KUBERNETES FOR STARTUPS

Table of Content

kubernetes

Who Is This Book for?

There are a lot of guides for setting up Kubernetes on the internet. Most of them deal with creating a trivial cluster that could never support actual traffic, or they deal with topics your company won't need until much later in its life. This book focuses on what you need to launch your startup's app on Kubernetes for the first time. Reading this book will help you understand essential Kubernetes topics, build your first cluster, deploy your application, monitor your cluster, and learn about some next steps to implement as your app scales.

The first section of this book covers the components that make up a Kubernetes installation. After that, we build and deploy your cluster. Next, we'll talk about how to monitor your cluster. Finally, we'll talk about some tools and topics you've probably heard about in your research that you can implement later.

About Blue Matador

We've been there. This is the blueprint we followed as we built and deployed our own webapp in Kubernetes. We understand that when you're a fast moving startup, you don't have the luxury of spending days figuring out exactly what you need in a Kubernetes cluster.

We've also spent a lot of time working with and thinking about Kubernetes itself. Our software automatically monitors Kubernetes and alerts you when things go wrong without any configuration. With Blue Matador, you don't need to know what alerts to set in Kubernetes. You can spend your time building your startup, not monitoring your infrastructure.

An Intro to Kubernetes Components

First let's go over some of the basic terminology used with Kubernetes. Many of these terms should be familiar and Kubernetes itself is actually a very intuitive system once you understand how the different components work together.

What is a Pod?

A pod is the basic unit of an application running in Kubernetes. A pod is a group of one or more Docker containers (though more often than not, it's just one) that has one purpose, whether that's a web server, a job, or a cache. Pods are configured to use a specific amount of CPU and memory, which helps Kubernetes know how and where to schedule the pod.

What is a Node?

A node is a server running in a Kubernetes cluster. In traditional webapps, you would run a single microservice on a node, but Kubernetes will schedule multiple pods on a single node. Nodes that run application pods are called worker nodes, while nodes that run system pods for Kubernetes administration are called master nodes.

What is a Namespace?

A namespace is a virtual subset of your Kubernetes cluster. You can use Namespaces to organize your other Kubernetes components and control access to them using Role-Based Access Control. Most Kubernetes components such as pods, services, deployments, and daemonsets belong to a namespace while low-level components such as nodes and persistent volumes do not. When you are just getting started, you can usually use the **default** namespace, but as you become more familiar with Kubernetes you will want to create additional namespaces to organize your growing infrastructure.

What is a Deployment?

Using a deployment, you can define how many instances of a particular pod you'd like to have running at any time. If a pod dies, the deployment will spin up another instance. Deployments also allow you to conduct rolling updates where you can roll out a new version of a container a couple of pods at a time. This feature lets you maintain high availability while updating your app.

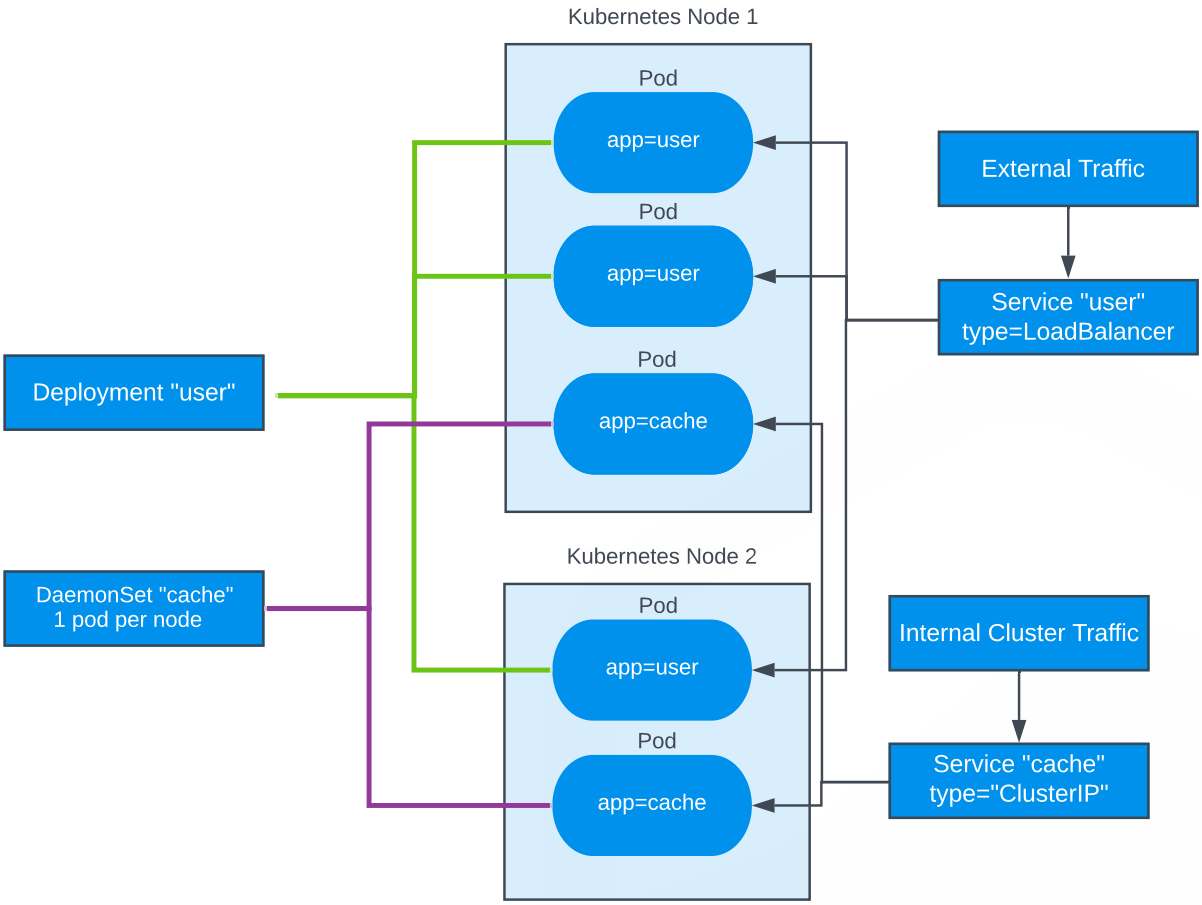
What is a DaemonSet?

While a deployment will schedule pods wherever there is capacity, a DaemonSet will make sure that exactly one instance of your pod is scheduled on each node. This is useful for ensuring monitoring tools or utilities like caches are available to all nodes in your cluster.

