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 (https://platform9.com/blog/kubernetes-storage-dynamic-volumes-and-the-container-storage-interface/) have been developed to natively support stateful applications (https://platform9.com/resource/kubernetes-stateful-applications-in-hybrid-cloud-environments/).

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.

Action Plan for Architecting Kubernetes Deployments

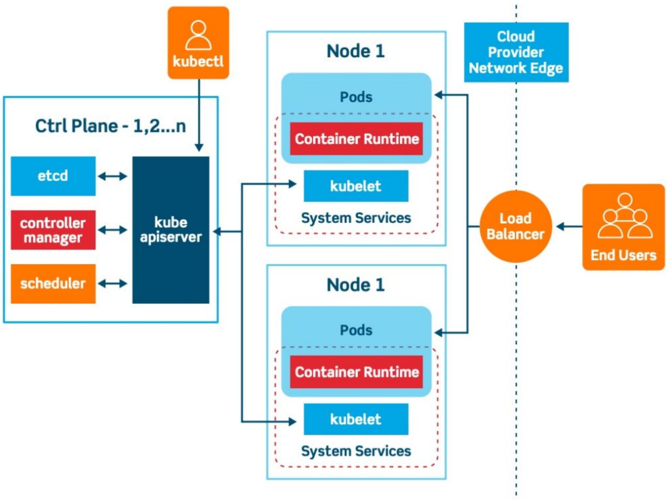
After reading this eBook, you'll will understand:

- Detailed overview of the chief architectural concepts
- How to distinguish the pros and cons of running Kubernetes on premises, in the cloud or on bare metal.
- How the key parts of the Kubernetes platform architecture-such as services, service meshes and runtimes fit together and interact with one another

Get the eBook (https://platform9.com/resource/action-plan-for-architecting-kubernetes-deployments/)

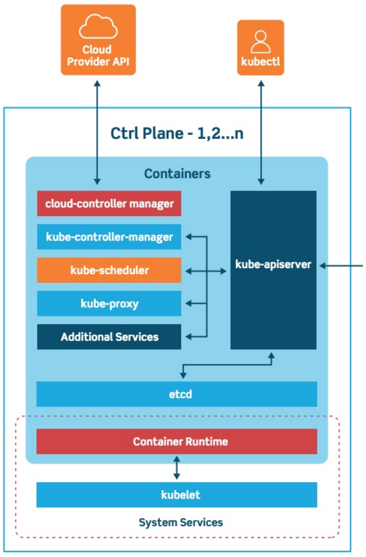
Kubernetes Architecture and Concepts

From a high level, a Kubernetes environment consists of a **control plane (master)**, a **distributed storage system** for keeping the cluster state consistent (**etcd** (https://platform9.com/blog/kubernetes-in-production-operating-etcd-with-etcdadm/)), and a number of **cluster nodes (Kubelets).**



(https://platform9.com/wp-content/uploads/2019/05/kubernetes-constructs-concepts-architecture.jpg)
Architectural overview of Kubernetes

Kubernetes Control Plane



(https://platform9.com/wp-content/uploads/2019/05/Kubernetes-control-plane-taxonomy.jpg)
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.

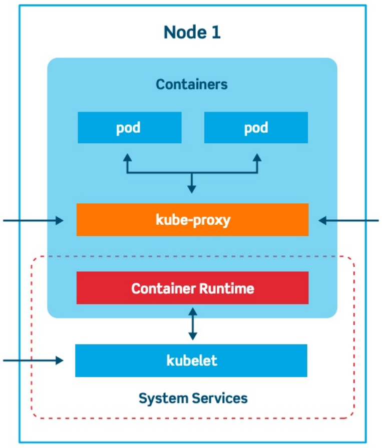
The **API Server** provides APIs to support lifecycle orchestration (scaling, updates, and so on) for different types of applications. It also acts as the gateway to the cluster, so the API server must be accessible by clients from outside the cluster. Clients authenticate via the API Server, and also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



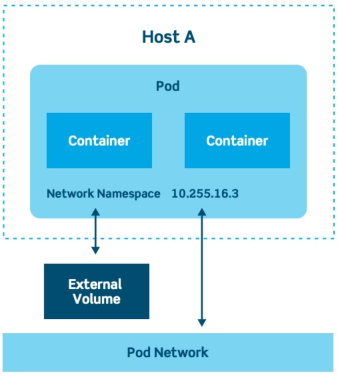
Kubernetes node taxonomy

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.



