencies between services and the business context in which they operate.

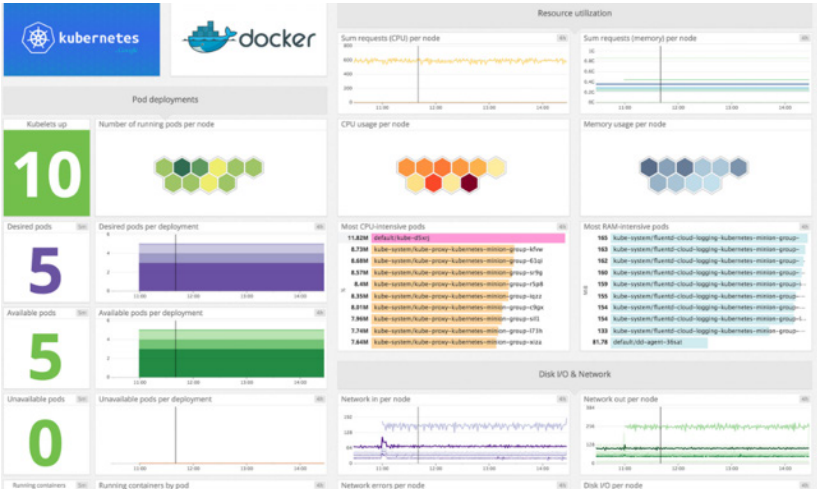


Figure 11: The DataDog Dashboard for Kubernetes.

This is where solutions like DataDog (**Figure 11**), NewRelic and AppDynamics come in. While these are proprietary solutions, they cover the whole stack: from infrastructure, Kubernetes and containers, to application tracing. This provides a complete picture of an application, as well as transactional traces across the entire system for monitoring of the end-user experience.

These complete solutions offer a unified metrics and monitoring experience and include rich dashboarding and alerting feature sets. Often, these products include default dashboard visualizations for monitoring Kubernetes, encompassing many standard integrations with components in the application stack to monitor up and down.

An open source metrics solution is Prometheus (**Figure 12**), which can natively monitor the clusters, nodes, pods, and other Kubernetes objects. It's easily deployed via kube-prometheus, which includes AlertManager for alerting, Grafana (**Figure 13**) for dashboards, and Prometheus rules combined with documentation and scripts. It provides an easy to operate, end-to-end Kubernetes cluster monitoring solution.

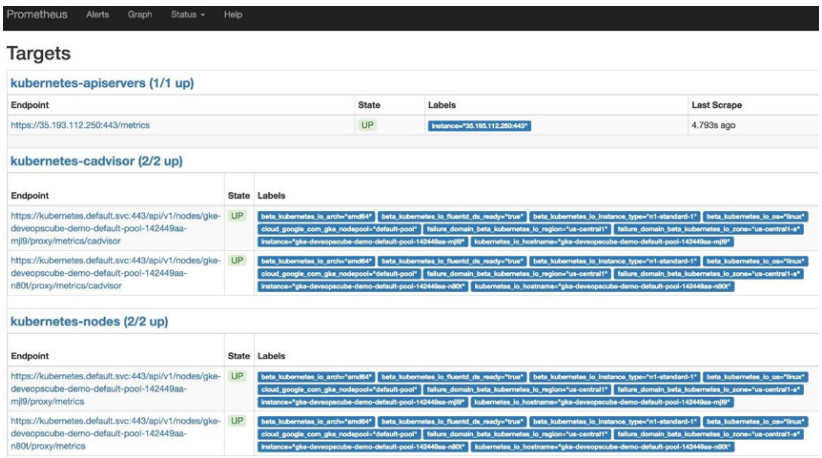


Figure 12: Prometheus Targets for Kubernetes.

Figure 13: The Grafana Dashboard for Kubernetes. https://itnext.io/kubernetes-monitoring-with-prometheus-in-15-minutes-8e54d1de2e13

This stack initially monitors the Kubernetes cluster, so it's pre-configured to collect metrics from all Kubernetes components. It also delivers a default set of dashboards and alerting rules. But it's easily extended to target multiple other metric APIs to monitor end-to-end application chains.

Prometheus can monitor custom application code and has integrations with many database systems, messaging systems, storage systems, public cloud services, and logging systems. It automatically discovers new services running on Kubernetes.

But collecting metrics is just part of the puzzle. In a microservices landscape, we need to observe behavior across the multitude of microservices to get a better understanding of the application's performance. For this reason, we need both tracing and logging (**Figure 14**).

