Operating Kubernetes: Everything You Need to Know

Platform9 Kubernetes Journey eBook series

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

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

Table of Contents

Introduction

If planning a Kubernetes installation is a challenge, operating Kubernetes once it is up and running is more difficult still. That is due in no small part to the fact that Kubernetes is not just one tool, but a collection of a dozen-odd components that provide functionality ranging from application deployments and upgrades, to logging and monitoring, to persistent data storage.

What's more, the specific tools used to provide each of these areas of functionality may vary from one Kubernetes instance to the next. Although some components are built into Kubernetes, the platform is also compatible with a range of different third-party solutions to address needs such as monitoring and data storage.

For both of these reasons, operating Kubernetes is more like flying a jet than driving a car: The teams in charge must master a range of different tools and processes, all tailored to whichever Kubernetes architectures and software stacks they are leveraging.

To help provide some guidance, this eBook walks through all stages of the key areas of functionality that teams typically must cover when operating Kubernetes. We won't discuss every tool that is available for these purposes, but we'll provide an overview of the main options. More importantly, we'll focus on the best practices that Kubernetes admins should follow when working with the various aspects of Kubernetes.

Combined with the other eBooks in this series, which focus on use cases for Kubernetes, Kubernetes architecture planning, and best practices for managing particularly complicated aspects of Kubernetes (such as security), this eBook provides the knowledge that teams need to take full advantage of Kubernetes, in all its complexity.

Kubernetes Monitoring Best Practices and Tooling Options

In any IT environment involving servers and virtual machines (VMs), application and system logs provide useful insights into the environment activities. They are helpful when dealing with tasks like debugging and activity monitoring, but the concept of monitoring and logging in containers is very different from traditional servers and VM monitoring.

Traditional monitoring applications were mostly monolithic, focusing on monitoring the infrastructure at bare metal or at the individual server level. In a microservices-based Kubernetes environment, however, your focus should be on the application level.

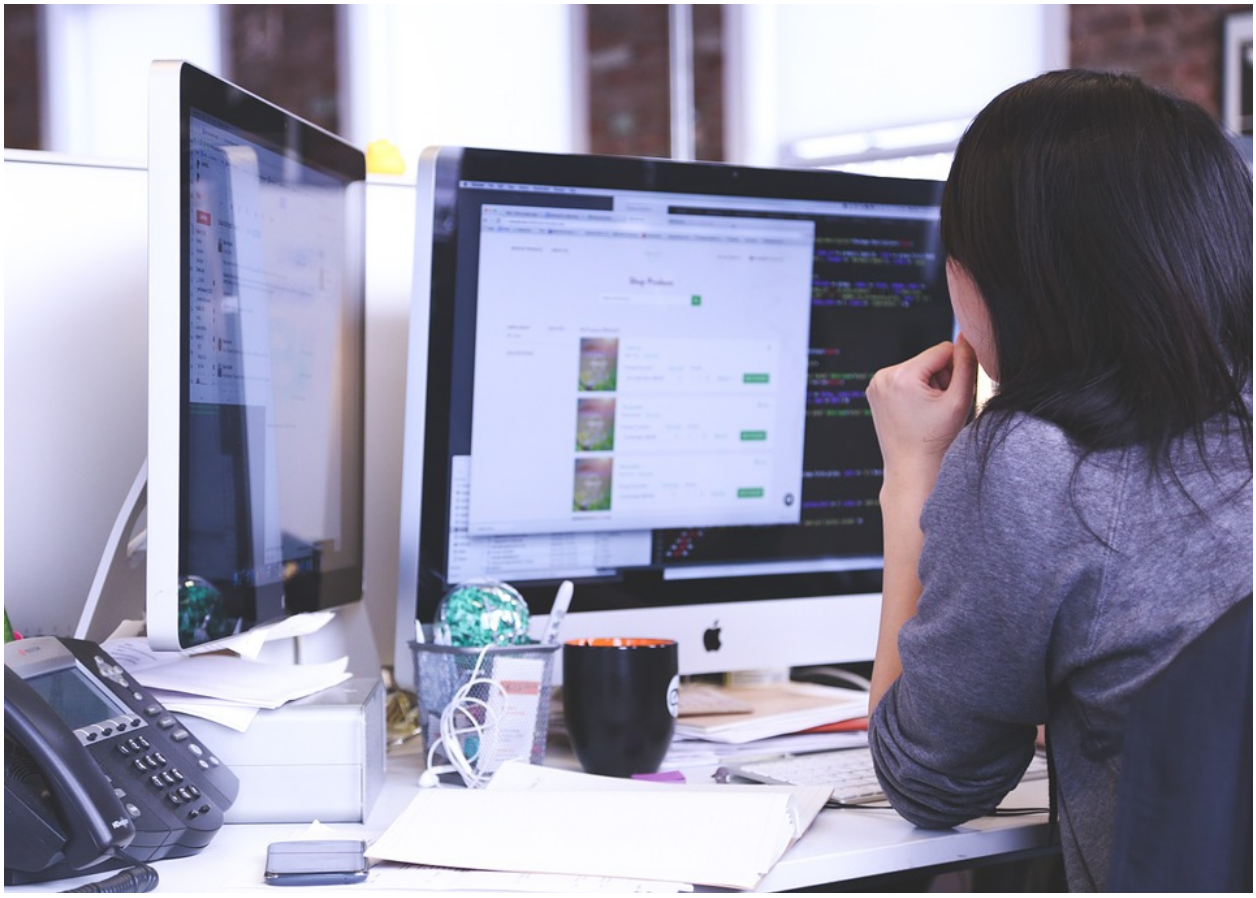
Most container engines support writing logs via standard output and standard error streams to enable the monitoring of hosted applications, but this native functionality is often insufficient because it lacks a few essential characteristics. For example, the availability of the logs is bound to the availability of the container pod itself, and if anything goes wrong with the container and it's unavailable online, the logs are often unavailable too. To ensure the availability of logs beyond a container's lifetime, you need to deploy a separate logging solution that is independent of nodes, pods, or containers.

Monitoring Kubernetes

The concept of cluster-level logging for Kubernetes containers requires independent storage of logs that will have their own independent lifecycle. Fortunately, Kubernetes provides ways to integrate with third-party logging solutions like Prometheus.

Prometheus is a monitoring and alerting toolkit, initially developed by SoundCloud in 2012. It is now a standalone open-source project maintained by an independent group of developers. In 2016, the Prometheus project was adopted by the Cloud Native Computing Foundation (CNCF), which is now responsible for the project's governance structure. Prometheus is just the second project hosted by CNCF, the first one being Kubernetes.

At its core, Prometheus is a multi-dimensional data model with support for time series data. It is comprised of the main Prometheus server, which collects and stores time-series data from its endpoints at a regular interval of time. The data is identified by metric name and key/value pairs and can be queried using the flexible query language called PromQL. There are special-purpose exporters which support services like HAProxy, StatsD, and Graphite. These exporters are modular and adaptable, enabling you to get started with specific monitoring requirements quickly.



Source: Pixabay.com

Best Practices for Multi-Tenancy

Multi-tenancy refers to the practice of enabling multiple users to share the same cluster for resource optimization and cost efficiency. With Kubernetes clusters, there are several ways of achieving multi-tenancy as well:

- Logical isolation using Namespaces - Use logical isolation (Namespaces) to separate all of the pods and resources of different user accounts. Ideally, the resources allocated to one namespace should not be able to access the resources in another namespace.
- Network policies for isolation - Have policies for all network activities at the lowest possible granular level. Clearly define the list of accepted IPs and allowed ports for the pods, and restrict all other communications.
- Use out-of-box applications - Use the built-in Managed Apps available out-of-the-box and avoid or minimize the use of third-party apps. Restrict all the ingress activities and permit the essential egress activities to the internet, but not to the cluster.
- Restrict cluster-level changes from within a pod - Do not allow any user to spin-up new pods, change the resource allocations, or even delete other pods from within a pod. For this, you should prevent Kubernetes API access by restricting the mounting of token to each pod.
- Strict resource constraints - Have limitations on the resources allocated for each pod based on the actual resource requirements. If any pod is trying to churn more resources, take appropriate actions like sending automated alerts to the admin of that pod, or killing the container or the application consuming the resources beyond limits. This can help ensure that neighboring pods are not impacted.
- Setup auto-scaling to avoid any overload situation - Autoscaling can help ensure that you do not run

out of resources due to any resource-hungry neighbor in the tenant pool.

- Have strong security policies - Define and enforce strong security policies, like setting cgroups to configure kernel-level limitations on resources and preventing an open-internet exposure of the pods hosting critical applications.

In addition, you should also consider the following best practices when using Prometheus:

- Use labels instead of multiple matrices - When tracking a set of similar tasks, instead of creating multiple matrices, use a single metric with supporting labels. For instance, when tracking the HTTP response count, instead of creating a separate matrix for each HTTP response (like *http_responses_500*, *http_responses_403* etc.), create a single matrix (*http_responses)* and use a label for the codes. In general, all metrics names should be static, and no name should be generated procedurally. At the same time, the overuse of labels should be avoided, because labelset is an additional time series that costs RAM, CPU, disk, and network resources. In general, any metric having cardinality more than 100 should be reconsidered for reducing the number of dimensions or should be skipped from monitoring to avoid any overhead.
- Use Timestamp, not Time Since - For doing time-lapse calculations, always use the Unix timestamp of the activity and perform the difference calculations instead of directly fetching the values for 'time since'.
- Optimal use of Instrumentation - Due to the high cost of resources associated with instrumentation, carefully consider how many matrices you update each second, especially for performance-critical or overused applications (having more than 100k function calls per second).

Common Dashboards/Events for Different Types of Applications

Kubernetes lacks several enterprise capabilities such as monitoring, logging, storage, databases, CI-CD pipelines, and visualizations. Although there is support for third-party integrations, this often means additional overhead for the DevOps Teams. They need to learn every new app, including its configurations, backup plans, APIs, and storage. This makes the overall management complicated.

Source: *Grafana.com*

Platform9 Managed Apps offers essential additional applications like Prometheus, EFK, and MySQL out-of-the-box. Platform9 also provides a single console for managing the entire technology stack with a hybrid cloud experience.

You can plug your existing environments into Platform9's SaaS-managed control panel for better administration.

These dashboards can be used for both stateful applications (which store data in persistent storage) as well as stateless applications (in which data is not stored on the device, but stays with the client). For instance, one dashboard can show a stateful MySQL application status with multiple slaves running asynchronous replication, while another dashboard may show the corresponding stateless frontend applications.

Platform9's continuous management technology automatically brings all of these environments under a centrally managed console, and it also enables automatic handling of operational tasks like ongoing monitoring, disaster recovery, and upgrades.

Enterprise-Grade Experiences

Besides better management of hosted applications, Managed Apps also enables you to perform overall administrative tasks like scaling up the cluster to increase its capacity. Infrastructure-as-a-Service (IaaS) platforms can be used to spin up new pods quickly via programmatic code without adding any operational overhead of separate monitoring or upgrades.

Platform9's Managed Kubernetes is infrastructure agnostic and supports public clouds as well as on-premises server infrastructure. Its out-of-the-box support for IP and IP encapsulation also allows uniform deployment across public as well as private clouds.

All the apps in the Managed Apps catalog are bound to the SLAs provided by Platform9. It also offers several additional capabilities like integration with other complementary managed applications, auto-scaling, auto-backup/recovery and more. You can set up each Managed App per specific clusters/namespaces or as a multi-tenant service across all clusters.

Source: Pixabay.com

Chapter Summary

To monitor Kubernetes pods, organizations require an entirely different mindset from the traditional VM and physical server monitoring practices. With Kubernetes, the monitoring practices need to be focused at the application level rather than the infrastructure level. Platform9 offers fully managed services supported with enterprise-grade SLAs so that your DevOps teams do not have to deal with the complexities and overhead of managing applications and their dependencies.

Updating Kubernetes Clusters: A Do-It-Yourself Guide

The process of installing a High Availability (HA) Kubernetes cluster on premises or in the Cloud is well documented and, in most cases, we don't have to perform many steps. There are additional tools like Kops or Kubespray that offer automation in this process.

Every so often, though, we are required to upgrade the cluster to keep up with the latest security features and bug fixes, as well as benefit from additional features being released on an on-gonig basis. This is especially important when we have installed a really outdated version (for example v1.9), or if we want to automate the process and always be on top of the latest supported version.

In general, when operating an HA Kubernetes Cluster, the upgrade process involves two separate tasks that may not overlap or be performed simultaneously: upgrading the Kubernetes Cluster; and, if needed, the etcd cluster, which is the distributed key-value backing store of Kubernetes. Let's see how we can perform those tasks with minimal disruptions.

Kubernetes Upgrade Paths

Note that this upgrade process is specifically for manually installing Kubernetes in the Cloud or on premises. It does not cover Managed Kubernetes Environments (like our own, where Upgrades are automatically handled by the platform), or Kubernetes services on public clouds (such as AWS' EKS or Azure Kubernetes Service), which have their own upgrade process. For the purposes of this tutorial, we assume that a healthy 3-node Kubernetes and Etcd Clusters have been provisioned. I've setup mine using six DigitalOcean Droplets (three Kubernetes masters plus three etcd nodes) plus one for the worker node.

Let's say that we have the following Kubernetes master nodes all running **v1.13**:

Name	Address	Hostname
kube-1	10.0.11.1	kube-1.example.com
kube-2	10.0.11.2	kube-2.example.com
kube-3	10.0.11.3	kube-3.example.com

Also, we have one worker node running **v1.13**:

Name	Address	Hostname
worker	10.0.12.1	worker..example.com

The process of upgrading the Kubernetes master nodes is documented on the Kubernetes documentation site. The following are the current paths:

- Upgrade from v1.12 to v1.13 HA

- Upgrade from v1.12 to v1.13
- Upgrade from v1.13 to v1.14
- Upgrade from v1.14 to v1.15

There is only one documented version for HA Clusters here, but we can reuse the steps for the other upgrade paths. In this example, we are going to see an upgrade path from v1.13 to v.1.14 HA. Skipping a version- for example, upgrading from v1.13 to v.1.15 - is not recommended.

Before we start, we should always check the release notes of the version that we intend to upgrade, just in case they mention breaking changes.

Upgrading Kubernetes: A Step-by-Step Guide

Let's follow the upgrade steps now:

Step 1: Login into the first node and upgrade the kubeadm tool only:

```
$ ssh admin@10.0.11.1
$ apt-mark unhold kubeadm && \
$ apt-get update && apt-get install -y kubeadm=1.13.0-00 && apt-mark hold
kubeadm
-
```

The reason why we run `apt-mark unhold` **`and`** `apt-mark hold` is because if we upgrade `kubeadm` then the installation will automatically upgrade the other components like kubelet to the latest version (which is v1.15) by default, so we would have a problem. To fix that, we use **hold** to mark a package as held back, which will prevent the package from being automatically installed, upgraded, or removed.

Step 2: Verify the upgrade plan:

```
$ kubeadm upgrade plan
...

COMPONENT            CURRENT   AVAILABLE
API Server           v1.13.0   v1.14.0
Controller Manager   v1.13.0   v1.14.0
Scheduler            v1.13.0   v1.14.0
Kube Proxy           v1.13.0   v1.14.0
...
```

Step 3: Apply the upgrade plan:

```
$ kubeadm upgrade plan apply v1.14.0
```

Step 4: Update Kubelet and restart the service:

```
$ apt-mark unhold kubelet && apt-get update && apt-get install -y
kubelet=1.14.0-00 && apt-mark hold kubelet
$ systemctl restart kubelet
```

Step 5: Apply the upgrade plan to the other master nodes:

```
$ ssh admin@10.0.11.2
```

```
$ kubeadm upgrade node experimental-control-plane

$ ssh admin@10.0.11.3

$ kubeadm upgrade node experimental-control-plane
```

Step 6: Upgrade kubectl on all master nodes:

```
$ apt-mark unhold kubectl && apt-get update && apt-get install -y
kubectl=1.14.0-00 && apt-mark hold kubectl
```

Step 7: Upgrade kubeadm on first worker node:

```
$ ssh worker@10.0.12.1

$ apt-mark unhold kubeadm && apt-get update && apt-get install -y
kubeadm=1.14.0-00 && apt-mark hold kubeadm
```

Step 8: Login to a master node and drain first worker node:

```
$ ssh admin@10.0.11.1

$ kubectl drain worker --ignore-daemonsets
```

Step 9: Upgrade kubelet config on worker node:

```
$ ssh worker@10.0.12.1

$ kubeadm upgrade node config --kubelet-version v1.14.0
```

Step 10: Upgrade kubelet on worker node and restart the service:

```
$ apt-mark unhold kubelet && apt-get update && apt-get install -y
kubelet=1.14.0-00 && apt-mark hold kubelet

$ systemctl restart kubelet
```

Step 11: Restore worker node:

```
$ ssh admin@10.0.11.1

$ kubectl uncordon worker
```

Step 12: Repeat steps 7-11 for the rest of the worker nodes.