What is a Service?

A service groups pods and exposes them to the rest of the cluster, or even outside the cluster. For example, a service can cover a set of pods running a microservice and make them accessible by the name of the service. Kubernetes 1.11+ comes with CoreDNS installed, which will automatically create a DNS name for your services. Services can also be created as LoadBalancers to distribute traffic between the pods and allow external access.

Here is a diagram that shows how these components are related in a simple example Kubernetes cluster:



Creating Your First Cluster

In order to really understand the power and utility that Kubernetes provides, you have to get your hands dirty and create a Kubernetes cluster. There are a ton of resources out there already for creating a Kubernetes cluster, so we will just go over a few methods briefly and give you the resources you need to fully configure your cluster in the environment of your choosing. The four environments we will cover are:

- Development
- Amazon Web Services
- Google Cloud
- Azure

Development

Setting up a Kubernetes cluster in development is a surprisingly simple process and is a great way to explore Kubernetes without committing to a specific vendor or paying the cost of running a production-ready cluster.

Minikube is the de-facto development installation of Kubernetes. Minikube essentially runs a single Kubernetes node on a VM on your local machine, and provides utilities for interacting with Kubernetes locally. It makes it easy to quickly install specific versions of Kubernetes, update your kubectl configuration to point to your local cluster(s) and even has a helper for mounting local filesystems into your cluster for quick development and testing of Kubernetes.

After installation, you can start minikube with Kubernetes version 1.13.0:

```
minikube start --kubernetes-version v1.13.0
```

Then, tell kubectl to switch to the minikube context so you can target the local cluster:

```
kubectl config use-context minikube
```

If you want to mount a local directory so it can be accessed from within minikube, you can use the **minikube mount** command. This is useful when using minikube in development so that you can access your local filesystem from within docker containers running on minikube. The following command mounts the current directory to **/app**:

```
minikube mount .:/app
```

You may run into issues with minikube mount if your pods try to mount a directory before it is mounted in minikube. In most cases you can just re-run minikube mount and then recreate your pods to resolve the issue.

Amazon Web Services

AWS is one of the most common places to run a Kubernetes cluster. Running your Kubernetes cluster in AWS is probably the right move if you are already using or plan to use lots of AWS services. There are two recommended ways to run a Kubernetes cluster on AWS: using kops or EKS.

Kops is an open-source tool created to allow for easy creation, upgrade, and maintenance of production Kubernetes clusters. Kops will create master nodes and worker nodes for your cluster, and has many utilities built-in to automatically set up high-availability, networking, and manage configuration for the EC2 instances your nodes run on.

The downside of Kops is that you are still running the Kubernetes master nodes on your infrastructure and have to maintain the security of those nodes. In addition, upgrading your cluster can be difficult with kops since its interaction with the AWS API to manage EC2 instances can encounter errors, and rolling back during an upgrade can be very tricky.

Another solution to running Kubernetes on AWS is Amazon EKS. EKS is the Amazon-managed Kubernetes solution. What can be confusing about EKS is that the marketing material makes it appear to be a fully-managed Kubernetes solution, but EKS actually only manages the control plane (master nodes, API services) for your cluster. You still have to set up worker nodes and join them to your cluster. You can use eksctl, a command-line tool similar to kops for creating EKS clusters, or you can use a Terraform module to manage your cluster config if you use Terraform. Either of these tools will make it much simpler to get started on EKS.

Google Cloud Platform

GCP is another common cloud to run Kubernetes on. Since Kubernetes was created by Google, their GKE (Google Kubernetes Engine) service is tightly integrated with Kubernetes. GKE has the simplest method of creating a cluster that is ready for Docker images:

```
gcloud container clusters create [CLUSTER_NAME]
```

GKE also has extensive documentation for cluster administration and support for many features that other clouds do not have like automatic remediation, first-class log management for Kubernetes, and the newest versions of Kubernetes available.

Azure

For anyone developing with Windows, Azure is a natural choice to run Kubernetes. AKS (Azure Kubernetes Service) is an offering similar to EKS and GKE to allow for quick provisioning of Kubernetes clusters. Azure offers a tutorial for creating a Kubernetes cluster in AKS. Since AKS is a newer managed Kubernetes service, it may not be intuitive at times to use, but this should improve as Microsoft invests more into Azure and AKS.

Building And Deploying Your Application

Getting started with your first Kubernetes deploy can be a little daunting if you are new to Docker and Kubernetes, but with a little bit of preparation your application will be running in no time. In this section we will cover the basic steps needed to build Docker images and deploy them to your Kubernetes cluster.

Docker Build

The first step to deploying your application to Kubernetes is to build your Docker images. I will assume you have already created Docker images in development to create your application, and we will focus on tagging and storing production-ready Docker images in an image repository.

The first step is to run docker image build. We pass in . as the only argument to specify that it should build using the current directory. This command looks for a Dockerfile in your current directory and attempts to build a docker image as described in the Dockerfile.

```
docker image build .
```

If your Dockerfile takes arguments such as **ARG app_name**, you can pass those arguments into the build command:

```
docker image build --build-arg “app_name=MyApp” .
```

You may run into a situation where you want to build your app from a different directory than the current one. This is especially useful if you are managing multiple Dockerfiles in separate directories for different applications which share some common files, and can help you write build scripts to handle more complex builds. Use the **-f** flag to specify which dockerfile to build with:

```
docker image build -f “MyApp/Dockerfile” .
```

When using this method, be mindful that the paths referenced in your Dockerfile will be relative to the directory passed as the final argument, not the directory the Dockerfile is located in. So in this example, we will build the Dockerfile located at **MyApp/Dockerfile** but all paths referenced in that Dockerfile for COPY and other operations will actually be relative to the **current working directory**, not MyApp.

Tagging

After your docker image has been built, you will then need to tag your image. Tagging is very important in a docker build and release pipeline since it is the only way to differentiate versions of your application. It is common practice to tag your newest images with the **latest** tag, but this will be insufficient for deploying to Kubernetes since you have to change the tag in your Kubernetes configuration to signal that a new image should be ran. Because of this, I recommend tagging your images with the git commit hash of the current commit. This way you can tie your docker images back to version control to see what has actually been deployed, and you have a unique identifier for each build.

