



Architecting a Kubernetes Deployment Tailored to Your Goals

Platform9 Kubernetes Journey eBook series

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

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

Table of Contents

Introduction	3
Kubernetes On-Premises: What, Why and How	4
The challenges of on-prem Kubernetes.....	4
Opportunities and benefits for Kubernetes on-prem:.....	4
Considerations for running DIY Kubernetes on-prem:	5
Infrastructure requirements and best practices for on-prem DIY Kubernetes implementation:.....	5
Kubernetes as an On-Premises “Operating System”	7
Releasing Software	7
Kubernetes applications on-prem	7
Kubernetes as an Operating System.....	8
Challenges.....	8
Solutions	10
Kubernetes on Bare Metal	11
Why Run Kubernetes on Bare Metal?	11
Challenges with Kubernetes on Bare Metal	11
Start your engines.....	11
Installing Docker on Linux	12
DIY Kubernetes.....	12
Managing it better	12
The bare necessities.....	12
Kubernetes Cloud Services - GKE vs. EKS vs. AKS	14
Amazon Web Services’ Elastic Kubernetes Services	14
Google Cloud Platform Kubernetes Engine.....	14
Microsoft Azure Kubernetes Service	14
Points of Comparison	15
Kubernetes Federation.....	17
What is Kubernetes Federation?	17
Multi-Tenant vs. Multi-Cluster vs. Federation	17
History of Kubernetes Federation	18
The basic structure of Kubernetes Federation	18
Setting up Kubernetes Federation	18
Configuration Flow within a Kubernetes Federation	19
Next Steps.....	21
Kata Containers, Docker and Kubernetes: How They All Fit Together	22
What are Kata containers?	22
Kata vs. Docker (and other container runtimes).....	23
Kata and Kubernetes.....	23
Kubernetes Service Discovery	25
Defining and deploying Services.....	25
Common Service Discovery Gotchas	31
Services Meshes and Kubernetes.....	33
The Rise of the Service Mesh.....	33
What is Service Mesh	33

Service Mesh Options for Kubernetes:.....	34
Consul Connect.....	34
Istio.....	34
Linkerd.....	35
Comparison of Istio, Linkerd and Consul Connect for Kubernetes Service Mesh.....	35
Common use cases to take advantage of Service Mesh today.....	39
Scaling Kubernetes for Modern Software Delivery.....	41
CI/CD platforms for Kubernetes.....	41
Use Case: Installing Jenkins.....	42
Use Case: Scaling CI/CD Jenkins Pipelines.....	43
Use of Autoscaling.....	47
Best Practices to use Kubernetes for CI/CD at scale.....	48
Conclusion.....	49

Introduction

Architecting a Kubernetes deployment is no mean feat – even for experienced IT architects. The main reason why is that, compared to most other types of IT platforms, Kubernetes is simply more complex. Not only does it consist of more than a half-dozen different components, but there are also a multitude of different ways to set up and deploy those various components across a set of servers.

As a result, designing the best Kubernetes architecture for a given set of priorities means considering where different parts of the cluster are hosted, how many clusters to deploy, how to segment workloads within clusters, how to connect clusters to storage and how to design networking layers – to name just some of the factors to weigh.

The best Kubernetes architecture for your organization depends on your needs and goals, of course. But in order to provide some guidance, this eBook offers a detailed overview of the chief architectural concepts in Kubernetes. It dives into the pros and cons of running Kubernetes on-premises, in the cloud and on bare metal. It also discusses how key parts of the Kubernetes platform architecture – such as services, service meshes and runtimes – fit together and interact with one another.

If you're designing a Kubernetes strategy for the first time, this eBook is for you. It's also for you if you already have a cluster (or several) up and running, but are seeking tips on how to optimize characteristics such as service configuration, or how to grow your cluster to keep pace with your application delivery needs.

Kubernetes On-Premises: What, Why and How

Kubernetes is often described as a cloud-native technology, and it certainly qualifies as one. However, the cloud-native concept does not exclude the use of on-premises infrastructure in cases where it makes sense. Sometimes, there are advantages to running Kubernetes on-premises.

This chapter explains those advantages, as well as the challenges of running Kubernetes on-prem, and best practices for addressing them.

The challenges of on-prem Kubernetes

Let's start with the challenges.

As opposed to using Kubernetes on AWS or Azure – where your public cloud provider essentially hides all the complexities from you – running Kubernetes on-prem means you're on your own for managing these complexities ... including etcd, load balancing, availability, auto-scaling, networking, roll-back on faulty deployments, persistent storage, and more.

In addition to building services to deal with the aforementioned complexities that public clouds generally solve for you, deploying Kubernetes on-prem in a DIY fashion also involves a considerable amount of core code modification, including:

- Changing hostnames by modifying the etcd/host. This is done because the default first interface is connected to a non-routable host-only network.
- Modifying host file configurations so hosts can communicate with each other by hostname.
- Verify each host has a unique MAC and product_uid.
- Configuring OS-level settings by enabling br_netfilter Kernel Module and disabling SWAP.

All of these concerns apply to bare metal deployment managed by hand. If the on-prem Kubernetes uses OpenStack/vSphere with software networking - where IPs are managed by the cloud platform - to manage the infrastructure as a private cloud, then you would use these to manage the infrastructure for the worker VMs.

But even in a bare-metal cluster, the worker nodes can be programmed to talk to a DNS system to get an IP which lives for their lifetime. kubeadm provides a single command to add/remove node from cluster given ip and ssh access

What's more, keep in mind that you'll need to upgrade your clusters roughly every 3 months when a new upstream version of Kubernetes is released (you'd typically use kubeadm for that), as well as address the complexities of managing Storage and monitoring of the health of your clusters in your on-prem Kubernetes environment.

Opportunities and benefits for Kubernetes on-prem:

Why then do organizations choose the path of running Kubernetes in their own data centers, compared to the relative "cake-walk" with public cloud providers?

Some organizations simply can't use the public cloud, as they are bound by stringent regulations related to compliance and data privacy issues. But more importantly, enterprises are looking to take advantage of Kubernetes leveraging their existing data centers to transform their business and be able to modernize their applications for cloud-native - while improving infrastructure utilization and saving costs.

Considerations for running DIY Kubernetes on-prem:

Kubernetes enables users to run it on on-premises infrastructure, but not in a straightforward way like you would hope. You can “repurpose” your environment to integrate with Kubernetes – using VMs, hypervisors or creating your own cluster from scratch on bare metal – but there’s no escaping the requirement for a deep understanding of the associated servers, storage systems, and networking infrastructure.

Since Kubernetes is rather new and expertise can be hard to find, the CNCF has introduced certifications like CKA (Kubernetes Administrator) and CKAD (Kubernetes Application Developer) that can be achieved by passing a test. While employing CNCF certified administrators is a great option, not everyone can hire new staff.

Also, you should plan for the fact that DIY projects in the enterprise often balloon to months-long (and even years-long) projects trying to tame and effectively manage the open source components at scale - accumulating costs and delaying time to market. For a [Managed Kubernetes service](#) that works on your existing infrastructure, check out our [PMK solution](#)

*Keep in mind that when running Kubernetes on your own in your own on-prem data centers, you'll need to manage all the **storage integrations, Load balancer, DNS, security management, Container registry, and monitoring infrastructure** yourself.*

In addition, each one of these components - from storage to networking - needs **its own monitoring and alerting system**, and you’ll need to set up your internal processes to monitor, troubleshoot and fix any common issues that might arise in these related services to ensure the health of your environments.

Infrastructure requirements and best practices for on-prem DIY Kubernetes implementation:

- Getting back to infrastructure, Kubernetes can technically run on one server, but ideally, it needs at least three: one for all the master components which include all the control plane components like the kube-apiserver, etcd, kube-scheduler and kube-controller-manager, and two for the worker nodes where you run kubelet.
- While master components can run on any machine, best practice dictates using a separate server for master and not running any user containers on this machine.
- One key feature of Kubernetes is the ability to recover from failures without losing data. It does this with a ‘political’ system of leaders, elections, and terms - referred to as Quorum - which requires “good” hardware to properly fulfill this capability. To be both available and recoverable, best practice dictates allotting three nodes with 2GB RAM and 8GB SSD each to this task, with three being the bare minimum and seven the maximum
- An SSD is recommended here since etcd writes to disk, and the smallest delay adversely affects performance. Lastly, always have an odd number of cluster members so a majority can be reached.
- Kubelet is a tool that along with kube-proxy, runs on each worker node, making sure all containers are running and meeting network regulations.
- You also want to run kubeadm on the master. Kubeadm is an installation tool that uses kubeadm init and kubeadm join as best practices to create clusters.
- For production environments, you would need a dedicated HAProxy load balancer node, as well as a client machine to run automation.
- It’s also a good idea to get a lot more power than what Kubernetes’ minimum requirements call for. Modern Kubernetes servers typically feature two CPUs with 32 cores each, 2TB of error-correcting RAM and at least four SSDs, eight SATA SSDs, and a couple of 10G network cards.
- It is best practice to run your clusters in a multi-master fashion in Production - to ensure high

availability and resiliency of the master components themselves. This means you'll need at least 3 Master nodes (an odd number, to ensure quorum). You'll further need to monitor the master(s) and fixing any issues in case one of the replicas is down.

- HA etcd: Etcd is an open-source distributed key-value store, and the persistent storage for Kubernetes. Kubernetes uses etcd to store all cluster-related data. This includes all the information that exists on your pods, nodes, and cluster. Accounting for this store is mission-critical, to say the least, since it's the last line of defense in case of cluster failure.
- Managing highly available, secured etcd clusters for large-scale production deployments is one of the key operational complexities you need to handle when managing Kubernetes on your own infrastructure. For production use, where availability and redundancy are important factors, running etcd as a cluster is critical.
- Bringing up a secure etcd cluster – particularly on-premises – is difficult. It involves downloading the right binaries, writing the initial cluster configuration on each etcd node, setting and bringing up etcd. This is in addition to configuring the certificate authority and certificates for secure connections. For an easier way to run etcd cluster on-prem, check out the open-source etcdadm tool.

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.

We are living in an age of Cloud-Native applications. The rise of microservices and the exponential growth in the scale of applications are fundamentally transforming modern software development and deployment. While this transformation is occurring first and foremost for applications that are born in the cloud, these days nearly any application is distributed, and requires auto-scaling and auto-failover and redundancy.

The evolution in modern software delivery is affecting the way software is being shipped – by both those who deliver a SaaS-based application that is accessed remotely, over the internet, as well as by traditional software vendors who ship software that needs to be installed locally. Let us explore how.

Releasing Software

While we do live in a SaaS, on-demand world, still – not all software today is being accessed “in the cloud”. While we no longer send CD-ROMs to customers with the installation file burnt on them, much of the software that we use today is still being delivered – and will continue to be delivered for a while – as an executable that is installed locally. We still use Visual Studio code, a myriad of desktop applications, organizations often prefer to have software installed internally, behind their firewall, and many security-conscious companies and government agencies use Github Enterprise instead of github.com. In addition, there are even more use-cases for shipping software for the IoT world where edges are not in a cloud environment.