Step 13: Verify the health of the cluster:

```
$ kubectl get nodes
```

Etcd Upgrade Paths

As you already know, etcd is the highly distributed key-value backing store for Kubernetes, and it's essentially the point of truth. When we are running an HA Kubernetes cluster, we also want to run an HA etcd cluster because we want to have a fallback just in case some nodes fail.

Typically, we would have a **minimum of 3 etcd nodes** running with the latest supported version. The process of upgrading the etcd nodes is documented in the etcd repo. These are the current paths:

- Upgrade from 2.3 to 3.0
- Upgrade from 3.0 to 3.1
- Upgrade from 3.1 to 3.2
- Upgrade from 3.2 to 3.3
- Upgrade from 3.3 to 3.4
- Upgrade from 3.4 to 3.5

When planning for etcd upgrades, you should always follow this plan:

- Check which version you are using. For example:

```
$ ./etcdctl endpoint status
```

- Do not jump more than one minor version. For example, do not upgrade from 3.3 to 3.5. Instead, go from 3.3 to 3.4, and then from 3.4 to 3.5.
- Use the bundled Kubernetes etcd image. The Kubernetes team bundles a custom etcd image located here which contains etcd and etcdctl binaries for multiple etcd versions as well as a migration operator utility for upgrading and downgrading etcd. This will help you automate the process of migrating and upgrading etcd instances.

Out of those paths, the most important change is the path from 2.3 to 3.0, as there is a major API change which is documented here. You should also take note that:

- Etcd v3 is able to handle requests for both the v2 and v3 data. For example, we can use the **ETCDCTL_API** env variable to specify the API version:

```
$ ETCDCTL_API=2 ./etcdctl endpoint status
```

- Running etcd v3 against the v2 data dir doesn't automatically upgrade the data dir to the v3 format.
- Using v2 api against etcd v3 only updates the v2 data stored in etcd.

You may also wonder which versions of Kubernetes have support for each etcd version. There is a small section in the documentation which says:

- **Kubernetes v1.0**: supports etcd2 only
- **Kubernetes v1.5.1**: etcd3 support added, new clusters still default to etcd
- **Kubernetes v1.6.0**: new clusters created with kube-up.sh default to etcd3, and kube-apiserver defaults to etcd3
- **Kubernetes v1.9.0**: deprecation of etcd2 storage backend announced
- **Kubernetes v1.13.0**: etcd2 storage backend removed, kube-apiserver will refuse to start with --storage-backend=etcd2, with the message etcd2 is no longer a supported storage backend

So, based on that information, if you are running Kubernetes **v1.12.0 with** etcd2, then you are required to

upgrade etcd to v3 when you upgrade Kubernetes to **v1.13.0 as --storage-backend=etcd3** is not supported. If you have Kubernetes **v1.12.0** and below, you can have both etcd2 and etcd3 running.

Before every step, we should always perform basic maintenance procedures such as periodic snapshots and periodic smoke rollbacks. Make sure to check the health of the cluster:

Let's say we have the following etcd cluster nodes:

Name	Address	Hostname
etcd-1	10.0.1.1	etcd-1.example.com
etcd-2	10.0.1.2	etcd-2.example.com
etcd-3	10.0.1.3	etcd-3.example.com

```
$ ./etcdctl cluster-health
member 6e3bd23ae5f1eae2 is healthy: got healthy result from
http://10.0.1.1:22379
member 924e2e83f93f2565 is healthy: got healthy result from
http://10.0.1.2:22379
member 8211f1d0a64f3269 is healthy: got healthy result from
http://10.0.1.3:22379
cluster is healthy
```

Upgrading etcd

Based on the above considerations, a typical upgrade etcd procedure consists of the following steps:

Step 1: Login to the first node and stop the existing etcd process:

```
$ ssh 10.0.1.1
$ kill `pgrep etcd`
```

Step 2: Backup the etcd data directory to provide a downgrade path in case of errors:

```
$ ./etcdctl backup \
      --data-dir %data_dir% \
      [--wal-dir %wal_dir%] \
      --backup-dir %backup_data_dir%
      [--backup-wal-dir %backup_wal_dir%]
```

Step 3: Download the new binary taken from etcd releases page and start the etcd server using the same configuration:

```
ETCD_VER=v3.3.15
```

```
# choose either URL
GOOGLE_URL=https://storage.googleapis.com/etcd
GITHUB_URL=https://github.com/etcd-io/etcd/releases/download
DOWNLOAD_URL=${GOOGLE_URL}

rm -f /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz
rm -rf /usr/local/etcd && mkdir -p /usr/local/etcd

curl -L ${DOWNLOAD_URL}/${ETCD_VER}/etcd-${ETCD_VER}-linux-amd64.tar.gz -o
/tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz
tar xzvf /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz -C /usr/local/etcd --strip-
components=1
rm -f /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz

/usr/local/etcd/etcd --version
ETCDCTL_API=3 /usr/local/etcd/etcdctl version
# start etcd server
/usr/local/etcd/etcd -name etcd-1 -listen-peer-urls http://10.0.1.1:2380 -
listen-client-urls http://10.0.1.1:2379,http://127.0.0.1:2379 -advertise-
client-urls http://10.0.1.1:2379,http://127.0.0.1:2379
```

Step 4: Repeat step 1 to step 3 for all other members.

Step 5: Verify that the cluster is healthy:

```
$ ./etcdctl endpoint health
10.0.1.1:12379 is healthy: successfully committed proposal: took =
10.0.1.2:12379 is healthy: successfully committed proposal: took =
10.0.1.3:12379 is healthy: successfully committed proposal: took =
```

Note: If you are having issues connecting to the cluster, you may need to provide HTTPS transport security certificates; for example:

```
$ ./etcdctl --ca-file=/etc/kubernetes/pki/etcd/ca.crt --cert-
file=/etc/kubernetes/pki/etcd/server.crt --key-
file=/etc/kubernetes/pki/etcd/server.key endpoint health
```

For convenience, you can use the following environmental variables:

```
ETCD_CA_FILE=/etc/kubernetes/pki/etcd/ca.crt
ETCD_CERT_FILE=/etc/kubernetes/pki/etcd/server.crt
ETCD_KEY_FILE=/etc/kubernetes/pki/etcd/server.key
```

Chapter Summary

In this chapter, we discussed step-by-step instructions on how to upgrade both Kubernetes and Etcd clusters. These are important maintenance procedures and eventualities for the day-to-day operations in a typical business environment. All participants who work with HA Kubernetes deployments should become familiar with the previous steps.

However, if you favor operational velocity and fewer maintenance tasks, you can consider using a fully managed Kubernetes deployment model like Platform9, which offers, among other things, zero-touch upgrades and no management overheads.

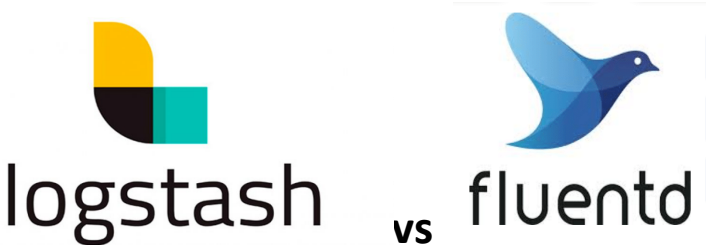
Kubernetes Logging: Comparing Fluentd and Logstash

Logging is an important part of the observability and operations requirements for any large-scale, distributed system. With Kubernetes being such a system, and with the growth of microservices applications, logging is more critical for the monitoring and troubleshooting of these systems, than ever before.

There are multiple log aggregators and analysis tools in the DevOps space, but two dominate Kubernetes logging: Fluentd and Logstash from the ELK stack.

Both log aggregators, Fluentd and Logstash, address the same DevOps functionalities, but are different in their approach, making one preferable to the other depending on your use case.

This chapter compares these log collectors against a set of critical features and capabilities. It also discusses which solution is preferable for different types of applications or environments.



The Logging Stack Components

Log analysis can't be done without log collectors. But to ensure the logging process is managed correctly, we need a logging stack. A logging stack is a set of components working together to ensure proper logging management.

Standard components of a logging stack are:

- a logs exporter (configure logs per host)
- a log collector listening for log input
- a logs storage medium
- a logs browser for visualization

As we already saw, Fluentd and Logstash are log collectors, how do they act in the logging stack?

Let's get acquainted with these tools, beginning with Logstash.

Logstash

Logstash is part of the popular ELK (logging stack), comprised of Elasticsearch, Logstash and Kibana.

Elasticsearch is the distributed and very scalable search engine. Raw data flows into Elasticsearch from different types of sources, including logs, system metrics, and web applications. *Data ingestion* is the process by which this raw data is parsed, normalized, and enriched before it is *indexed* in Elasticsearch. Once indexed in Elasticsearch, users can run queries against their data and use aggregations to retrieve summaries of their data. With Kibana, users can create powerful visualizations of their data, share dashboards, and manage the Elastic Stack.
Logstash is the ELK open-source data collection engine and it can do real-time pipelining. All components of Logstash are available under the Apache2 license.

Logstash can unify data from disparate sources dynamically and also normalize the data into destinations of your choice. Here is a great tutorial on configuring the ELK stack with Kubernetes.

Fluentd

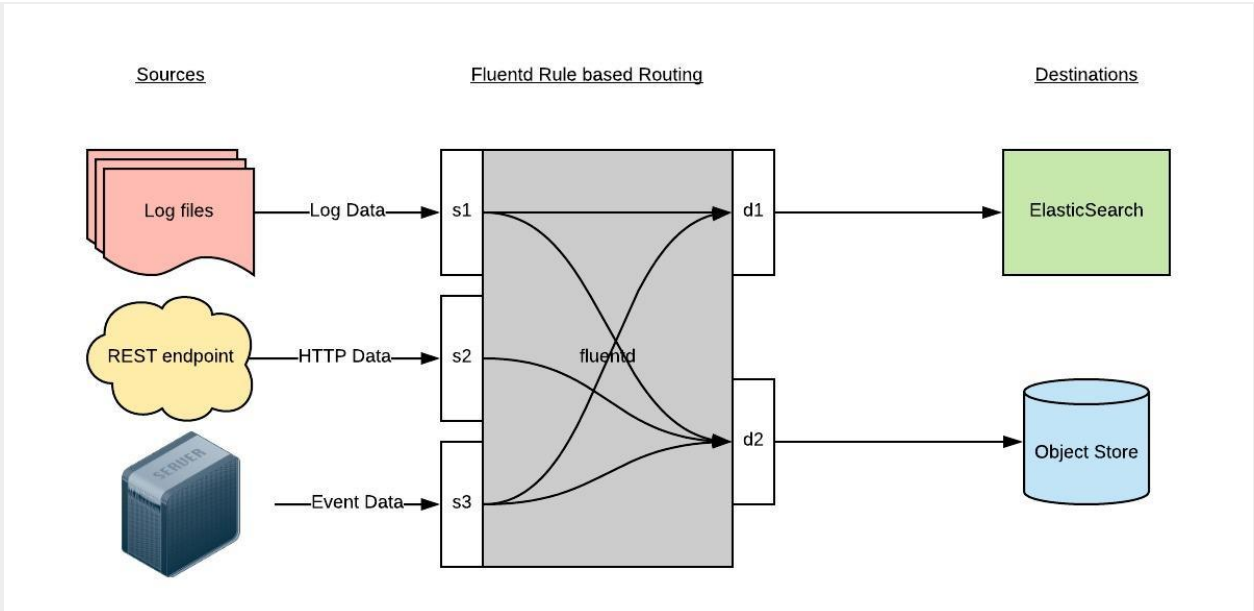
Fluentd is built by Treasure Data and is part of the CNCF foundation. All components of Fluentd are available under the Apache2 license.

Like Logstash, Fluentd is part of the ELK stack, also an open-source data collector, which lets you unify the data collection and consumption to allow a better use and insight of your data. Here is a great tutorial on configuring Fluentd with Kubernetes. Fluentd also works together with ElasticSearch and Kibana. This is known as the EFK stack.

How Does Fluentd Work?

Fluentd scraps logs from a given set of sources, processes them (converting into a structured data format) and then forwards them to other services like Elasticsearch, object storage etc. Fluentd is especially flexible when it comes to integrations – it works with 300+ log storage and analytic services.

- Fluentd gets data from multiple sources.
- It structures and tags data.
- It then sends the data to multiple destinations, based on matching tags



fluentd architecture

Source Configuration in Fluentd

For the purpose of this discussion, to capture all container logs on a Kubernetes node, the following source configuration is required:

```
<source>

@id fluentd-containers.log

@type tail

path /var/log/containers/*.log

pos_file /var/log/fluentd-containers.log.pos

time_format %Y-%m-%dT%H:%M:%S.%NZ

tag raw.kubernetes.*

format json

read_from_head true

</source>
```

1. id: A unique identifier to reference this source. This can be used for further filtering and routing of structured log data
2. type: Inbuilt directive understood by fluentd. In this case, “tail” instructs fluentd to gather data by tailing logs from a given location. Another example is “http” which instructs fluentd to collect data by using GET on http endpoint.
3. path: Specific to type “tail”. Instructs fluentd to collect all logs under /var/log/containers directory. This is the location used by docker daemon on a Kubernetes node to store stdout from running containers.
4. pos_file: Used as a checkpoint. In case the fluentd process restarts, it uses the position from this file to resume log data collection
5. tag: A custom string for matching source to destination/filters. fluentd matches source/destination tags to route log data

Routing Configuration in Fluentd

Lets look at the config instructing fluentd to send logs to Elasticsearch:

```
<match **>

@id elasticsearch

@type elasticsearch

@log_level info

include_tag_key true
```

```
type_name fluentd

host "#{ENV['OUTPUT_HOST']}"

port "#{ENV['OUTPUT_PORT']}"

logstash_format true

<buffer>

@type file

path /var/log/fluentd-buffers/kubernetes.system.buffer

flush_mode interval

retry_type exponential_backoff

flush_thread_count 2

flush_interval 5s

retry_forever

retry_max_interval 30

chunk_limit_size "#{ENV['OUTPUT_BUFFER_CHUNK_LIMIT']}"

queue_limit_length "#{ENV['OUTPUT_BUFFER_QUEUE_LIMIT']}"

overflow_action block

</buffer>
```

1. “match” tag indicates a destination. It is followed by a regular expression for matching the source. In this case, we want to capture all logs and send them to Elasticsearch, so simply use **
2. id: Unique identifier of the destination
3. type: Supported output plugin identifier. In this case, we are using ElasticSearch which is a built-in plugin of fluentd.
4. log_level: Indicates which logs to capture. In this case, any log with level “info” and above – INFO, WARNING, ERROR – will be routed to Elasticsearch.
5. host/port: ElasticSearch host/port. Credentials can be configured as well, but not shown here.
6. logstash_format: The Elasticsearch service builds reverse indices on log data forward by fluentd for searching. Hence, it needs to interpret the data. By setting logstash_format to “true”, fluentd forwards the structured log data in logstash format, which Elasticsearch understands.
7. Buffer: fluentd allows a buffer configuration in the event the destination becomes unavailable. e.g. If the network goes down or ElasticSearch is unavailable. Buffer configuration also helps reduce disk activity by batching writes.

Fluentd as Kubernetes Log Aggregator

To collect logs from a K8s cluster, fluentd is deployed as privileged daemonset. That way, it can read logs from a location on the Kubernetes node. Kubernetes ensures that exactly one fluentd container is always running on each node in the cluster. For the impatient, you can simply deploy it as helm chart with a command like this:

```
$ helm install stable/fluentd-elasticsearch
```

Comparing Logstash and Fluentd

Let's now compare the two tools against important DevOps features and capabilities.

Platform

Both tools run on both Windows and Linux

Event routing

Event routing is an important feature of a log collector. Logstash and Fluentd are different in their approach concerning event routing.

Logstash uses the if-else condition approach; this way we can define certain criteria with If..Then..Else statements - for performing actions on our data.

With Fluentd, the events are routed on tags. Fluentd uses tag based routing and every input (source) needs to be tagged. Fluentd then matches a tag against different outputs and then sends the event to the corresponding output.Tagging events is much easier than using if-then-else for each event type, so Fluentd has an advantage here.

