Coralogix

Logging in Kubernetes

BY CHRIS COONEY

<PART 1>

KUBERNETES 101

KUBERNETES 101

Kubernetes has become the de-facto standard for container orchestration. In July 2021, Red Hat found that out of 500 DevOps capable organizations surveyed, 88% were using Kubernetes. Where once, there was stiff competition between Kubernetes, Docker Swarm, Mesos Marathon, and others, now only Kubernetes (K8s) remains. It's easy to see why Kubernetes has become so popular.

There are many different ways to operate a Kubernetes cluster, and all of them have their benefits and drawbacks. In the world of logging and observability, these drawbacks can be the difference between operational success and total disaster. This eBook is all about helping you set some great foundations for your Kubernetes logging solution, using best practices from around the Kubernetes and cloud-native community.

The insights you'll gain from this eBook

Let's dive into the world of Kubernetes logging: the best practices, common pitfalls, and the quick wins that you can exploit to achieve true observability within your cluster rapidly. This eBook will show you how to employ common Kubernetes tools, patterns, and practices to achieve logging success, and it will regularly stop to inspect why a given approach has won over all others.

By the end of this book, you'll know:

- **How to deploy an Elasticsearch cluster to Kubernetes**
- **How to get your logs from your applications into Elasticsearch**
- **The key values to monitor in your Elasticsearch cluster and the operational challenges you may encounter while you're running Elasticsearch**
- **The engineering practices that will unlock the true power of your logs**

What is container orchestration, and why is Kubernetes so good at it?

When Docker became the standard mechanism for packaging and running applications, it became easier than ever to get your code deployed into production. The beauty of containerization is that your application can run in its own little environment, and that environment doesn't need to cater for anything else.

But the specialization of containers also meant that the sheer volume of containers increased. Once you have deployed 10 applications on the same virtual machine (VM) and are only worried about configuring the base VM, you now have 10 separate docker containers that look and behave a little like miniature VMs. Container orchestration is necessary to help lighten the load. Rather than manually monitoring and fixing 5,10, or 1,000 docker containers, you allow the most common tasks to be automatically handled by your container orchestration system.

K8s does this very well. If a long-running container dies, Kubernetes will restart it. If a job fails, it will rerun it. If you can't work out which VM to host your new container, don't worry, Kubernetes automatically tracks and schedules containers to an appropriate server. Organizations were regularly building home-grown mechanisms to solve these problems, but in the end, K8s have demonstrated their superiority.

What is Observability, and why do you need observable logs in Kubernetes?

Kubernetes is a complex element of engineering. The picture can only grow more confusing when you add your software. Containers running in a Kubernetes cluster may be restarted at any time. The servers that host them may be taken down for maintenance, terminated in a scale-down event, or simply fail. Rather than a single, static architecture that aims to keep everything running by changing nothing, Kubernetes is constantly optimizing your environment.

This natural state of change comes with significant benefits. If your servers are constantly cycling, you know that your disaster recovery will work in an actual server outage. However, it can also make for a tricky system to operate. Is a server going down because Kubernetes has decided it no longer needs it or is something wrong with the instance? Why has the control plane suddenly added 5 new nodes into the cluster? Answering these questions requires more profound insight.

LOGS PROVIDE THAT INSIGHT, BUT YOU NEED TO WORK FOR IT

The word “logs” immediately conjures up images of using the SSH command to jump onto a server and using *cat* or *less* to read the contents of an obscurely named *.txt* file. This paradigm simply doesn’t work in the world of microservices and doesn’t hold up in the shifting sands of a Kubernetes cluster.

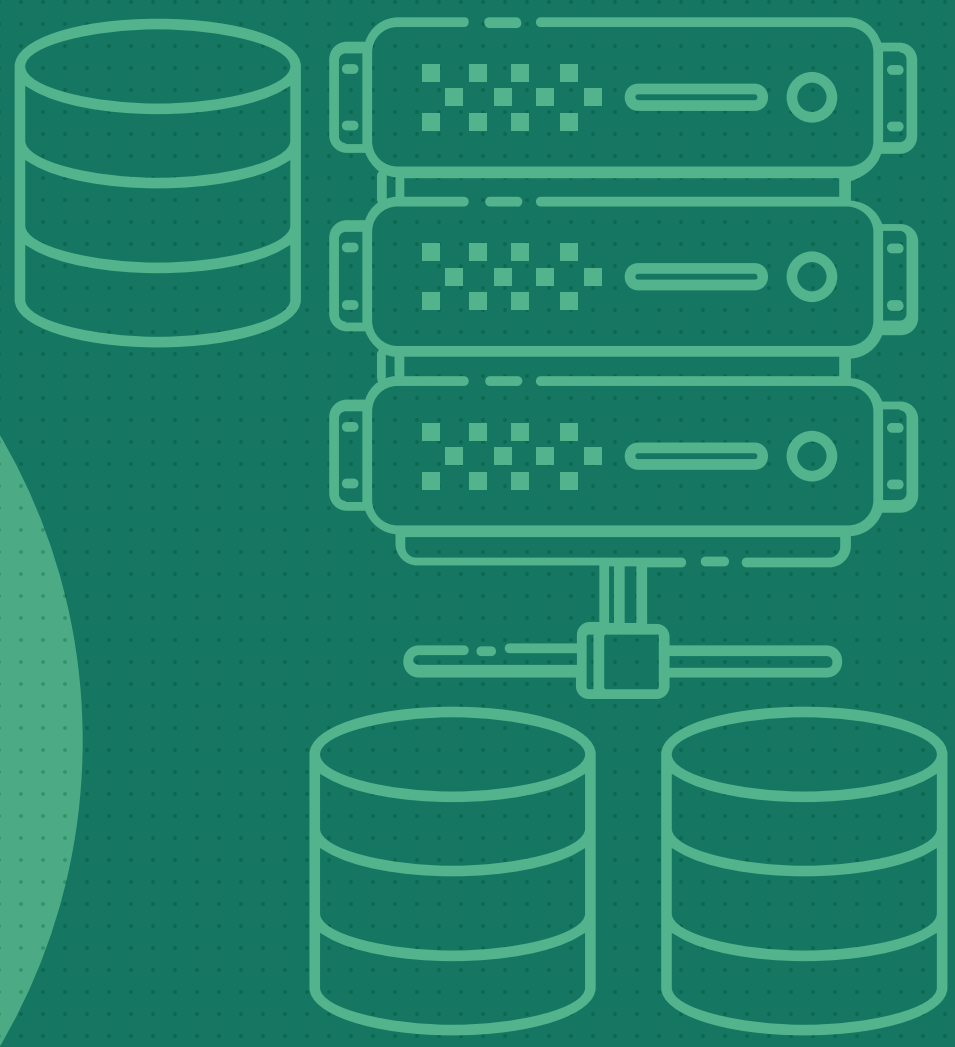
Kubernetes *pods* (container’s collections) regularly scale up, down, and move between servers. When they move, their logs move with them, so relying on the local disk of a stateless pod means you have no defined retention period for your logs. It also means that you have no central view of what is happening in your cluster.