Kubernetes applications on-prem

Applications can be delivered and consumed either as a SaaS product, or as a local installation – and this is true for both traditional 3-tier applications or desktop services, as well as for the newer container-based applications.

Ever since we released the first version of our [Managed Kubernetes](#) offering, we’ve seen many interesting companies using Kubernetes in a variety of ways – from mobile applications, video processing to AI. While some of these companies do run their Kubernetes infrastructure on the public cloud, many large enterprises also have Kubernetes-based applications that are for internal consumption, running on internal, on-premises hardware.

An interesting use case we’ve seen in this regard is from software providers that are adopting Kubernetes as the underlying platform for deploying their applications.

We primarily see two types of companies in this space:

1. SaaS Providers – going on-prem

These are companies offering an application that is consumed by end-users as a SaaS/cloud service. Most often, these applications are hosted in the cloud – either public or private – and are accessed by users via a web-portal.

Since many of these applications are cloud-native, we see companies that have chosen Kubernetes as the

underlying platform for the app, primarily to enable portability (more on this later).

Some of these SaaS providers deliver products that are used by Development and Operations teams in large enterprises (think GitHub Enterprise, or other software delivery, security, and data processing solutions.) As they grow their customer base to include fortune 1,000 companies and government agencies, often the requirement arise to access the service as a private SaaS offering, installed behind the firewall and operated internally.

Since these SaaS vendors already use Kubernetes as the backend for their applications, it is easier to use the same technology for running their on-premises version as well (rather than having to re-architect the application, have a separate product line, or compromise some of the key benefits that have led them to choose Kubernetes in the first place.)

So how do you run Kubernetes-based applications on-prem, as a private container offering?

2. Traditional Software Vendors – going cloud

These vendors often did not start in the cloud, and have been successful primarily with providing customers – particularly large enterprises – software that is installed either on local machines or on-premises data centers. They are now looking to expand their market opportunity and provide their software as a SaaS offering. Many of them are looking now at containers and Kubernetes as the underlying platform to enable them to modernize their offering and cloudifying it – so it is “cloud-native” (or “cloud-aware”.) This, too, prompts them to look for Kubernetes offering across different infrastructure – so that they can continue to develop and release their product both as a SaaS solution, as well as a local installation.

Kubernetes as an Operating System

Kubernetes has emerged as the platform of choice for deploying cloud-native applications. In essence, Kubernetes is emerging as an Operating System (not in the classical sense, but from the perspective of a distributed, cloud-native application.)

It is easy to see why. Kubernetes provides many of the features that are critical for running a cloud-native application:

- High Availability: Automatically start another instance if a service instance fails
- Granular, infinite, scalability: A group of services run behind a load-balancer, giving the ability to scale each service individually.
- Rolling Upgrade: You can upgrade each service independently of others, with rolling upgrade and rollback.
- Discovery: Discover other services through name service.
- Portability: Thanks to the containers and the availability of many ‘Kubernetes-as-a-Service’ platforms, Kubernetes-based applications are portable across any environment or infrastructure provider that uses Kubernetes.

While developers certainly care about these capabilities, you can see how crucial these are – particularly the last point – to IT Operations.

Challenges

It is easy to assume that Kubernetes by itself provides all the necessary features and services that would allow you to easily, and consistently, use it both in the cloud as well as on your own, on-premises infrastructure. The reality is far from it. There are critical gaps, specifically when you consider that majority of enterprises operate in a hybrid/multi-cloud environment and need to support, on average, 5 different clouds or several datacenters- in the cloud and internal.

Public Cloud Providers and Consistency

Different cloud providers provide their own version of Kubernetes. When you have different vendors providing a service (even with Open source software like Kubernetes), the consistency between different providers becomes questionable. For example, Amazon EKS supports Kubernetes version 1.10.3, Google's GKE supports the last 3 releases and Microsoft Azure still seems to be on version 1.9.

In addition to the different Kubernetes versions, the underlying OS (kernel version) also differs. This may or may not pose a problem to most application, but worth considering, especially when security and compliance are important.

Perhaps the biggest hurdle is the tight coupling and dependency of various cloud provider between Kubernetes and other services – like networking, storage, monitoring, authentication, and more. These related services are different, and are integrated differently, from cloud to cloud. So, for example, a Kubernetes application on AWS would integrate with certain AWS-specific services – for example it uses cloud-watch monitoring or IAM for authentication – that are different from comparable services that are available on a different cloud provider.

Private Clouds

While there is a lot of buzz around Kubernetes solutions provided by public cloud provider, [KubeCon survey results](#) show there are also real use cases and adoption already happening in on-prem data centers. From CI/CD to security-sensitive applications, bare-metal occupies > 50% of the container workload.

While there are solutions available to create a single Kubernetes cluster on a private cloud, there is a clear lack of solutions that can provide Multi-Clusters or a simple Kubernetes-as-a-Service (KAAS) on these bare metal servers, that are typically inside a private data-center.

Managed Edge Cloud

With the advent of smart devices like Alexa, Nest, Ring, IoT becomes ubiquitous in our always-connected world. Similarly, Point-of-Sale (POS) devices in retail store present a use case, not unlike that of IoT – of silo'd, small computing footprint at the edge. As Kubernetes becomes the de-facto Operating System of choice, there is a need for managing these 'edges'. See the following blog from [Chik-Fil-A describing their experience with running such clusters](#) at their local stores.

As you can imagine the challenges here are different, mostly around auto-deployment of clusters, recovery, backup in case of failure and remote management for troubleshooting and upgrades.

Un-Managed Edges

While managed-edges are great for retail use cases, there is a whole class of use cases where Kubernetes is being used as an underlying OS for deploying distributed applications that are installed locally – for example, Kubernetes-based applications that are behind a firewall, etc... The challenges here are similar to Edge use cases, requiring that a Kubernetes cluster comes up:

- Without any internet connection (i.e cannot download images dynamically)
- Almost without user interaction: so that the auto-deploy of a cluster on a fixed set of nodes is done automatically
- With high availability built-in: requires multi-master support for failover.
- With backup and recovery built-in: That it already has the tools and processes for automatically backing up the state of the cluster and recovering from it.
- In an "OEM-able" state: so that the end user only sees and interacts with the application itself, while

the underlying Kubernetes infrastructure is completely invisible.

Solutions

For the reasons outlined above, many of us have already made a decision to use Kubernetes as the Operating System- for both our SaaS/cloud applications, as well as for our on-premises, locally installed software.

To accelerating development, simplifying updates and ongoing operations, and reducing costs – you want to limit your software/management variations. To ensure success, you first need to make sure you have consistency and interoperability of the application code and management processes between the different types of infrastructure. So that regardless of where the app is running – private cloud, public cloud or edge appliances – it is always developed, deployed and operated in the same way.

You need to choose a solution that would help you:

- Ensure fidelity across different infrastructure: This means one API to deploy and maintain clusters across different infrastructure including cloud providers.
- Give you the control you need: Be it the choice of network or storage provider, or simply using SAML2 for authentication, you should be able to integrate with the processes and tools of your choice.
- Managed Service: You want to focus on your application and not on the infrastructure management – so choose a solution that manages your clusters footprint across different infrastructure for you- including version upgrades, patches, performance optimization, and more.
- Un-Managed: Lastly since you are releasing software to your customers, pick a solution that will let you OEM the solution so it can be delivered as a self-sufficient, 'boxed' platform on an environment or edge appliance that may not have any internet connectivity.

Kubernetes on Bare Metal

Just as on-premises may not be the first deployment strategy that comes to mind for a cloud-native technology like Kubernetes, running Kubernetes on bare-metal servers may also seem outdated. After all, the era when most applications ran directly on bare metal ended two decades ago.

Here again, however, in some cases this approach offers critical advantages. This chapter examines why you might benefit from hosting Kubernetes clusters on bare metal in certain circumstances.

Why Run Kubernetes on Bare Metal?

When deploying on VMs, you incur a lot of overhead with all the “wrappers” around each and every container. By bypassing the hypervisor, Kubernetes on bare metal improves:

- Resource utilization: Kubernetes on bare metal takes up on average about a third of the resources, compared to VMs
- Density: you can deploy and manage more containers on the same server than you can with VMs
- Costs: Improved utilization and density, in turn, reduce infrastructure costs.
- Latency: Kubernetes on bare metal sees less latency than on VMs, making it popular for Edge use cases.

Challenges with Kubernetes on Bare Metal

The biggest challenge is that if you already used to consume Kubernetes through one of the services offered by the public cloud providers, then much of the complexities of deploying and operating Kubernetes infrastructure at scale behind the scenes have been obfuscated from you. You’ll need to buckle up to be able to take on these challenges yourself now. These include day1 deployment - anything from installing Kubernetes, manual IP allocation, load balancing, auto-scaling, etcd, persistent storage and volume management, and more - in addition to all the data center management work- such as installing operating systems, etc. Then, you’ll need the skills and the resources in place to ensure Day2 operations at scale - infrastructure HA, monitoring, upgrades, security patching, troubleshooting, and more.

This is because, unlike in virtualized environments where Kubernetes creates your VMs and restarts them automatically when they fail, bare metal entails manually configuring every node, as well as a manual restart/repair every time one fails. Additionally, bare metal makes it your prerogative to stay up-to-date with the fast and frequent community releases of Kubernetes.

You could also consider first implementing a solution that can help you enable bare-metal as a service- so you could manage this infrastructure as if it were VMs, and then deploy Kubernetes on top of that. Look at [Ironic](#), [Packet](#), or other solutions in the space to evaluate your options.

Start your engines

Before you get down to installing an orchestrator on bare metal, you need something that powers containers

– aka a container engine. This is where Docker comes in; and while Kubernetes undoubtedly abstracts infrastructure and automates the process of scaling and managing containers, it's Docker that makes it all possible on a single operating system.

Note: this isn't to say Kubernetes can't work with a different container engine like rkt, for example; it's just that there's a lot of experience that's already been encapsulated in the Docker-Kubernetes integration and it would be a waste to go another way right now.

So, the first step is to install Docker. Keep in mind that this process requires both basic Linux/Unix skills and an understanding of Kubernetes terminology, and you probably want to start by adding the official Docker GPG key to your system.

Installing Docker on Linux

This step is to make sure your downloads are valid and authentic. It is done using the curl command and the apt-key command that's used to manage the list of keys used to authenticate packages.

<code>

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

</code>

The above example is for Ubuntu-based systems. If you're using Red Hat or CentOS, you're going to have to change the URL as well as use the yum command instead of apt-key.

DIY Kubernetes

While the next logical step would be to similarly install Kubernetes repositories and then Kubernetes itself, Kubernetes 1.8 and above don't work with swap enabled for Linux, so you need to disable that first with `sudo swapoff -a`.

Once that's done, add Google's repository keys for Ubuntu the same way you did for Docker with the curl command.