Plugins

Contrary to Fluentd, the Logstash plugin ecosystem is centralized under a single GitHub repository. Fluentd has an official repository, but most of the plugins are hosted elsewhere.
It depends on the user's taste preference for how they want to manage and collect the plugins, from a centralized place (Logstash) or from several places (Fluentd). Efficiency wise, a centralized place is usually preferable.

For both tools, the plugins extend the tool's functionality. See Logstash GitHub for the central repo and here is also an example for a Fluentd plugin repo.

Transport

Inputs – like files, syslog and data stores – are used to get data into Logstash. Logstash is limited to an in-memory queue that holds 20 events and, therefore, relies on an external queue, like Redis, for persistence across restart. Often, Redis is facilitated as a "broker" in a centralized Logstash installation, queueing Logstash events from remote Logstash "shippers." This is not the case with Fluentd, which has a configurable in-memory or on-disk buffering system.

This means for Logstash you need a second party for getting data into Logstash, which adds another dependency to the system. This is not the case with Fluentd, which is independent in getting its data.

Regarding architecture, for Logstash you need an external party, further complicating the architecture and increasing the risk of failure, which is not the case with Fluentd. Fluentd therefore is 'safer' in se than Logstash regarding data transport.

Performance

This really depends on the situation. It is known that Logstash consumes more memory, but both tools have lightweight products: Elastic Beats vs Fluent-bit.

Enterprise Support

Both tools have vendors offering enterprise support for them, however Logstash is part of the ELK stack and, when used with ElasticSearch and Kibana, could have better enterprise support.
However, Fluentd is a CNCF project, and when used with Kubernetes (also CNCF), Fluentd could be a better choice.

Coding

Logstash can be coded with JRuby and Fluentd with CRuby. This means Fluentd has an advantage here, because no java runtime is required.

Event and Error Logging

Data logging can be divided in 2 areas: event and error logging.
Fluentd and Logstash can handle both logging types.

However, if the use case also requires Docker, Fluentd is more appropriate, because Docker has a built-in logging driver for Fluentd and not for Logstash. No extra agent is required on the container to push logs to Fluentd. Logs are directly shipped to Fluentd service from STDOUT and no extra log file is required. Contrary to Logstash where a plugin (eg. filebeat) is necessary to read the application logs from STDOUT before they can be sent to Logstash. Thus, when using Docker in a logging use case, Fluentd is the preferred candidate, it makes the architecture less complex and this makes it less risky for logging mistakes.

What if there are high volumes of data involved, which needs to be logged?

High Volume Logging

We already discussed the lightweight variants Elastic Beats and Fluent-bit. Let's have a look at Fluent-bit.

Fluentd uses Ruby and Ruby Gems for configuration of its 500+ plugins. Ruby is an interpreted language: it uses a lot of C extensions for parsing log files and forwarding data to provide the necessary speed. However, due to the volume of logs ingested, performance problems are expected, because a lot of C extensions (extra code next to Ruby!) is necessary
Fluent-bit is recommended when using small or embedded applications. Fluent-bit, in contrast to is implemented primarily in C. But it can provide all the functionality you need and meets performance expectations.

Elastic beats is the lightweight variant of Logstash, but most importantly, if your use case, next to data transport, also needs data aggregation - data pulling -, you need both Logstash and Elastic Beats.

Log Parsing

The components for log parsing are different per logging tool. Fluentd uses standard built-in parsers (JSON, regex, csv etc.) and Logstash uses plugins for this. This makes Fluentd favorable over Logstash, because it does not need extra plugins installed, making the architecture more complex and more prone to errors.

Metric Data Collection

Fluentd can't collect container metrics on its own; it can use the Prometheus exporter for this.
Logstash, as part of the ELK stack, also uses this exporter, next to MetricBeat.

Chapter Summary: Which Option is Best

Fluentd and Logstash can both be used for different use cases and can even co-exist in ecosystems; they can work for both VMs and legacy applications, as well as Kubernetes-based microservices.

Regarding Kubernetes, Fluentd is the ideal candidate because Kubernetes and Fluentd are both part of the CNCF. Fluentd has, contrary to Logstash, a built-in Docker logging driver and parsers, thus making it unnecessary for an extra agent to be present on the container to push logs to Fluentd. In comparison with Logstash, this makes the architecture less complex and also makes it less risky for logging mistakes.

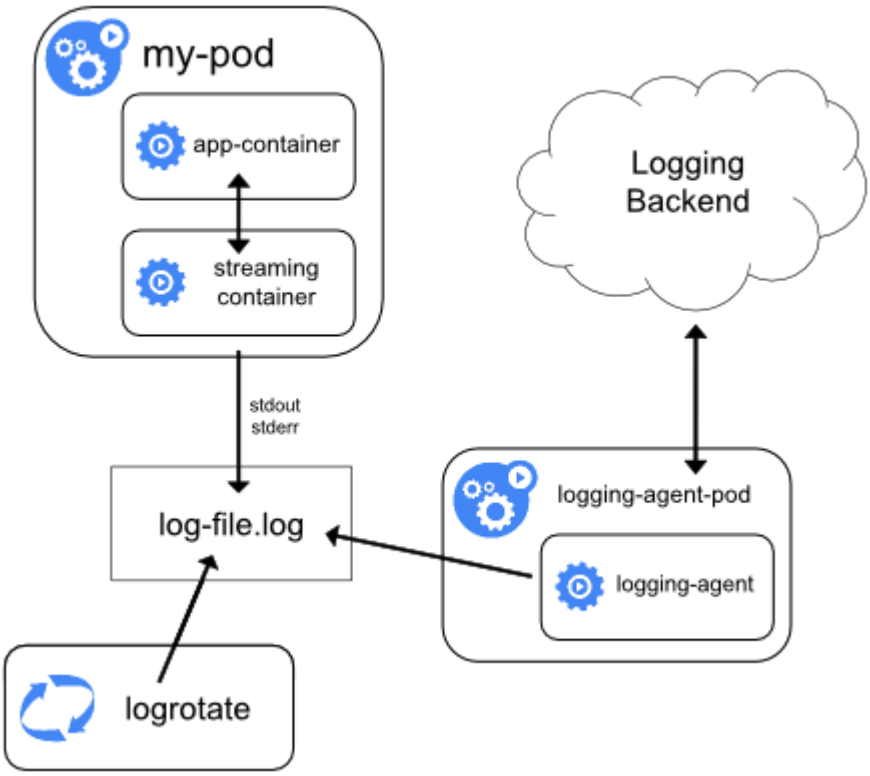
Overall, when it comes to logging for Kubernetes, Fluentd is the ideal candidate.

K8s Logging Best Practices

Kubernetes helps manage the lifecycle of hundreds of containers deployed in pods. It is highly distributed and its parts are dynamic. An implemented Kubernetes environment involves several systems with clusters and nodes that host hundreds of containers that are constantly being spun up and destroyed based on workloads.

When dealing with a large pool of containerized applications and workloads in Kubernetes, it is important to be proactive with monitoring and debugging errors. These errors are seen at the container, node, or cluster level. Kubernetes' logging mechanism is a crucial element to manage and monitor services and infrastructure. In the case of Kubernetes, logs allow you to track errors and even to fine-tune the performance of containers that host applications.

Configure stdout and stderr Streams



Source: Kubernetes

The first step is to understand how logs are generated. With Kubernetes, logs are sent to two streams – stdout and stderr. These streams of logs are written to a JSON file and this process is handled internally by Kubernetes. You can configure which logs you'd like to send to which stream. A best practice is to send all application logs to stdout and all error logs to stderr.

Decide Whether to Sidecar or Not to Sidecar

Kubernetes recommends using sidecar containers to collect logs. In this approach, every application container would have a neighboring 'streaming container' that streams all logs to stdout and stderr. The sidecar model helps to avoid exposing logs at the node level, and it gives you control over logs at the container level.

The problem with this model, however, is that it works well for low-volume logging, but at scale, it can be a resource drain. This is because you need to run a separate logging container for every application container that's running. The K8s docs say that this model is 'hardly a significant overhead.' It's up to you to try this model and see the kind of resources it consumes before opting for it.

The alternative is to use a logging agent that collects logs at the node level. This accounts for little overhead and ensures the logs are handled securely. Fluentd has emerged as the best option to aggregate Kubernetes logs at scale. It acts as a bridge between Kubernetes and any number of endpoints where you'd like to consume Kubernetes logs. Opting for a managed K8s service like Platform9 even gives you the ease of a fully-managed Fluentd instance without you having to manually configure or maintain it.

Once you've decided on Fluentd to better aggregate and route log data, the next step is to decide how you'll store and analyze the log data.

Pick Your Log Analysis Tool - EFK or Dedicated Logging

Traditionally, with on-prem server-centric systems, application logs are stored in log files located in the system. These files can be seen in a defined location or can be moved to a central server. But in the case of Kubernetes, all logs are sent to a JSON file on disk at `/var/log`. This type of aggregation of logs isn't safe because pods in the nodes can be temporary or short-lived. The log files would be lost when the pod is deleted. This can be an issue when trying to troubleshoot with part of the log data missing.

Kubernetes recommends two options: send all logs to Elasticsearch, or use a third-party logging tool of your choice. Here again, there is a choice to make. Going the Elasticsearch route means you buy into a complete stack – The EFK stack – that includes Elasticsearch, Fluentd, and Kibana. Each tool has its own role to play. As mentioned above, Fluentd aggregates and routes logs. Elasticsearch is the powerhouse that analyzes raw log data and gives out readable output. Kibana is an open-source data visualization tool that creates beautiful, custom-made dashboards from your log data. This is a completely open-source stack and is a powerful solution for logging with Kubernetes.

Still, there are things to keep in mind. Elasticsearch is built and maintained by an organization called Elastic, and a huge community of open source developers. While it is battle-tested to be blazing fast and very powerful at running queries on large scale data, it also has its quirks when operating at scale. Self-managed Elasticsearch needs someone who knows how to architect the platform for scale.

The alternative is to use a cloud-based log analysis tool to store and analyze Kubernetes logs. There are many examples of these tools like Sumo Logic and Splunk. Some of these tools leverage Fluentd to route logs to their platform while others may have their own custom logging agent that sits at the node level within Kubernetes. The setup for these tools is easy, and it takes the least amount of time to go from zero to viewing logs in beautiful dashboards.

Control Access to Logs with RBAC

The Authentication mechanism in Kubernetes uses role-based access control (RBAC) to validate a user's access and permissions with the system. The audit logs generated during the operation are annotated based on whether a user has privileges (authorization.k8s.io/decision) and a reason (authorization.k8s.io/reason) access is being given to the user. Audit logs are not activated by default. Activating it to track authentication issues is recommended and can be set up with 'kubectl'.

Keep Log Formats Consistent

Kubernetes logs are generated by different parts of the Kubernetes architecture. These aggregated logs should be in a consistent format so that it is easier for log aggregation tools like fluentd or FluentBit to process them. This should be kept in mind when configuring stdout and stderr, or when assigning labels and metadata using Fluentd, for example. Such structured logs, once provided to Elasticsearch, reduce latency during log analysis.

Set Resource Limits on Log Collection Daemons

With the high volume of logs generated, it can get hard to manage the logs at the cluster level. DaemonSet is used in Kubernetes in a similar way as Linux. It runs in the background to perform a specific task. Fluentd and filebeat are two daemons that Kubernetes supports for log collection. It is imperative to set up a resource limit per daemon so that the collection of log files will be optimized according to the available system resources.

Chapter Summary

Kubernetes contains multiple layers and components that should be monitored well and should be tracked. Kubernetes encourages logging with external 'Kubernetes Native' tools that integrate seamlessly to make logging easier for admins. The practices mentioned here are important to have a robust logging architecture that works well in any situation. They consume computing resources in an optimized way and keep the Kubernetes environment secure and performant.

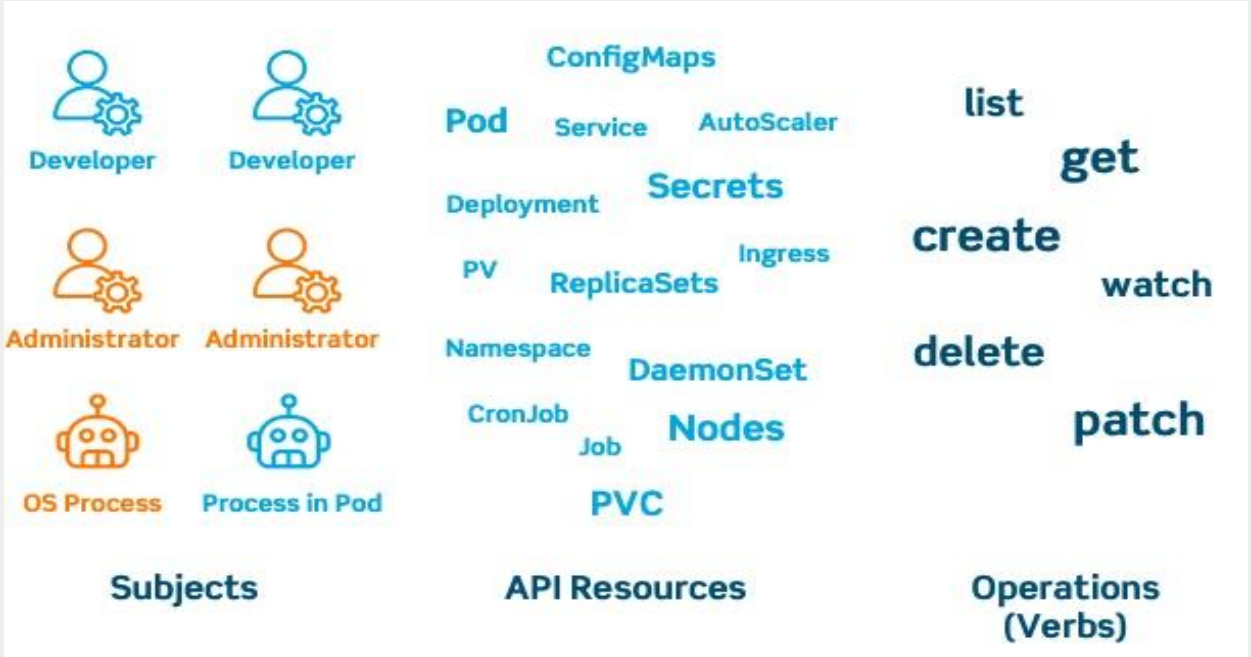
Kubernetes RBAC: Giving Users Access

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

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

RBAC in Kubernetes is Based on Three Key Concepts:

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



The types of Role Based Access Control used by Kubernetes.

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

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

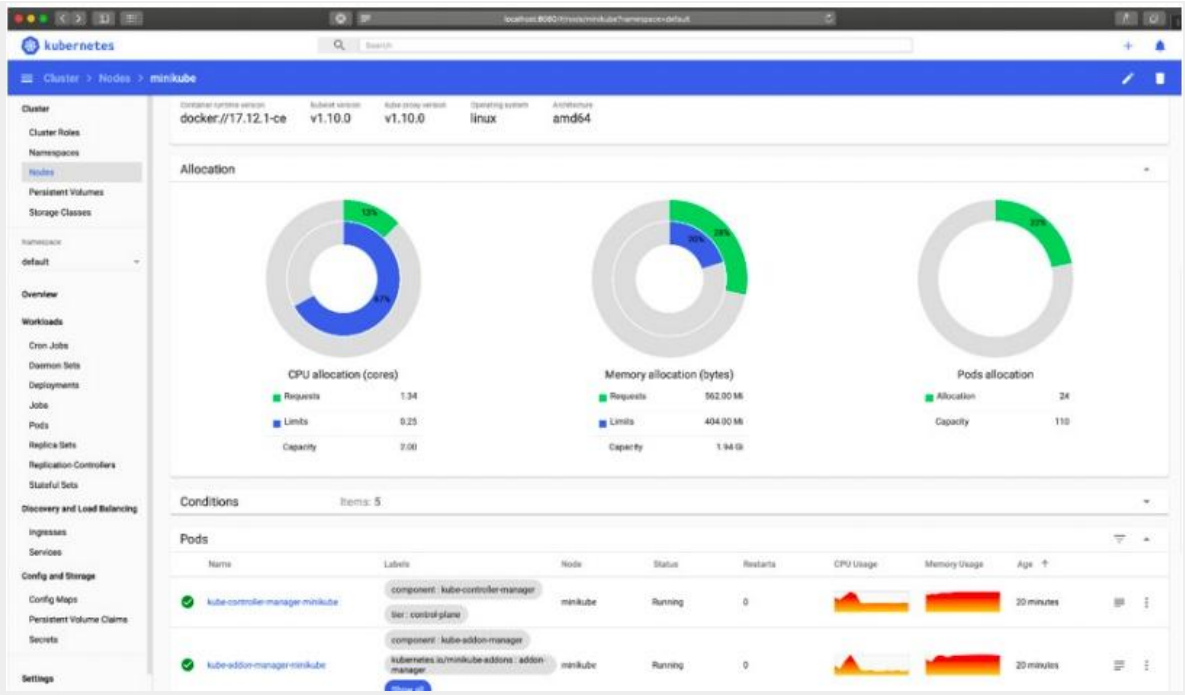
There are two kinds of users: service accounts managed by Kubernetes, and normal users. These normal users come from an identity store outside Kubernetes. This means that accessing Kubernetes with multiple users, or even multiple roles, is something that needs to be carefully thought out. Which identity source will you use? Which access control mode most suits you? Which attributes or roles should you define? For larger deployments, it's become standard to give each app a dedicated service account and launch the app with it. Ideally, each app would run in a dedicated namespace, as it's fairly easy to assign roles to namespaces.