This is why log collection has become the ubiquitous practice that it is today. Rather than hopping between *pods* and hoping that the application log files are intact, developers can store them in a single place that offers fine-grained control over retention, visualization, and insight. It also allows engineers to query their logs and present them in many different ways, which is the essential capability of an observable logging platform.

It’s not enough to pull back the logs for a single instance. You need to pull logs across multiple applications, VMs, and Kubernetes clusters. Then, once you have the data, you need to slice it and join it however you see fit. As your architecture grows, the ability to simply know the state of your cluster will become indispensable, and logs are critical here, enabling you to harvest the best possible insight and respond to the sublime chaos of a working Kubernetes cluster.

<PART 2>

UNLOCKING THE POTENTIAL IN YOUR LOGS



UNLOCKING THE POTENTIAL IN YOUR LOGS

When working with Kubernetes, operators often wonder why something happened. If Kubernetes scales up another pod, it may be clear why from the container metrics of the other pods. For example, if they're all beginning to run out of memory, it is safe to infer that this new instance has been brought in to increase the available memory for workloads.

Metrics can sometimes give engineers a clear picture of what is happening, but these same metrics cease to be useful when the processes aren't working as they should be. For example, why *isn't* a new pod created when all of my memory is maxed out on my existing instances? In this case, the answer is likely a combination of many different metrics. Joining these metrics together in a meaningful way is hard enough, but finding them in the first place is the true challenge.

For some experienced K8 operators reading this, you may be wondering why you would even bother with metrics in this situation. The truth is, you wouldn't. You'd look at the logs produced by the Kubernetes scheduler, understand why new pods are not scheduled onto nodes, or you would look at the logs of the Kubernetes autoscaler and understand why new virtual machines aren't created.

Logs are very information-rich

Rather than trying to piece together the numerical description of an event, the logs can tell you the intent of an application. Is it trying to scale up the nodes and failing, or is it not even trying? You might infer this intent from the metrics, but only the logs can tell you this outright.

The challenge is how to do anything with your logs at scale. In a microservices architecture, you may have hundreds of different applications, all producing a myriad of various logs that describe anything from the number of bytes received in an HTTP request through to a user making a sale on the site. These logs can be in different formats, use different terminology, attach different fields, measure the same thing slightly differently, and more. This means that the immense value of the logs is obfuscated behind the unstructured nature of the data. The flexibility and inconsistency of logs are both the source of their value and their complexity. This calls for a solution that brings you all the tools you need to index, query, analyze, and visualize your logs. Conveniently, one such solution has existed for a long time. Elasticsearch.

Elasticsearch and why it has become the go-to for modern logging

It's unlikely that you'll spend more than five minutes reading about logging before running into Elasticsearch. At its core, Elasticsearch is a document database that allows you to store, analyze, query, and visualize large volumes of data. Elasticsearch comes packed with configurable settings that enable you to fine-tune your cluster precisely what you need.

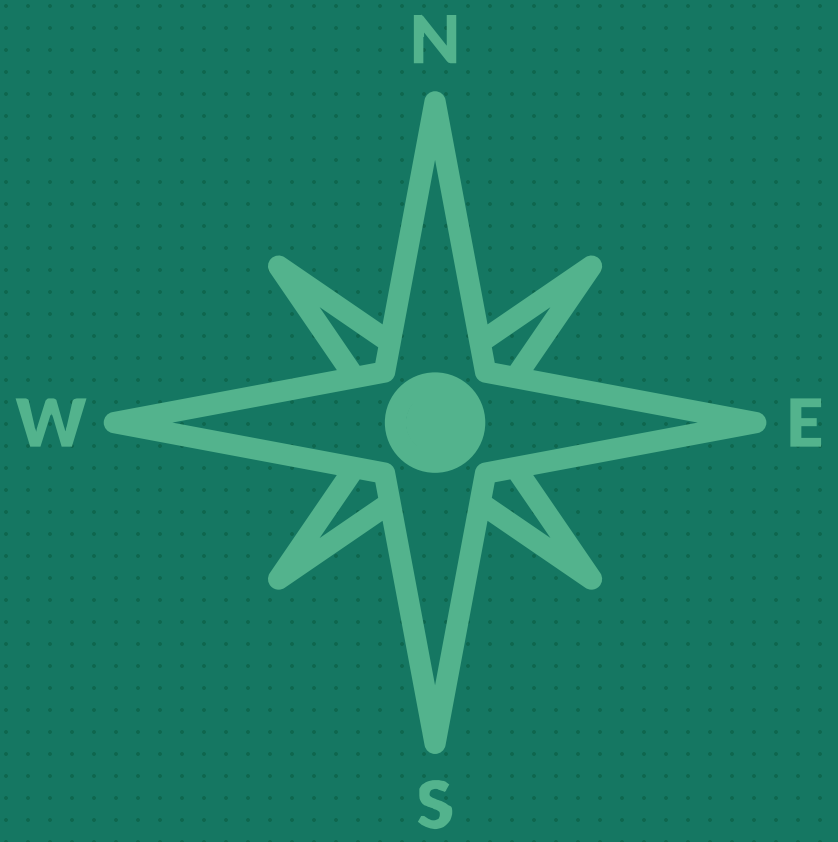
UNLOCKING THE POTENTIAL IN YOUR LOGS

Its popularity is inextricably linked to its ability to rapidly process and index logs and transform them from opaque lines in a text file into indexed objects that can be queried and analyzed. It can do this with millions, even billions, of log lines and, when configured correctly, can be built to scale to almost any logging challenges.

Later on, this eBook will discuss just how complex managing a high-performance Elasticsearch cluster is and some of the resources you may wish to read if you plan on going down this road into production.

<PART 3>

THE MOST COMMON DEPLOYMENT OPTIONS IN KUBERNETES



THE MOST COMMON DEPLOYMENT OPTIONS IN KUBERNETES

In the early days of Kubernetes, this choice didn't exist. You simply applied a mountain of YAML files to your cluster and tweaked them until your cluster began to behave itself. Unsurprisingly, the engineers operating these pioneer clusters were quite unhappy with this workflow. It was messy and complex and very open to human error. It was also tough to see why something was broken.