Then, use apt-get to install Kubernetes repositories, followed by your 3 core components, kubelet, kubeadm and Kubernetes-cni. Only after all the previous steps have been successfully completed, is the kubeadm init command finally used to initialize your cluster with kubeadm.

Managing it better

This is a pretty basic example, however, and it becomes more complex in an enterprise scenario. The good news is that there are a number of tools and platforms specially designed to solve the complexities involved with installing Kubernetes on bare metal.

Platform9 uses OpenStack Ironic to offer users what is now being called "Bare-metal-as-a-Service."

The bare necessities

If you really want to do it the hard way and brave the bare metal yourself, remember that this means you're all on your own and solely responsible for everything, including manually configuring each worker node

(using kubeadm join tokens). You also have to create users with administrative privileges to manage the cluster, as well as provision for storage, monitoring, HA, networking and the like.

While deploying Kubernetes on bare metal still seems like a difficult proposition for a beginner, the applications in modern edge technology, as well as with machine learning algorithms, make it a valuable skill to learn. In cases where bare metal is the business mandate, it helps to play to the strengths of bare metal and find ways to minimize its complexities. There are solutions available that strike a great balance between having full control over bare metal servers while still enjoying the simplicity of the cloud.

Kubernetes Cloud Services - GKE vs. EKS vs. AKS

Now that we've discussed the case for Kubernetes on bare metal and in the cloud, let's look at the other side of the coin: Running Kubernetes in the cloud.

Specifically, this chapter examines the Kubernetes offerings from each of the "Big Three" public cloud providers: Amazon Elastic Kubernetes Service (EKS) from Amazon Web Services (AWS), Azure Kubernetes Service (AKS) from Microsoft Azure, and Google Kubernetes Engine (GKE) from Google Cloud.

When selecting which hosted Kubernetes offering is best for you, you have to look beyond just price. Additional considerations like scalability, standardization, update frequency, recovery, and whether or not a service mesh is included are all critical to making the best decision.

Amazon Web Services' Elastic Kubernetes Services

[Elastic Kubernetes Services \(EKS\)](#) is one of the managed container offerings that are available on AWS, and is the least integrated offering as far as interacting with other AWS services like CI/CD pipelines. Elastic Container Service (ECS) that preceded EKS and Fargate are more preferred offerings within the AWS ecosystem; but as EKS is based on Kubernetes, most everything you will need to connect to it will work, as the industry is moving towards supporting Kubernetes as a deployment target for applications and data source for logs and application performance metrics.

EKS is a good choice if you already have a large AWS footprint and are either experimenting with Kubernetes or want to migrate workloads from Kubernetes on other clouds.

Google Cloud Platform Kubernetes Engine

The Google Cloud Platform (GCP) entry in the hosted Kubernetes space is [Google Kubernetes Engine \(GKE\)](#). GKE is the most resilient and well-rounded Kubernetes offering when compared to AKS and EKS. It has the highest SLA for uptime (see table below) and is the only one with a marketplace to deploy applications from. It has support for the Istio service mesh, and gvisor for an extra layer of security between running containers. It also has an on-premises offering in development as part of [Google's Anthos offering](#) for hybrid/multi cloud environments on dedicated hardware.

Microsoft Azure Kubernetes Service

[AKS](#) is the Microsoft developed Kubernetes offering that runs on Azure Public Cloud, Government Cloud, and even Azure Stack for on-premises. It is deeply integrated with the rest of the Microsoft cloud services and has managed worker nodes (unlike EKS). Like most things Microsoft does, it is definitely best-of-breed when it comes to seamless integration with their cross-platform development tools, including VS Code and DevOps (formerly Visual Studio Team Services).

If you have an established relationship with Microsoft, and no strong preference for another cloud, then AKS will fit your needs.

Note: Microsoft also offers Azure Red Hat OpenShift (ARO) as a managed tier-1 service on Azure. However, the ARO service abstracts Kubernetes functionality away from the cloud user, and is not a real Kubernetes offering, but merely embeds Kubernetes for its own use.

Points of Comparison

- **Pricing**
 - All services charge standard compute rates for worker nodes and they are roughly competitive. AWS EKS is the only one to charge for the control plane at a cost of \$0.20/hour.
- **Kubernetes Release**
 - As of October 2019, Kubernetes has released version 1.16. GKE, EKS, and AKS have 1.14 as their regular stable release, though GKE has “rapid” stream that runs 1.15.
- **Global Availability**
 - All three providers have their offering available in most regions globally. A notable exception is that EKS is not available in the AWS government cloud; AKS, however, is in at least one Azure government cloud. (For reference, Google has no government clouds.)
- **Upgrades**
 - AKS and GKE will both handle security patches on the control plane and nodes; the nodes in EKS are not managed in the same way. You can enable automatic upgrades to the control plane on GKE, and all three offer on-demand upgrades of the control plane to newer versions.
- **Node Groups**
 - EKS and GKE both allow nodes in a cluster to be grouped so applications can be targeted at specific nodes, and the entire cluster does not need to have more expensive nodes like GPU enabled ones. AKS recommends separate clusters in these scenarios.
- **Bare Metal Nodes**
 - Only EKS allows the use of bare metal nodes. GKE and AKS can only use virtual machines. It is fair to note that EKS defaults to virtual machines, as bare metal are much more expensive.
- **Management via CLI**
 - GKE and AKS have full support to manage all aspects of their Kubernetes clusters via their CLI tool. EKS has partial support via the CLI, which makes some automation harder to enact without 3rd party tools.
- **Resource Monitoring**
 - GKE (StackDriver) and AKS (Azure Monitor) have native support for resource monitoring within their Kubernetes cluster, with StackDriver having far more capabilities. EKS requires the use of a third party product and recommends Prometheus like any other non-hosted Kubernetes offering would use.

In addition to these comparison points, there are many more that may be of interest. Below is a table that summarizes the ones already mentioned, and includes a few more.

	AKS	EKS	GKE
Year Released	2017	2018	2014
Kubernetes Versions	1.12, 1.13, 1.14	1.12, 1.13, 1.14	1.13, 1.14, 1.15
Global Availability	Yes + Government	Yes	Yes
SLA	99.5%	99.9%	99.5% (zone) 99.95% (regional)
Control Plane Cost	Free	\$0.20/hr	Free
Control Plane Upgrades	On-Demand	On-Demand	Automation and On-Demand
Worker Upgrades	Yes	No	Yes

Bare Metal Nodes	No	Yes	No
GPU Nodes	Yes	Yes	Yes
Linux Containers	Yes	Yes	Yes
Windows Containers	Yes	Yes	Yes
Resource Monitoring	Yes (Azure Monitor)	3rd Party	Yes (StackDriver)
Nodes per Cluster	500	100	5000
App Secret Encryption	No	No	Yes
RBAC	Yes	Yes	Yes
Network Policies	Beta	3rd Party	Yes
KNative Support	Yes	No	Yes
Load Balancer	Yes	Yes	Yes
Global Load Balancing	Yes (Traffic Manager)	Yes (Manually)	Yes
Service Mesh	No (In Development)	Yes (App Mesh)	Beta (Istio)
DNS Support	Integrated (Free)	Integrated (\$)	No
Marketplace	No	No	Yes
Compliance	HIPAA, SOC, ISO, PCI DSS	HIPAA, SOC, ISO, PCI DSS	HIPAA, SOC, ISO, PCI DSS
FedRAMP	High	High	Moderate
Documentation	Extensive official documentation and a strong community	Weak but complete, and a strong community	Not very thorough, but has an active community
CLI Support	Yes	Partial	Yes

Kubernetes Federation

No matter where you host your Kubernetes clusters – whether on-premises or in the cloud – Kubernetes’s federation feature is a powerful way to keep Kubernetes deployments more manageable and efficient. This chapter explains how federation works in Kubernetes, why you may want to take advantage of it and how you can go about doing it (as well as important caveats regarding the state of development of this feature).

What is Kubernetes Federation?

There are many SaaS and open-source solutions that have the capability to manage multiple, disparate Kubernetes clusters – regardless of the specific distribution or what hosted offering is in play – as long as it is a certified Kubernetes platform. Multi-cluster management focuses broadly on giving a single view to interact with, and report on each and every cluster that the manager is connected to.

Kubernetes Federation takes this concept of multi-cluster management and introduces the capabilities to have a master configuration managed through a single API in the host cluster, and have all or part of that configuration applied to any clusters that are members of the federation.

Multi-Tenant vs. Multi-Cluster vs. Federation

In the world of Kubernetes there are some concepts which are similar, and can share some features, but are distinct deployment models. If we start with the basic deployment model of a single cluster used by one development team where no partitioning of any kind is needed then it is great but that is not the best use of compute resources to have all the overhead associated with a cluster for a single development team. This is amplified when security and quality assurance teams also want their own place to test and deploy the same applications.

Multi-tenancy is the ability to have multiple distinct areas within a single cluster where quotas and security policies can be applied at the boundaries to improve overall resource utilization and simplify the overall infrastructure required to support an environment.

There are times where you still need to have multiple separate clusters in play whether it is to separate development and production, run sensitive workloads, or even across multiple data centers. This scenario can still have multi-tenancy in some or all of the clusters, but it has become a multi-cluster deployment. Multi-cluster management allows for a single interface which can see and control aspects of every cluster that is connected to the multi-cluster management plane.

The last type is Federation which is a spin on Multi-cluster, but does not cover all the multi-cluster scenarios. When multiple clusters are federated then they actually share pieces of their configuration which is managed by what is called the host cluster and sent to the member clusters of the federation. The benefits of having the high order parts of a configuration shared is that any resources configured to take advantage of the federation will use treat all member clusters as a single distributed cluster with oversight by the host cluster.

One of the most common scenarios where Federation is desirable is to scale an application across multiple data

centers. The application is packaged in a deployment which is set to leverage the federation and it equally spreads out its desired number of replicas across worker nodes in all the member clusters. Since not all the configuration is pushed this allows local variables in each member cluster to be supplemental to the federated configuration to accommodate cluster specific things like domain names and network policies.

History of Kubernetes Federation

Kubernetes Federation is an approach released and managed by the multicluster Special Interest Group (SIG) under the umbrella of the Kubernetes project in the CNCF. Version 1 of the standard was originally maintained by the core Kubernetes team when it was released in Kubernetes 1.5 version until about version 1.8, at which point it was transitioned to the SIG. The SIG realized the approach wasn't in line with how configuration management was moving in the main Kubernetes distributions and started working on version 2, which is a complete rewrite, and will work on Kubernetes 1.13 and newer.

Kubernetes Federation v2 is commonly referred to as "KubeFed," along with the new standard approach that leverages commonly-used components in Kubernetes, like CRDs. The SIG has a `kubefedctl` CLI tool in early stages of development on GitHub that will manage the configuration within a federation.

The basic structure of Kubernetes Federation

The core concept behind Kubernetes Federation is the host cluster that contains any configuration that will be propagated to what are called member clusters. The host cluster can be a member and run real workloads too, but typically organizations will have the host cluster as a standalone cluster for simplicity.