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

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

Monitoring and Ensuring Cluster Health

The easiest way of manually checking your cluster after deployment is via the Kubernetes Dashboard. This is the default dashboard and is usually included in new clusters. The dashboard gives a graphical overview of resource usage, namespaces, nodes, volumes, and pods. The dashboard provides a quick and easy way to display information about the cluster. Because of its ease of use, it's usually the first step in a health check.



The Default Kubernetes Dashboard

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

applications using a wizard.

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

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

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

Monitoring and Ensuring Application Health

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

Kubernetes Cluster Monitoring

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

Monitoring tools look at infrastructure telemetry: compute, storage, and network. They look at (potential) bottlenecks in the infrastructure, such as processor and memory usage, or disk and network I/O. These resources are an important part of your monitoring strategy, as they're limited to the capacity procured, and costly to expand.

There's another important reason to study these metrics: they define the behavior of the infrastructure on which the applications run, and they can serve as an early warning sign of potential issues. Should

issues be identified, you can mitigate the issue before applications dependent on that infrastructure are impacted.

Kubernetes Pod Monitoring

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

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

This is where solutions like DataDog, NewRelic and AppD come in. While these are proprietary solutions, they cover the whole stack: from infrastructure, Kubernetes and containers, to application tracing. This provides a complete picture of an application, as well as transactional traces across the entire system for monitoring of the end-user experience. These solutions offer a unified metrics and monitoring experience and include rich dashboarding and alerting feature sets. Often, these products include default dashboard visualizations for monitoring Kubernetes, encompassing many standard integrations with components in the application stack to monitor up and down.

Datadog Kubernetes Dashboard Example

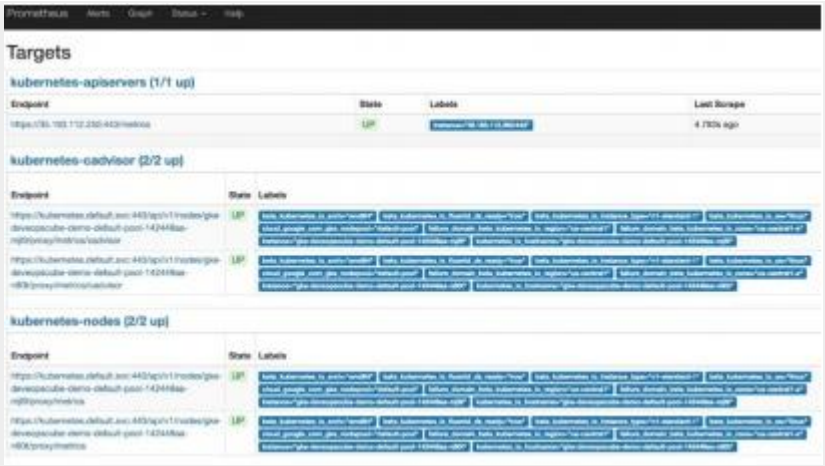
Kubernetes Monitoring with Prometheus

The most popular Kubernetes monitoring solution is the open source Prometheus, which can natively monitor the clusters, nodes, pods, and other Kubernetes objects.

It's easily deployed via kube-prometheus, which includes AlertManager for alerting, Grafana for dashboards, and Prometheus rules combined with documentation and scripts. It provides an easy to operate, end-to-end Kubernetes cluster monitoring solution.

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

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



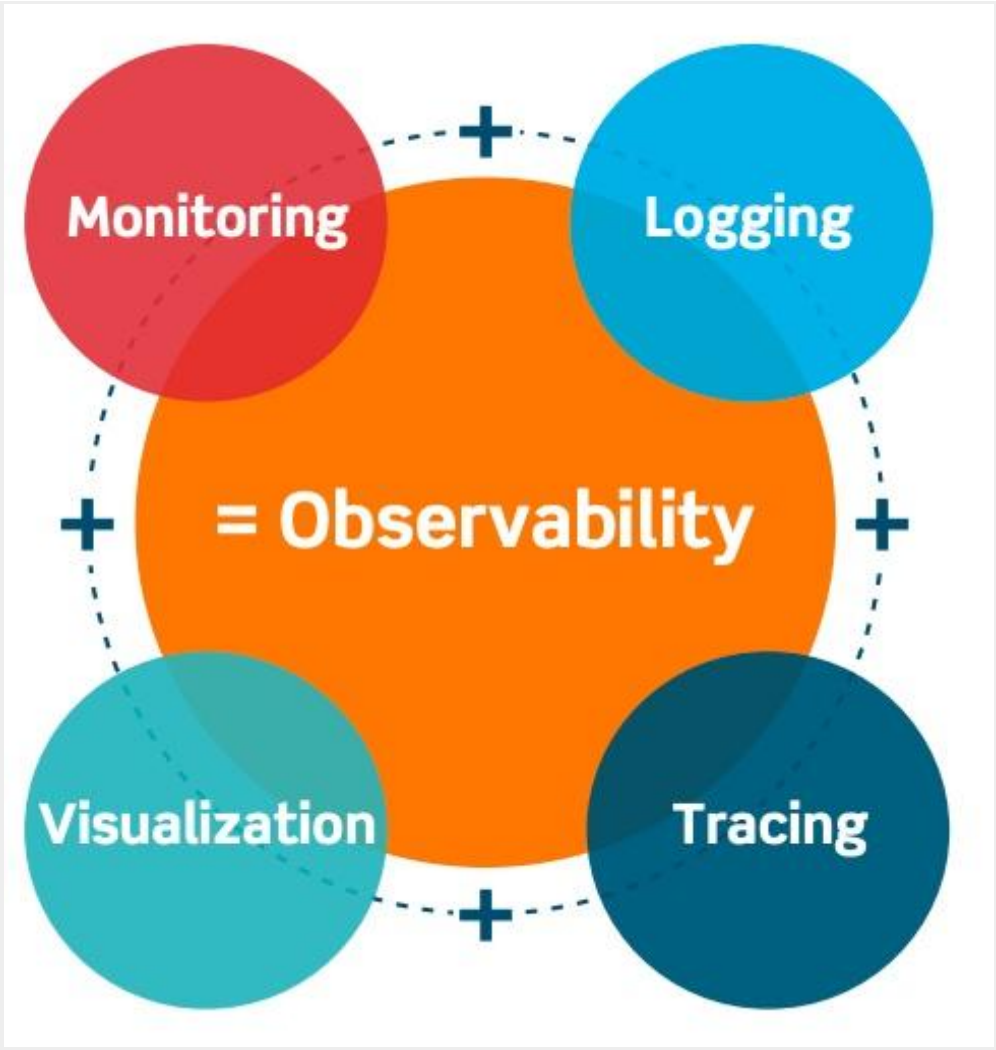
Prometheus Targets for Kubernetes

The Grafana Dashboard for Kubernetes

* BTW – If you don't want to go through the trouble setting up and managing Prometheus on your own, check out our Managed Prometheus solution for multi-tenant, out-of-the-box Prometheus monitoring with 99.9% SLA on any environment.

Kubernetes Logging and Tracing

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



The elements of observability

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

To dive deeper into Kubernetes Logging and monitoring, see our blog series on the EFK Stack.

Persistent Storage

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

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

To learn more about Dynamic Volumes and the Container Storage Interface (CSI) see our Kubernetes Storage blog, which also includes a Minikube tutorial so you can hack on the inner workings of storage in Kubernetes!

Chapter Summary

RBAC is a critical tool for helping to secure resources within a Kubernetes environment. When used in conjunction with proper cluster monitoring and logging tools, RBAC keeps Kubernetes running stably and securely, no matter what the scale of your deployments.

Using Elasticsearch with Kubernetes

Elasticsearch is a robust search engine and document store. Kibana is the UI companion of Elasticsearch, simplifying visualization and querying.

From the rich feature set of Elasticsearch, the following ones are most useful for log management:

- It is schema-less, and can store data from various log sources
- It can automatically guess data types, unlike traditional data stores, and does not require schema definitions.
- It can store very large datasets (Petabyte Scale)
- It can efficiently search across a very large dataset
- It scales horizontally, and can tolerate node failures.

As you might have guessed, Elasticsearch is the most complex piece in our EFK stack for Kubernetes log aggregation and monitoring solution.

Elasticsearch Architecture:

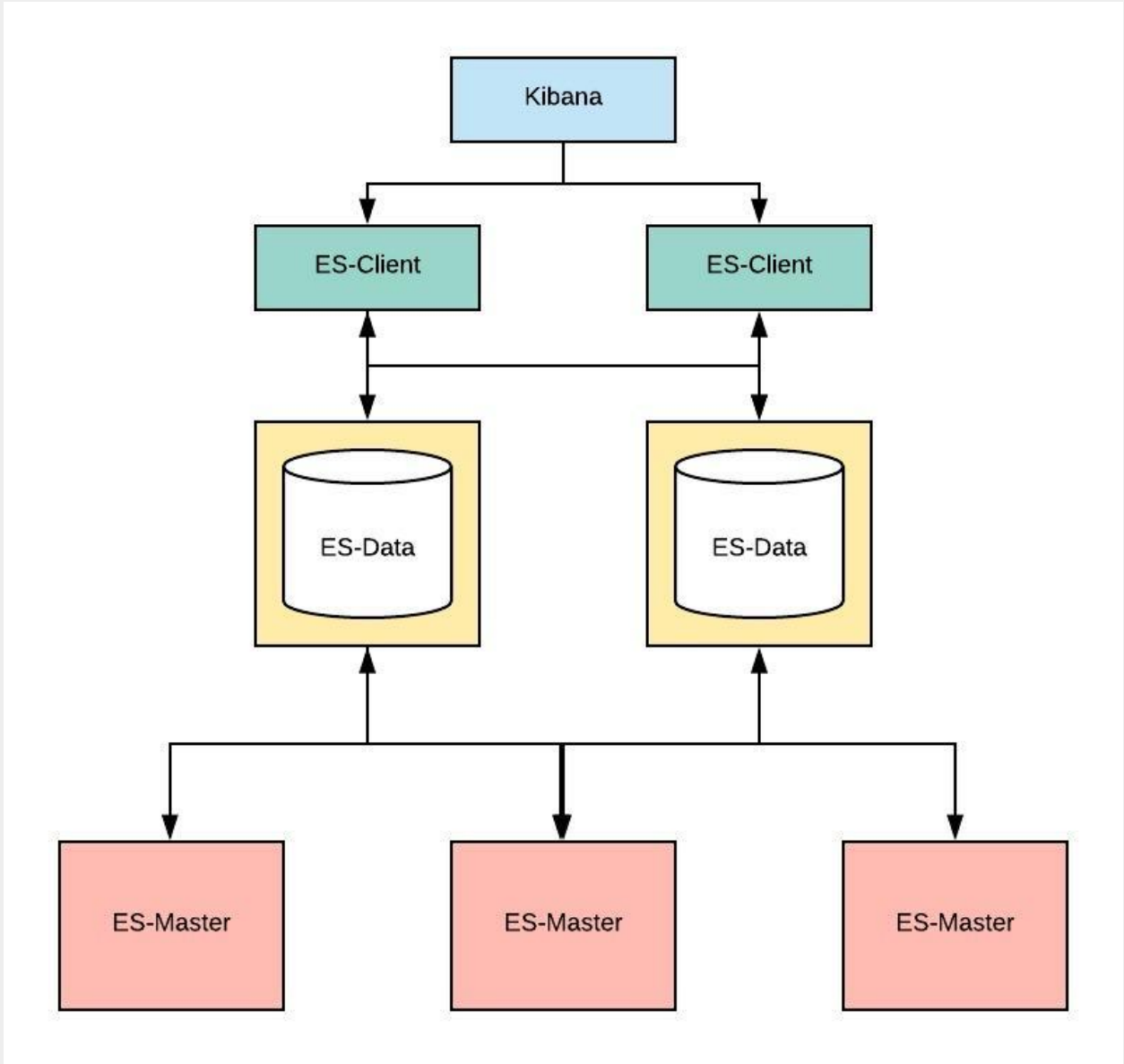
Let's review the Elasticsearch architecture and key concepts that are critical to the EFK stack deployment:

- Cluster: Any non-trivial Elasticsearch deployment consists of multiple instances forming a cluster. Distributed consensus is used to keep track of master/replica relationships.
- Node: A single Elasticsearch instance.
- Index: A collection of documents. This is similar to a database in the traditional terminology. Each data provider (like fluentd logs from a single Kubernetes cluster) should use a separate index to store and search logs. An index is stored across multiple nodes to make data highly available.
- Shard: Because Elasticsearch is a distributed search engine, an index is usually split into elements known as shards that are distributed across multiple nodes.(Elasticsearch automatically manages the arrangement of these shards. It also re-balances the shards as necessary, so users need not worry about these.)
- Replica: By default, Elasticsearch creates five primary shards and one replica for each index. This means that each index will consist of five primary shards, and each shard will have one copy.

Deployment:

ElasticSearch deployment consists of three node types:

- Client: These nodes provide the API endpoint and can be used for queries. In a Kubernetes-based deployment these are deployed a service so that a logical dns endpoint can be used for queries regardless of number of client nodes.
- Master: These nodes provide coordination. A single master is elected at a time by using distributed consensus. That node is responsible for deciding shard placement, reindexing and rebalancing operations.
- Data: These nodes store the data and inverted index. Clients query Data nodes directly. The data is sharded and replicated so that a given number of data nodes can fail, without impacting availability.



Elasticsearch Components

Installing Elasticsearch Cluster with Fluentd and Kibana

Now that we understand the basics of Elasticsearch, let us set up an Elasticsearch cluster with fluentd and Kibana on Kubernetes.

For this purpose, we will need a Kubernetes cluster with following capabilities.

- Ability to run privileged containers.
- Helm and tiller enabled.
- Statefulsets and dynamic volume provisioning capability: Elasticsearch is deployed as stateful set on Kubernetes. It's best to use latest version of Kubernetes (v 1.10 as of this writing)

Step by Step Guide:

- Make sure you have incubator repo enabled in helm:
  helm repo add incubator https://kubernetes-charts-incubator.storage.googleapis.com/
- Update repos:
  helm repo update
- As a good practice, lets keep logging services in their own namespace:
  kubectl create ns logging
- Install Elasticsearch
  helm install incubator/elasticsearch --namespace logging --name elasticsearch --set data.terminationGracePeriodSeconds=0
  Note that the install is customized. As Elasticsearch can use replicas, the individual processes can terminate immediately, without the risk of data loss.
- Install Fluentd
  helm install --name fluentd --namespace logging stable/fluentd-elasticsearch --set elasticsearch.host=elasticsearch-client.logging.svc.cluster.local,elasticsearch.port=9200
  Note that above command configured Fluentd so that it can send logs to right Elasticsearch endpoint. This deployment does not use explicit authentication. The fluentd-elasticsearch chart injects the right fluentd configuration so that it can pull logs from all containers in the Kubernetes cluster and forward them to ElasticSearch in logstash format.
- Install Kibana
  helm install --name kibana --namespace logging stable/kibana --set env.ELASTICSEARCH_URL=http://elasticsearch-client.logging.svc.cluster.local:9200,env.SERVER_BASEPATH=/api/v1/namespaces/logging/services/kibana/proxy
  Same as with fluentd, Kibana chart variables are set to point it to the deployed Elasticsearch. The SERVER_BASEPATH is a quirk in Kibana UI. When deployed on Kubernetes, we need to set it to an internal endpoint.