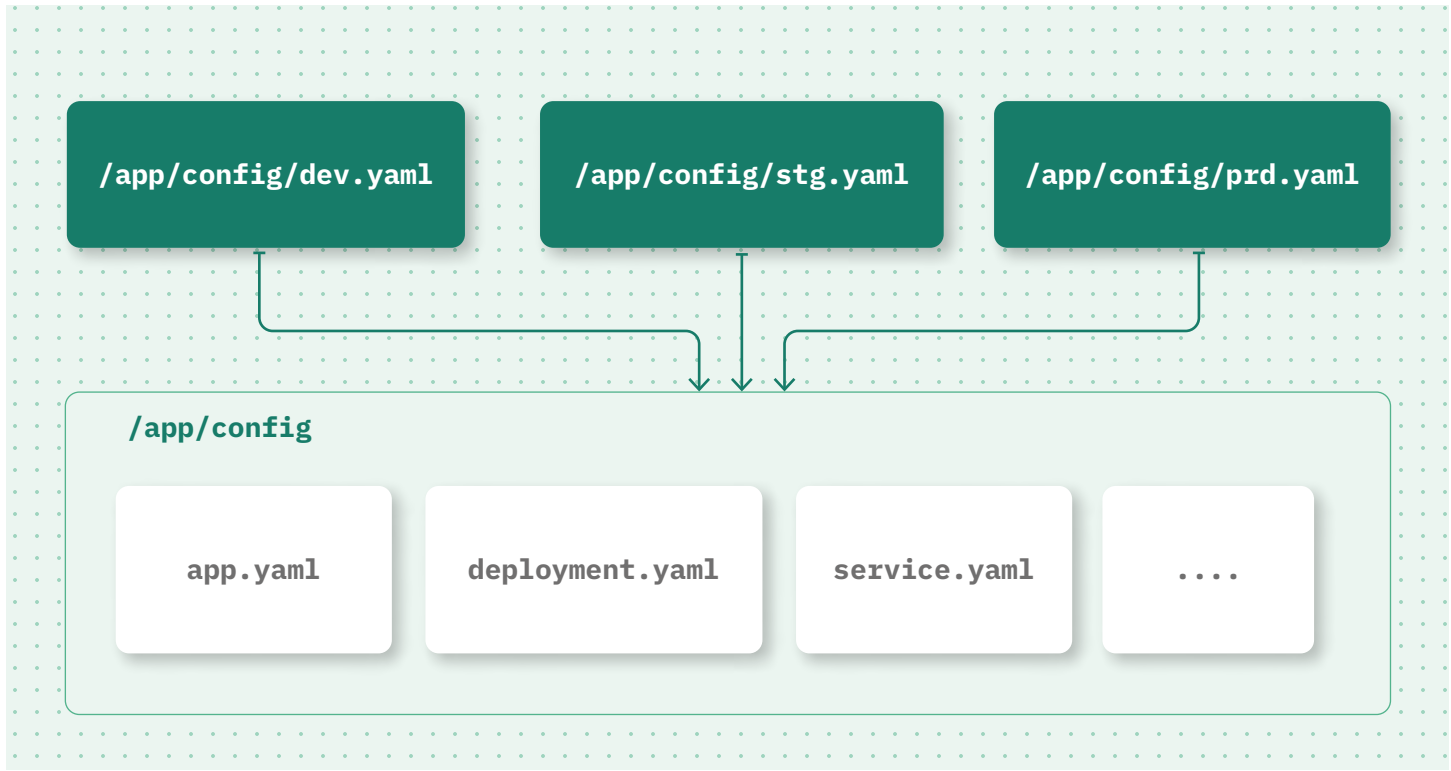
You had to scroll through pages of YAML files until the problem presented itself, and as applications began to scale, this became a more significant portion of the K8s operator's time. Not to mention the drift that occurred between different copies of the duplicate YAML files. Kubernetes had exposed a great, consistent API, but it had a challenge ahead if it was going to solve this problem.

Instead, they needed to group YAML files into a single, coherent, logical object. Necessity drove the creation of a new tool that quickly became the standard across the Kubernetes world - Helm.

Helm charts and why they're amazing

Helm charts are a collection of YAML files templated with different values. For example, you may have a collection of core YAML files describing how to deploy your application, but their values may differ for different environments. You may wish to switch on *DEBUG* level logging in your development environment but not in your production environment. In the old world, this would have meant copying and pasting YAML files into separate folders. You simply apply a different set of values to your YAML templates with helm.

THE MOST COMMON DEPLOYMENT OPTIONS IN KUBERNETES



This opened up a new ecosystem of shared Helm charts. Now, rather than understanding every minor component of a piece of software, you would quite simply make use of a remote helm chart and install it, configuring only the parts you were interested in. Therefore, allowing vendors to release helm charts with best practices *baked into* the chart itself!

Helm charts made installing 3rd party software on Kubernetes more straightforward. The community quickly swarmed around Helm and began using it voraciously, building helm chart after helm chart to solve common problems and sharing them amongst Kubernetes operators worldwide.

As with all innovations, though, it had some drawbacks. Now, no one knew what they were applying to their clusters, which made some operators and engineers initially uncomfortable. Hidden inside a helm chart could be a manifest file to deploy a suite of Bitcoin miners to your

cluster. This unknown was enough to drive the more cautious part of the community away initially, but many have now accepted this risk in recent years and moved on.

There was, however, one persistent drawback to this approach. This approach made it easier for vendors to put the “best practice” into their helm charts so that the software’s layout, once installed, conformed to the optimum standards. The result removed the burden from the engineer, but there was still another gap - a chasm began to present itself any time the software was due for an upgrade.

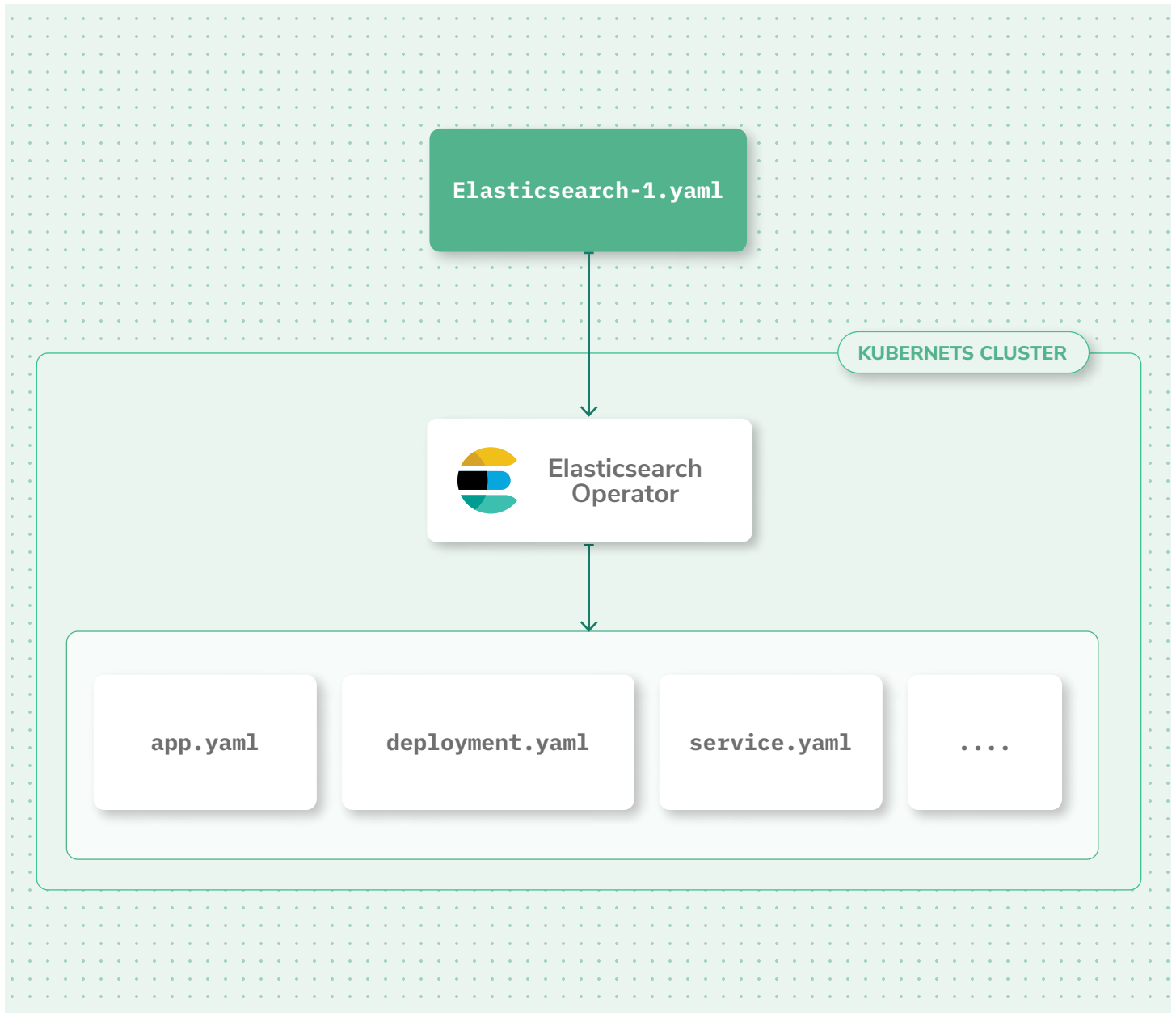
So far, Helm has managed to abstract the specific YAML files that you have installed onto your cluster. While it’s a significant step forward, the industry needed to answer a fundamental question that had existed ever since one company installed the software from another company. How do you upgrade without breaking everything?