All cluster-wide configurations are handled through a single API, and define both what is in scope of the federation and what clusters that piece of configuration will be pushed too. The details of what is included in the federated configuration are defined by a series of templates, policies, and cluster-specific overrides.

The federated configuration manages the DNS entries for any multi-cluster services in addition to needing access to all the member clusters to create, apply, and remove any configuration items including deployments. Typically every deployment will have its own namespace, which will be consistent across the member clusters.

For a more detailed breakdown please visit the [site maintained by the KubeFed team](#).

So, for example, a Kubernetes Federation is set up with three-member clusters. The first runs in Google Cloud in Canada, the second in Azure in Singapore, and the third in IBM's UK cloud. You have the flexibility to deploy application A in namespace `test-app1` across `gke-canada` and `ibm-uk`, while application B will be deployed in `gke-canada` and `aks-singapore`.

Setting up Kubernetes Federation

As with all things Kubernetes, the concept is simple, but the execution can get as complicated as you want to make it. In the "happy day scenario," if all the prerequisites are in place in the clusters that will be federated, it can be as simple as a few commands. To see what is possible, [KubeFed Katacoda](#) takes just under 30 minutes to show off a simple use case.

The basic flow picks the cluster that will become the host cluster, and configures the roles and service accounts that are needed and deploy the federation control plane. From the [documentation](#) you can also see there is a HelmChart available for the installation, if you prefer.

Create RBAC for Helm

```
$ cat << EOF | kubectl apply -f -
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: tiller
  namespace: kube-system
EOF

$ helm init --service-account tiller
```

Install KubeFed Control Plane via Helm

```
$ helm repo add kubefed-charts \
  https://raw.githubusercontent.com/kubernetes-sigs/kubefed/master/charts
$ helm install kubefed-charts/kubefed --name kubefed --version=<x.x.x> \
  --namespace kube-federation-system
```

Configuration Flow within a Kubernetes Federation

Now that Federation is up and running between a couple Kubernetes clusters, it is time to start using the capabilities to make service and application administration much easier. The types of items that leverage Federations most often are deployments and services that point at the applications contained in those deployments.

The first two pieces of the puzzle are need a central namespace created on the host cluster to contain the federated deployment resources and a federated namespace with the same name on each cluster that will contain the deployed resources:

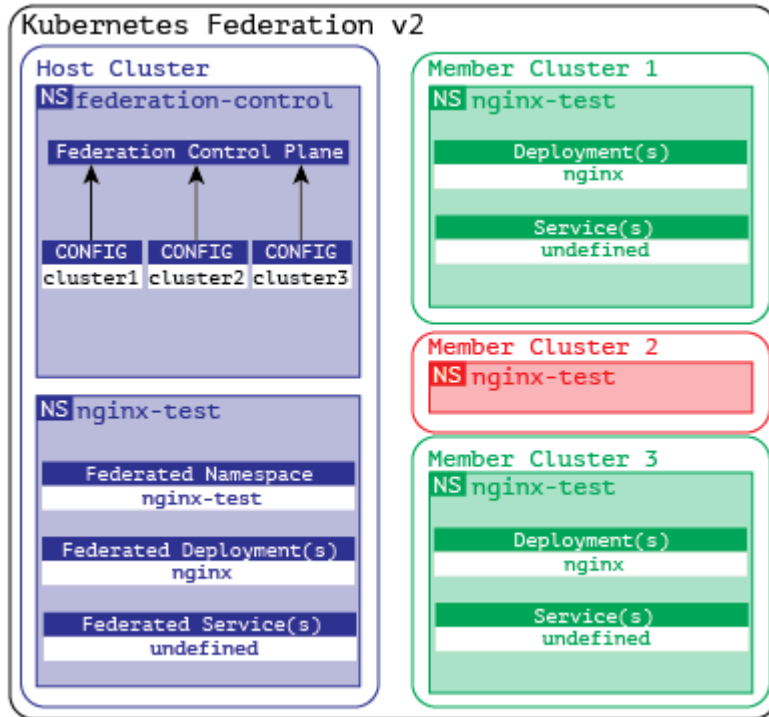
```
apiVersion: v1
kind: Namespace
metadata:
  name: nginx-test
---
apiVersion: types.kubefed.io/v1beta1
kind: FederatedNamespace
metadata:
  name: nginx-test
  namespace: nginx-test
spec:
  placement:
```

```
clusters:
- name: cluster1
- name: cluster2
- name: cluster3
```

The functional piece of the puzzle is the actual federated deployment of nginx that will create four replicas that will be spread evenly over the clusters it has specified and will use the nginx-test namespace on all clusters:

```
apiVersion: types.kubefed.io/v1beta1
kind: FederatedDeployment
metadata:
  name: nginx-test
  namespace: nginx-test
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      replicas: 4
      selector:
        matchLabels:
          app: nginx
      template:
        metadata:
          labels:
            app: nginx
        spec:
          containers:
            - image: nginx
              name: nginx
  placement:
    clusters:
      - name: cluster1
      - name: cluster3
```

This is a diagram that shows how that flows in the live clusters which are Federated including how it didn't flow to the cluster it excluded from its configuration. Notice that while cluster2 has the namespace it does not have any deployed artifacts which means that simply because the namespace exists on the member cluster does not mean everything automatically propagates into it without being declared in the federated deployment:



Next Steps

While Kubernetes Federation looks promising and is well on its way, it is not generally available yet. So for now, only those comfortable with the bleeding edge will be deploying it in production, and will rely on leading multi-cluster management products like Platform9.

To stay current with the status of KubeFed, the special interest group has mailing lists and regular meetings, which can be found on their [official information page](#).

Kata Containers, Docker and Kubernetes: How They All Fit Together

Kubernetes supports a number of container runtimes, and planning the right Kubernetes architecture requires choosing the right one. This chapter discusses two important runtime options – Kata and Docker – and explains how they relate to Kubernetes.

By now, virtually everyone has heard of Docker containers. But you may still be unfamiliar with [Kata](#), an open source container project launched in December of 2017. Kata emerged at a time when the container ecosystem was already crowded with other projects, making it easy to miss.

Yet, despite being a late arrival to the containerization party, Kata is developing into an important project -- not least because it promises to let developers and IT teams have their cake and eat it, too, by delivering both the performance of Docker containers and the security of virtual machines.

Here's a look at how Kata does that, how it is similar to and different from Docker, and (the question we know you're also asking) what all of this has to do with Kubernetes.

What are Kata containers?

In a nutshell, Kata is a container runtime designed to provide greater isolation between containers while still enabling the performance and efficiency provided by other runtimes.

Now, you may be thinking, "Why!? The last thing the world needs is yet another container runtime." That's a fair question to ask; between cri-o, containerd, rktlet, and Docker (to name just the most widely used runtimes), there was no shortage of runtime options before Kata came along.

But Kata is not just another container runtime. Although Kata is similar to other runtimes in most respects, there is one critical difference: the Kata runtime enforces a deeper level of isolation between containers than other runtimes.

It does this in two main ways:

- In Kata, each container runs its own kernel instead of sharing the host system's kernel with the host and other containers using cgroups. By extension, each container also gets its own I/O, memory access, and other low-level resources, without having to share them.
- In most cases, Kata containers can also take advantage of security features provided by hardware-level virtualization (meaning virtualization that is built into CPUs and made available using VT extensions).

And Kata does both of these things while avoiding the heavy resource consumption that comes with traditional virtualization. Unlike virtual machines, which can take a minute or two to start and waste a fair amount of hardware resources on isolation, Kata containers aim to start just as fast and consume resources just as efficiently as other containers.

Thus, the chief objective of the Kata project is to allow developers and IT teams to enjoy all the flexibility of traditional container runtimes, without the worry that a security breach in one container will escalate to affect other containers running on the same host. It eliminates the need to deploy different sets of containers inside virtual machines in order to achieve rigid isolation between the sets -- a practice that, to date, has been

widespread, but which undercuts one of the core benefits of containers: their ability to run faster and more efficiently than virtual machines.

Kata vs. Docker (and other container runtimes)

Given Kata's ambitions of doing containers better than Docker, the platform that brought containers into the mainstream starting in 2013, it's natural to want to compare Kata to Docker.

Such a comparison only makes partial sense, though, because Kata and Docker are not the same things. Kata is just a runtime, whereas Docker is a full suite of tools (some commercial, some open source) designed to create, orchestrate, and manage containerized applications.

Still, we can draw several major distinctions between Kata and Docker (as well as other container runtimes that are not Kata):

- **Security and isolation:** The most obvious difference is Kata's deeper level of isolation and security between containers, as described above.
- **Operating system support:** Currently, Kata runs only on Linux. Some other container runtimes, such as containerd, support both Linux and Windows.
- **Architecture support:** Kata can run on CPUs built using the x86_64 architecture, as well as on ARM, IBM p-series, and IBM z-series architectures. Most other container runtimes provide full support only for x86_64. (Some also support ARM, but there are limitations in most cases.) For this reason, Kata may become an important containerization solution for enterprise infrastructures that are built using architectures that haven't received much love from the Docker community, which has focused on commodity x86 servers.
- **Backers:** From an ecosystem standpoint, Kata stands out because the project is led by the OpenStack Foundation, a nonprofit best-known for its role in overseeing development of the OpenStack private cloud platform. OpenStack has declined in influence in recent years, and Kata might be the OpenStack Foundation's way of reasserting its relevance in the modern, container-focused age (which has been dominated so far by the Cloud Native Computing Foundation, the open source group behind Kubernetes). It's unclear what, if anything, the origins of Kata might mean for how the technology is actually used, but it could have implications for which integrations for the technology are built.

Kata and Kubernetes

If you're wondering whether Kata can be used with Kubernetes, the answer is a resounding yes. Despite the fact that Kata and Kubernetes are developed under the auspices of different organizations, they are not intended to compete with each other. Kata is a container runtime, whereas Kubernetes is a container orchestrator that can work with containers created using many different runtimes.

To put it in more technical terms, Kata adheres to the Open Container Initiative (OCI) standard, which Kubernetes supports. You can therefore use Kubernetes to orchestrate your Kata containers very easily.

In fact, if you want to test out Kata under Kubernetes, the Kata project has a [prebuilt deployment configuration](#) that you apply to your cluster with just a couple of Kubectl commands.

By way of conclusion, let us emphasize that Kata is not a new kid on the block who is out to compete with established container technologies like Docker and Kubernetes. It's a project that adds a fundamentally new type of functionality to the container ecosystem by providing a stronger isolation model. It squares the circle separating containers from virtual machines, allowing teams to get the best of both worlds.

Kata is well worth a look if you've always wanted to use containers, but were scared off by their comparatively

weak isolation architecture relative to virtual machines -- or if you have been deploying containers inside virtual machines in order to achieve more isolation, but are tired of waiting minutes for those virtual machines to start.