To get the current commit hash programmatically and then tag your image, run:

```
git rev-parse --verify HEAD
```

You can then tag your image like so:

```
docker image tag $IMAGE MyApp:$COMMIT
```

Tagging your image after it is built can be useful for fixing up old images, but you can and should tag them as part of the build command using the **-t** argument. With everything put together, you could write a simple bash script to build and tag your image:

```
#!/bin/bash
COMMIT=$(git rev-parse --verify HEAD)
docker image build -f “MyApp/Dockerfile” . \
  --build-arg “app_name=MyApp” \
  -t “MyApp:latest” \
  -t “MyApp:${COMMIT}”
```

Docker Repositories

Now that you have your Docker images built and tagged, you need to store them somewhere besides on your laptop. Your Kubernetes cluster needs a fast and reliable Docker repository from which to pull your images, and there are many options for this.

One of the most popular Docker image repositories is dockerhub. For open source projects or public repositories, dockerhub is completely free. For private repositories, dockerhub has very reasonable pricing.

To push images to dockerhub, you must tag your images with the name of the dockerhub repository you created, and then push each tag. Here is an example of tagging and pushing the latest image built above:

```
docker image tag MyApp:latest myrepo/MyApp:latest
docker login
docker push myrepo/MyApp:latest
```

As with any tag, you can tag your image during the build using the **-t** argument instead of tagging it later. When pushing tags to your remote repository, you will need to push **each** tag that you want access to. Even if your latest tag is the same image as another tag, they must be pushed separately to the remote repo so each of them can be used in your Kubernetes configuration.

For anyone already using Amazon Web Services, Amazon Elastic Container Registry provides cheap and private docker repositories. You can similarly tag and push docker images to your ECR repository if you have the AWS CLI installed. Just replace **ECR_URL** in the following example with the actual URL for your ECR repository, which can be viewed in the AWS Web Console.

```
docker image tag MyApp:latest ECR_URL/MyApp:latest
eval $(aws ecr get-login --no-include-email)
docker push ECR_URL/MyApp:latest
```

GCP users can use Container Registry to store their Docker images. Simply configure your GKE instances to have access to your registry, and then use the **gcloud** tool to authenticate with the repo. Replace PROJECT_ID with your GCP project ID:

```
gcloud auth configure-docker
docker image tag MyApp:latest gcr.io/PROJECT_ID/MyApp:latest
docker push gcr.io/PROJECT_ID/MyApp:latest
```

Azure also has a private container registry with similar features to dockerhub, ECR, and Container Registry. You can follow this tutorial to set up and push images to the Azure Container Registry.

Deploying

Now that you have built and pushed your Docker images, you can deploy them to your Kubernetes cluster. The quickest way to get started is by using **kubectl**. You can create a Deployment in your cluster by following the Kubernetes documentation. Here is an example configuration for a Deployment to run 3 copies of the example MyApp image and expose port 80:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: MyApp
  labels:
    app: MyApp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: MyApp
  template:
    metadata:
      labels:
        app: MyApp
    spec:
      containers:
      - name: MyApp
        image: myrepo/MyApp:latest
        ports:
         - containerPort: 80
```

Once you have configured your deployment, you will not need to modify most of the options when you update your app except for the **image** attribute on your containers.

You'll notice in the example I have used the **latest** tag. When you first create a Deployment, it will pull the correct image and run it. If you update the latest tag to a newer Docker image, Kubernetes will have no way of knowing that it needs to pull a new image and deploy new Pods. This is why we should not use latest when deploying to Kubernetes. By using another tag such as the git commit, we can simply update the deployment using **kubectl edit deploy/MyApp**, change the **image** attribute, and save. Now Kubernetes will detect that the MyApp Deployment has changed, and will rotate out the old pods for new ones automatically.

For an in-depth look at how Deployments update your pods, you can check out our detailed blog post.

A common issue you may run into is permissions issues when pulling your Docker image from the repository. For public repositories, this should not be a problem. For private repositories, you will have to dig into the documentation for dockerhub, Amazon ECR, Google Container Registry, or Azure Container Registry to figure out how your Kubernetes worker nodes should authenticate with the registry.

When creating or updating a Deployment, you can check its update status with the following command:

```
kubectl rollout status deploy/MyApp
```

Once the deploy is complete, we can now make a Service to expose our pods for access. Let's create a basic **ClusterIP** Service that exposes our pods only within the Kubernetes cluster:

```
apiVersion: v1
kind: Service
metadata:
  name: MyApp
  namespace: default
  labels:
    app: MyApp
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: MyApp
  type: ClusterIP
```

Notice how the **selector** for our Service matches one of the **labels** from our Deployment. This is what allows Kubernetes to route traffic directed at our service to our pods. Now you can use the dns name **MyApp** to send traffic to your pods from within the cluster:

```
> curl http://MyApp:80
> Hello World
```

Most applications will want to allow external access at some point. This can be accomplished by using a Service with type LoadBalancer. LoadBalancer services will create an internal Kubernetes Service that is connected to a Load Balancer provided by your cloud provider (AWS, GCP, or Azure). This will create a publicly addressable set of IP addresses and a DNS name that can be used to access your cluster from an external source.

```
apiVersion: v1
kind: Service
metadata:
  name: MyApp-public
  namespace:default
  labels:
    app: MyApp
spec:
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
  selector:
    app: MyApp
  type: LoadBalancer
```

The specifics on how the cloud's Load Balancer can be configured are specific to the cloud. Read the Kubernetes documentation for debugging issues with your specific cloud provider.

Now that your LoadBalancer service is created you should be able to see a corresponding resource in your cloud provider's dashboard. You can use the public DNS and IP addresses to access your service externally.

Security Essentials

Now that you know how to create your first Kubernetes cluster and deploy some applications to it, you should take a moment to think about security. We all know that there will always be another vulnerability, and another breach, but we have to do our best to secure the things we control as best as we can. This is by no means an exhaustive list of security items to check, but should get you started on the right path.

Upgrading

Kubernetes has over 2,000 individual contributors and is updated frequently. With more eyes on it, security vulnerabilities are also being discovered and patched more frequently. It is important to stay reasonably up-to-date on Kubernetes versions especially as it matures. How you upgrade your cluster depends on what tool or service you used to create it:

- Upgrading with Kops
- EKS Cluster Upgrade
- GKE Cluster Upgrade
- AKS Cluster Upgrade

Try to stay no more than 1 or 2 major versions behind on Kubernetes, and take advantage of the existing tools to help you upgrade often and without service disruption.

Restrict API Access

Most cloud implementations for Kubernetes already restrict access to the Kubernetes API for your cluster by using IAM (Identity & Access Management), RBAC (Role-Based Access Control), or AD (Active Directory). If your cluster does not use these methods, you can usually set up one of these methods using open source projects for interacting with various authentication methods. We also recommend restricting API access by IP address if at all possible, only allowing access from trusted IPs such as a VPN or bastion host.

Restrict SSH Access

Another easy and essential security policy to implement in your new cluster is to restrict SSH access to your Kubernetes nodes. Ideally you would not have port 22 open on any node, but you may need it to debug issues at some point. You can configure your nodes via your cloud provider to block all access to port 22 except via your organization's VPN or a bastion host. This way you can quickly get SSH access but outside attackers will not be able to.

Namespaces

If your cluster acts as a multi-tenant environment, you can and should use Namespaces to restrict access to resources within the cluster. Namespaces, together with RBAC, will let you create user accounts that have access only to particular resources. In this example, we create a user **MyDevUser** that only has access to resources in the **development** namespace:

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: MyDevUser
  namespace: development
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: MyDevUser
  namespace: development
rules:
- apiGroups: [“”, “extensions”, “apps”]
  resources: [“*”]
  verbs: [“*”]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: MyDevUser
  namespace: development
subjects:
- kind: ServiceAccount
  name: MyDevUser
  namespace: development
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: MyDevUser
```