(https://platform9.com/wp-content/uploads/2019/05/kubernetes-Pod-architecture.jpg)
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.

Action Plan for Architecting Kubernetes Deployments

After reading this eBook, you'll will understand:

- Detailed overview of the chief architectural concepts
- How to distinguish the pros and cons of running Kubernetes on premises, in the cloud or on bare metal.
- How the key parts of the Kubernetes platform architecture-such as services, service meshes and runtimes fit together and interact with one another

Get the eBook (https://platform9.com/resource/action-plan-for-architecting-kubernetes-deployments/)

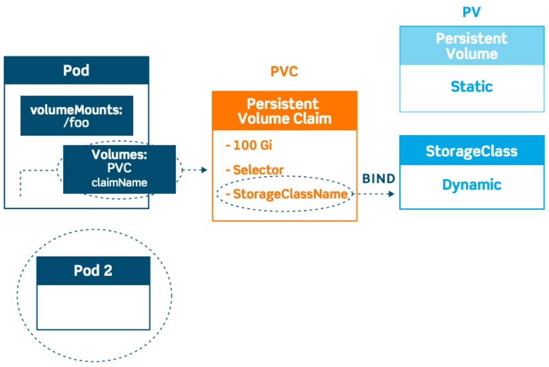
Kubernetes Networking

Networking Kubernetes has a distinctive networking model for cluster-wide, podto-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, v cl

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

Persistent Storage in Kubernetes



(https://platform9.com/wp-content/uploads/2019/05/kubernetes-Persistent-volumes-claims-storage-classes.jpg)
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.

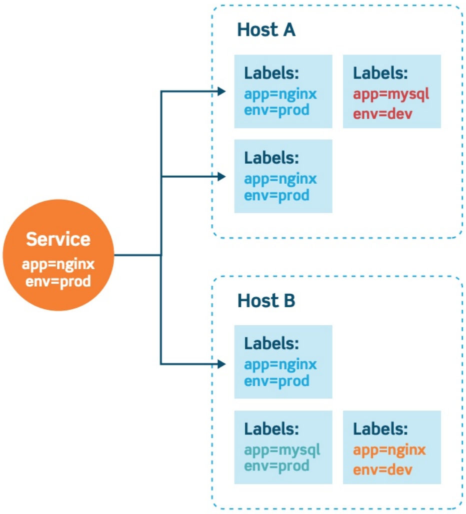
There are a few special kinds, like configMap and Secrets, used for injecting information stored within Kubernetes into the pod or emptyDir, commonly used as scratch space.

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


Host A
Labels: app=nginx env=prod
Labels: app=mysql env=dev
Labels: app=nginx env=prod
Service app=nginx env=prod
Host B
Labels: app=nginx env=prod
Labels: app=mysql env=prod
Labels: app=nginx env=dev

(https://platform9.com/wp-content/uploads/2019/05/kubernetes-service-discovery.jpg)
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:

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.

Kubernetes Tooling and Clients:

Here are the basic tools you should know:

1. **Kubeadm** (https://github.com/kubernetes/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).
2. **Kubectl** (https://github.com/kubernetes/kubectl) is a tool for interacting with your existing cluster.
3. **Minikube** (https://github.com/kubernetes/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.

If you're ready to get started, you can deploy a free Kubernetes cluster on AWS or on-premises under five minutes: https://platform9.com/signup/ (https://platform9.com/signup/)

Action Plan for Architecting Kubernetes Deployments

After reading this eBook, you'll will understand:

- Detailed overview of the chief architectural concepts
- How to distinguish the pros and cons of running Kubernetes on premises, in the cloud or on bare metal.
- How the key parts of the Kubernetes platform architecture-such as services, service meshes and runtimes fit together and interact with one another

Get the eBook (https://platform9.com/resource/action-plan-for-architecting-kubernetes-deployments/)

Learn More About Enterprise Kubernetes, Kubernetes Best Practices, and more

Kubernetes is notoriously difficult to deploy and operate at scale — particularly for enterprises managing both on-premises and public cloud infrastructure. Numerous Kubernetes solutions and products have emerged in the industry (from both startups and established traditional vendors) aimed to solve some of the challenges around Kubernetes. The space has become crowded, and difficult for organizations to navigate and compare the various offerings. Our additional articles below can help you learn more about Kubernetes best practices.