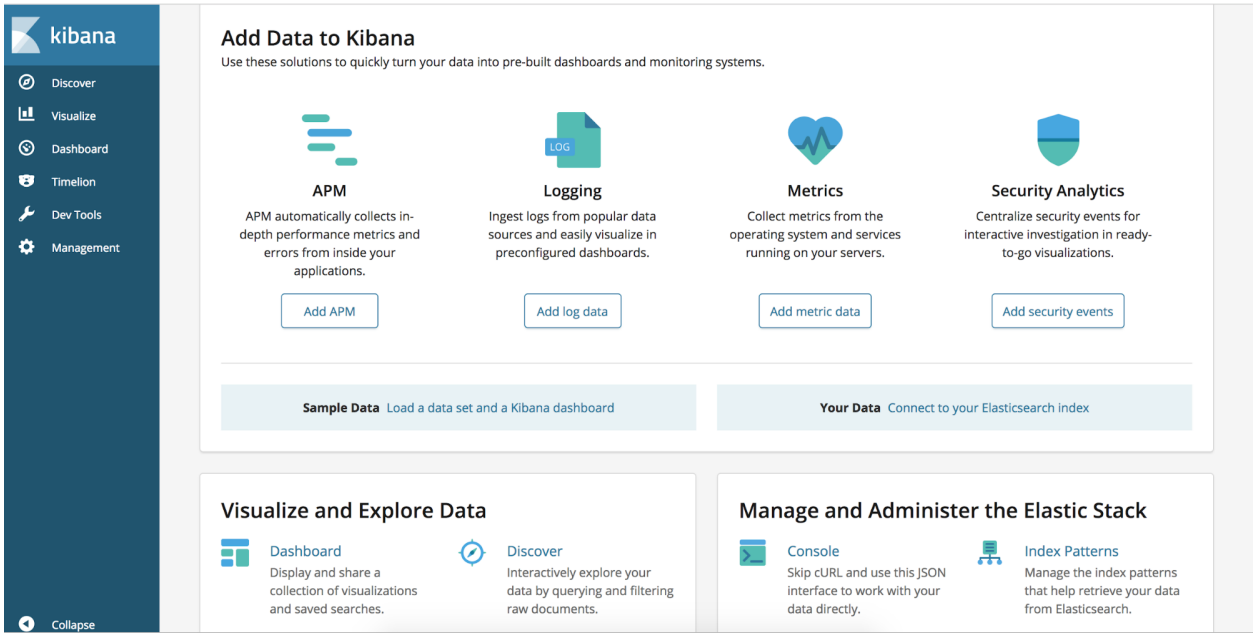
That's it!

Once these services come up, the Kibana UI can be accessed by forwarding the port of the Kibana pod to your machine or with a Kubernetes proxy command:

```
kubectl proxy 8001
```

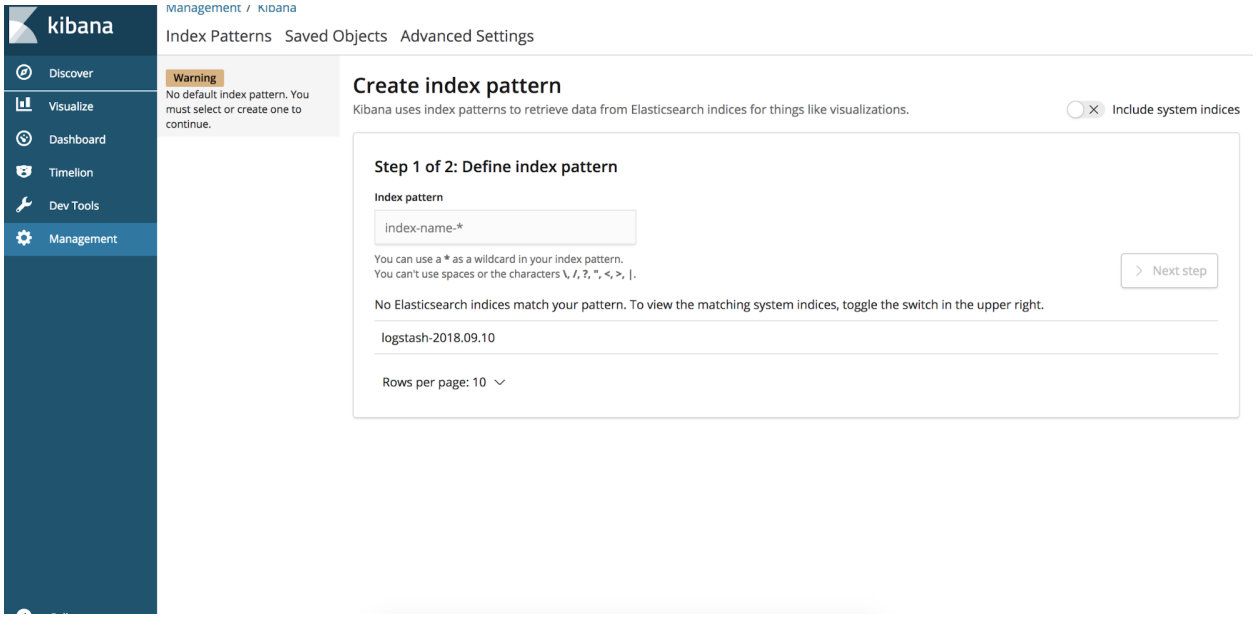
Then go to http://localhost:8001/api/v1/namespaces/logging/services/kibana:443/proxy/

You should see the following screen:

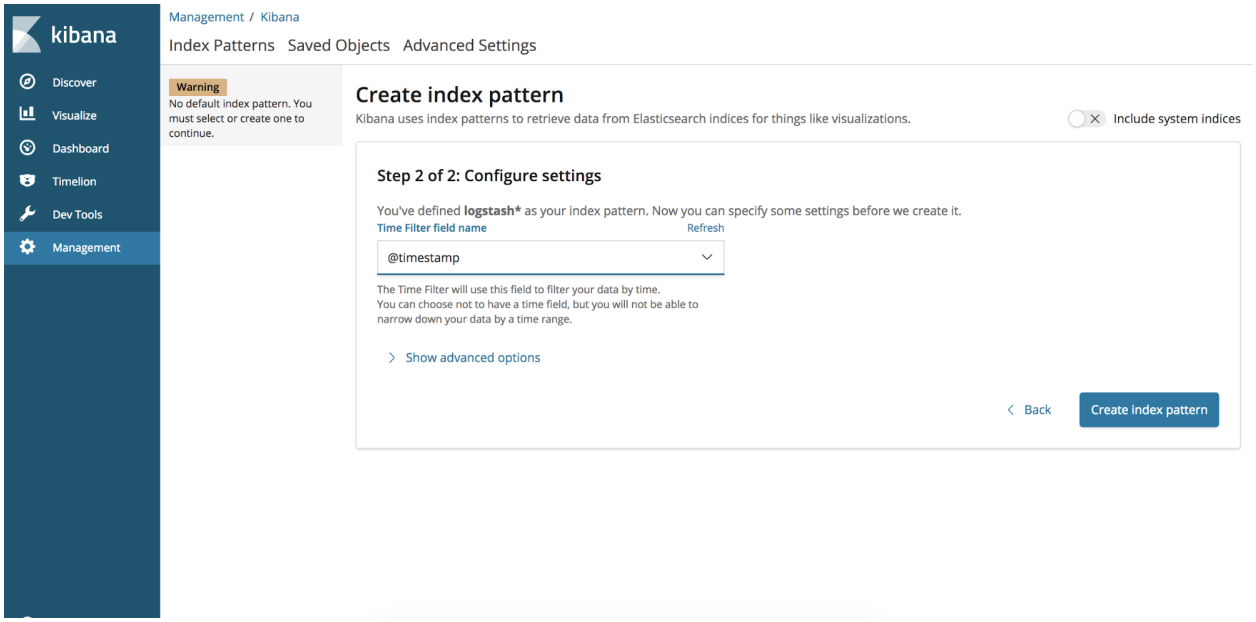


Now, Let's Add Data for Kibana to Display

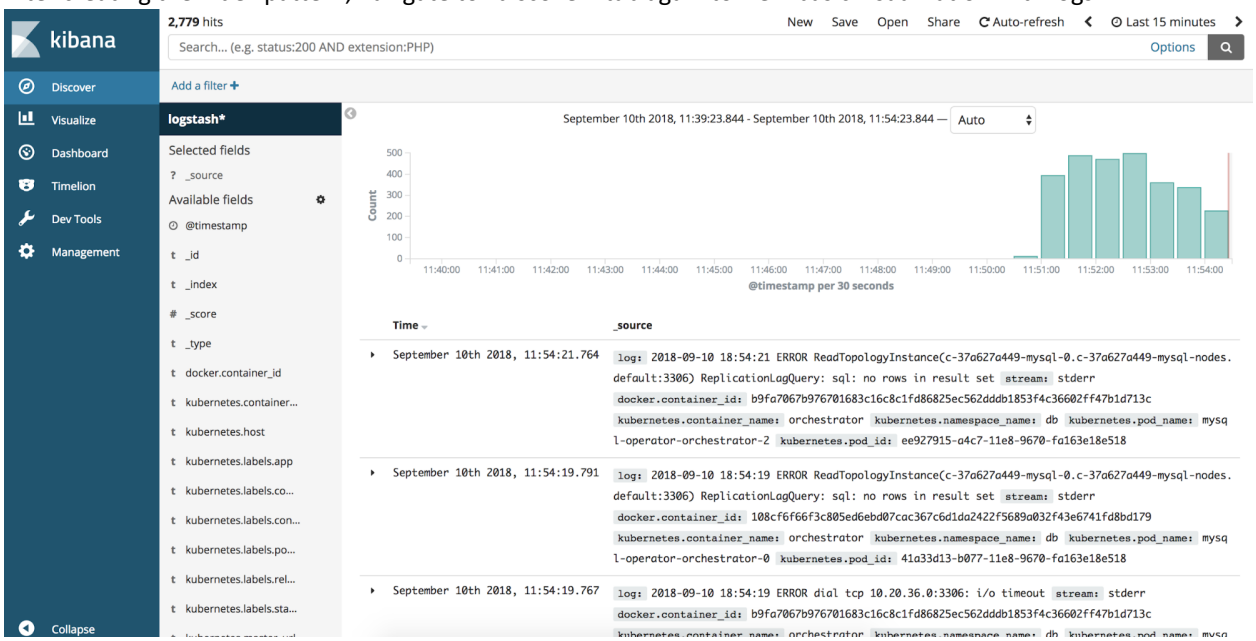
- Navigate to "Discover" tab.



- You can use the fluentd config to send log data in logstash format to Elasticsearch. The logstash format is also recognized by Kibana.
  Type logstash* in the index pattern dialog box.
- Timestamp-based filtering is usually a good idea for logs, Kibana lets you configure it.



1. After creating the index pattern, navigate to “discover” tab again to view basic visualization with logs.



Chapter Summary: You’re Done!

Congratulations! you know have a working ELF stack for logging and monitoring of Kubernetes.

Kubernetes Capacity Planning

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

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

Tuning beyond the minimums is best determined by application performance profiling based on projected usage; but that is a whole other topic that includes everything from using benchmarking as a service to load testing tools to watching real trends with application performance management suites.

Where There is Kubernetes, There is etcd

At the core of almost every Kubernetes deployment is an etcd cluster. In many distributions, this starts as a three-node etcd cluster that is co-located on the management nodes. It can often grow into a five-node dedicated cluster once the Kubernetes cluster starts to see any substantial load. The largest etcd clusters will become constrained by CPU limitations, but for most moderate-sized clusters, two or four CPU cores are enough and disk I/O will be far more critical; therefore, making sure the fastest disks available are used.

When dealing with the largest cloud providers, IOPS increases with the size of the disk provisioned; so often, you will need far more disk space provisioned than required to meet the performance numbers required.

To start to plan the capacity required to run your cluster, getting etcd taken care of is critical. To start off with a small etcd cluster serving under 200 Kubernetes nodes will be three servers with two cores each, 8GB of RAM, 20GB of disk space per node, and greater than 3000 concurrent IOPS.

More details on how to plan for the capacity requirements for etcd are available in the official ops guide on etcd.io.

Kubernetes Management Nodes (Control Plane)

The management nodes that are deployed as part of any Kubernetes cluster are not the most resource-intensive applications you will ever run because a lot of their performance and capacity requirements are tied to their underpinning etcd instance. This is why a lot of public clouds charge a nominal fee, or no fee at all, for the nodes.

To be highly available, a minimum of two management nodes are required. But since many smaller clusters have the underpinning etcd cluster co-located on the management nodes, the control plane becomes a three-node cluster. You could have a single node for both etcd and the management control plane, but that increases risk, as all management capabilities will be unavailable as you recover the single node. At one point, Google Kubernetes

Engine (GKE) used this configuration for etcd when the cluster was under 500 nodes.

On small clusters with under 100 nodes, the control plane will fit into the unused capacity from the etcd cluster as long as the nodes have four CPU cores and 16GB of memory, which is what is commonly recommended by all major distributions.

As a cluster grows, the number of deployments that need to be managed will increase, as will the number of incoming requests. While each of the manager functions only use a few hundred MB of memory, as the number of connections grows, there is also a need to scale the number of api server instances. They need to scale faster than other components to avoid becoming a bottleneck for all communications. Each instance can only handle a set number of requests at a time which defaults to 400.

Kubernetes Worker Nodes (Memory and Storage vs CPU cores)

The actual deployed applications are running in pods on the worker nodes. This includes DaemonSets, which run on every node and regular applications that have a set number of replicas that can be configured to dynamically change with autoscaling. For the sake of simplicity, though, we will leave autoscaling out of the basic planning since that is just an additive to whatever capacity plans are created.

With this scheduling algorithm, it is important to remember that CPU is compressible, Unless it is reserved by a quota, Kubernetes will just keep piling things onto the same nodes, regardless if CPU is overcommitted or not.

Memory and storage are considered incompressible; so if the space is not available, the scheduling will fail on that node or storage device. Even the autoscaler functionality in Kubernetes uses memory as its default metric to know when to add additional pods or even nodes. There are two types of quotas on a pod: the requested amount, which is the minimum; and the limit, which is the maximum. If a pod asks for memory above its request but below its limit, the scheduler will grant the request.

Kubernetes schedules applications across worker nodes in a round-robin fashion. If the requested memory size isn't available, it will fail to schedule; then the scheduler will move to the next node and continue until it finds a host with the available resources. The same logic will apply if quotas are in use for CPU and there aren't enough uncommitted CPU cycles.

An additional consideration is that the default number of pods that can run on a single host is 100. In some instances it can be increased to 250, but in other instances, like AKS, it is limited to 30 pods on a host. This metric will make a big difference in what types of compute instances you can use, which will impact calculations for everything.

For example, if you calculated 3000 pods in an AKS cluster and you'll also be running Fluent, Prometheus, and Dynatrace as DaemonSets, then you'll have a working limit of 27 pods on each node. This leads to the requirement of 112 nodes at no extra capacity, just based on the per-node limit, and not counting CPU usage or physical memory limitations.

When capacity planning for the Kubernetes worker nodes, after you know your per-node limit, calculate the amount of memory you will need per node. Ideally, everything will have quotas set, which will allow for exact calculations. There are multiple ways to set default resource limits, but for our purposes here, we will assume the average pod will have a requested memory of 256Mb. So, the same 3000 pods will require 750GB of total memory across all the nodes, plus the 1GB of base per-node memory already required (kubelet, plus the three DaemonSets listed above). This means on AKS, each of the 112 nodes would need 8GB of RAM at the absolute minimum. But GKE or OpenShift (with 100 pods per node as the default limit) would have 31 nodes with 26GB of RAM minimum.

Chapter Summary

While capacity management is often its own specialization in the world of IT planning, once the base levels for the clusters your organization will deploy have been established, maintaining the most efficiently-sized clusters is not as difficult as it is with many other platforms. Due to the nature of containers and how Kubernetes was architected for scaling from 'Day One,' it only takes some straightforward math to calculate how much to increase or decrease workloads to maintain the most efficient use of resources. This also minimizes wasted compute time, which can save your organization a lot of money over time, as the public cloud can be expensive if you aren't careful.

Autoscaling in Kubernetes

Kubernetes is hailed for making it simple to provision the resources you need, when you need them. However, it's challenging to know the exact size and number of nodes that best fit your application, especially when you can't predict what load you will want to support in the future. If you are allocating resources manually, you may not be quick enough to respond to the changing needs of your application. Fortunately, Kubernetes provides multiple layers of autoscaling functionality: the Horizontal Pod Autoscaler, the Vertical Pod Autoscaler, and the Cluster Autoscaler. Together, these allow you to ensure that each pod and cluster is just the right size to meet your current needs.