Kubernetes Service Discovery

One of the reasons why Kubernetes is the leading Container Orchestration platform today is because it offers a nice separation of concerns in terms of describing workloads.

Essentially, Kubernetes consists of a collection of API-oriented, very fast binaries, that are well-documented and easy to contribute to and build applications upon.

The basic building block starts with the Pod, which is just a resource that can be created and destroyed on demand. Because a Pod can be moved or rescheduled to another Node, any internal IPs that this Pod is assigned can change over time.

If we were to connect to this Pod to access our application, it would not work on the next re-deployment. To make a Pod reachable to external networks or clusters without relying on any internal IPs, we need another layer of abstraction. K8s offers that abstraction with what we call a Service Deployment.

Services provide network connectivity to Pods that work uniformly across clusters. Service discovery is the actual process of figuring out how to connect to a service.

In this chapter, we are going to explore some of the fundamental operations of service discovery in K8s along with some common troubleshooting issues and gotchas that appear in large-scale apps.

For the following examples, we assume you have a configured Kubernetes cluster. For testing purposes, we created one using [DigitalOcean](#) with one node (which costs about \$10 per month). We are using [hello-kubernetes image](#).

Defining and deploying Services

First, we create an initial infrastructure to grasp the concepts of services and how they work in practice.

Create two namespaces, one for **develop** and one for **production**:

```
$ cat << ENDL > develop-namespace.yml
apiVersion: v1
kind: Namespace
metadata:
  name: develop
ENDL
```

```
$ cat << ENDL > production-namespace.yml
apiVersion: v1
kind: Namespace
metadata:
  name: production
```

ENDL

```
$ kubectl apply -f develop-namespace.yml
namespace/develop created
```

```
$ kubectl apply -f production-namespace.yml
namespace/production created
```

Deploy our example application on both namespaces:

```
$ cat << ENDL > app-deployment-develop.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello
  namespace: develop
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hello
  template:
    metadata:
      labels:
        app: hello
    spec:
      containers:
      - name: hello-kubernetes
        image: paulbouwer/hello-kubernetes:1.5
        ports:
        - containerPort: 8080
```

ENDL

```
cat << ENDL > app-deployment-production.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello
  namespace: production
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hello
  template:
    metadata:
      labels:
        app: hello
    spec:
```

```
    containers:
      - name: hello-kubernetes
        image: paulbouwer/hello-kubernetes:1.5
        ports:
          - containerPort: 8080
ENDL

$ kubectl apply -f app-deployment-develop.yml
$ kubectl apply -f app-deployment-production.yml
```

Now we can describe the Pods and gather their IP addresses:

```
$ kubectl describe pods --all-namespaces

...
Name:                hello-f878d7d65-qjmbp
Namespace:           production
IP:                  10.244.0.112
...
```

In the output above, the IPs for each Pod are internal and specific to each instance. If we were to redeploy the application, it would be assigned a new IP each time.

We verify that we can ping a Pod within the cluster. Create a temporary Pod in the default namespace and ping that IP address:

```
$ cat << ENDL >> jumpod.yml
apiVersion: v1
kind: Pod
metadata:
  name: jumpod
  namespace: default
spec:
  containers:
    - name: busybox
      image: busybox:1.28
      command:
        - sleep
        - "3600"
      imagePullPolicy: IfNotPresent
      restartPolicy: Always
ENDL

$ kubectl apply -f jumpod.yml
pod/jumpod created

$ kubectl exec -it jumpod ping 10.244.0.149
PING 10.244.0.149 (10.244.0.149): 56 data bytes
```

Using the **nslookup** tool, we can get the FQDN of the Pod:

```
$ kubectl exec -it jumpod nslookup 10.244.0.149
Server:      10.245.0.10
Address 1: 10.245.0.10 kube-dns.kube-system.svc.cluster.local

Name:       10.244.0.149
Address 1: 10.244.0.149 10-244-0-
149.hello.production.svc.cluster.local
```

Next, we use a service deployment to expose our application to the Internet. Here are examples for each namespace:

```
$ cat << ENDL > app-service-develop.yml
apiVersion: v1
kind: Service (1)
metadata:
  name: hello
  namespace: develop
spec:
  type: LoadBalancer (2)
  ports:
    - port: 80 (3)
      targetPort: 8080
  selector:
    app: hello (4)
ENDL

$ cat << ENDL > app-service-production.yml
apiVersion: v1
kind: Service (1)
metadata:
  name: hello
  namespace: production
spec:
  type: LoadBalancer (2)
  ports:
    - port: 80 (3)
      targetPort: 8080
  selector:
    app: hello (4)
ENDL

$ kubectl apply -f app-service-develop.yml
$ kubectl apply -f app-service-production.yml
```

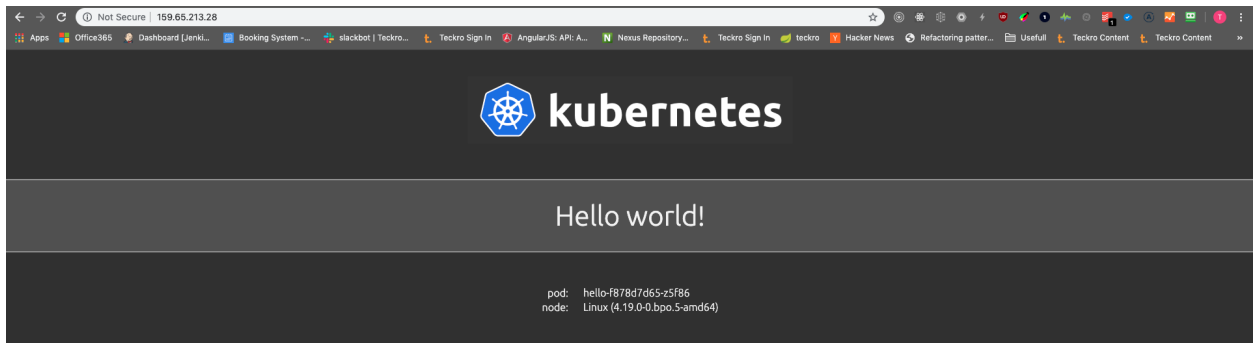
Where:

- (1) We specify the kind of workload to be a Service.
- (2) We specify the Service type to be a **LoadBalancer**. This means that we expose the service externally using a cloud provider's load balancer. Other options include **ClusterIP**, which makes the Service only reachable from within the cluster, **NodePort**, which uses the Node IP and a static port to expose the service outside the cluster, and finally, **ExternalName**, which maps a Service to a DNS name that we configure elsewhere.
- (3) This is the port to expose to the external network.
- (4) This is the name of the deployment that we want to connect to.


Verify that we can visit the public IPs of the services that we deployed:

```
$ kubectl get services --all-namespaces
NAMESPACE   NAME           TYPE           CLUSTER-IP   EXTERNAL-IP   AGE
default     kubernetes    ClusterIP      10.245.0.1    <none>        443/TCP
develop     hello         LoadBalancer  10.245.150.103  159.65.213.28 80:31129/TCP
kube-system kube-dns       ClusterIP      10.245.0.10   <none>        53/UDP, 53/TCP, 9153/TCP
production  hello         LoadBalancer  10.245.143.177 159.65.214.225 80:31705/TCP
```

If we visit either <http://159.65.213.28> or <http://159.65.214.225>, we should be able to see the application running:



Note: If we were to use a NodePort Service type instead, then Kubernetes would expose the service in a random port within the range 30000-32767 using the Node's primary IP address. For example, let's look at the Droplet that was created when we provisioned the Kubernetes Cluster:

Name	IP Address	Created	Tags
 pool-aotcpfyu4-rtcd 2 GB / 50 GB Disk / LON1 - Debian do-kube-115.2-do.0	167.71.141.161	4 hours ago	k8s + 2 More ▾

Here, the Node IP is **167.71.141.161**, and when we create a NodePort Service, we get the following assignment:

```
$ kubectl get services --all-namespaces
NAMESPACE      NAME           TYPE           CLUSTER-IP      EXTERNAL-IP
PORT(S)                AGE
default        kubernetes    ClusterIP      10.245.0.1      <none>
443/TCP                4h16m
develop        hello         NodePort       10.245.209.176  <none>
80:31519/TCP          17s
kube-system    kube-dns      ClusterIP      10.245.0.10     <none>
53/UDP,53/TCP,9153/TCP 4h16m
production     hello         LoadBalancer  10.245.143.177  159.65.214.225
80:31705/TCP                164m
```

So, based on that information, we can visit <http://167.71.141.161:31519/> and access the application again.

Also, note that the **ClusterIP** that was assigned here for both the develop and production namespaces is the **LoadBalancer** address. This means that it will balance the requests based on an algorithm (default is Round-robin) and it is only accessible within the cluster. We can verify this using the following commands:

```
$ kubectl exec -it jumpod nslookup 10.245.150.103
Server:      10.245.0.10
Address 1: 10.245.0.10 kube-dns.kube-system.svc.cluster.local
```

```
Name:        10.245.150.103
Address 1: 10.245.150.103 hello.develop.svc.cluster.local
```

```
$ nslookup 10.245.150.103
Server:      8.8.8.8
Address:     8.8.8.8#53
```

```
** server can't find 103.150.245.10.in-addr.arpa: NXDOMAIN
```

Additionally, we can access the applications from within the cluster using the following HTTP urls:
<http://hello.develop> or <http://hello.production>

```
$ kubectl exec -it jumpod wget -- -O - http://hello.develop
Connecting to hello.develop (10.245.150.103:80)
<!DOCTYPE html>
<html>
<head>
  <title>Hello Kubernetes!</title>
  <link rel="stylesheet" type="text/css" href="/css/main.css">
  <link rel="stylesheet"
href="https://fonts.googleapis.com/css?family=Ubuntu:300" >
</head>
<body>
...
```

The previous examples worked because this Kubernetes cluster automatically configures an internal DNS service to provide a lightweight mechanism for service discovery by default.