Kubernetes On-premises: Why, and How

In this blog post you'll learn about,

- Challenges with Kubernetes On-premises
- Opportunities and benefits for Kubernetes on-prem
- Considerations for running DIY Kubernetes on-prem
- Infrastructure requirements and best practices for on-prem DIY Kubernetes implementation
- And more

Read more: Kubernetes On-premises: Why, and How (https://platform9.com/blog/kubernetes-on-premises-why-and-how

---

Kubernetes as an On-Premises "Operating System"

To drive home the message about the reasons why you might choose to run Kubernetes on-premises, let's now examine how Kubernetes can function not just as another tool running in your data center, but as a way to build an "operating system" for your entire on-premises infrastructure.

Read more: Kubernetes as an On-Premises "Operating System" (https://platform9.com/blog/kubernetes-as-a-cloud-native-operating-system-on-premises-too/)

Kubernetes Resource Limits: Kubernetes Capacity Planning

Capacity planning is a critical step in successfully building and deploying a stable and cost-effective infrastructure. The need for proper resource planning is amplified within a Kubernetes cluster, as it does hard checks and will kill and move workloads around without hesitation and based on nothing but current resource usage.

This article will highlight areas that are important to consider, such as: how many DaemonSets are deployed, if a service mesh is involved, and if quotas are being actively used. Focusing on these areas when capacity planning makes it much easier to calculate the minimum requirements for a cluster that will allow everything to run.

Read more: Kubernetes Resource Limits: Kubernetes Capacity Planning (https://platform9.com/blog/kubernetes-capacity-planning/)

Kubernetes Cluster Sizing – How Large Should a Kubernetes Cluster Be?

When it comes to Kubernetes clusters, size matters. The number of nodes in your cluster plays an important role in determining the overall availability and performance of your workloads. So does the number of namespaces, in a way.

This does not mean, however, that bigger is always better. A Kubernetes cluster sizing strategy that aims to maximize node count will not always deliver the best results – certainly not from a cost perspective, and perhaps not from an overall availability or performance perspective, either. And maximizing namespaces is hardly ever a smart strategy.

Instead, calculating the number of nodes to include in a cluster requires careful consideration of a variety of factors. Keep reading for an overview – if not a precise recommendation on how large your cluster should be, because only you can decide that for yourself.

Read more: Kubernetes Cluster Sizing – How Large Should a Kubernetes Cluster Be? (https://platform9.com/blog/kubernetes-cluster-sizing-how-large-should-a-kubernetes-cluster-be/)

Kubernetes for CI/CD at Scale

one of the main use cases of Kubernetes is to run Continuous Integration or Continuous Delivery (CI/CD) pipelines. That is, we deploy a unique instance of a CI/CD container that will monitor a code version control system, so whenever we push to that repository, the container will run pipeline steps. The end goal is to achieve a 'true or false' status. True, if the commit passes the various tests in the Integration phase; false, if it does not.

In this blog post you'll learn about,

- CI/CD platforms for Kubernetes
- How to Install Jenkins on Kubernetes
- Scaling CI/CD Jenkins Pipelines with Kubernetes
- Best Practices to use Kubernetes for CI/CD at scale

Read more: Kubernetes for CI/CD at scale (https://platform9.com/blog/kubernetes-for-ci-cd-at-scale/)

Kubernetes Security: Architecture & Best Practices

When it comes to security, there is a lot that Kubernetes does. There is also a lot that it doesn't do.

To secure Kubernetes effectively for real-world deployment, you must understand which built-in security features Kubernetes offers and which it doesn't, and how to leverage Kubernetes's security capabilities at scale.

In this blog post you'll learn Kubernetes's security architecture and best practices for securing production Kubernetes deployments.

Read more: Kubernetes Security: Architecture & Best Practices (https://platform9.com/blog/kubernetes-security-what-and-what-not-to-expect/)

There's More:

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

In Partnership with PLATFORM9

ENTERING THE JUNGLE

(https://platform9.com/resource/the-gorilla-guide-to-kubernetes-in-the-enterprise/)On the next posts we'll dive deeper into the Kubernetes deployments (https://platform9.com/blog/kubernetes-enterprise-chapter vp infrastructure, Kubernetes use cases (https://platform9.com/blog/kubernetes-use-cases/), and best practices for ope ale.

Are you already in production or just getting started?

Can't wait?