Operators and why they’re becoming the standard

Helm charts made it easy to deploy the correct YAML in the configuration that the vendor recommended, but it didn’t make it easy to *change* that YAML once deployed. It simply relied on the Kubernetes control plane to determine which resources needed to be replaced. This is fine for totally stateless applications that can be dropped and redeployed at any time. For applications like Elasticsearch, however, there is an order of play to a smooth upgrade, and the Kubernetes API isn’t the correct place to bake this logic.

THE MOST COMMON DEPLOYMENT OPTIONS IN KUBERNETES

Instead, the industry created operators. Operators are software that runs on the Kubernetes cluster. They function by listening for signals in the cluster and making a change when those signals change. These signals often come in the form of resources declared on the cluster. When a new resource is declared, the operator will read it and attempt to *reconcile* the cluster state with what it sees in the resource. In the context of deploying new software, this means that rather than installing new software directly, the operator is deployed (which is typically very straightforward). You, the engineer, *ask that operator* to install the software on your behalf.



THE MOST COMMON DEPLOYMENT OPTIONS IN KUBERNETES

The operator architecture has some profound benefits. It allows vendors to create operators that install the application according to the best practice and upgrade it in the smoothest possible way. It also further reduces duplication of effort. Now, not every company needs complex logic to decide if a deployment went well or not. Instead, it can use the operator and simply *declare* which version of the software it wants. The operator handles the rest.

Naturally, the same people who were a little nervous about Helm charts were also worried about operators, but in much the same fashion, they're marching forward anyway. Operators represent the most advanced mechanism that Kubernetes users have for deploying 3rd party software, and they dramatically reduce the operational burden. Operators are a new technology, so bugs are expected, but the potential wrapped up in this deployment mechanism is boundless.

<PART 4>

RUNNING ELASTICSEARCH ON KUBERNETES

RUNNING ELASTICSEARCH ON KUBERNETES

Okay, so this eBook has covered why Elasticsearch is excellent and why it has become the standard choice for most companies looking to unlock the power of their logs. Now, you're left with some engineering questions.

- ➜ **What is the best way to deploy Elasticsearch onto your cluster?**
- ➜ **How do you monitor Elasticsearch once you have it?**
- ➜ **How do you make it as easy as possible to upgrade your Elasticsearch cluster?**
- ➜ **What are the operational concerns with Elasticsearch?**

The following chapters will explore the best practices for deploying Elasticsearch onto a modern Kubernetes cluster. Of course, this will include an explanation of common engineering patterns in Kubernetes that lift much of the operational burden away from you and place it into the hands of your cluster.

Prerequisites

You're going to need a few things if you wish to run through this section from start to finish:

- A running Kubernetes cluster, which you have sufficient permissions to install new Kubernetes resources, for example, Minikube.
- A basic understanding of Kubernetes commands.
- A deployed Helm CLI that is available.

Once these are in place, it's time to make a choice. Do you install it, or does your cluster install it on your behalf? To understand this choice, it's time to explore a common challenge that engineers face when deciding on the best way to install a new tool onto Kubernetes.

Elasticsearch deployments, two ways

You have two options when it comes to deploying Elasticsearch onto your Kubernetes cluster. The first is the simple, direct method - use helm to install Elasticsearch, which will get you up and running quickly. The second, and now recommended mechanism, is to use the Elasticsearch operator.

DIRECT INSTALL USING HELM

Installing Elasticsearch via the Helm chart is very straightforward. To begin, you'll need first to make the Elastic Helm repository available to your local Helm installation. So run the following command:

```
helm repo add elastic https://helm.elastic.co
```

This command adds the elastic helm charts to the local helm CLI. It should output something like this:

```
“elastic” has been added to your repositories
```

Next, you need to get your hands on a values file. As you saw earlier, a values file is used to template the YAML with some common values that you can tweak. You can do this from the command line, but it’s just as easy to navigate to one of the example files on Github and download it.

> ➜ **NOTE: The values file is in no way to be considered “production-ready.” You and your specific requirements with Elasticsearch define the production readiness. This book covers some operational readiness later on.**

Once you have your file and you’re ready to go, the next step is simple!

```
helm install elasticsearch elastic/elasticsearch -f .
/values.yaml
```

This step will begin the installation process of Elasticsearch. You didn’t need to inspect any YAML to get this working. You simply declared what you would like, and the Elasticsearch helm chart did the rest for you. You’ll see some helpful output from the helm chart:

```
NAME: elasticsearch
LAST DEPLOYED: Fri Jan 28 07:29:14 2022
NAMESPACE: default
STATUS: deployed
REVISION: 1
```

➜ NOTES:

```
1. Watch all cluster members come up.
   $ kubectl get pods --namespace=default -l app=elasticsearch-
master -w
2. Test cluster health using Helm test.
   $ helm --namespace=default test elasticsearch
```

These commands in the NOTES section are some simple commands you can run to watch the installation happen and test the rollout's success. You can watch the pods and wait until the master nodes have entered into a "ready" state. Then simply run the *helm test* command:

```
helm --namespace=default test elasticsearch
```

You should see some output indicating that the pods have been deployed. Congratulations, you now have a simple Elasticsearch cluster running on your Kubernetes platform!

Installing using the ECK Operator

Now you've done the direct install, let's look at the difference when you install via the operator. It's time to apply two separate things. The first is the installation for the operator, and the second is the custom resource that defines what the desired end-state looks like. Finally, this section will cover a cluster upgrade to see how the operator handles the process.

➜ **NOTE: Version 1.9.1 of the ECK operator has been specified. You should look to see what the latest stable version of the operator is before installing.**

Let's install the custom resource definitions into your cluster. These custom resource definitions are the ECK operator's interface to communicate with the user. Most operators use custom resources because it allows them to define their own API.

```
kubectl create -f https://download.elastic.co/downloads/eck/1.9.1/crds.yaml
```

If you want, you can navigate to the URL and inspect the YAML files to understand better what you're installing on your cluster. Once the CRDs are in place, it's just a case of installing the operator.