Horizontal Pod Autoscaler

The Horizontal Pod Autoscaler (HPA) scales the number of pods available in a cluster in response to the present computational needs. You specify the metrics that will determine the number of pods needed, and set the thresholds at which pods should be created or removed. The usual metrics are CPU and memory usage, but you can also specify your own custom metrics. Once you've set up the HPA, it will continuously check the metrics you've chosen (the default value for checking metrics is 30-second intervals). If one of the thresholds you've specified is met, the HPA updates the number of pod replicas inside the deployment controller. This triggers the deployment controller to scale the number of pods, up or down, to meet the desired number of replicas.

Note that, in order for the HPA to have the data it needs to determine the best number of pod-replicas, you must install the metrics-server on your Kubernetes clusters. This will give the HPA access to CPU and memory metrics. If you wish to use custom metrics to determine how the HPA scales your pods, you will need to link Kubernetes to a time series database (such as Prometheus) with the metrics you wish to use.

One concern with autoscaling is "thrashing": you don't want the number of pods constantly fluctuating in response to minute changes in your resource consumption. However, you do want the autoscaling functionality to be sensitive enough to your changing needs that you always have the correct level of resources. Kubernetes offers users the ability to influence the **scale velocity** of the HPA in two ways:

1. You can customize how long the autoscaler has to wait before another downscale operation can be performed after the current one has finished (the default value is five minutes).
2. To control how fast the target can scale up, you can specify the minimum number of additional replicas needed to trigger a scale-up event.

Vertical Pod Autoscaler

Where the HPA allocates pod replicas in order to manage resources, the Vertical Pod Autoscaler (VPA) simply allocates more (or less) CPUs and memory to existing pods. This can be used just to initialize the resources given to each pod at creation, or to actively monitor and scale each pod's resources over its lifetime. Technically, the VPA does not alter the resources for existing pods; rather, it Checks which of the managed pods have correct resources set and, if not, kills them so that they can be **recreated by their controllers** with the updated requests.

The VPA includes a tool called the **VPA Recommender**, which monitors the current and past resource consumption and, based on that data, provides recommended values for the containers' CPU and memory requests. Even if you don't trust the VPA to manage your pods, you can still use the VPA to get recommendations about what resources

would best fit your current load.

The HPA and VPA are both useful tools, so you may be tempted to put both to work managing your container resources. However, this practice has the potential to put the HPA and VPA in **direct conflict with one another.** If they both detect that more memory is needed, they will both try to resolve this issue at the same time, resulting in the wrong allocation of resources. However, it is possible to use the HPA and VPA together, provided that they rely on different metrics. This will prevent them from being triggered by the same events. The VPA only uses CPU and memory consumption to generate its recommendations, but if you set your HPA to use custom metrics, then both tools can function in parallel.

Cluster Autoscaler

While the HPA and VPA allow you to scale pods, the Cluster Autoscaler (CA) scales your **node clusters** based on the number of **pending pods**. It checks to see whether there are any pending pods and increases the size of the cluster so that these pods can be created. The CA also **deallocates idle nodes** to keep the cluster at the optimal size. In order to provision more nodes, the CA can interface directly with cloud providers and request the resources needed. It can also use cloud provider-specific logic to specify strategies for scaling clusters.

The Cluster Autoscaler relies on different metrics and has a different goal than either the HPA or VPA. Thus, you can use CA in addition to either the HPA or VPA without conflict. The HPA and CA complement each other for truly efficient scaling. If the load increases, HPA will create new replicas. If there isn't enough space for these replicas, CA will provision some nodes, so that the HPA-created pods have a place to run.
However, the Kubernetes Cluster Autoscaler should not be used alongside CPU-based cluster autoscalers offered by some cloud-providers. CPU-usage-based cluster autoscalers do not take into account pods when scaling up and down. As a result, they may add a node that will not have any pods, or remove a node that has some system-critical pods on it.

As with the HPA, one concern to keep in mind is the **speed** at which the CA will deploy (or deallocate) resources. If the scaling is too sensitive, your clusters are unstable, but if there is too much latency, then your application may experience downtime. In practice, it can take a few minutes for the CA to create a new node. One way to ensure additional pods are immediately available is to configure your pods to include **"pause pods"** with low priority and can be terminated to make room for new pods. In essence, this saves time by reserving space in the pod for additional clusters. On the other hand, if you want to slow the scale-up velocity, you can configure a **delay interval**. To smooth out the scale-down operations, you can configure the CA using the PodDistruptionBudgets tag to prevent pods from being deleted too abruptly.

Chapter Summary

Kubernetes's autoscaling features can save your team money by ensuring you are not over-provisioning, while still ensuring that your application has all the resources it needs to stay operational, despite unpredictable loads. However, configuring them correctly may be a headache, even if you avoid all the pitfalls listed in this chapter.

Whether to use HPA, VPA, CA, or some combination, depends on the needs of your application. Experimentation is the most reliable way to find which option works best for you, so it might take a few tries to find the right setup. Mastering autoscaling in Kubernetes is a journey, and will require continuous learning as

these tools mature.

If you want the advantages of autoscaling but want to shortcut the learning process, you may consider a Kubernetes-as-a-Service platform, which implements and manages autoscaling for you. No matter how you implement them though, this suite of autoscalers can help you realize the promise of Kubernetes in a right-sized on-demand infrastructure.

Running Stateful Applications in Kubernetes

When containers became mainstream, they were designed to support ephemeral - stateless - workloads. Since then, a lot of effort has been made to support stateful applications in the container ecosystem, with a lot of that focus targeted towards better support from core Kubernetes. Stateful applications - and the data they contain - are extremely common in most organizations and are vital to the business. Being able to support data-driven applications with Kubernetes enables more organizations to take advantage of containers for modernizing their legacy apps as well as for supporting additional mission-critical use cases - which are often stateful.

This chapter is intended as a crash course on the basics required to get started running any stateful application in Kubernetes.

Kubernetes Storage Constructs:

Stateful applications require, at minimum, persistent storage. Let's first examine the Kubernetes storage constructs to understand how you would persist data in Kubernetes. The most basic distinction to start with is between local storage vs. Persistent Volumes.

Kubernetes Volumes

Volumes are the basic unit of storage in Kubernetes. A **Volume** is storage that's attached - and dependent - to the pod and its lifecycle. A volume has no persistence at all and is mostly used for storing temporary, local data that doesn't need to exist outside the pod's lifecycle. Once the pod is destroyed, its local volume is also released.

Volumes can mount nfs, ceph, gluster, aws block storage, azure or google disk, git repos, secrets, ConfigMaps, hostpath, and more. In these cases the pod will not create or destroy the storage, it will simply **attach** the volume to whatever mount points are identified in the pod specification.

An exception to that is a type of volume called **emptyDir**. emptyDir is a special case where the pod will create its own temporary storage and mount it to the containers in the pod so they can all share files back and forth. The shared storage is deleted forever when the pod is removed from the node.

As an example, below is a very simple pod specification with a container using emptyDir on different mount points so the containers can all share files:

```
apiVersion: v1
kind: Pod
metadata:
  name: emptyDir-pod
spec:
  containers:
  - image: nginx
    name: ed-nginx
    volumeMounts:
```

```
    - mountPath: /a
      name: ed-volume
  containers:
  - image: redis
    name: ed-redis
    volumeMounts:
    - mountPath: /b
      name: ed-volume
  volumes:
  - name: ed-volume
    emptyDir: {}
```

Kubernetes Persistent Volumes

Now that we've identified what a 'regular' volume is in Kubernetes it is easy to see some of its limitations around portability, persistence, and scalability. Stateful applications require that data that is used or generated by the app is persisted, retained, backed up and accessible outside of the particular hosts that run the application. This is where Persistent Volumes (PV) come into play.

Where basic volumes are essentially unmanaged, a Persistent Volume is managed by the cluster. Persistent volumes remain available outside of the pod lifecycle and can be claimed by other pods. Their data can be retained and backed up.

When creating a PV, the administrator specifies for the Kubernetes cluster which storage filesystem to provision, and with which configuration - including size, volume IDs, names, access modes, and other specification. The configuration is specified in a **StorageClass.**

PVs are **resources** in a cluster. **Persistent Storage Claim (PVC)** are requests for these resources, made with a specific StorageClass for the desired configuration.

The Kubernetes master continuously listens for new pods being created with PVC requests. When a new PVC is identified, the Master will find the matching PV and bind it to the PVC. The bound volume would then be mounted to a pod.

With that, each pod is created with the required storage (and its config and environment variables), and each replica would have the same storage type attached and mounted. These pods can then scale with StategulSet (more on that later) so that new pods that join the distributed application have the same storage attached.

Creating a Persistent Volume in Kubernetes:

The steps involved in creating a persistent volume and attaching it to a container in a pod are:

1) Create a **StorageClass** which defines the type of storage that will be used. This could be AWS, EBS, Portworx, etc.

   If a provisioner is defined (for example when using a Kubernetes cloud service, or if your distribution has a default provisioner) then Kubernetes will connect to the storage provider and allocate a new persistent volume whenever a claim is made. Otherwise the cluster administrator needs to manually add **new persistent volumes** before each claim can be made.

   Sample StorageClass

```
apiVersion: storage.k8s.io/v1
```

```
kind: StorageClass
metadata:
  name: px-high-io
provisioner: kubernetes.io/portworx-volume
parameters:
  repl: "1"
  snap_interval:   "120"
  io_priority:  "high"
```

Sample PersistentVolume (PV) - for manual creation

```
kind: PersistentVolume
apiVersion: v1
metadata:
 name: app-fourty-two-pv
 labels:
  type: local
spec:
 storageClassName: manual
 capacity:
  storage: 1Gi
 accessModes:
  - ReadWriteOnce
 hostPath:
  path: "/data"
```

PVs can also be created dynamically. To learn more about dynamic volumes, CSI and how to hack on your storage configuration in Kubernetes, see this deep-dive Kubernetes Storage how-to article.

2) Create a **PersistentVolumeClaim** (PVC) which will have the cluster set aside storage to be used by your application in its pod specifications.

Sample PersistentVolumeClaim

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
 name: app-fourty-two-pv-claim
spec:
 storageClassName: px-high-io
 accessModes:
  - ReadWriteOnce
 resources:
  requests:
   storage: 1Gi
```

3) Define the **volume** you want to use in the deployment (pod, statefulset, etc.)

Sample deployment using a PersistentVolumeClaim

```
apiVersion: v1
kind: Pod
metadata:
  name: mysql-app
spec:
    containers:
    - name: mysql
      image: mysql
      env:
```

```
    - name: MYSQL_ROOT_PASSWORD
      value: "rootpasswd"
    volumeMounts:
    - mountPath: /var/lib/mysql
      name: data
      subPath: mysql
  volumes:
  - name: data
    persistentVolumeClaim:
      claimName: app-fourty-two-pv-claim
```

Deployments with StatefulSet

Deploying a stateful application into Kubernetes can now leverage a specific model called **StatefulSet**. A StatefulSet is essentially a Kubernetes deployment object with unique characteristics specifically for stateful applications. Like 'regular' deployments or ReplicaSet, StatefulSet manages deploying of Pods that are based on a certain container spec. But unlike a regular deployment, it allows you to specify the order and dependencies of the deployment to maintain sticky identities for each of the pods. This enables persistence across pod (re)scheduling.

StatefulSet Deployments provide:

- **Stable, unique network identifiers**
  Each pod in a StatefulSet is given a hostname that is based on the application name and increment. For example, web1, web2, web3 and web4, for a StatefulSet named "web" that has 4 instances running.

- **Stable, persistent storage**
  Each and every pod in the cluster is given its own persistent volume based on the storage class defined, or the default, if none are defined. Deleting or scaling down pods will not automatically delete the volumes associated with them- so that the data persists. To purge unneeded resources, you could scale the StatefulSet down to 0 first, prior to deletion of the unused pods.

- **Ordered, graceful deployment and scaling**
  Pods for the StatefulSet are created and brought online in order, from 1 to n, and they are shut down in reverse order to ensure a reliable and repeatable deployment and runtime. The StatefulSet will not even scale until all the required pods are running, so if one dies, it recreates the pod before attempting to add additional instances to meet the scaling criteria.

- **Ordered, automated rolling updates**
  StatefulSets have the ability to handle upgrades in a rolling manner where it shuts down and rebuilds each node in the order it was created originally, continuing this until all the old versions have been shut down and cleaned up. **Persistent volumes are reused, and data is automatically migrated to the upgraded version.**

When deploying a Kubernetes application using the regular deployment and a ReplicaSet or a StatefulSet, you define the application as a Kubernetes Service, so other applications can interact with it. Session affinity is achieved by enabling "sticky sessions," allowing clients to go back to the same instance as often as possible, which helps with performance – especially for stateful applications with caching.

Sample StatefulSet for Cassandra database with multiple instances each with their own persistent volume. (This contains the storage class but would need to be exposed by a service.)

```
apiVersion: apps/v1
```

```
kind: StatefulSet
metadata:
  name: cassandra
  labels:
    app: cassandra
spec:
  serviceName: cassandra
  replicas: 3
  selector:
    matchLabels:
      app: cassandra
  template:
    metadata:
      labels:
        app: cassandra
    spec:
      terminationGracePeriodSeconds: 1800
      containers:
      - name: cassandra
        image: gcr.io/google-samples/cassandra:v13
        imagePullPolicy: Always
        ports:
        - containerPort: 7000
          name: intra-node
        - containerPort: 7001
          name: tls-intra-node
        - containerPort: 7199
          name: jmx
        - containerPort: 9042
          name: cql
        resources:
          limits:
            cpu: "500m"
            memory: 1Gi
          requests:
            cpu: "500m"
            memory: 1Gi
        securityContext:
          capabilities:
            add:
              - IPC_LOCK
        lifecycle:
          preStop:
            exec:
              command:
              - /bin/sh
              - -c
              - nodetool drain
        env:
          - name: MAX_HEAP_SIZE
            value: 512M
          - name: HEAP_NEWSIZE
            value: 100M
          - name: CASSANDRA_SEEDS
            value: "cassandra-0.cassandra.default.svc.cluster.local"
          - name: CASSANDRA_CLUSTER_NAME
            value: "K8Demo"
          - name: CASSANDRA_DC
            value: "DC1-K8Demo"
          - name: CASSANDRA_RACK
            value: "Rack1-K8Demo"
          - name: POD_IP
```

```
            valueFrom:
              fieldRef:
                fieldPath: status.podIP
        readinessProbe:
          exec:
            command:
            - /bin/bash
            - -c
            - /ready-probe.sh
          initialDelaySeconds: 15
          timeoutSeconds: 5
        # These volume mounts are persistent. They are like inline claims,
        # but not exactly because the names need to match exactly one of
        # the stateful pod volumes.
        volumeMounts:
        - name: cassandra-data
          mountPath: /cassandra_data
  # These are converted to volume claims by the controller
  # and mounted at the paths mentioned above.
  # do not use these in production until ssd GCEPersistentDisk or other ssd pd
  volumeClaimTemplates:
  - metadata:
      name: cassandra-data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: fast
      resources:
        requests:
          storage: 1Gi
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast
provisioner: k8s.io/minikube-hostpath
parameters:
  type: pd-ssd
```

Operators Can Help

While operators are not necessary, they are more robust than a deployment or StatefulSet, and can help run stateful apps on Kubernetes with features like application-level HA management, backups and restore.

You can use existing Operators or develop your own.

The operator package includes all the configuration needed to deploy and manage the application from a Kubernetes point of view - from a StatefulSet to be used to any required storage, rollout strategies, persistence and affinity configuration, and more. Kubernetes will then rely on the operator to validate instances of the application against the specification to ensure it runs in the same way across instances in all clusters it is deployed in.

Stateful App Scenarios Examples:

Let's look at two common scenarios for Kubernetes stateful application: apps powered by a NoSQL/sharded database, and apps using a relational database for their backend.

In both these cases, we'd use PV and PVCs to have Kubernetes provision and manage the persistent storage.

1) **Cassandra or other NoSQL/sharded databases**
   In these cases, the database is designed to be fault tolerant and easier scaling. For example, in the case of Cassandra you already have 3 copies of the data typically, and all the nodes are equal (no master/slave designation). If one node fails the other nodes are still accepting data and the application doesn't need to be aware of any DB availability issues. In the case of NoSQL databases, a best practice is to not create too many replicas ((keep it at 3) to accelerate start-up time if a node fails and a new replica is automatically created.

2) **A PostgreSQL database that is backing a business application**
   PostgreSQL, like most relational databases, typically runs as a single instance, so there is no cluster to maintain data. When running a relational database in Kubernetes, try to keep it small as much as possible so that the in-flight surface is smaller. That way, if a pod dies and becomes available on a different node your start-up time will be faster to restore in-flight transactions from the binary logs.

   Container-based storage solutions that work natively with Kubernetes and offer built-in replication and abstraction across environments are also helpful. The storage class in Kubernetes could point to anything from an EBS block storage to NFS share for this usage; or, when performance matters, an enterprise-class storage solution like Ceph, or a physical SAN over Fibre Channel. Container-friendly software-defined storage like Ceph, GlusterFS, or Portworx can co-exist in the same Kubernetes cluster, but would be hosted on nodes with extra storage capacity in the form of dedicated solid state drives.

Chapter Summary

Stateful applications are one of the most common types of applications being containerized and moved to Kubernetes-managed environments. With advancements in Kubernetes storage constructs and operations, you can now support data-driven applications on Kubernetes as well.

For more information on the Kubernetes components mentioned, check out the latest documentation on kubernetes.io.

Building Helm Charts for Kubernetes Management

Kubernetes is a very popular container orchestration system used extensively in DevOps. However, it can become very complex: you have to handle all of the objects (ConfigMaps, pods, etc.), and you also have to manage your releases. Both can be accomplished with Kubernetes Helm, the Kubernetes packaging solution.

This chapter will discuss Helm packages (also known as Helm charts), how they are created, and what you should be aware of when you create them for maximum reusability, code quality, and efficient deployments.

In a Nutshell: How Does Kubernetes Work?

Kubernetes is an open source container orchestration system. The containers that make up an application are grouped into logical units. To set up a single application, you have to create multiple independent resources – such as pods, services and deployments. Each requires you to write a YAML manifest file.

Helm: the Kubernetes Package Manager

Helm is a Kubernetes package manager designed to easily package, configure, and deploy applications and services onto Kubernetes clusters – it's the apt/yum/homebrew for Kubernetes. It contains the resource definitions necessary to run an application, tool, or service inside of a Kubernetes cluster.

What is a Helm chart?

Charts are packages of Kubernetes resources. A Helm chart is basically a collection of files inside of a directory. The directory name is the name of the chart (without the versioning information). Helm charts consist of a self-descriptor file - yaml file - and one or more Kubernetes manifest files called templates. The YAML file describes the default configuration data for the templates in a structured format. Helm Chart templates are written in the Go template language.

All template files are stored in a chart's templates/ folder. When Helm renders the charts, it will pass every file in that directory through the template engine.

The values for the templates can be supplied in one of two ways:

- Chart developers can supply a file called values.yaml inside of a chart. This file may contain default values.
- Chart users can supply a YAML file that contains values. This can be provided in the command line with the helm install-command.

When a user supplies custom values, these values will override the values in the chart's values.yaml file.

Here is the basic directory structure of a Helm chart:

```
package-name/
```

charts/
templates/
Chart.yaml
LICENSE
README.md
requirements.yaml
values.yaml

Helm charts are very useful when deploying your software. If you can't find an existing chart for the software that you are deploying, you may want to create your own.

How Do You Create a Helm Chart?

You can create your first Helm chart very easily with the Helm create command:

```
$ helm create mychart
Creating mychart
```

This will create a folder with the files and directories seen above, which gives you a base for developing your charts further.

As we noted earlier, the templates are the manifests of the Helm package, with configuration data that is either derived by default from the YAML file or customized by the author.

So what should you be aware of when you create your templates in order to ensure maximum reusability, code quality, and efficient deployments?

1. **Respect the general conventions**

Always be aware of the Helm rules to be applied when working with Helm charts. The chart name should be lower case letters and numbers. The words can be separated with dashes (-); for example:

wordpress
docker
aws-test1

When the directory contains a chart, the directory name must be the same as the chart name. When versioning is applied, the version numbers follow the SemVer format. YAML files should be indented using two spaces. Helm refers to the project as a whole and the client-side command. The term "chart" should not be capitalized (with the exception of Chart.yaml), because the filename is case-sensitive.

2. **Take care of the values**

The name of a variable should begin with a lowercase letter, and words should be separated using camel case:

pea: true
peaSoup: true

Due to YAML's flexible format, values can be deeply nested or flattened. Flat is better for development because of its simplicity. On the other hand, you have to perform an existence check for every layer of nesting, and you should do a check for every nested value at every level. This is not recommended with flattening, because it will make the template less readable.

You should quote all strings for type conversion; that way, a code reader won't be confused about whether or not it's a string.

Every property defined in values.yaml should be documented. The documentation string should start with the property's name and a one-sentence description.

3. **Mind the template structure**

The templates/ directory should be structured as follows:

a. Template files should have the extension **.yaml** if they produce YAML output. The extension **.tpl** can be used for template files that produce no formatted content.

b. Template file names should use dash notation, not camel case.

c. Each resource definition should be in its own template file.

d. Template file names should have the resource type in the name: test-pod.yaml, test2-svc.yaml

4. **Dependencies**

For a chart, you should use version ranges instead of referring to an exact version:

Version: ~ 1.2.3

If there are optional dependencies, you should add conditions or tags. The same tags can be applied to subcharts that contain features (optional).

5. **Using labels and annotations**

The metadata of an item should be used as a label when it is:

- used by Kubernetes to identify this resource.
- useful to expose to operators for the purpose of querying the system.

For example:

helm.sh/chart: NAME-VERSION

If an item of metadata is not used for querying, it should be set as an annotation. Helm hooks are always annotations.

6. **Working with CRD**

When working with Custom Resource Definitions (CRD), keep in mind that the CRD is declared for your chart (see the YAML file called CustomResourceDefinition).
If your CRD defines **test.example.com/v2**, every resource with **apiVersion: test.example.com/v2** is a resource

using the CRD.

7. **RBAC**

Role-Based Access Control is important for the following resources in chart manifests:

- ServiceAccount (namespaced)
- Role (namespaced)
- ClusterRole
- RoleBinding (namespaced)
- ClusterRoleBinding

Keep in mind that the RBAC and ServiceAccount configurations should be done under separate keys in YAML, because they represent different functionality:

```
rbac:
  create: true

serviceAccount:
  create: true
```

The RBAC resources should always be created by default (set to true).

Chapter Summary

In this chapter, we discussed some of the main best practices for creating Helm charts, but there are many more that can help ensure maximum reusability, code quality, and efficient deployment. See the Chart Best Practices Guide for Helm for more best practices.

Container orchestration is all about managing the lifecycles of containers, especially in large, dynamic environments. Through container orchestration, DevOps teams can control and automate many tasks such as the provisioning and deployment of containers.

Kubernetes is a very popular container orchestration system, but it can become very complex when handling objects and releases. Helm is a very useful packaging solution for making Kubernetes more manageable.

Considerations for Running Multi-Master Kubernetes in Production

Kubernetes has been around for five years and, at this point, has become a stable platform that is commonly used throughout development and production environments to run applications. In the time it has been a project, the base project has become more capable. It makes it easier to build and deploy a reliable cluster on your own, and then add in the components you want, so it meets your individual requirements.

Production Considerations

In development, things like having a single etcd node and single master control plane aren't a problem, since when they are down, it doesn't actually stop anything from running; you just can't manage the cluster while they are down. Worst case is that a critical bug is found in an application and it crashes, which means it won't be brought fully back online until the master server or etcd host are back online.

Beyond the high availability consideration for the masters, when looking at deploying a home-built Kubernetes cluster into production, there are numerous other items to consider.

- Monitoring all the metrics available within a Kubernetes cluster for health and usage patterns, using a proven tool like Prometheus
- Graphical console (like Dashboard) with observability of monitor metrics, with a tool like Grafana
- Watching and analysing log files by centralizing them in a system like elastic search.
- Security patching from the operating system layer, all through the Kubernetes cluster
- Application binary scanning with an open source tool like Clair, or from one of many commercial vendors (Aqua Security, Palo Alto Networks, Trend Micro, Synopsys, etc).
- Security integration between RBAC and external authentication and authorization services.
- Load balancers and their publicly-facing SSL certificates. Let's Encrypt is the low cost entry point for independent SSL certificates.
- Networking layer – Is the default wide-open policy enough, or do you need multi-tenancy or even network policy support? There are many vendors that can provide tools from the purely open source Open vSwitch, to the open source based Tigera Calico, to purely commercial VMware NSX-T.

Clustering High-Availability etcd

New etcd Database Cluster

When creating a new etcd database cluster, it is as easy as specifying all the instances on the command line on each node as you start the cluster, and letting the instances figure it out. If you can, this is always the best way to start. Three nodes are the recommended minimum; five nodes are recommended for most reasonably-sized production clusters (1000+ pods).

For clusters using static discovery, each node needs to know the IP of the other nodes. Following, are the exact commands to run on the three nodes in a new cluster. These commands also enable in-transit encryption of all communication channels.

First up is creating the certificates that will be used across the nodes. This can be done on any Mac or Linux host with openssl installed. In this case the hostnames are infra0, infra1, and infra2.

Create the Certificate Authority (and private key):

```
openssl genrsa -out ca-key.pem 2048
openssl req -x509 -new -nodes -key ca-key.pem -days 10000 -out ca.pem -subj "/CN=etcd-ca"
```

Create the configuration used to create the client certificate:

```
# vi openssl.conf
[req]
req_extensions = v3_req
distinguished_name = req_distinguished_name

[req_distinguished_name]

[ v3_req ]
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
subjectAltName = @alt_names

[ ssl_client ]
extendedKeyUsage = clientAuth, serverAuth
basicConstraints = CA:FALSE
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid,issuer
subjectAltName = @alt_names

[ v3_ca ]
basicConstraints = CA:TRUE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
subjectAltName = @alt_names
authorityKeyIdentifier=keyid:always,issuer

[alt_names]
DNS.1 = localhost
DNS.2 = infra0
DNS.3 = infra1
DNS.4 = infra2
IP.1 = 127.0.0.1
IP.2 = 10.1.2.11
IP.3 = 10.1.2.12
IP.4 = 10.1.2.13
```

Load the configuration into the environment and then create the client key, client certificate, and sign it.

```
CONFIG=openssl.conf
openssl genrsa -out infra-key.pem 2048
openssl req -new -key infra-key.pem -out infra.csr \
  -subj "/CN=etcd-01" -config ${CONFIG}
openssl x509 -req -in infra.csr -CA ca.pem -CAkey ca-key.pem \
  -CAcreateserial -out infra.pem -days 3650 \
  -extensions ssl_client -extfile ${CONFIG}
```

And finally, copy the certificates to the nodes that will host etcd in a new directory named /etc/ssl/etcd/ssl/. These commands will copy the files and preserve permissions.

```
ssh root@10.1.2.11 mkdir -p /etc/ssl/etcd/ssl/
scp -p ca.pem ca-key.pem infra-key.pem infra.pem infra.csr root@10.1.2.11:/etc/ssl/etcd/ssl/
```

On each node you install etcd: a couple environment configurations, a setup command, then the same four commands will create the startup file and start the service. This is based on CentOS host, but other platforms have a very similar configuration setup:

Node 1:

```
sudo yum -y install etcd
```

```
export NODE1_IP=10.1.2.11
export NODE2_IP=10.1.2.12
export NODE3_IP=10.1.2.13
export INTERNAL_IP=${NODE1_IP}
export ETCD_NAME=infra0
cat <<EOF | sudo tee /etc/etcd/etcd.conf
ETCD_DATA_DIR="/var/lib/etcd/p9"
ETCD_LISTEN_CLIENT_URLS="https://${INTERNAL_IP}:2379"
ETCD_LISTEN_PEER_URLS="https://${INTERNAL_IP}:2380"
ETCD_NAME="${ETCD_NAME}"
ETCD_ELECTION_TIMEOUT=5000
ETCD_HEARTBEAT_INTERVAL=250

[Clustering]
ETCD_ADVERTISE_CLIENT_URLS="https://${INTERNAL_IP}:2379"
ETCD_INITIAL_ADVERTISE_PEER_URLS=https://${INTERNAL_IP}:2380
ETCD_INITIAL_CLUSTER="infra0=https://${NODE1_IP}:2380,infra1=https://${NODE2_IP}:2380,infra2=https://${NODE3_IP}:238
0"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster-0"
ETCD_INITIAL_CLUSTER_STATE="new"

[Proxy]
ETCD_PROXY=off

[Security]
ETCD_TRUSTED_CA_FILE=/etc/ssl/etcd/ssl/ca.pem
ETCD_CERT_FILE=/etc/ssl/etcd/ssl/infra.pem
ETCD_KEY_FILE=/etc/ssl/etcd/ssl/infra-key.pem
ETCD_PEER_TRUSTED_CA_FILE=/etc/ssl/etcd/ssl/ca.pem
ETCD_PEER_CERT_FILE=/etc/ssl/etcd/ssl/infra.pem
ETCD_PEER_KEY_FILE=/etc/ssl/etcd/ssl/infra-key.pem
ETCD_PEER_CLIENT_CERT_AUTH=true
EOF
```

Node 2:

```
sudo yum -y install etcd

export NODE1_IP=10.1.2.11
export NODE2_IP=10.1.2.12
export NODE3_IP=10.1.2.13
export INTERNAL_IP=${NODE2_IP}
export ETCD_NAME=infra1
cat <<EOF | sudo tee /etc/etcd/etcd.conf
ETCD_DATA_DIR="/var/lib/etcd/p9"
ETCD_LISTEN_CLIENT_URLS="https://${INTERNAL_IP}:2379"
ETCD_LISTEN_PEER_URLS="https://${INTERNAL_IP}:2380"
ETCD_NAME="${ETCD_NAME}"
ETCD_ELECTION_TIMEOUT=5000
ETCD_HEARTBEAT_INTERVAL=250

[Clustering]
ETCD_ADVERTISE_CLIENT_URLS="https://${INTERNAL_IP}:2379"
ETCD_INITIAL_ADVERTISE_PEER_URLS=https://${INTERNAL_IP}:2380
ETCD_INITIAL_CLUSTER="infra0=https://${NODE1_IP}:2380,infra1=https://${NODE2_IP}:2380,infra2=https://${NODE3_IP}:238
0"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster-0"
ETCD_INITIAL_CLUSTER_STATE="new"

[Proxy]
ETCD_PROXY=off

[Security]
ETCD_TRUSTED_CA_FILE=/etc/ssl/etcd/ssl/ca.pem
ETCD_CERT_FILE=/etc/ssl/etcd/ssl/infra.pem
ETCD_KEY_FILE=/etc/ssl/etcd/ssl/infra-key.pem
ETCD_PEER_TRUSTED_CA_FILE=/etc/ssl/etcd/ssl/ca.pem
```

```
ETCD_PEER_CERT_FILE=/etc/ssl/etcd/ssl/infra.pem
ETCD_PEER_KEY_FILE=/etc/ssl/etcd/ssl/infra-key.pem
ETCD_PEER_CLIENT_CERT_AUTH=true
EOF
```

Node 3:

```
sudo yum -y install etcd

export NODE1_IP=10.1.2.11
export NODE2_IP=10.1.2.12
export NODE3_IP=10.1.2.13
export INTERNAL_IP=${NODE3_IP}
export ETCD_NAME=infra2
cat <<EOF | sudo tee /etc/etcd/etcd.conf
ETCD_DATA_DIR="/var/lib/etcd/p9"
ETCD_LISTEN_CLIENT_URLS="https://${INTERNAL_IP}:2379"
ETCD_LISTEN_PEER_URLS="https://${INTERNAL_IP}:2380"
ETCD_NAME="${ETCD_NAME}"
ETCD_ELECTION_TIMEOUT=5000
ETCD_HEARTBEAT_INTERVAL=250

[Clustering]
ETCD_ADVERTISE_CLIENT_URLS="https://${INTERNAL_IP}:2379"
ETCD_INITIAL_ADVERTISE_PEER_URLS=https://${INTERNAL_IP}:2380
ETCD_INITIAL_CLUSTER="infra0=https://${NODE1_IP}:2380,infra1=https://${NODE2_IP}:2380,infra2=https://${NODE3_IP}:238
0"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster-0"
ETCD_INITIAL_CLUSTER_STATE="new"

[Proxy]
ETCD_PROXY=off

[Security]
ETCD_TRUSTED_CA_FILE=/etc/ssl/etcd/ssl/ca.pem
ETCD_CERT_FILE=/etc/ssl/etcd/ssl/infra.pem
ETCD_KEY_FILE=/etc/ssl/etcd/ssl/infra-key.pem
ETCD_PEER_TRUSTED_CA_FILE=/etc/ssl/etcd/ssl/ca.pem
ETCD_PEER_CERT_FILE=/etc/ssl/etcd/ssl/infra.pem
ETCD_PEER_KEY_FILE=/etc/ssl/etcd/ssl/infra-key.pem
ETCD_PEER_CLIENT_CERT_AUTH=true
EOF
```

On all the nodes, reload the daemon-config; enable, and start etcd on all nodes:

```
sudo systemctl enable --now etcd
# This command sets the cluster to existing for the next start
sed -i s'/ETCD_INITIAL_CLUSTER_STATE="new"/ETCD_INITIAL_CLUSTER_STATE="existing"/'g \
  /etc/etcd/etcd.conf
```

And now, verify its running

```
# etcdctl -C https://10.1.2.11:2379 --ca-file /etc/ssl/etcd/ssl/ca.pem cluster-health
member 21ce7d4f60f4ade0 is healthy: got healthy result from https://10.1.2.11:2379
member 7e5b4986949de455 is healthy: got healthy result from https://10.1.2.12:2379
member 85313efd8bbb270f is healthy: got healthy result from https://10.1.2.13:2379
cluster is healthy
```

Adding HA to Existing etcd Database

If you have an existing etcd database running, before you can add or remove nodes, a quorum – the majority of nodes – needs to be active, or it will not allow write operations. So in a database cluster with three instances, two need to be alive for you to make any changes to the database or its runtime configuration. If the cluster has less

than three instances, then all instances need to be alive to make changes. This is because of the way the formula – which is basically (n/2)+1 – works.
So, if you have a single instance or two instances, in the event of a failure, no changes can be made until all are back online.

With a quorum in place, adding new nodes involves configuring additional nodes as per the setup instructions listed above, including updating and distributing any certificates.

Once the new nodes are configured with their type set to *existing not new*, simply update the existing cluster's runtime configuration to have the new peer addresses before starting the etcd service on those new nodes. The runtime update is fairly straightforward.