You can also configure your namespaces to restrict the amount of memory and CPU that are allowed to run in that namespace. This can help prevent rogue deployments in development or QA from affecting the available resources in production.

Network Policies

Network policies also allow you to restrict access to services within your Kubernetes cluster. You can also use them to restrict access to your cloud's metadata API from pods in your cluster. Follow this documentation to set up a network policy.

Do Not Run As Root

One of the most overlooked security issues is running the containers in your Pods as the root user. In Kubernetes, the UID of the user running a container is mapped directly to the host. This means that if your container runs as UID 0 (root) it will also appear as root on the node it is running on. Kubernetes has built-in protections to prevent escalation of privileges with this mechanism, but there is always the risk of a security vulnerability or exploit where a container could escalate privileges this way.

The way around this is usually quite simple: do not run your containers as root. You can accomplish this by modifying the Dockerfile for your built containers to create and use a user with a known UID. For example, here the beginning of a Dockerfile that adds a user named **user** with UID 1000 to an image for Java 8:

```
FROM openjdk:8-jre-slim-stretch

USER root

RUN groupadd -r user --gid=”1000” \
  && adduser --home “/home/user” --gid “1000” --disabled-password --disabled-login --gecos ‘’
--shell “/bin/bash” --uid “1000” user \
  && chown -R user /home/user

USER 1000
```

Notice that we use **USER 1000** instead of **USER user** to declare which user is used going forward. We do this for the sake of consistency with Kubernetes. When you configure your Kubernetes manifest to run your container, you can specify what UID the container must run as to enforce that the correct user is used. This is especially useful for larger teams where cluster security may be enforced by a different team than the one writing the Dockerfiles. Simply add these lines to your **spec.containers** to enforce that the container is ran as UID 1000.

```
      securityContext:
        runAsUser: 1000
        allowPrivilegeEscalation: false
```

You can also enforce that non-root users are used using PodSecurityPolicies. This feature is in beta as of Kubernetes v1.14, and is documented here.

IAM Access

One of the benefits of running Kubernetes in one of AWS, GCP, or Azure is the ability to use their managed services to run your DNS, databases, load balancing, and monitoring. You will likely need to both grant and restrict access to these services from your Kubernetes cluster so you can fully integrate Kubernetes.

Google cloud uses Cloud IAM to control access to its services. This is integrated with GKE using RBAC as described here. You can restrict your GCP users and roles to certain access within your Kubernetes cluster, but there is no built-in way to assign an IAM role to a pod and restrict its access to services; a pod will have the same access as the node it runs on.

Azure's AKS uses Active Directory to manage access to resources. This documentation describes how you can use AD to not only restrict user access to your cluster, but you can also assign Pod Identities for fine-grained control over how pods access other Azure services.

Amazon's EKS by default uses IAM to restrict user access to your EKS cluster. There is no built-in method for restricting pod access to other AWS services, but the open-source projects kiam and kube2iam provide this functionality. On EKS clusters, kiam is more difficult to set up because of the client-server model that project uses, but both solutions will work on a kops-managed cluster. For an in-depth look at managing IAM permissions for Kubernetes in AWS specifically, check out our blog series.

Security Reviews

As a startup, it can be easy to forget about one of the most mundane security tasks: getting an external security review. It is extremely important to validate the work you've done on your cluster with a 3rd party if your application will be handling any sensitive user data, and even if it is not it is a good practice to do annual security reviews to make sure you are on top of all of the issues mentioned above.

Monitoring Your Cluster

Now that you're running your app in Kubernetes, you'll want to make sure you're keeping it healthy. In this section, we'll discuss how to view your current cluster state and monitor your Kubernetes cluster over time.

First, to list the currently running pods, as well as some details about each pod, run the following command:

```
kubectl get pods -o wide
```

Once you have the list of pods, you can use it to view logs for a particular pod, which can be really useful when trying to find the source of a bug. You can tail the logs from a particular pod by issuing this command:

```
kubectl logs -f <podname>
```

Finally, you can use the **kubectl top** command with pods and nodes to see resource utilization and find troublesome pods.

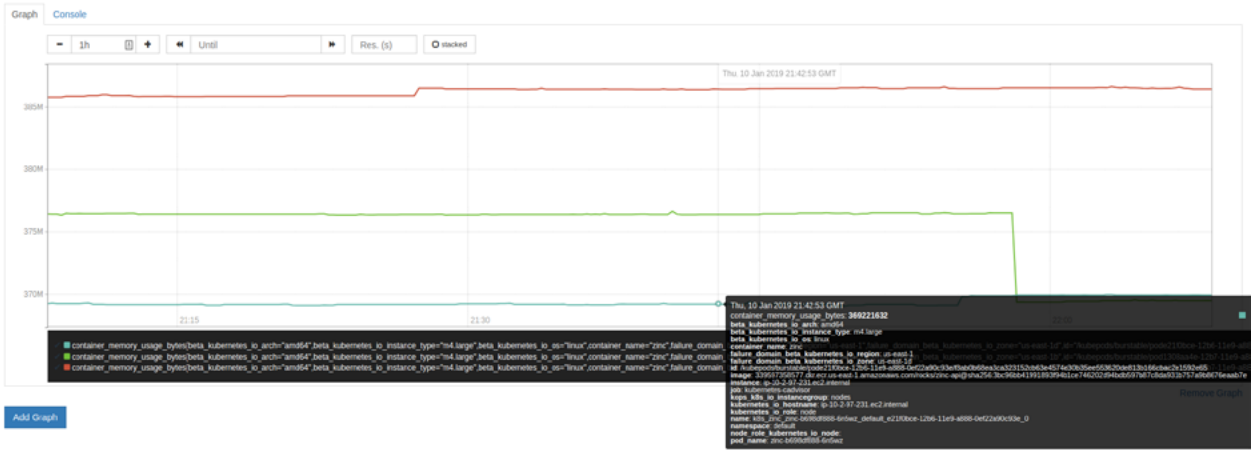
```
kubectl top pods
kubectl top nodes
```