For centralized log aggregation, there are numerous options. The default option is Fluentd, a sister project of Kubernetes. On top of that, transactional tracing systems like Jaeger give insights into the user experience as they traverse the microservice landscape.

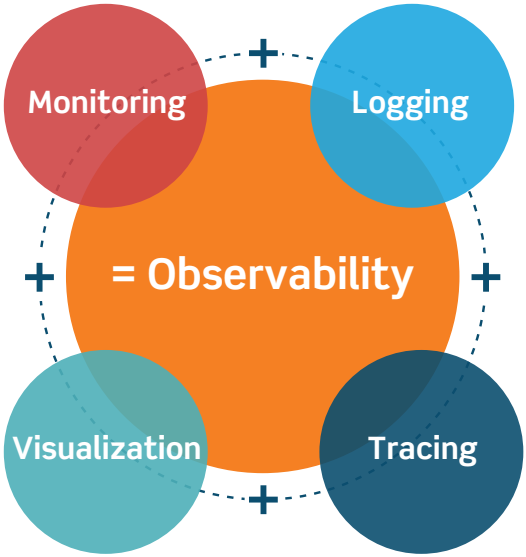


Figure 14: The elements of observability.

Persistent Storage

Managing storage in production is traditionally one of the most complex and time-consuming administrative tasks. Kubernetes simplifies this by separating supply and demand.

Admins make existing, physical storage and cloud storage environments alike available using PersistentVolumes. Developers can consume these resources using Claims, without any intervention of the admins at development or deploy time. This makes the developer experience much smoother and less dependent on the admin, who in turn is freed up from responding ad-hoc to developer requests.

CHAPTER 5

Managed Kubernetes Solutions

Running Kubernetes is still a lot of work, and requires deep domain expertise. More importantly, it takes time to design, implement and get into production. While the benefits of Kubernetes are massive, managing Kubernetes isn't just set-it-and-forget-it.

Every business must give thought to the *buy vs. rent* paradigm. It comes down to evaluating cost and risk, and judging whether they're outweighed by the advantages of a particular solution.

Building your own Kubernetes deployment is daunting. The consequences of making the wrong choices are long-lasting, and can impact application availability, performance, and agility. While building a solution in-house may be cheaper, your design might be of lesser quality or have flaws that will only be realized once you're in production.

And really, what is the business problem you're trying to solve with Kubernetes? It probably has to do with developer velocity, creating flow and reducing the work in progress. Having a sub-optimal platform will hurt those goals.

This is the true value of a managed service: making sure the service delivered is top-notch. The service provider makes sure their Kubernetes environment is highly available, resilient, flexible, up-to-date, secure, and efficient. Their job is to remove the toil and hassles of Kubernetes.

Focus on Moving the Business Forward

While installing and managing Kubernetes doesn't move the business forward, quickly deploying new applications and versions to customers does. To developers – Kubernetes' end users – platform availability is the key. They don't care who builds it or how it gets there: they just want to get their hands on it, and for it to work well.

But although developers don't care about the implementation details of a cluster or its operational state, someone still has to do the operational work to make sure the cluster is up to date, healthy, and secure.

A large part of that work consists of jobs like node lifecycle management, deploying new hosts, and making sure the hosts are kept up to date with the latest OS patches and container runtime versions.

Setting up a monitoring solution to keep an eye on Kubernetes infrastructure is notoriously difficult, and troubleshooting issues that arise is even more so. This can result in a huge time sink for admins and other infrastructure specialists, keeping them from helping developers solve their problems.

Simplifying Open Source

These reasons make it clear that for many organizations, a managed Kubernetes solution is the best option. The turnkey aspect of the service enables organizations to start working with an ecosystem of open source tools, like Kubernetes, quickly and securely.

Because for many organizations, it's not just Kubernetes. Efforts usually also include Jenkins and other CI/CD tools, a suite of observability tools for logging, metrics, tracing and dashboards, various databases and key/value stores, pub/sub message queueing systems, and more.

Setting all these up before a developer can start using the ecosystem simply takes too much time, and can overwhelm an IT department.

They need help from domain experts. Hiring outside consultants only partially solves this issue, as responsibility for any future work doesn't shift. With a managed service, on the other hand, it does.

Choosing a Managed Kubernetes Solution

It's important to choose a solution that solves these specific challenges:

- **Deployment time**
  Deploying a Kubernetes Control Plane is a piece of cake for the MSP. Plugging in your various environments should be easy, quick and frictionless.

- **Monitoring and troubleshooting**
  Any issues in the environment should be surfaced and resolved automatically (where possible). For issues where manual intervention is needed, alerts should go out to the customer.