Two methods are currently available for installing the operator. The first is a good, old-fashioned YAML application, which is the recommended method. The second is to use the experimental helm chart that will allow you to use all of Helm's great features for bundling and templating your YAML. Let's go with the recommended method right now, but keep track of the helm chart because it will likely become the standard soon enough.

```
kubectl apply -f https://download.elastic.co/downloads/eck/1.9.1/operator.yaml
```

Once you've applied this, you can check the progress by inspecting the pods, using the following command:

```
kubectl get pods -n elastic-system
```

Or you can inspect the logs of the operator with the following command:

```
kubectl -n elastic-system logs -f statefulset.apps/elastic-operator
```

You'll know when the operator is healthy because the pod will go into a ready state. When you get the pods in your *elastic-system* namespace, you should see an output that looks something like this:

NAME	READY	STATUS	RESTARTS	AGE
elastic-operator-0	1/1	Running	0	96s

Define your credentials

Before you declare your cluster, set a simple password, just for this section. In the future, you can set something more robust:

```
export PASS="whatever-you-like"
kubectl create secret generic my-simple-cluster-es-elastic-user
--from-literal=elastic=${PASS}
```

This will set the Elasticsearch username to "Elastic" and the password to a value of your choosing. If you don't specify this secret, the operator will declare a random secret for you. While this is quite secure, it can be tricky to have little control over your cluster's credentials.

And now to declare the new cluster

Once the operator is installed, declaring a new cluster is straightforward. You'll need to define a new custom resource that you apply to the cluster. This acts as an instruction to the operator. The beauty of this approach is that you're not telling the operator how to do it. You're simply declaring that you would like a given version of Elasticsearch and letting the operator handle the details.

This YAML will work for a straightforward Elasticsearch cluster, to begin with, but this can grow into a more complex and finely tuned platform as you deepen your understanding:

```
apiVersion: elasticsearch.k8s.elastic.co/v1
kind: Elasticsearch
metadata:
  Name: my-simple-cluster
spec:
  version: 7.16.3
  nodeSets:
  - name: nodes
    count: 1
    config:
     node.store.allow_mmap: false
```

There are a few details to note here:

- This custom resource is not part of the core Kubernetes API. You can see this from the `apiVersion` field. It is an entirely new API that the ECK operator has installed on the cluster.

- Version 7.16.3 is specified. You should investigate to see the latest version to avoid falling behind valuable patches.

- The cluber only has a single “node set” with only one node in there. A production Elasticsearch cluster would almost universally have three nodes.

➜ **This custom resource does NOT need to live in the `elastic-system` namespace alongside the operator. This means you can keep the operator out of the way to avoid any core operator mistakes. At the same time, you define the best Elasticsearch cluster for your organization.**

➜ **`mmap` has been disabled, which is a performance optimization that most production clusters will need.**

Write the custom resource text above into a file, and call it `cluster.yaml.` Then, it's simply a case of applying it, as you would with any other YAML file.

```
kubectl apply -f cluster.yaml
```

What you'll see is the custom resource was created. The output won't mention anything about any other resources being made. This is because the operator is handling it for you. Now, if you look at pods in your default namespace, you'll see the cluster coming to life:

```
kubectl get pods
```

You'll see that you now have a single pod in your default namespace, which is the operator dutifully going about its business. Once it's healthy, you have a running Elasticsearch cluster.

The operator also presents a read API

The beauty of custom resources is that they can present APIs via the Kubernetes interface specialized to your question. Try running the following command:

```
kubectl get elasticsearch
```

You'll see that you can get a high-level cluster status check right there in the CLI, which simply wouldn't be possible with the direct Helm installation.

What about changing the version?

Changing the version of your cluster is as simple as opening your cluster.yaml file and changing the version in there. Simply reapply the YAML with whichever you like. This is the beauty of making use of the operator. Now, you don't need to expressly know how the upgrade (or, indeed, rollback) will be executed. You declare your intention and let the Kubernetes cluster reconcile.

Let's try rolling back a patch version just to see what happens. Open up your file and replace the contents with:

```
apiVersion: elasticsearch.k8s.elastic.co/v1
kind: Elasticsearch
metadata:
  Name: my-simple-cluster
spec:
  version: 7.16.2
  nodeSets:
  - name: nodes
    count: 1
    config:
        node.store.allow_mmap: false
```

The version has gone from *7.16.3* to *7.16.2*, which is common in situations where the latest patch has some incompatibility or bug. Now, as before, it's simply a case of running:

```
kubectl apply -f cluster.yaml
```

Your pod will automatically be taken down and replaced with a new pod. If you rerun the status command, you can see your new version:

```
> kubectl get elasticsearch
```

NAME	HEALTH	NODES	VERSION	PHASE	AGE
my-simple-cluster	green	1	7.16.2	Ready	14m

Let's put Kibana in front of your cluster

Okay, so you have some logs, but you can't see what's going on! Let's get Kibana deployed into your Kubernetes environment, so that you have a nice, friendly interface into your cluster. This is where the operator shines. Create a new file called *kibana.yaml* and add the following contents:

```
apiVersion: kibana.k8s.elastic.co/v1
kind: Kibana
metadata:
  name: my-kibana-deployment
spec:
  version: 7.16.3
  count: 1
  elasticsearchRef:
    name: my-simple-cluster
```

The beauty of this resource is that you don't need to specify things like hostnames. You could simply refer to the previous cluster you declared

and let the operator pull the details. This further demonstrates why the operator is a brilliant mechanism for deploying your cluster. Apply this file and watch the magic happen.

```
kubectl apply -f kibana.yaml
```

Now, you can use the Operator API to check the health of your kibana instance:

```
kubectl get kibana
```

Once the health of your Kibana instance goes green, you're set! If you don't have any ingress set up for your cluster, you may need to work out the best way of viewing this Kibana instance. To quickly verify, you can set up port forwarding to see the traffic's output.

First, look for the name of your service in your cluster:

```
kubectl get svc
```

Once you have the name of the service, you can set up the port forwarding. Forward traffic to port 5601 on the container. Port 5601 is the default HTTP interface for Kibana. For example, with a service called, *my-kibana-deployment-kb-http*, the command looks like this:

```
kubectl port-forward service/my-kibana-deployment-kb-http
5601:5601
```