In order to run this command, you'll need to install metrics-server in your cluster. This command is particularly useful to keep an eye on things when deploying, or when an emergency occurs. For a list of other useful kubectl commands, check out the cheat sheet in the Kubernetes documentation.

Prometheus and Grafana

The Kubernetes documentation specifically recommends using Prometheus, which is an open source metric collector. Once Prometheus is installed in your cluster, it'll begin collecting performance metrics. All you need to do is create a cluster role, a config map, and a deployment for Prometheus, most of which can be copied and pasted from any number of tutorials online (here's one to get you started). It takes only a couple minutes to get set up.

While you can view metric graphs in Prometheus, they leave something to be desired and you can only view one metric at a time:



This is unlikely to cover your visualization needs, so most people install Grafana, an open source dashboard application, in their clusters as well. Grafana's setup is a breeze because it has an integration that pulls data from Prometheus. It provides a lot of dashboarding functionality and is easy on the eyes

.

When something goes wrong in your cluster, you're unlikely to happen to be watching it. You'll need an alerting system to notify you. To get notifications, you'll need to install AlertManager, which is the Prometheus ecosystem's alerting system. AlertManager can be configured to alert on any metric in Prometheus, but it's most helpful to be watching CPU and memory usage.

Paid Services

While setting up monitoring for your Kubernetes cluster within said cluster is very easy, it can be dangerous in an actual downtime event.

> If your application is misbehaving and is running in the same cluster as your monitoring solution, it will likely break your monitoring solution. You will be flying blind when you need monitoring the most!

As such, it makes a lot of sense to run your monitoring solution elsewhere. While you could spin up another cluster, sometimes it's easier and cheaper just to pay for a monitoring service. There are many services that monitor Kubernetes:

- Datadog - an all-in-one metrics monitoring solution that combines an agent that collects metrics, visualizations, and alert configuration
- New Relic - an APM that can also monitor your cluster's resource usage
- Dynatrace - another APM that can also monitor your cluster's resource usage
- Blue Matador - an automated monitoring solution that watches your cluster and alerts you of issues without the need for any configuration

Kubernetes Events

Kubernetes also provides a stream of events that are occurring within your cluster. Many of these events are just info level events, but critical events are also sent to the stream. To view all the events in your cluster, use:

```
kubectl get events
```

Many of the paid solutions mentioned above also collect critical Kubernetes events, including Blue Matador. For an in depth dive into Kubernetes events, check out our blog post on the subject.

Log Management for Kubernetes

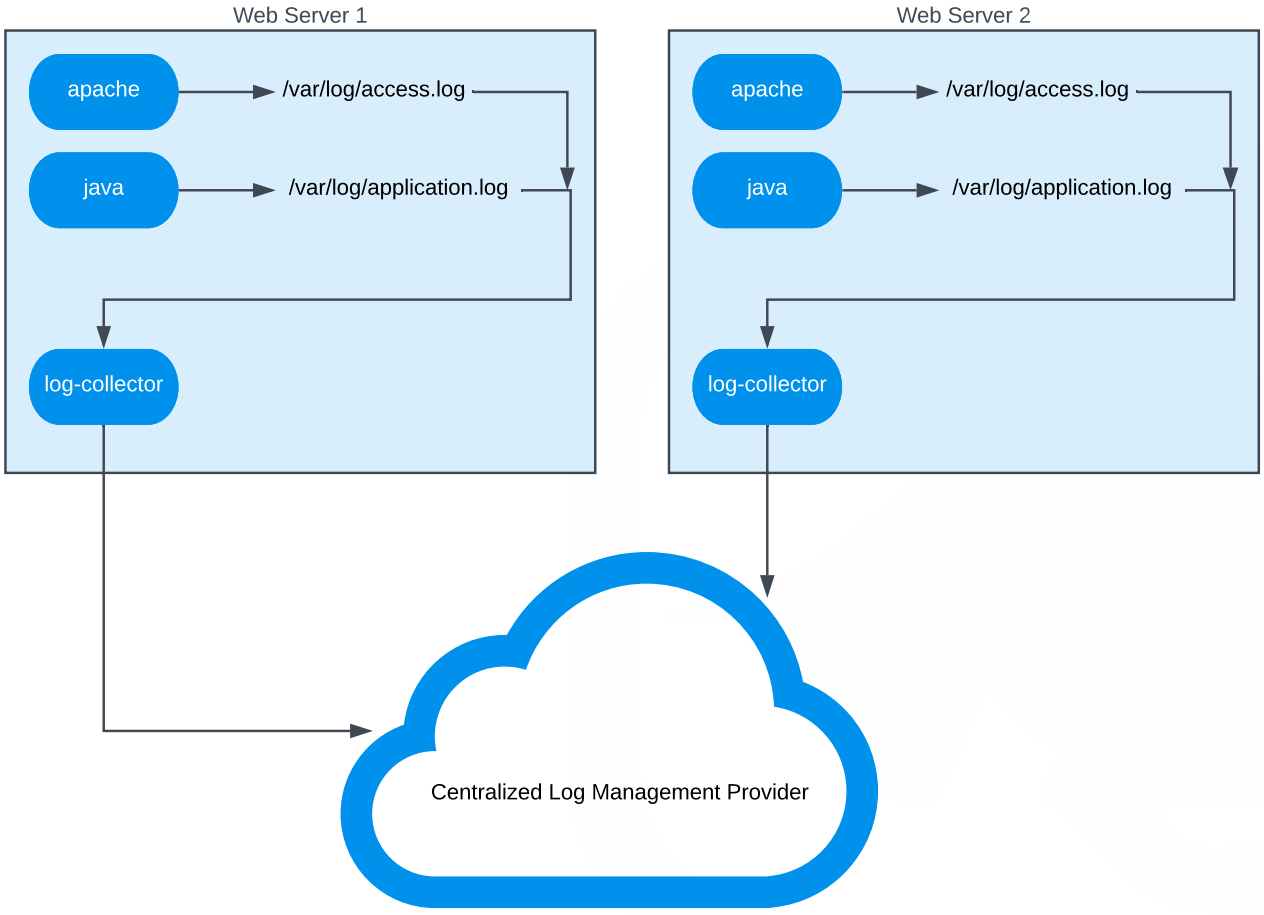
Log messages help us to understand data flow through applications, as well as spot when and where errors are occurring. There are a lot of resources for how to store and view logs for applications running on traditional services, but Kubernetes breaks the existing model by running many applications per server and abstracting away most of the maintenance for your applications. In this section, we focus on log management for applications running in Kubernetes by reviewing the following topics:

- How Logging Works in Kubernetes
- Viewing Logs
- Centralized Log Management

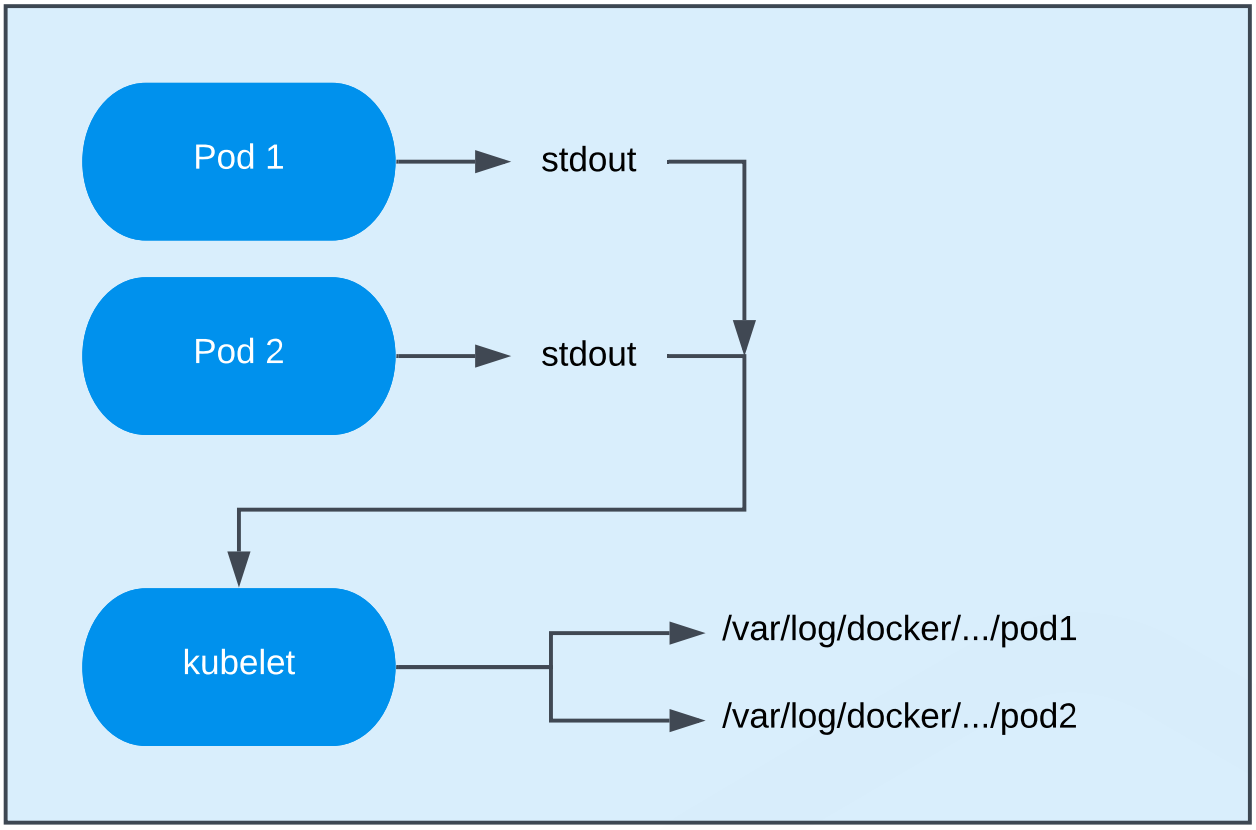
By the end of this section, you should have a good high level understanding of the essential concepts for logging with Kubernetes, and should be ready to begin implementing it for your cluster and logging use case.

How Logging Works in Kubernetes

In a traditional server setup, application logs are written to a file such as **/var/log/application.log** and then viewed either on each server, or collected by a logging agent and sent to a centralized location.



With Kubernetes, writing logs to disk from a pod is discouraged since you would then have to manage log files for pods that can be numerous and short-lived. Instead, your application should output logs to **stdout** and **stderr**. The kubelet running on each Kubernetes node will collect the stdout and stderr of each running pod and combine them into a log file that is managed by Kubernetes. Kubernetes will automatically manage logs for each container in a pod and restrict the log file size, with most installations keeping the most recent 10Mb of logs.



Viewing Logs

Pod logs can be accessed using **kubectl log**. By using kubectl you avoid accessing individual nodes to access the logs for pods running on those nodes, and are able to view logs from pods running on different nodes in real time. Here are a few examples of using kubectl to view logs for a pod:

View all available logs for a pod:

```
kubectl logs <pod>
```

View logs for the last hour for a pod:

```
kubectl logs --since=1h <pod>
```

View logs and then follow the live stream of logs for a pod:

```
kubectl logs -f <pod>
```

View logs for a specific container of a pod in another namespace

```
kubectl -n <namespace> logs <pod> -c <container>
```

View logs for a random pod in a deployment:

```
kubectl logs deployment/<deployment>
```

These options can be combined and then used with **grep** to easily filter logs. Here is an example looking for recent exceptions on a pod

```
kubectl logs --since=1h -f <pod> | grep Exception
```

As you can see, **kubectl logs** is limited to viewing a single pod's logs at a time. This can be fine for quick debugging or smaller systems, but eventually you will want a quick way to live-tail logs from many pods. This is where kubetail comes in. Kubetail is a small binary that essentially runs **kubectl logs -f** on multiple pods and combines the log data into a single stream. Kubetail supports many of the same options as kubectl, and the basic usage will cover most cases:

View logs for all pods with "my-app" in their name:

```
kubetail my-app
```

View 15 minutes of logs for "my-app":

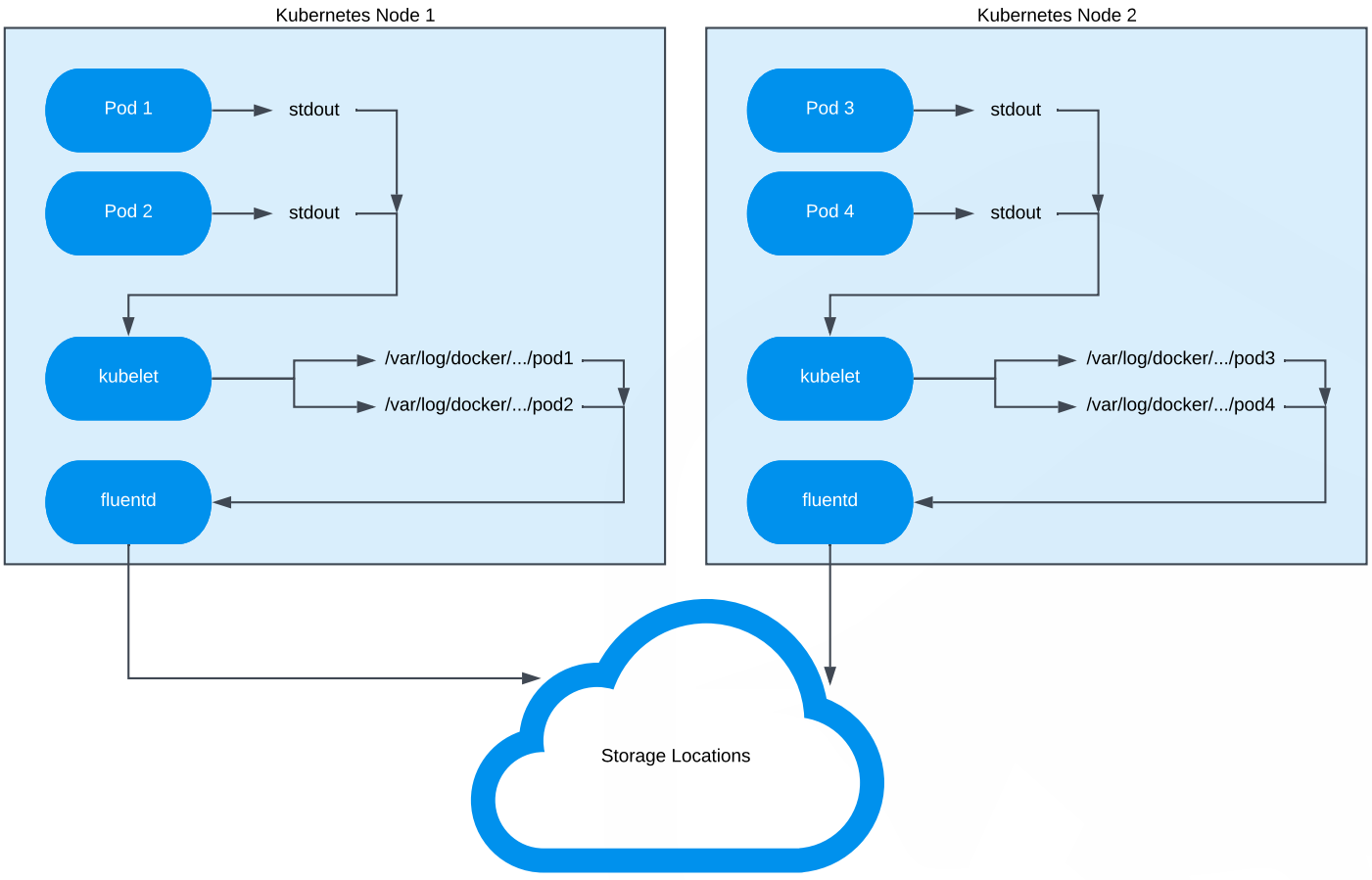
```
kubetail my-app -s 15m
```

Centralized Log Management

Viewing logs using **kubectl logs** or **kubetail** is very convenient for live log streams, but it does not allow you to look at historical log data past a couple of hours or for pods that have been terminated. At some point, you will have to implement centralized log management for your Kubernetes logs in order to meet security and quality requirements for your organization.

There are many solutions for collecting pod logs and shipping them to a centralized location, but we will focus on one of the most widely-used products to aggregate logs in Kubernetes: fluentd. Essentially, fluentd acts as a middleware that collects and parses logs from many sources, and then ships them to one or multiple destinations. Fluentd has a huge amount of plugins available and is flexible enough to collect and parse essentially any log type from any location, and send them to any other location.

In a Kubernetes cluster, we rely on fluentd to collect the pod logs stored on the node filesystem, parse them from various formats (json, apache2, mysql, etc.) and ship them to a logging provider so they can be searched. This is accomplished by running a Fluentd DaemonSet. This DaemonSet runs a pod that collects logs from kubelet, the Kubernetes API server, and all of your running pods on each node. These logs are buffered in the fluentd pods and then sent to wherever you want logs stored: Amazon S3, Elasticsearch, or a third party log management tool.



You should **not** store your Kubernetes logs in the same Kubernetes cluster that generates the logs. Running your own Elasticsearch setup tends to be tedious and time consuming when issues do occur, and running it inside the same cluster that generates logs can put you in a bad spot. If your cluster begins experiencing issues, your Elasticsearch setup will likely also be affected. Running queries in Elasticsearch can put extra strain on your nodes that are already having issues. Additionally, managing disk space, retention, and access to a self-hosted Elasticsearch cluster likely takes valuable time away from your product development, and is best left to log management companies.

Here are a few providers, in no particular order, that offer support for Kubernetes logging in some fashion, and some of them even have free tiers for smaller amounts of log data!

- Sumologic
- LogDNA
- Logz.io
- Splunk
- Elastic

If you insist on keeping the logs within your infrastructure instead of using a third party, there are many blogs detailing this setup:

- Logging in Kubernetes with Elasticsearch, Kibana, and Fluentd
- How To Set Up an Elasticsearch, Fluentd and Kibana (EFK) Logging Stack on Kubernetes

Whether you use a third party provider to store your logs, or run your own Elasticsearch setup, you will likely need to configure your fluentd pods to collect and parse logs that are specific to your application. The fluentd Quickstart Guide is a great resource to understand how fluentd works and to find ways to configure sources, filters, and outputs.