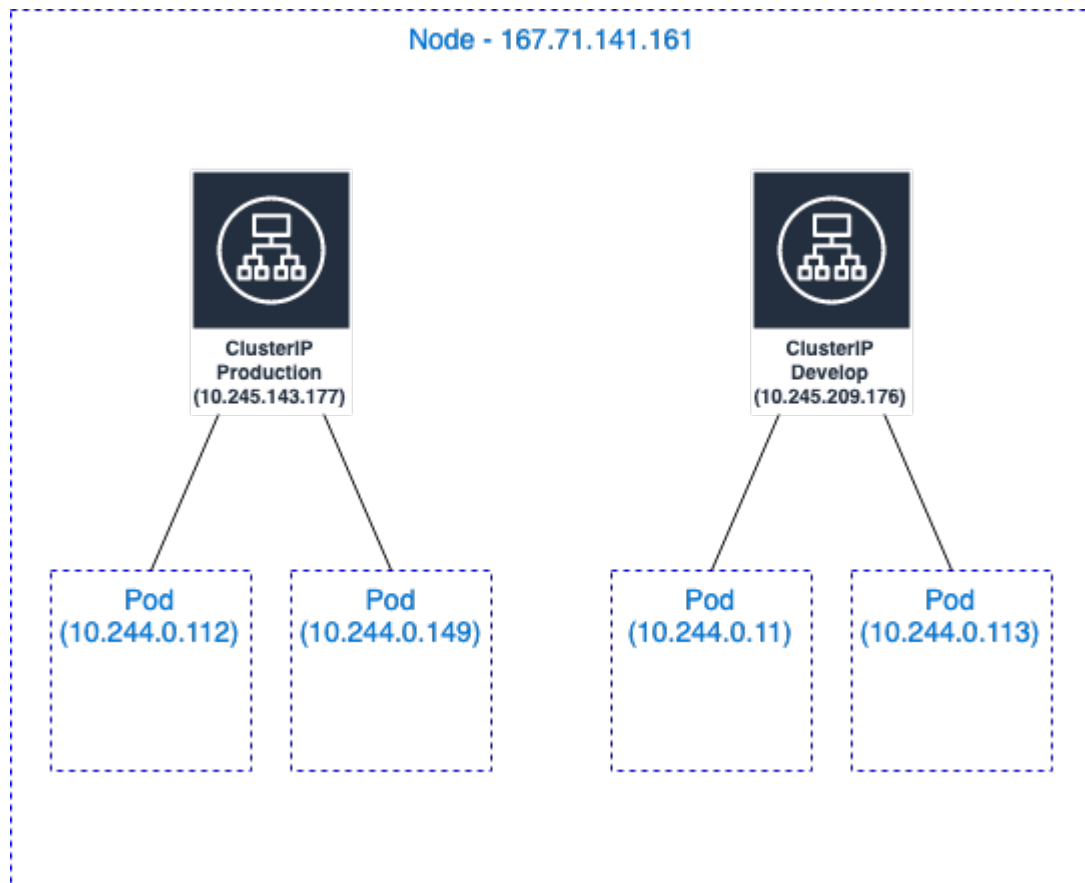
This is not, however, the only way to do it. We can use environment variables as a last resort, but they are not ideal in cases where the Pod might fail to reach a service that was created after the Pod.

In our example application, Kubernetes exports a set of environment variables on the node where the Pod

gets created:

```
$ kubectl exec -it hello-f878d7d65-2qctb env --namespace=develop
...
HELLO_SERVICE_PORT=80
HELLO_PORT=tcp://10.245.150.103:80
HELLO_SERVICE_HOST=10.245.150.103
HELLO_PORT_80_TCP=tcp://10.245.150.103:80
HELLO_PORT_80_TCP_PROTO=tcp
HELLO_PORT_80_TCP_ADDR=10.245.150.103
HELLO_PORT_80_TCP_PORT=80
NPM_CONFIG_LOGLEVEL=info
NODE_VERSION=8.1.0
YARN_VERSION=0.24.6
HOME=/root
```

Below we can see the final schematic of the Node and ClusterIP layout:



Common Service Discovery Gotchas

Now that you understand the basics of Service Discovery with Kubernetes, let's look at some of the most common troubleshooting scenarios.

- **We cannot find the service Environmental Variables**
You have created a Pod that needs to access a Service, and you discover that there are no environment

variables defined other than the default ones. In this case, you may have to redeploy the Pods again, because the Service will not inject the variables after the Pod is created.

The quickest way to redeploy them is to scale them down and up again:

```
$ kubectl scale deployment hello --replicas=0 --namespace=develop
$ kubectl scale deployment hello --replicas=2 --namespace=develop
```

If you inspect the Pod environment again, you should be able to read the service variables from the environment.

- **DNS Service Discovery is not working**

First, make sure that the DNS addon service is installed:

```
$ kubectl get pods --namespace=kube-system -l k8s-app=kube-dns
NAME                                READY   STATUS    RESTARTS   AGE
coredns-9d6bf9876-lnk5w             1/1    Running   0          174m
coredns-9d6bf9876-mshvs             1/1    Running   0          174m
```

Then, verify that the service IP address is assigned:

```
$ kubectl get services --namespace=kube-system -l k8s-app=kube-dns
NAME          TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)
AGE
kube-dns      ClusterIP     10.245.0.10   <none>
53/UDP,53/TCP,9153/TCP  176m
```

Since DNS is an addon to Kubernetes, it may not be installed or configured correctly.

Now, when a new Pod is created, Kubernetes adds the following entry in **/etc/resolv.conf** with the nameserver option to the cluster IP of the kube-dns service and appropriate search options to allow for shorter hostnames:

```
$ cat /etc/resolv.conf
nameserver <kube-dns ClusterIP>
search <namespace>.svc.<cluster_domain> svc.<cluster_domain>
<cluster_domain> <additional ...>
options ndots:5
```

If we try to nslookup the **kubernetes.default** name and it fails, then there is a problem with the **coredns/kube-dns** add-on:

```
$ nslookup kubernetes.default
Server: 10.32.0.10
Address 1: 10.32.0.10
```

```
nslookup: can't resolve 'kubernetes.default'
```

We also need to make sure that the server address matches the kube-dns service IP that we found earlier (for our example 10.245.0.10).

Note: DNS Configuration for multi-cloud Kubernetes (federated) environments is not yet GA. You can look at the relevant [kubefed](#) project for more information.

Services Meshes and Kubernetes

If managing all of the services in Kubernetes seems daunting, there is a tool to help: The service mesh. Actually, there are several such tools, because there are multiple service mesh platforms developed by different vendors or open source projects.

Let's take a look at what service meshes are and why they are important in the context of Kubernetes – as well as which service mesh offerings are available today.

The Rise of the Service Mesh

Cloud-native applications are often architected as a constellation of distributed microservices, which are running in Containers. Increasingly, these containerized applications are Kubernetes-based, as it has become the de-facto standard for container orchestration.

One outcome that most companies using microservices architecture don't fully understand the impact of until they are well down the path is microservices sprawl. Like the suburbs around a city, the number of small services that are deployed seems to expand exponentially.

This exponential growth in microservices creates challenges around figuring out how to enforce and standardize things like routing between multiple services/versions, authentication and authorization, encryption, and load balancing within a Kubernetes cluster.

Building on Service Mesh helps resolve some of these issues, and more. As containers abstract away the operating system from the application, Service Meshes abstract away how inter-process communications are handled.

What is Service Mesh

While Service Mesh technology has been around prior to Kubernetes, the proliferation of microservices that are built on Kubernetes has contributed to the growing interest in Service Mesh solutions.

The thing that is most crucial to understand about microservices is that they are heavily reliant on the network.

Service Mesh manages the network traffic between services. It does that in a much more graceful and scalable way compared to what would otherwise require a lot of manual, error-prone work and operational burden that is not sustainable in the long-run.

In general, service mesh layers on top of your Kubernetes infrastructure and is making communications between services over the network safe and reliable.

Think about service mesh like a routing and tracking service for a package shipped in the mail: it keeps track of the routing rules and dynamically directs the traffic and package route to accelerate delivery and ensure receipt.

Service mesh allows you to separate the business logic of the application from observability, and network and security policies. It allows you to connect, secure, and monitor your microservices.

- Connect: Service Mesh enables services to discover and talk to each other. It enables intelligent routing to control the flow of traffic and API calls between services/endpoints. These also enable advanced deployment strategies such as blue/green, canaries or rolling upgrades, and more.

- Secure: Service Mesh allows you secure communication between services. It can enforce policies to allow or deny communication. E.g. you can configure a policy to deny access to production services from a client service running in development environment.
- Monitor: Service Mesh enables observability of your distributed microservices system. Service Mesh often integrates out-of-the-box with monitoring and tracing tools (such as Prometheus and Jaeger in the case of Kubernetes) to allow you to discover and visualize dependencies between services, traffic flow, API latencies, and tracing.

These key capabilities provide operational control and observability into the behavior of the entire network of distributed microservices that make up a complex cloud-native application.

Service Mesh is critical when you're dealing with web-scale or hyper-scale microservices workloads (think Netflix, Amazon, etc.). But, as we'll see below, there's plenty that you can already get out of service mesh now – while you're still growing – as a framework to support massive scale in the future.

Service Mesh Options for Kubernetes:

There are three leading contenders in the Kubernetes ecosystem for Service Mesh. All of these solutions are open source. Each solution has its own benefits and downfalls, but using any of them will put your DevOps teams in a better position to thrive as they develop and maintain more and more microservices.



Consul Connect

Consul is a full-feature service management framework, and the addition of Connect in v1.2 gives it service discovery capabilities which make it a full Service Mesh. Consul is part of HashiCorp's suite of infrastructure management products; it started as a way to manage services running on Nomad and has grown to support multiple other data center and container management platforms including Kubernetes.

Consul Connect uses an agent installed on every node as a DaemonSet which communicates with the Envoy sidecar proxies that handles routing & forwarding of traffic.

Architecture diagrams and more product information is available at [Consul.io](https://consul.io).



Istio

Istio is a Kubernetes-native solution that was initially released by Lyft, and a large number of major technology companies have chosen to back it as their service mesh of choice. Google, IBM, and Microsoft rely on Istio as the default service mesh that is offered in their respective Kubernetes cloud services. A [fully-managed service of Istio](#) for hybrid environments will soon be available from Platform9 [Managed Kubernetes](#) service.

Istio was the first to include additional features that developers really wanted, like deep-dive analytics.

Istio has separated its data and control planes by using a sidecar loaded proxy which caches information so that it

does not need to go back to the control plane for every call. The control planes are pods that also run in the Kubernetes cluster, allowing for better resilience in the event that there is a failure of a single pod in any part of the service mesh.

Architecture diagrams and more product information is available at [Istio.io](https://istio.io).



LINKERD




Linkerd

Linkerd is arguably the second most popular service mesh on Kubernetes and, due to its rewrite in v2, its architecture mirrors Istio’s closely, with an initial focus on simplicity instead of flexibility. This fact, along with it being a Kubernetes-only solution, results in fewer moving pieces, which means that Linkerd has less complexity overall. While Linkerd v1.x is still supported, and it supports more container platforms than Kubernetes; new features (like blue/green deployments) are focused on v2. primarily.

Linkerd is unique in that it is part of the Cloud Native Foundation (CNCF), which is the organization responsible for Kubernetes. No other service mesh is backed by an independent foundation.

Architecture diagrams and additional product information is available at [Linkerd.io](https://linkerd.io).