```
$ etcdctl member add infra3 --peer-urls=http://10.1.2.14:2380
added member 9bf1b35fc7761a23 to cluster
```

After the address is successfully added, start the services on the new nodes. If you need to add a new node to replace an existing failed node, then always perform the *remove* operation first, as that makes it easy for the cluster to reach the quorum.

Note: Don't forget to update the configuration files like /etc/etcd/etcd.conf on any existing nodes to reflect the new host, if static discovery is being used as in the example above.

Going to a Multi-Master Configuration

The beautiful thing about the actual Kubernetes deployment of its control plane is, it is identical on every single node. Regardless if you have one node or 100 control plane nodes, all the configurations are stored in the etcd database. So, if you need to add or remove control plane nodes, it is as simple as duplicating any TLS certificates, configuration, and binaries from an existing host to all new hosts. As long as they can connect to the etcd datastore over the network (TLS certificates are a big part of this), then all is well.

The basic steps are to get the binaries

```
wget -q --show-progress --https-only --timestamping \
  "https://dl.k8s.io/v1.17.0/kubernetes-server-linux-amd64.tar.gz"
tar xzf kubernetes-server-linux-amd64.tar.gz
```

Move any certificates, private keys, and secrets configuration to the runtime folder

```
sudo mv ca.pem ca-key.pem kubernetes-key.pem \
    service-account-key.pem service-account.pem \
    encryption-config.yaml /etc/kubernetes/
```

Then enable the services using systemd, initV, or kubelet initialization parameters. Finally, add any new nodes to the load balancer that is required to balance traffic across all the API Server instances.

Chapter Summary

Now that your multi-master cluster is in place, it is time to pick all the individual components that are required to satisfy items like your monitoring and security controls. A great place to find all these solutions – including centralized management – is with SaaS providers like Platform9, which can simply and easily

provide a very solid base to bring any Kubernetes cluster up to a manageable and maintainable state. At that point it is more about filling in the few missing gaps unique to your environment, instead of building out proven core management capabilities.

Challenges of Running Kubernetes at the Edge

The age of Edge computing has finally dawned. The rapid developments in digital and mobile technologies have made Edge computing increasingly more prevalent, and more critical to the success of businesses across a wide range of industries.

What is Edge Computing?

Edge computing essentially takes memory and computing out of the traditional data center to bring them as close as possible to the location where they are needed – often in the form of hand-held or local devices, appliances, or point-of-sale or physical units that are distributed across different locations.

Edge means different things to different industries. For automotive, for example, it may mean the growing importance of compute capacity in smart cars or in hand-held devices used by technicians and service centers. For retail, it might mean new kinds of compute capacity available at point-of-sale systems and new experiences being delivered to customers in storefronts. Even in the fast food industry, Chick-fil-A shared it was running edge devices with container-based applications in every restaurant.

Edge applications which interact the closest with the local devices in the field are getting more sophisticated and intelligent with every passing quarter. There's a lot of opportunity and promise in edge computing – for both end consumers and for business. These applications can offer customers a seamless and personalized experience, help improve business processes, and more.

Let us first examine some of the key promising use cases for Edge computing. We'll then discuss the challenges with Edge computing and some of the key questions business leaders should ask themselves around supporting the intelligent Edge.

Five Key Edge Use Cases:

- **Field and Industrial IoT** – Various sensors and other field devices across verticals like Manufacturing, Transportation, Power are a prime candidate for Edge computing. These devices can be HVAC systems, Energy Meters, Aircraft engines, Oil rigs, Scanners in Retail, Wind turbines, Connected cars, RFIDs in Supply chain, Robotics, AR, and much more. These are often characterized by applications that collect data from edge devices and analyze it for different business use cases – security management, predictive maintenance, performance or usage tracking, demand forecasting, etc.
- **Smart Cities and Architecture** – Many cities across the globe are vying for the tag of a Smart City. IoT devices will make living in such cities easier for citizens. The use cases here range from municipalities providing faster urban services (repair of equipment), traffic management (to reduce gridlock), public safety and green energy provisioning
- **Customer Experience in Retail and Hospitality** – Customer sentiment data and social media data is collected and analyzed to improve customer experience. Data here is being captured by a kiosk or a Point of Sale (POS) system or Terminal.
- **Connected Vehicles** – For example, telematics data used for navigation, or to influence dynamic pricing for auto insurance, predict required maintenance, and so on.
- **Facial and image recognition** – as a way of identifying customers and reducing fraud in verticals such as Retail, Banking, and Entertainment.

The Edge Represents a Unique Computing Challenge

Edge computing is very different from traditional data center environments for the following reasons:

- **Compute and hardware constraints**: Many edge environments are constrained from the standpoint

of technical computing footprint. For example, in the case of embedded devices, you can't fit as much hardware as on a full-scale data center.

- **Accessibility and Operations constraints**: Often, Edge applications pose logistical difficulties in deploying human IT resources to manage them and do not allow for high operator cost. Companies can not have a dedicated admin to monitor and service each and every Edge location. For example, in the case of wind turbines spread across thousands of miles, or sensors located in the depth of oil wells or mining sites, or for every payment processing device at every checkout line at a department store, or thermostats located in people's private domains.These operator limitations – either due to distance, the volume of devices, geographical accessibility, and other costs/ROI considerations mandate that Edge applications be not just very low on computing footprint but also on technical IT overhead. They have to be "plug & play" from installation and on-going operations perspectives.
- **Remote management:** In many environments, skilled personnel are not available to deploy & manage the solution on a regular basis. An unskilled operator may need to perform simple plug and play deployments. This includes delivering secure edge application updates, debug-ability in the case of problems and deployment of additional devices. The Edge applications need to be highly sophisticated and should be able to provide a range of features: data caching in case of lost connections, raw data stream processing to filter, analyze relevant data, message brokering for event-based applications, device management, fault tolerance, etc. Saving bandwidth costs of constrained networks is also another important consideration.
- **Connectivity**: The ability of the technology provider to work with all sorts of latency and jitter issues is also key.
- Support for Air-gapped deployments – the ability to manage remote, air-gapped devices in compute constrained locations without resorting to manual intervention is a key need in Edge Computing. High latency to the central cloud can cause delays and interfere with the workings of the application. This also means that assumptions that originate in "normal operations mode" of datacenter networking often do not hold true in Edge environments.
- **Security is a foundational consideration**. This includes secure communication from the datacenter to the Edge, ensuring the privacy of data both at rest and in motion – anonymizing sensitive customer data stored at the Edge. Other security requirements include establishing mutual trust between the central datacenter and Edge devices, the ability to find and stop rogue devices in the event of an attack and secure communication over the WAN.
- **Unified architecture and release processes** that span both the Edge deployment targets, as well as traditional datacenters. This is a major challenge since many Edge applications also need to be deployed across other environments or data centers, creating a complex and practically unmanageable matrix of code bases, pipelines, deployment processes and operational practices. These architecture silos are as much a cause of technical debt as are the data and processes silos.

Digital Transformation Via the Intelligent Edge

In light of the challenges above, the following are some of the key questions technology leaders should ask themselves around Edge applications' release:

- How can we impact the customer experience in a connected world? What ecosystem partnerships can be beneficial for specific use cases? For instance, if you are a retailer, how can real-time customer traffic affect dynamic promotions to drive sales? Can you partner with Banks or players in complementary verticals to gain customers?
- How will the new edge experience(s) integrate with existing channels and processes for increasing customer engagement? Based on the above example, can we suggest other products to the customer based on their previous purchases?
- How can existing business workflows be augmented using Edge insights? Going back to our retail example, how do popular items affect "just in time" manufacturing, procurement, and supply chain

workflows? Can these be augmented with this fresh data?

- What is the right architecture stack that enables the business to accomplish these capabilities? What does that look from a Cloud, Data, and Middleware design standpoint? Do we need a combination of VM and Containers running on the edge?
- What does this mean in terms of enabling self-service across the technology stack and the various stakeholders? Should business transformation be about autonomous capabilities? IT should not become a bottleneck. For example, can certain busier stores receive more compute capacity on the fly without a long manual provisioning cycle?
- How can we do so while eliminating most of the grunt work around building, operating and managing clouds? In short, how can our IT avoid becoming “Cloud janitors”?
- How can these Edge systems continuously learn and improve? Can robots or Drones deployed in distribution centers learn how to assemble boxes and stock those in the right areas? Can a mathematical model be deployed which enables a robot to understand metrics such as mean time to restock, error rate, accuracy in business processes, and more.
- How can the data being collected across edge devices help reduce unnecessary inventory, damage, and other quality issues?
- From a compute cloud standpoint, the key requirements are to support low latency, a high degree of workload parallelism and fault tolerance.

Chapter Summary

Platform9 presents a series of edge-computing use cases that Kubernetes is well positioned to address. However, running Kubernetes successfully on the cloud edge requires addressing some special considerations, such as managing the added complexity that edge architectures introduce and ensuring efficient network performance.

Kubernetes for Machine Learning

Machine learning is rapidly becoming essential to businesses and institutions across the globe. Each organization must meet the challenge of provisioning a computational infrastructure that can support a resource-intensive machine-learning pipeline. While moving computation to the cloud has become the standard response to the challenge of scalability, machine learning (ML) teams have specific needs that must be considered.

Fortunately, the popular containerization orchestrator - Kubernetes - addresses these requirements so that teams can apply the flexibility of cloud-native development and infrastructure to their machine-learning applications.

Auto-scaling

The phases of the machine learning process vary in the amount, type, and timing of the resources they require. Data preparation requires consistent, intensive computation while model inference is less resource heavy. Deep learning is characterized by bursts of activity that will bring the process to a halt if sufficient resources are not available. The machine learning workflow works best when each step in the process can be scaled-up when needed, and scaled back down when done.

The ability to scale resources on-demand is one of the primary benefits of cloud-computing. Kubernetes makes this process simple to orchestrate: users can allocate additional resources as needed simply by adding more physical or virtual servers to their clusters. Additionally, Kubernetes offers the ability to track which resources are provisioned and in use, such as the type and number of CPUs or GPUs present, or the amount of RAM available. Thus the existing allocation is taken into account when scheduling jobs to nodes, ensuring that resources are allocated efficiently.

While simply being able to scale resources is a great advantage over needing to physically allocate servers, this scaling must be automated in order to ensure efficient utilization. This is especially true in machine-learning workloads that require unpredictable bursts of additional computation power.

Kubernetes includes the Horizontal Pod Autoscaler, which scales the number of pod replicas; and the Vertical Pod Autoscaler, which allocates more memory or CPUs to existing pods. If you need another layer of automation on top of this, you can also autoscale clusters so that new clusters are created to manage the extra pods generated by these autoscalers.

While the inherent scalability of Kubernetes ensures that ML teams *can* manage their resources efficiently, the ability to automate that scaling ensures that they actually *will,* even when they can't predict their needs.

GPU Support

If possible, GPUs (graphic processing units) should be used for deep-learning training, due to their significant speed when compared to CPUs. However, managing an end-to-end GPU stack requires layers of drivers and dependencies wherein small discrepancies can lead to big headaches. By using containers, it is easy to reproduce the environment necessary to support computation on GPUs.

Currently, Kubernetes includes experimental support for managing AMD and NVIDIA GPUs, and NVIDIA offers "Kubernetes on Nvidia GPUs" software that automates the management of GPU-accelerated application containers. These tools enable ML teams to leverage the speed of GPUs within a containerized workflow.

Data Management

Machine Learning requires large sets of data for training and testing. Although early feature engineering may be done manually, the transformation of this data should be automated as soon as possible in order to ensure reproducibility. The results of these transformations may also need to be saved temporarily, resulting in a sudden doubling of the team's storage needs. A flexible data repository, such as a cloud-based data lake, is a more efficient way to meet these needs than static, on-premises hardware. This solution can also support the high-throughput access required by training and inference workloads, without requiring additional data replication.

Kubernetes provides a single access point for diverse data sources and manages volume lifecycle, enabling teams to provision exactly the cloud-based storage they require, while reducing complexity. Kubernetes also works with a host of third-party tools, like Apache Hadoop or Spark, which allow complex data transformations to be automated and executed across clusters.

Multitenancy

The machine learning process is made up of a diverse set of workloads, which are often managed by separate teams. However, separating these workloads into their own dedicated hardware environments creates unnecessary siloing and inefficient use of resources. It is both simpler and more efficient to build the process on shared environments that can support the needs of multiple concurrent workloads.

Kubernetes offers the 'namespaces' feature, which enables a single cluster to be partitioned into multiple virtual clusters. Each namespace can be configured with its own resource quotas and access control policies. This allows a single cluster to more easily support different steps of the machine learning workflow.

Abstraction

Orchestrating a machine learning pipeline is a tricky business, and managing the details of how the input from one step gets passed to the next can be overwhelming. Many machine learning teams rely on libraries like TensorFlow or Keras to abstract the process so they can focus on the overall logic. Fortunately, these libraries are compatible with Kubernetes and there is ample documentation available for how to incorporate them into a containerized workflow.

As much as possible, the details of the infrastructure should be abstracted as well. The task of managing the resources that run machine learning models should not be more complex than the task of developing them. Kubernetes provides a layer of abstraction for managing containers that provides a number of easy-to-access interfaces from which workloads can be manipulated. Thus, while managing containers is certainly a skill that takes time to learn, with Kubernetes, users are at least shielded from the complexity of the underlying tech stack.

For even further abstraction, there are platforms available which simplify the process of getting set up with Kubernetes, and streamline its management. One popular option is Kubeflow, which offers a user interface from which teams can configure and monitor their containerized machine learning pipeline. Seldon is an additional tool, which extends Kubeflow and is specifically built for Kubernetes deployments. For teams considering a complex Kubernetes workflow, such as a hybrid-cloud model, these services are particularly useful. By enlisting the help of these third-party tools, data scientists can reap the benefits of Kubernetes while still focusing on their models.

Chapter Summary

A containerized cloud-based workflow orchestrated by Kubernetes meets many of the challenges posed by the computational requirements of machine learning. It provides scalable access to CPUs and GPUs that automatically ramps up to when computation needs spike. It provides multiple teams with access to data storage that grows to meet their needs and supports the tools they rely on for manipulating those datasets.

A layer of abstraction gives data scientists access to these services without worrying about the details of the infrastructure underneath it. As more groups look to leverage machine learning to make sense of their data, Kubernetes makes it easier for them to access the resources they need.

Conclusion

As the chapters of this eBook highlight, Kubernetes is a powerful platform with a wide range of functionality. Not only does it support the deployment of simple, stateless applications, but it can also be leveraged in more complex situations for use cases such as edge computing and even machine learning.

With this wide-ranging power and functionality, however, comes complexity. No matter which use cases you adopt Kubernetes for, or what the scale of your environments is, it is essential to have in place tools and processes for managing all of the requirements of a successful Kubernetes operation. Those range from monitoring and logging, to managing RBAC and beyond.

For further guidance on operating Kubernetes successfully, reach out to Platform9's Kubernetes experts anytime. Or, request a free trial of Platform9's fully managed Kubernetes offering, which uses an SaaS delivery model to eliminate the need for customers to worry about the tedious details of operating Kubernetes. With Platform9 Kubernetes, your team can focus on its applications, not the complexity of Kubernetes itself.