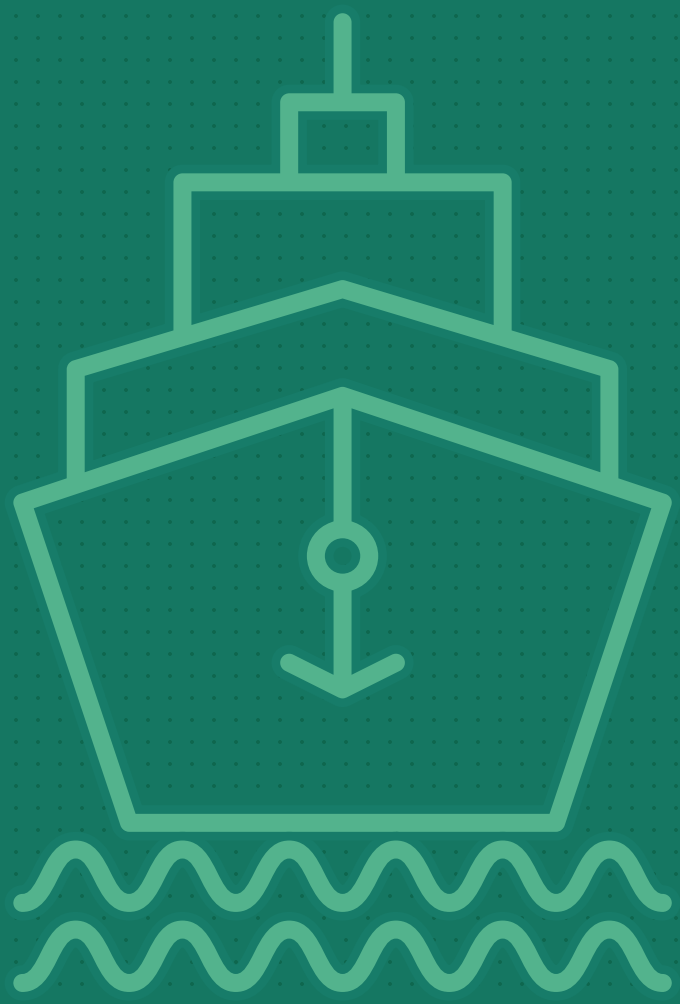
This is routing traffic from your local port 5601 to the container port. Now, if you navigate to this local host, you will see your Kibana instance!

➜ **NOTE: You may get an error about invalid certificates because Kibana uses a self-signed certificate when you install it like this. This is straightforward to fix and should be done for all production deployments.**

Finally, you'll simply need to log in with the password you specified earlier in this section. There you go, a running Kibana instance communicating with your new Elasticsearch cluster. However, once you explore a little, you'll realize that there's nothing much there to work with! Now, let's start sending logs from an application.

<PART 5>

SHIPPING YOUR LOGS INTO ELASTICSEARCH



SHIPPING YOUR LOGS INTO ELASTICSEARCH

So, whichever approach you decide to take, you've now got a working Elasticsearch cluster, sitting in Kubernetes, ready to index and analyze your logs as you need. You've also got a Kibana instance that will let you visualize your logs. Now, you need to start sending your logs to your cluster.

There are a few options for sending logs from an application into a centralized place.

- **Application libraries will automatically stream logs to Elasticsearch**
- **Logging agents can run on the virtual machine that will automatically pick up docker logs and send them to your cluster**
- **You can use the Kubernetes DaemonSet to deploy a pod per VM that will integrate with Elasticsearch**
- **You can leverage the ECK operator and let it work for you**

This section will work with the ECK operator to wire up a Filebeat instance that will automatically scrape and push your logs into your Elasticsearch instance. In the same way as deploying Kibana, let's begin by creating a local YAML file, `filebeat.yaml`.

SHIPPING YOUR LOGS INTO ELASTICSEARCH

```
apiVersion: beat.k8s.elastic.co/v1beta1
kind: Beat
metadata:
  name: my-filebeat-installation
spec:
  type: filebeat
  version: 7.16.3
  elasticsearchRef:
    name: my-simple-cluster
  config:
    filebeat.inputs:
    - type: container
      paths:
      - /var/log/containers/*.log
  daemonSet:
    podTemplate:
      spec:
        dnsPolicy: ClusterFirstWithHostNet
        hostNetwork: true
        securityContext:
          runAsUser: 0
        containers:
        - name: filebeat
          volumeMounts:
          - name: varlogcontainers
            mountPath: /var/log/containers
          - name: varlogpods
```

SHIPPING YOUR LOGS INTO ELASTICSEARCH

```
mountPath: /var/log/pods
   - name: varlibdockercontainers
     mountPath: /var/lib/docker/containers
 volumes:
 - name: varlogcontainers
   hostPath:
     path: /var/log/containers
 - name: varlogpods
   hostPath:
     path: /var/log/pods
 - name: varlibdockercontainers
   hostPath:
     path: /var/lib/docker/containers
```

Let's break down what this file is declaring:

- ➜ Call the custom resource "my-filebeat-installation."
- ➜ Specify in the spec that the type of this Beat resource is filebeat - this will differentiate it from other beats.
- ➜ Request version 7.16.3, in line with the cluster and Kibana.
- ➜ Define a DaemonSet inside the resource. You can see that a series of folders are listed in the DaemonSet definition. These folders contain the logs that are sent out from Kubernetes containers.

SHIPPING YOUR LOGS INTO ELASTICSEARCH

Once you're happy, apply this in the same way as everything else:

```
kubectl apply -f filebeat.yaml
```

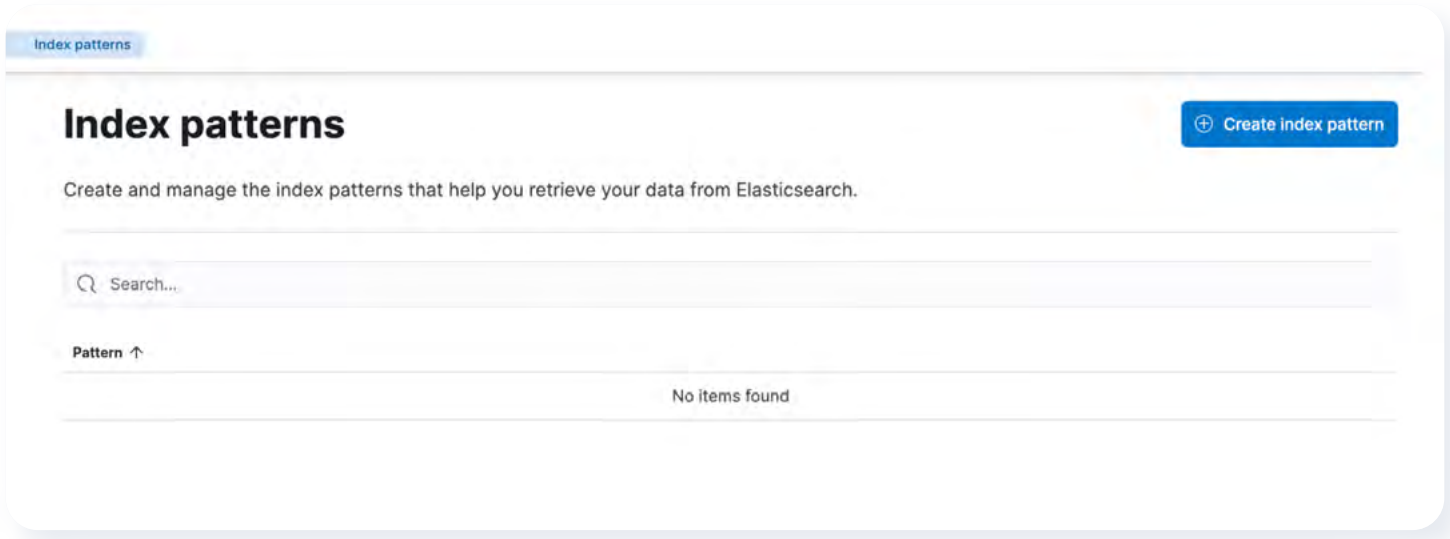
As before, you can simply fetch the high-level status of your filebeat instance using the Operator API.

```
kubectl get beat
```