Comparison of Istio, Linkerd and Consul Connect for Kubernetes Service Mesh

	 Istio Istio	 LINKERD Linkerd v2	 HashiCorp Consul Consul
Supported Workloads	Does it support both VMs-based applications and Kubernetes?		
Workloads	Kubernetes + VMs	Kubernetes only	Kubernetes + VMs

Architecture	The solution’s architecture has implications on operation overhead.		
Single point of failure	No – uses sidecar per pod	No	No. But added complexity managing HA due to having to install the Consul server and its quorum operations, etc., vs. using the native K8s master primitives.
Sidecar Proxy	Yes (Envoy)	Yes	Yes (Envoy)
Per-node agent	No	No	Yes
Secure Communication	All services support mutual TLS encryption (mTLS), and native certificate management so that you can rotate certificates or revoke them if they are compromised.		
mTLS	Yes	Yes	Yes
Certificate Management	Yes	Yes	Yes
Authentication and Authorization	Yes	Yes	Yes

Communication Protocols

TCP	Yes	Yes	Yes
HTTP/1.x	Yes	Yes	Yes
HTTP/2	Yes	Yes	Yes
gRPC	Yes	Yes	Yes

Traffic Management

Blue/Green Deployments	Yes	Yes	Yes
Circuit Breaking	Yes	No	Yes
Fault Injection	Yes	Yes	Yes
Rate Limiting	Yes	No	Yes

Chaos Monkey-style Testing	Traffic management features allow you to introduce delays or failures to some of the requests in order to improve the resiliency of your system and harden your operations		
Testing	Yes- you can configure services to delay or outright fail a certain percentage of requests	Limited	No
Observability	In order to identify and troubleshoot incidents, you need distributed monitoring and tracing.		
Monitoring	Yes, with Prometheus	Yes, with Prometheus	Yes, with Prometheus
Distributed Tracing	Yes	Some	Yes
Multicluster Support			
Multicluster	Yes	No	Yes
Installation			

Deployment	Install via Helm and Operator	Helm	Helm
Operations Complexity	How difficult is it to install, configure and operate		
Complexity	High	Low	Medium

Any of these service meshes will solve your basic needs. The choice comes down to whether you want more than the basics.

Istio has the most features and flexibility of any of these three service meshes by far, but remember that flexibility means complexity, so your team needs to be ready for that.

For a minimalistic approach supporting just Kubernetes, Linkerd may be the best choice. If you want to support a heterogeneous environment that includes both Kubernetes and VMs and do not need the complexity of Istio, then Consul would probably be your best bet.

Migrating between service mesh solutions

Note that service mesh is not as an intrusive transformation as the one from monolithic applications to microservices, or from VMs to Kubernetes-based applications. Since most meshes use the sidecar model, most services don't know that they run as a mesh. However, replacing one service mesh with another is complex, particularly when you want to standardize on the service mesh as a solution to scale across all your services.

So it's important to choose wisely! Start with a sample project(s) and see which solution you prefer.

Istio is quickly becoming the standard for service mesh on Kubernetes. It is the most mature, but also the most complex to deploy. For a managed experience of consuming Istio at scale, stay tuned for when we announce our [Managed Istio solution](#), as part of our [Kubernetes managed apps](#)!

Common use cases to take advantage of Service Mesh today

From an Operations point of view, Service Mesh is useful for any type of microservices architecture since it helps you control traffic, security, permissions, and observability.

Once you have a Kubernetes infrastructure + Microservices architecture consider the below use cases in order to take advantage of Service Mesh in your organization today, regardless of the scale of your applications.

By getting your feet wet with these, you can start standardizing on Service Mesh in your system design to lay the building blocks and the critical components for large-scale operations in the future.

- **Improving observability into distributed services:** with service-level visibility, tracing, and monitoring abilities. Some of the key capabilities of service mesh dramatically improve visibility as well as your ability to troubleshoot and mitigate incidents. For example, If one service in the architecture becomes a bottleneck, the common way to handle it is through re-tries, but that can worsen the bottleneck due to timeouts. With service mesh, you can easily break the circuit to failed services to disable non-functioning replicas and keep the API responsive.
- **Blue/green deployments:** with the ability to control traffic. Service mesh allows you to implement Blue/Green deployments to safely rollout new upgrades of the applications without risking service interruption. First, you expose only a small subset of users to the new version, validate it, then proceed to release it to all instances in Production.
- **Chaos monkey/ testing in production scenarios:** with the ability to inject delays, faults to improve the robustness of deployments
- **'Bridge' / enabler for modernizing legacy applications:** If you're in the throes of modernizing your existing applications to Kubernetes-based microservices, you can use service mesh as a 'bridge' while you're de-composing your apps. You can register your existing applications as 'services' in the Istio service catalog and then start migrating them gradually to Kubernetes without changing the mode of communication between services – like a DNS router. This use case is similar to using Service Directory.
- **API Gateway:** If you're bought into the vision of service mesh and want to start the rollout, but don't yet have Kubernetes applications up and running, you can already have your Operations team start learning the ropes of using service mesh by deploying it simply to measure your API usage.

In its most mature implementation, Service mesh becomes the dashboard for microservices architecture. It's the place for troubleshooting issues, enforcing traffic policies, rate limits, and testing new code. It's your hub for monitoring, tracing and controlling the interactions between all services – how they are connected, perform and secured.

Scaling Kubernetes for Modern Software Delivery

The bulk of this eBook has focused on different architectural approaches to Kubernetes itself. But at the end of the day, what matters is how well Kubernetes supports the use cases it is meant to advance.

One of the main use cases of Kubernetes is to run 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 Integration phase; false, if it does not.

The previous steps describe the CI part or Continuous Integration. Another pipeline can pick up the CD part, which is the Continuous Delivery, so it will attempt to roll out the Application Container to production. The key thing to understand is that those operations run on demand, so we need to have a mechanism to spin up individual nodes to run those pipeline steps and retire them when they are unneeded. This way we preserve resources and reduce down costs.

The key mechanism here is, of course, Kubernetes, which – with its declarative structure and customizability – allows you to efficiently schedule jobs, nodes and pods for any kind of scenarios.

This chapter is comprised of three parts. In the first part we will explore the most popular CI/CD platforms that work on Kubernetes at the moment.

After that, we will look at two use cases. In the first case, we will install Jenkins on Kubernetes in simple steps and see how we can configure it. In the second case, we will take this Jenkins deployment to the next level. Here we provide tips and recommendations for scaling up CI/CD pipelines in Kubernetes. Finally, we discuss the most reasonable ways and practices to run Kubernetes for CI/CD at scale.

The goal of this chapter is to provide you with a thorough understanding of the efficiency of Kubernetes in handling those workloads.

Let's get started.

CI/CD platforms for Kubernetes

Kubernetes is an ideal platform for running CI/CD platforms, as it has [plenty of features](#) that make it easy to do so. How many CI/CD platforms can run on Kubernetes at the moment you ask? Well, as long as they can be packaged in a container, Kubernetes can run them. There are quite a few options that work more closely with Kubernetes and are worth mentioning:

- **Jenkins:** Jenkins is the most popular and the most stable CI/CD platform. It's used by thousands of enterprises around the world due to its vast ecosystem and extensibility. If you plan to use it with Kubernetes, it's recommended to install the [official plugin](#).
- **JenkinsX:** This is a transformation of Jenkins, suited for the cloud-native world. It is suitable to operate more harmoniously with Kubernetes and offer better integration features like GitOps, Automated CI/CD and Preview Environments.
- **Drone:** This is a versatile, cloud-native CD platform with many features. It can be run in Kubernetes using the [associated Runner](#).
- **GoCD:** Another CI/CD platform from Thoughtworks that offers a variety of workflows and features

suited for cloud native deployments. It can be run in Kubernetes as a [Helm Chart](#).

- **Spinnaker:** Is a CD platform for scalable multi-cloud deployments, with backing from Netflix. To install it, we can use the relevant [Helm Chart](#).

Additionally, there are cloud services that work closely with Kubernetes and provide CI/CD pipelines like CircleCI and Travis, so it's equally helpful if you don't plan to have hosted CI/CD platforms.

Let's see how we can get our hands dirty by installing JenkinsX on a Kubernetes Cluster.

Use Case: Installing Jenkins

For the purposes of this tutorial, we are going to use DigitalOcean as our Kubernetes Service to install Jenkins using Helm.

Note that installing JenkinsX on a public Kubernetes provider is a tricky business. Currently, JenkinsX is a work in progress when it comes to deploying it to a working cluster, as there are lots of issues to solve.

First we need to install Helm, which is the package manager for Kubernetes:

```
$ curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3
> get_helm.sh
$ chmod 700 get_helm.sh
$ ./get_helm.sh -v v2.15.0
```

Also, we need to install Tiller, for Helm to run properly:

```
$ kubectl -n kube-system create serviceaccount tiller
serviceaccount/tiller created

~/kube
$ kubectl create clusterrolebinding tiller --clusterrole cluster-admin --
serviceaccount=kube-system:tiller
clusterrolebinding.rbac.authorization.k8s.io/tiller created

~/kube
$ helm init --service-account tiller
$HELM_HOME has been configured at /Users/itspare/.helm.
```

Once we've done those steps, we need to run inspect command to check the configuration values of the deployment:

```
$ helm inspect values stable/jenkins > values.yml
```

Carefully inspect the configuration values and make changes if required. Then install the chart:

```
$ helm install stable/jenkins --tls \
  --name jenkins \
  --namespace jenkins
```

The installation process will spit out some instructions for what to do next:

NOTES:

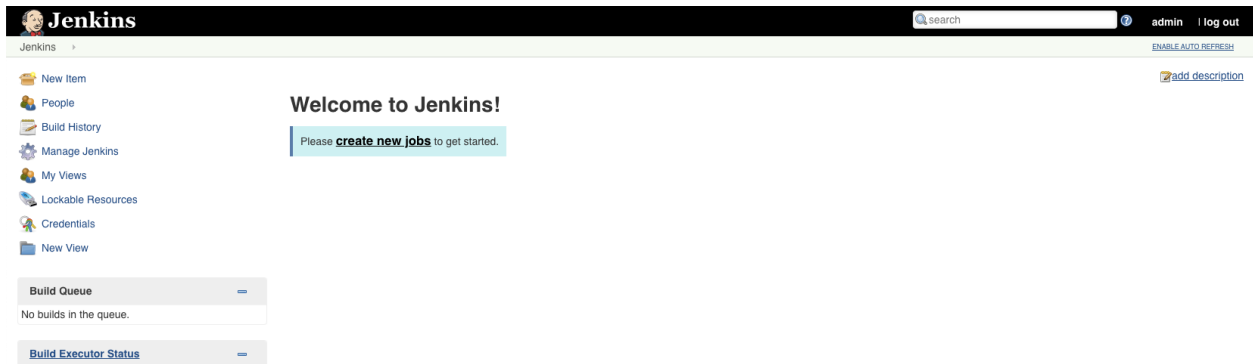
1. Get your 'admin' user password by running:

```

printf $(kubectl get secret --namespace default my-jenkins -o
jsonpath="{.data.jenkins-admin-password}" | base64 --decode);echo
2. Get the Jenkins URL to visit by running these commands in the same shell:
export POD_NAME=$(kubectl get pods --namespace default -l
"app.kubernetes.io/component=jenkins-master" -l
"app.kubernetes.io/instance=my-jenkins" -o
jsonpath="{.items[0].metadata.name}")
echo http://127.0.0.1:8080
kubectl --namespace default port-forward $POD_NAME 8080:8080

```

Follow those steps and they will start the proxy server at <http://127.0.0.1:8080>.
 Navigate there and enter your username and password. Voila! Your own Jenkins server in 5 minutes:



Bear in mind that there is still a lot of configuration options left to apply, so you should visit the [chart documentation page](#) for more information. By default, this server comes installed with the most basic plugins such as Git, and Kubernetes-Jenkins, and we can install more on demand.