- **Ongoing operations for upgrades and fixes**
  Managing Kubernetes is their specialty. That means the provider does the ongoing management and operations of the control plane, as well as the nodes. Upgrading Kubernetes versions and keeping the nodes up to date should be an invisible and smooth background process.

The Importance of 'As-a-Service' and SLAs

The risk of outsourcing a part of your infrastructure is sub-par quality from the MSP. Making sure the provider keeps up their end of the deal, which is to provide a highly-available, secure and performant Kubernetes control plane, is key for a successful Kubernetes deployment.

This is why a service is preferred over a software product. A product still needs to be installed, configured, and maintained by the

customer; a service is a much more hands-off experience, with service-level agreements (SLAs) in place to offer and maintain a suitable service level.

Truly Multi-cloud and Hybrid Cloud

Public cloud has caught developers' attention. They love the flexibility with which they can provision a new VM, container, key/value store or pub/sub message queue.

The downside is paying those public cloud bills. Even though public cloud is great, not everything must, or can, run in that environment for financial or data governance reasons. This means that the managed Kubernetes solution needs to be able to manage nodes across on-premises and public cloud boundaries.

Seamless Support for Other Workload Types

Containers solve a set of specific problems, but they're not a silver bullet for everything. In most IT shops, there's still a need for physical infrastructure and VMs, as well as higher-level abstractions like serverless. There's also a plethora of services that offer ready-to-go functionality that you wouldn't want to develop in-house, like mass-messaging via SMS or e-mail.

Kubernetes has a place inside this broader ecosystem, but has to seamlessly support VMs, serverless computing and third-party services. It's critically important that the managed Kubernetes provider doesn't force you to use their services for VMs, serverless or third-party services.

CHAPTER 6

Top Use Cases

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.

(Note also that there are many use cases and tutorials beyond the ones detailed here, available at the Kubernetes website.[3])

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:
```

[3] https://kubernetes.io/docs/tutorials/

```
      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. (There's a more detailed explanation available as well.[4])

[4] https://betsol.com/2018/11/devops-using-jenkins-docker-and-kubernetes/

CHAPTER 7

Platform9 Managed Kubernetes

Platform9 Managed Kubernetes (PMK) is the industry's only SaaS-based, continuously managed Kubernetes service that guarantees 24x7x365 SLA and works across any infrastructure: on-premises data centers, public clouds, and at the edge.

PMK provides comprehensive Day-Two operations for Kubernetes production clusters, with guaranteed 99.9% uptime availability. PMK's management plane performs around-the-clock,

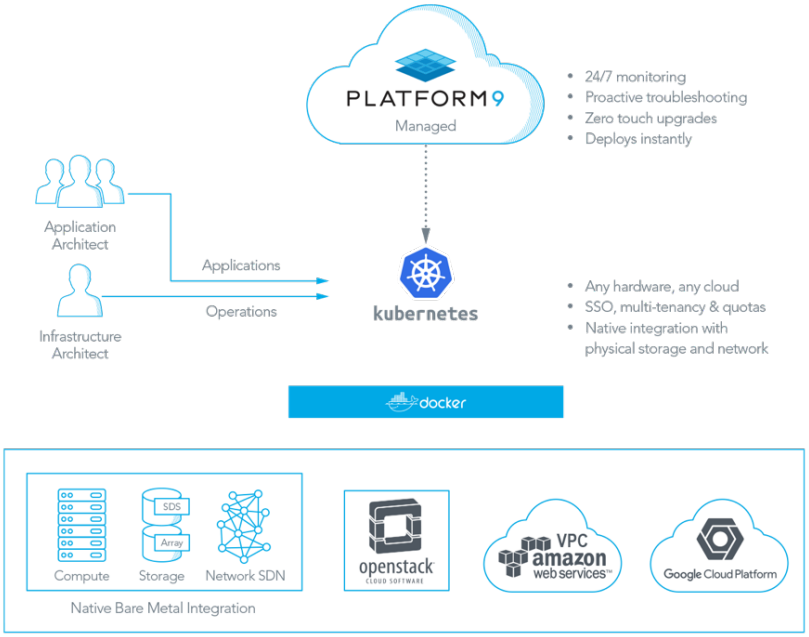


Figure 15: Managed Kubernetes Overview

automated, real-time monitoring and anomaly detection. Alerts are automatically generated, which trigger automated trouble-shooting and remediation processes.

Platform9 provides the industry's leading zero-touch upgrade process and instant security patching. Their certified Kubernetes experts provide expert guidance on best practices and architecture, and are available 24x7 to perform proactive remediation to guarantee uptime availability.