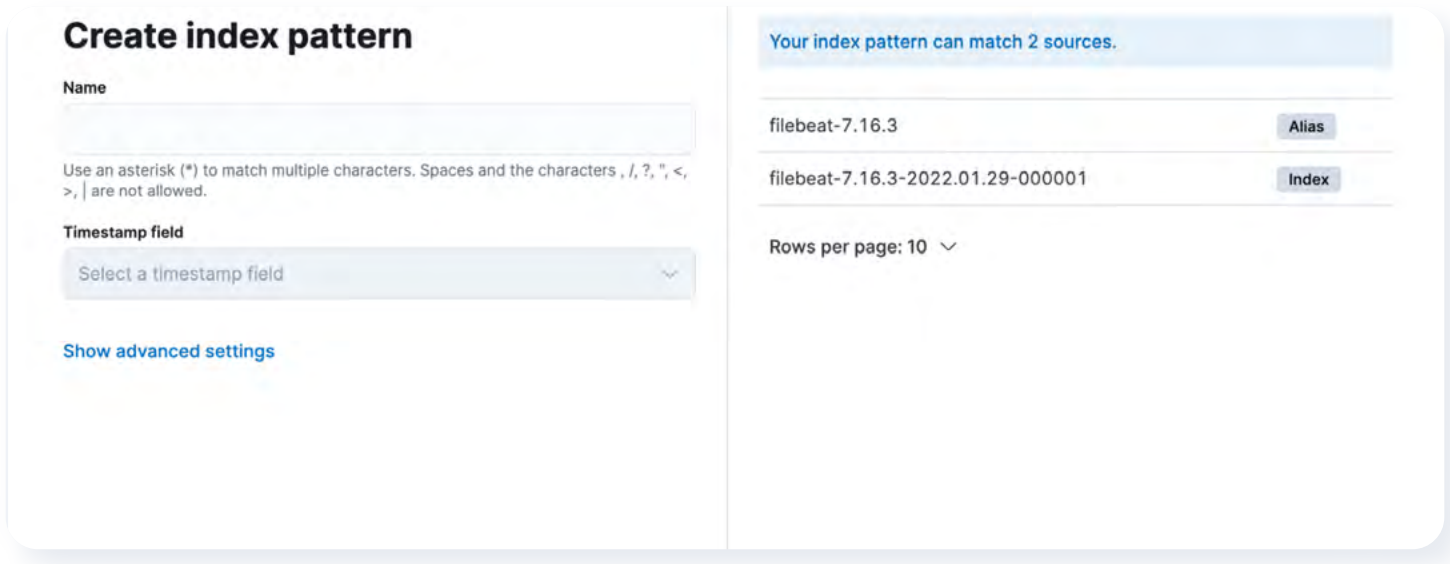
Wait until that goes green, and you have a pipeline of logs going straight to your Elasticsearch cluster. Despite having no applications deployed, you still have operator logs and control plane logs running on your cluster. This means that logs will already be shipped into your Elasticsearch instance.

Set up a simple Index Pattern

Now you have some logs you'll need to create an index pattern to query them. Filebeat would have already made some indices in your cluster, so your index pattern simply needs to match those. With your port forwarding open from before, merely navigate to this local host, which will bring up the management UI and allow you to view your index patterns.

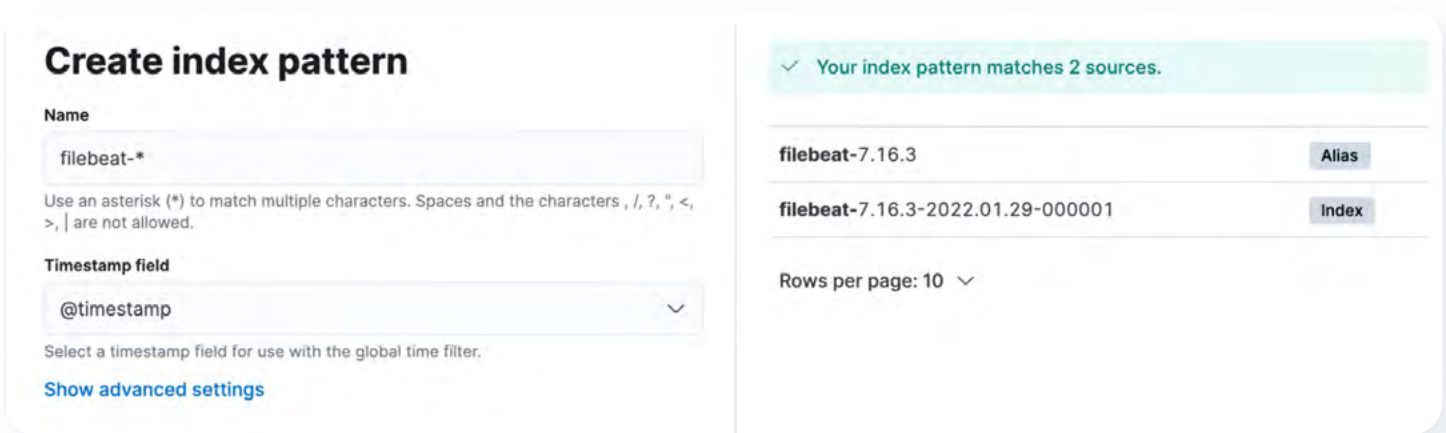


Click on the button in the top right of the screen to create a new index pattern. When you click it, you'll see something like this.