What Not To Do (Yet)

We've covered Kubernetes components, how to create a Kubernetes cluster, building and deploying your application to Kubernetes, securing your cluster, and how to use monitoring and logging tools to stay on top of your application and cluster. Since Kubernetes is new, there are tons of related topics, articles, blogs, how-to's, and documentation for all kinds of features that are really exciting.

For a startup that is just trying to get their business going, many of these topics can lead you astray from the primary goal of running your application, and they should only be implemented as needed. Let's go over a few of these and how and when they can fit into your organization.

Config Management

When you are just starting up, it is usually easier to just run one-line commands and configure things on the fly. For any projects that you expect to last, you will want to consider how you are organizing and keeping track of changes to your infrastructure, and that includes your Kubernetes components.

All Kubernetes components can be expressed in either JSON or YAML in manifest files. These can be organized in a file structure that makes sense to you, and will allow you to recreate your Kubernetes components in the event of a failure. Kubectl allows you to specify a filename for creating or updating resources using manifest files like so:

```
kubectl create -f my_app_deployment.yaml
```

If you need to edit a resource, you can instead edit the YAML manifest file and then update using the file:

```
kubectl apply -f my_app_deployment.yaml
```

You can then store these files in a git repository, and commit changes to these files to your git history. In this way you can easily look at how your environment has changed over time, replicate your entire setup, and eventually create automated workflows based on changes to these files.

Another commonly used tool for managing Kubernetes is helm. Helm essentially acts as a repository for creating and managing Kubernetes resources from the community, and you can also use it for your own resources. A lot of helm users like the ability to use templates for Kubernetes resources which can speed up deployment for apps across multiple environments such as dev, QA, and production. One criticism is the added complexity, and the tiller process which must run to keep configuration in sync. In any case, using either of these methods is better than having no control and history of your Kubernetes configuration changes.

Continuous Integration/Deployment

CI/CD is the ultimate goal of every engineering organization these days. The ability to fully automate the building, testing, and deployment of your application sounds amazing on paper, but for a startup it can be an early death sentence. Early on in your application's lifecycle, you will be creating features, reworking the entire system, and writing bugs daily. Spending hours (which will turn into days) to try and automate a build system for something that is constantly evolving is a poor use of your time early on.

The right time to invest in an automated build system depends on your team. When building and deploying is a noticeable drain on your time, and you have a few days of free time to spend on it, you should at least invest in an automatic build system. Recognize that you can do automatic builds without spending even more time on automatic deploys, and you should do so. Until you are extremely confident that your automated tests and builds are 100% trusted, we do not recommend doing automatic deploys into a Kubernetes cluster.

When you do decide to automate your build and deploy system, Jenkins is one of the most popular open-source build tools, and it can be integrated with Docker and Kubernetes to automate your build. If you are already familiar with Jenkins, then this is the approach we recommend. Other people have written detailed guides for integrating Jenkins into a Kubernetes pipeline, so we will defer to them.

If you are not familiar with Jenkins, then you can look at the free offerings of TravisCI and CircleCI. Both of these SaaS products offer ways to build your docker images automatically and they have reasonable paid plans for when you are at a scale that requires it.

It can be convenient to run Jenkins or another open-source solution inside of your production Kubernetes environment, but it is not recommended. If your production cluster is having issues, you do not want to be in a position where you cannot build or deploy your application. Treat your build system as a first-class citizen in the same way you do for your log management and alerting system. At the very least, have a backup method for building and deploying that you test regularly in case your primary method is unavailable.

Autoscaling

Another popular feature of Kubernetes is the ability to both vertically and horizontally autoscale your pods and nodes. One of the advantages of Kubernetes is definitely the ability to abstractly run your microservices in the cluster without worrying about resources, but autoscaling in the early stages of your application is probably unnecessary and can become expensive. If your app is not seeing consistent week-over-week growth then it is likely easier to just slightly over-provision your cluster and pods and monitor things manually until you can put in the time to set up autoscaling. Scaling up in Kubernetes the "manual" way is generally pretty easy if you've been following this guide. You can quickly add more nodes and increase the size of your deployment when you need it, you just have to set up some basic alerts to tell you when CPU and memory utilization are high in your cluster.

Service Mesh

Service meshes are the next big thing after Kubernetes. The idea behind a service mesh is that at a certain scale, it becomes difficult to manage dozens of microservices in production and understand how they interact with each other. A service mesh can provide you with performance metrics, service discovery, load balancing, failure recovery, deployment strategies, end-to-end authentication, and many other features that make life easier for large deployments. However, these features come at a cost.

There is a learning curve to expressing your application as a service mesh and a literal performance cost since it sits between all of your services to gather request metrics. Implementing a service mesh is probably a bad call until you have a team dedicated to DevOps, or infrastructure, or at least SRE that can actually see the value it provides.

Istio is one of the most popular service meshes currently, and integrates tightly with Kubernetes. It has all of the features that are promised of a service mesh, including interesting deployment techniques that allow for canary rollouts, A/B testing, fault injection, and circuit breaker patterns. Keep an eye on istio as you scale your application into multiple regions and achieve CI/CD as a way to help you stay on top of what will likely become a very complex Kubernetes setup.

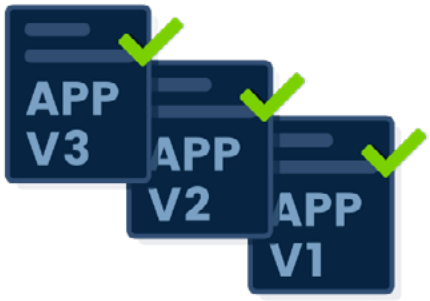
bluematador

Alerts without the toil.

The typical, manual monitoring approach requires significant time and toil to manage every alert on every resource. Blue Matador eliminates the need to manually set up alerts by automatically configuring full monitoring coverage out-of-the-box.

Know about critical production issues.

Issues in your infrastructure can pop-up at any time and unless you create alerts for each event and resource, you will be caught unaware. Blue Matador identifies previously unknown issues, ensuring you see the problems first—instead of hearing about them from your customers.



Deploy faster, rest easier.

Agile teams are looking to rapidly deliver features and delight customers, but that leaves little time to configure proper alerting. Blue Matador supports agile teams deploying multiple times per day by ensuring that they will be alerted of any potential issues.

MAKE MONITORING EASIER
START YOUR FREE TRIAL TODAY

START YOUR FREE TRIAL


bluematador