PMK's simple-to-use self-service portal allows developers and operations teams to deploy multi-cluster Kubernetes in under 20 minutes, on any infrastructure, with a simple wizard. Administrators can centrally manage clusters on different cloud providers and data centers via the dashboard (**Figure 16**). Developers can connect to their clusters through the UI or the built-in CLI accessible from the portal. You can also access your Kubernetes dashboard with a single click. Developers can deploy application templates or related services and processes via the App Catalog.

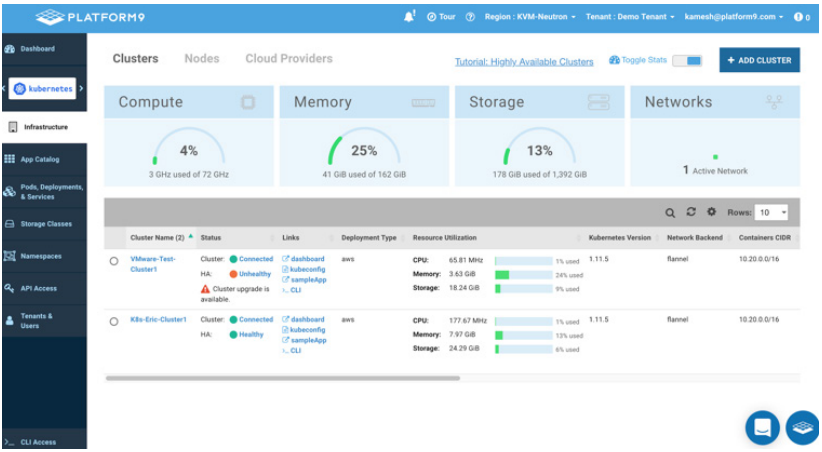


Figure 16: The Platform9 dashboard.

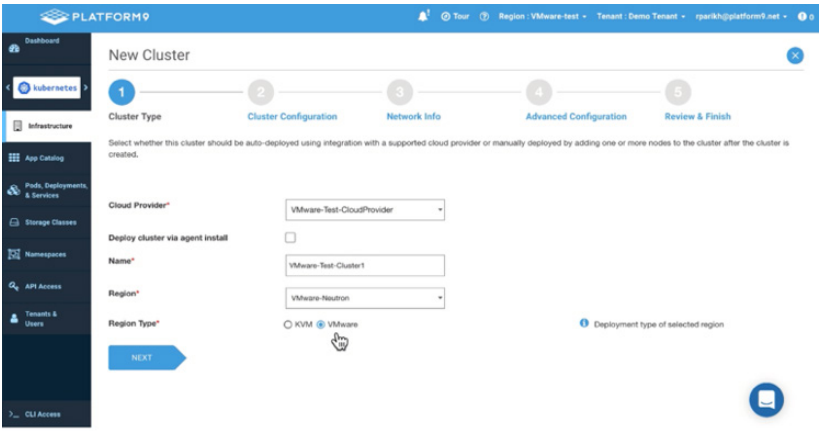


Figure 17: Creating a new Kubernetes cluster.

Getting Started with Platform9 Managed Kubernetes

The simple prep process makes it quick and easy to get a cluster up and running. For on-premises deployments, adding nodes is a matter of pushing the Platform9 agent to the machines. For cloud-based deployments, this step is automated.

A simple graphical wizard helps first-time users make the right choices, with clear explanations for options like running workloads on master nodes, the container and service CIDR IP Blocks, privileged mode, and more (**Figure 17**).

Platform9 deliberately chose an upstream, open source version of Kubernetes for its deployment and operations. Being CNCF certified also means that applications written for and tested on Platform9 Managed Kubernetes will run on any other open source-certified Kubernetes, promoting portability and avoiding lock-in to proprietary distributions.