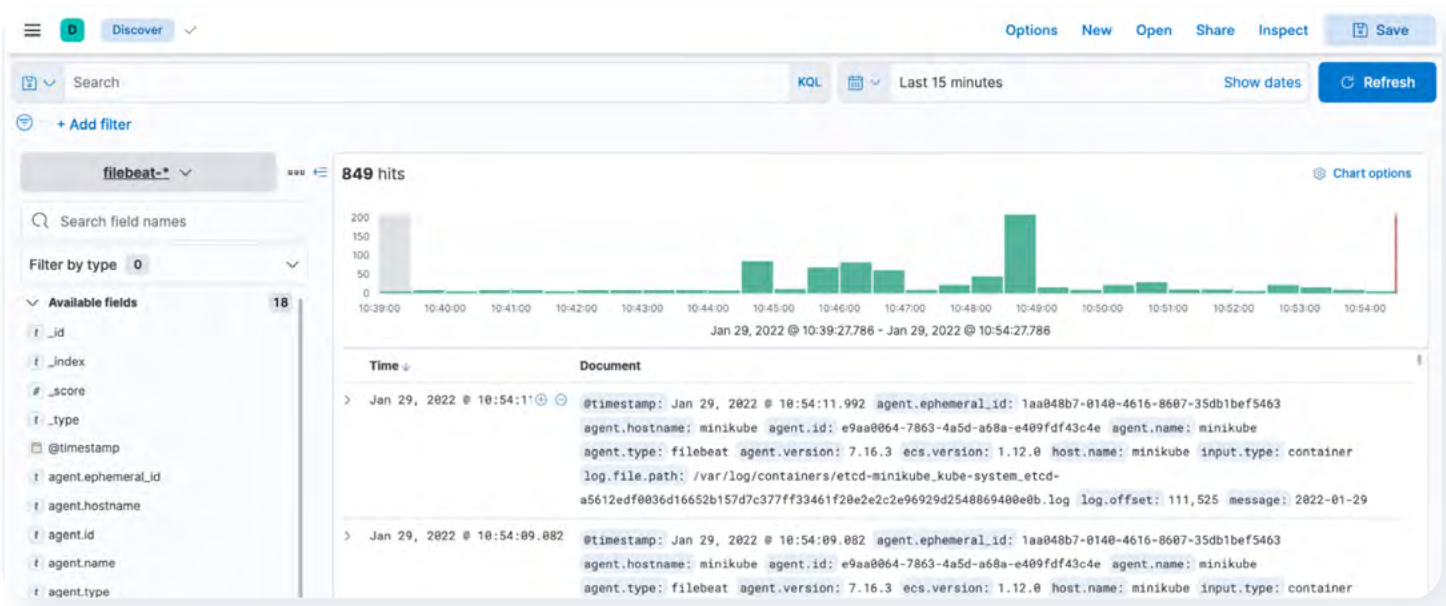


SHIPPING YOUR LOGS INTO ELASTICSEARCH

You can create a simple index pattern that will match all of your filebeat logs by typing *filebeat-** into the name for your index pattern. You should get a prompt to indicate that this will match your two filebeat indices. Finally, select the @*timestamp* field as your timestamp field.



Once you're happy, click on the button at the bottom of the screen to create your index pattern. It will take you to the index management page, where you will see all of your fields. Field mappings are a complex and exciting practice when operating an Elasticsearch cluster. Simply navigate back to the Discover section, and you should have your logs available to you:



A full pipeline

Congratulations! You have Elasticsearch storing, indexing, and analyzing your logs. You have Kibana, which allows you to query and view your logs, and you have Filebeat moving logs from your containers and into your Elasticsearch cluster. This represents one of the most common logging configurations on production clusters. While this is not quite a production setup just yet, you have the raw materials you need to create a great logging pipeline.

So can I start shipping logs?

It's not really recommended - configuring Elasticsearch can be a long and complex process. It will take some time and testing before you get it right. There are some things that the helm chart has done out of the box for you:

- **Configured some master nodes that are great for handling API requests**
- **Given some primary resources to each pod. Each pod has 512MB of memory and can burst up to a single core**
- **Set up pod anti-affinity meaning that two of your Elasticsearch pods won't deploy to the same node**

However, what it hasn't done (and this isn't an exhaustive list), but should be considered:

- **Performance tested your cluster to make sure it can handle the sorts of volumes you're expecting in production**

SHIPPING YOUR LOGS INTO ELASTICSEARCH

- ➜ **Set up authentication so that your information remains secure**
- ➜ **Test your clusters ability to recover from a node outage**
- ➜ **Set up nodes with different roles, such as data, ingest, voting, and coordinating, depending on what you need from your cluster**

Configuring and optimizing Elasticsearch is an eBook all on its own, but now that you have the basic deployment out of the way, you can now focus on which configuration is the best for your use case. Here are some friendly tips:

- ➜ **Consider the number, size, and type of node you need. It's not just about sizing CPU and memory but also about the underlying instance types for your virtual machines.**
- ➜ **To avoid the split brain problem, you will typically have at least 3 "master eligible" nodes.**
- ➜ **You should think about retention periods on your logs.**

<PART 6>

HOW TO GET THE MOST OUT OF YOUR LOGS

HOW TO GET THE MOST OUT OF YOUR LOGS

Once you have ingested your logs, you're going to find that they can be quite challenging to analyze. This is because lots of systems write logs in different formats, using different fields, conventions, etc. The challenge, then, is to work out which conventions you want to put in place to make your logs as usable and traceable as possible. Here are some ideas for getting the most out of your application logs.

Log in a structured format like JSON

JSON is a machine-readable, straightforward format that can easily be indexed and analyzed by Elasticsearch. This consistency makes it much easier and simpler to manage your logs. Ensuring that your applications are logging in JSON will encourage engineers to:

- ➜ **Put all of the necessary information on one logline, not split over multiple**
- ➜ **Allow for more fine-grained indexing of fields in your logs**
- ➜ **Allow 3rd party tooling to attach values onto your logs, such as the IP address**

Many logging agents support converting logs into JSON before sending them off to Elasticsearch. This ensures transparency in the application, but it does somewhat water down the benefits of focusing on structured logging.

Use a *Mapped Diagnostic Context*

Automatically inject thread local values into your logs without constantly adding them into your log lines. So rather than code that looks something like this:

```
log.info("Session started", { sessionId });
log.info("Session in progress", { sessionId });
log.info("Session finished", { sessionId });
```

You can simply set it once and automatically inject it into your logs.

```
MDC.add({ sessionId })

log.info("Session started");
log.info("Session in progress");
log.info("Session finished");
```

Some form of MDC library exists in almost every major language, and it completely removes the need for engineers to remember every key value in the logs. It also keeps the code nice and clean. This isn't a significant overhead with one value, but with 10 or 20, the MDC becomes necessary.

Should you build all of this yourself?

There is a crucial debate whenever a company embarks on its logging journey or any engineering journey. To build or to buy? Engineers typically lean towards build, while management sees the utility in purchasing an off-the-shelf solution.

There are no silver bullet answers for this, but you can begin by asking generic questions.

HOW TO GET THE MOST OUT OF YOUR LOGS

- **Is this a capability you're looking to create for your company?**
- **Does the engineering cost stack up against an existing product's upfront or subscription cost?**

They are tricky but common questions that need investigating at the beginning of any complex piece of work, but in the world of logging, there are some more difficult questions you need to ask:

- **Should you impact feature development while you're building your Elasticsearch cluster?**
- **Do you have the skills or the means to hire the right skills to maintain this cluster once you've built it?**
- **How soon do you need the benefit from your cluster?**

These questions are more challenging to answer and require serious investigation into the motivations behind investing in your observability. Of course, SaaS alternatives offer managed Elasticsearch instances, with rich features that include cost optimization, machine learning driven anomaly detection, alerting, metrics, and much more.

As you've seen in this eBook, a new Elasticsearch cluster is not something that you should approach lightly. It is a commitment of time, effort, and money. SaaS may be a viable alternative to help you quickly get the value out of your logs without incurring a brutal upfront cost and potentially even running at a lower operational price, with more features at your disposal.

TAKE AWAY

We have looked at Kubernetes patterns, how to deploy Elasticsearch, directly and indirectly, using K8s operators, and how to ship your logs from your application into your cluster. The final points covered the steps you need to consider to get to an operationally sound, production-ready Elasticsearch cluster.

The challenge ahead of you is significant, but with some clever engineering and creative thinking, you'll have a battle-ready Elasticsearch cluster that will give you new insights and drive your operational success for years to come.

Start solving your
production issues faster

Managed, scaled,
and compliant monitoring,
built for CI/CD