Overall, the experience of installing Jenkins with Helm is largely effortless; the same can't be said for JenkinsX, which was ... well, painful, if this chapter is any indication.

Use Case: Scaling CI/CD Jenkins Pipelines

Now that we have a rough understanding of how CI/CDs fit together with Kubernetes, let's see an example use case of deploying a highly-scalable Jenkins deployment in Kubernetes. This is in actual use (with small modifications) at Teckro to handle CI/CDs in our infrastructure, so it's more opinionated than the general case. Let's start:

Use of an Opinionated Jenkins Distro

While the official Jenkins image is a good choice for starting out, it needs more configuring than we may want. With Teckro, we opted for a more opinionated distro using [my-bloody-jenkins](#), which offers quite a long list of pre-installed plugins and configuration options. Out of the available plugins, we use the [saml plugin](#), SonarQubeRunner, Maven and Gradle.

Although, it can be installed via a Helm Chart using the following commands:

```
$ helm repo add odavid https://odavid.github.io/k8s-helm-charts
$ helm install odavid/my-bloody-jenkins
```

We chose to deploy a custom image instead, using the following Dockerfile;

```
FROM odavid/my-bloody-jenkins:2.190.2-161

USER jenkins

COPY plugins.txt /usr/share/jenkins/ref/
RUN /usr/local/bin/install-plugins.sh < /usr/share/jenkins/ref/plugins.txt

USER root
```

Where the plugins.txt file is a list of additional plugins we want to pre-install into the image:

```
build-monitor-plugin
xcode-plugin
rich-text-publisher-plugin
jacoco
scoverage
dependency-check-jenkins-plugin
greenballs
shiningpanda
pyenv-pipeline
s3
pipeline-aws
appcenter
multiple-scms
Testng-plugin
```

Then we use this generic Jenkinsfile to build the master whenever the dockerfile changes:

```
#!/usr/bin/env groovy

node('generic') {
    try {

        def dockerTag, jenkins_master

        stage('Checkout') {
```

```

        checkout([
            $class: 'GitSCM',
            branches: scm.branches,
            doGenerateSubmoduleConfigurations:
scm.doGenerateSubmoduleConfigurations,
            extensions: [[${class: 'CloneOption', noTags: false, shallow:
false, depth: 0, reference: ''}],
            userRemoteConfigs: scm.userRemoteConfigs,
        ])

        def version = sh(returnStdout: true, script: "git describe --tags
`git rev-list --tags --max-count=1`").trim()
        def tag = sh(returnStdout: true, script: "git rev-parse --short
HEAD").trim()
        dockerTag = version + "-" + tag
        println("Tag: " + tag + " Version: " + version)
    }

    stage('Build Master') {
        jenkins_master = docker.build("jenkins-master", "--network=host
.")
    }

    stage('Push images') {
        docker.withRegistry("https://\$env.DOCKER\_REGISTRY", 'ecr:eu-west-
2:jenkins-aws-credentials') {
            jenkins_master.push("${dockerTag}")
        }
    }

    if(env.BRANCH_NAME == 'master') {

        stage('Push Latest images') {
            docker.withRegistry("https://\$env.DOCKER\_REGISTRY", 'ecr:eu-
west-2:jenkins-aws-credentials') {
                jenkins_master.push("latest")
            }
        }
    }

```

```

    stage('Deploy to K8s cluster') {
        withKubeConfig([credentialsId: 'dev-tools-eks-jenkins-secret',
            serverUrl: env.TOOLS_EKS_URL]) {
            sh "kubectl set image statefulset jenkins
jenkins=$env.DOCKER_REGISTRY/jenkins-master:${dockerTag}"
        }
    }
}
currentBuild.result = 'SUCCESS'
} catch(e) {
    currentBuild.result = 'FAILURE'
    throw e
}
}
}

```

We use a dedicated cluster composed of a few large and medium instances in AWS for Jenkins jobs, which brings us to the next section.

Use of Dedicated Jenkins Slaves and labels

For scaling some of our Jenkins slaves we use Pod Templates and assign labels to specific agents, so in our Jenkinsfiles we can reference them for jobs. For example, we have some agents that need to build Android applications. So we reference the following label:

```

pipeline {
    agent { label "android" }
    ...
}

```

And that will use the specific pod template for Android. We use this Dockerfile, for example:

```

FROM dkr.ecr.eu-west-2.amazonaws.com/jenkins-jnlp-slave:latest

RUN apt-get update && apt-get install -y -f --no-install-recommends
xmlstarlet

ARG GULP_VERSION=4.0.0
ARG CORDOVA_VERSION=8.0.0

# SDK version and build-tools version should be different
ENV SDK_VERSION      25.2.3

```

```

ENV BUILD_TOOLS_VERSION 26.0.2
ENV SDK_CHECKSUM
1b35bcb94e9a686dff6460c8bca903aa0281c6696001067f34ec00093145b560
ENV ANDROID_HOME /opt/android-sdk
ENV SDK_UPDATE tools,platform-tools,build-tools-25.0.2,android-
25,android-24,android-23,android-22,android-21,sys-img-armeabi-v7a-android-
26,sys-img-x86-android-23
ENV LD_LIBRARY_PATH
${ANDROID_HOME}/tools/lib64/qt:${ANDROID_HOME}/tools/lib/libQt5:${LD_LIBRARY_P
ATH/
ENV PATH ${PATH}:${ANDROID_HOME}/tools:${ANDROID_HOME}/platform-
tools

RUN curl -SLO
"https://dl.google.com/android/repository/tools_r${SDK_VERSION}-linux.zip" \
  && echo "${SDK_CHECKSUM} tools_r${SDK_VERSION}-linux.zip" | sha256sum -c
- \
  && mkdir -p "${ANDROID_HOME}" \
  && unzip -qq "tools_r${SDK_VERSION}-linux.zip" -d "${ANDROID_HOME}" \
  && rm -Rf "tools_r${SDK_VERSION}-linux.zip" \
  && echo y | ${ANDROID_HOME}/tools/android update sdk --filter
${SDK_UPDATE} --all --no-ui --force \
  && mkdir -p ${ANDROID_HOME}/tools/keymaps \
  && touch ${ANDROID_HOME}/tools/keymaps/en-us \
  && yes | ${ANDROID_HOME}/tools/bin/sdkmanager --update

RUN chmod -R 777 ${ANDROID_HOME} && chown -R jenkins:jenkins ${ANDROID_HOME}

```

We also use a Jenkinsfile that works similarly with the previous one for building the master. Whenever we make changes to the Dockerfile, the agent will rebuild the image. This gives enormous flexibility with our CI/CD infrastructure.

Use of Autoscaling

Although we have assigned a specific number of nodes for deployments, we can do even more by enabling cluster autoscaling. This means that in cases of increased workloads and spikes, we can spin up extra nodes to handle the jobs. Currently, if we have a fixed number of nodes, we can only handle a fixed number of jobs. This is roughly estimated taken the fact that the minimum resources for each slave assigned typically 500ms CPU and 256MB memory, and setting a concurrency limit too high might not be respected at all.

This can happen when, for example, we have a major release cut and we need to deploy lots of microservices. Then a storm of jobs will pile up the pipeline, imposing significant delays.

In such cases we can increase the number of nodes for that phase. For example, we can add extra VM instances in the pool, and at the end of the process we remove them.

We can do this from the command line using the auto scaling option for configuring either the [Vertical](#) or [Cluster](#) autoscaling option. (For Azure and GC users, the links are [here](#) and [here](#).) However, this approach requires careful planning and configuration, as sometimes the following will happen:

- An increased number of jobs reach a plateau.
- Autoscaler adds new nodes, but takes 10 minutes to deploy and assign.
- Old jobs will have finished by now and new jobs fill the gap, lowering the need for new nodes.
- The new nodes are available, but remain stale and underutilized for at least X amount of minutes defined by the [--scale-down-unneeded-time](#) flag.
- Same effect happens multiple times per day.

In that case it's better if we either configure it based on our specific needs, or just increase the nodes for that day and revert them back when the process is finished. This all has to do with finding the optimal way to utilize all resources and minimize costs.

In either case we have a scalable and simple-to-use Jenkins cluster in place. For each job, a pod gets created to run a specific pipeline, and gets destroyed after it is done.

Best Practices to use Kubernetes for CI/CD at scale

Now that we have learned which CI/CD platforms exist for Kubernetes and how to install one, we will discuss some of the ways to operate them at scale.

To begin with, the choice of Kubernetes Distribution is one of the most critical factors that we need to take into account. Finding the most suitable one needs to be an informed, technical decision, as it's not trivial to find good compatibility between them. One example of a solid production-grade Kubernetes distribution is from [Platform9](#), as it offers first-class solutions that cater to different needs like [devops-automation](#) and [hybrid clouds](#).

Second, the choice of a Docker registry and package management is equally important. We are looking for secure and reliable image management that can be retrieved on-demand, as quickly as possible. Using Helm as a package manager is a sound choice as it's the best way to discover, share, and use software built for Kubernetes.

Third, using modern integration flows such as GitOps and ChatOps offer significant benefits in terms of ease of use and predictability. Using Git as a single source of truth allows us to run "operations by pull requests," which simplifies control of the infrastructure and application deployments. Using team chat tools such as Slack or Mattermost for triggering automated tasks for CI/CD pipelines assist us in eliminating repetitiveness, and simplifies integration.

Overall, if we want to go more low-level, we could customize and develop our own [Kubernetes Operator](#) that works more closely with Kubernetes API. The benefits of using a custom operator are numerous, as they aim to build a greater automation experience.

As a final word, we could say that Kubernetes and CI/CD platforms form a great match and it's the recommended way to run them. If you are just entering the Kubernetes ecosystem, then the first task you could try to integrate is a CI/CD pipeline. This is a good way to learn more about its inner workings. The key is to allow room for experimentation, which can make it easier to change in the future.

Conclusion

Building an efficient, reliable and scalable Kubernetes cluster is – to put it simply – hard work. There is no single “best” approach to designing a Kubernetes architecture or determining where to host your cluster or clusters. The best strategy for your team depends on a range of factors, including which use cases you are aiming to support, which type of infrastructure is available to you and how much performance matters relative to factors such as scalability.

So, before you go and spin up a Kubernetes cluster using whichever tool or hosting environment is within easiest reach, evaluate all of the different options you have in designing and implementing your Kubernetes deployment. It’s only through careful design and management that you’ll build the architecture best suited to your needs.



PLATFORM9

Freedom in Cloud Computing

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

Sign up today: platform9.com/signup