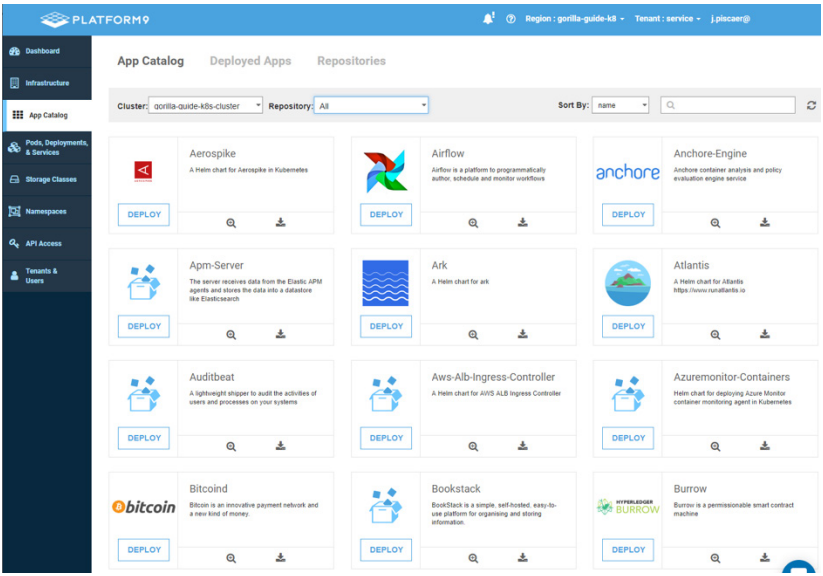


Figure 18: The Platform9 App Catalog.

Built-in Application Catalog

Platform9 Managed Kubernetes has a built-in application catalog (**Figure 18**) with hundreds of applications from the Kubernetes community. It comes with the Fission repository for easy deployment of Fission – Platform9's serverless framework – onto the Kubernetes platform.

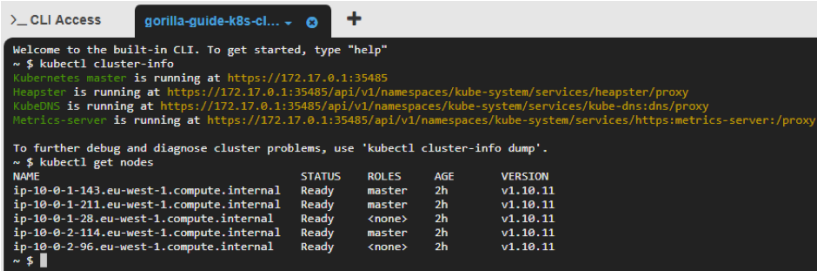


Figure 19: Platform9's Web CLI.

Web CLI

Platform9 Managed Kubernetes includes a Web-based CLI for simple access to command-line tools like kubectl and Helm. You can see an example of the CLI in **Figure 19**.

The CLI allows access to those tools in situations where there's no terminal available. It's also accessible via mobile devices. This is great for a quick fix of an issue, or for activities like checking cluster and node health.

CHAPTER 8

The Big Decision

This Gorilla Guide has taken you on a journey through the depths of Kubernetes, including its taxonomy, design decisions, and how to deploy a production-grade Kubernetes cluster.

There's no doubt that Kubernetes plays a vital role in the cloud-native landscape: the advantages to the developer workflow are undeniable, and its use cases are constantly expanding.

Still, deploying enterprise-grade Kubernetes is a daunting task, with many variables and complex tasks, requiring highly specialized knowledge. Using a managed solution like PMK makes deployment a simple, five-minute task.

It includes all the best practices that make Kubernetes enterprise-ready, like a highly-available multi-master control plane and integrated cluster monitoring. It allows managed operations like automatic cluster upgrades, integrated identity and access management, the Helm package manager and much more (See **Figure 20**).

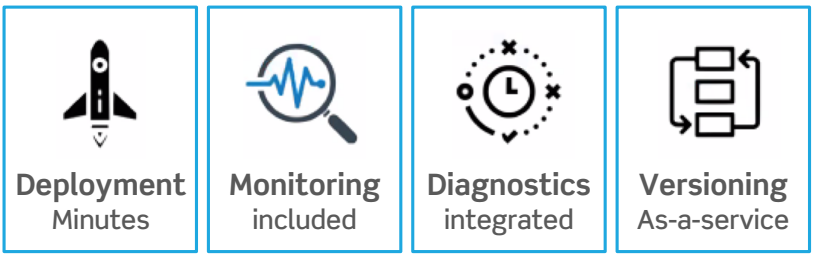


Figure 20: The advantages of a Platform9 solution.

Play in the Sandbox

It's now time to take one of those use cases, or come up with your own, to start with a managed Kubernetes deployment.

Platform9 offers a free sandbox with Kubernetes installed to get started quickly and painlessly. The sandbox includes a guided walk-through for SaaS-managed Kubernetes.

Alternatively, you can deploy your Kubernetes cluster on a public cloud such as Amazon Web Services by supplying user credentials for your public cloud environment.

If you're not using containers yet, the question is why? The benefits are massive, and an enterprise-grade management system like Kubernetes makes it easier than ever.

If you're ready to see what Kubernetes can do for you, visit Platform9[5] for more information, and start playing in your free sandbox.

[5] https://platform9.com/managed-kubernetes/