StackRox

16-Point Checklist for Building Production-Ready Kubernetes Clusters

Table of Contents

Overview

Kubernetes is a powerful tool for building highly scalable systems. As a result, many companies have begun, or are planning, to use it to orchestrate production services. Unfortunately, like most powerful technologies, Kubernetes is complex. How do you know you’ve set things up correctly and it’s safe to flip the switch and open the network floodgates to your services? We’ve compiled the following checklist to help you prepare your containers and kube clusters for production traffic.

Containers Done Right

Kubernetes provides a way to orchestrate containerized services, so if you don't have your containers in order, your cluster isn't going to be in good shape from the get-go. Follow these tips to start out on the right foot.

1. Use minimal base images

What: Containers are application stacks built into a system image. Everything from your business logic to the kernel gets packed inside. Minimal images strip out as much of the OS as possible and force you to explicitly add back any components you need.

Why: Including in your container only the software you intend to use has both performance and security benefits. You have fewer bytes on disk, less network traffic for images being copied, and fewer tools for potential attackers to access.

How: Alpine Linux is a popular choice and has broad support.

2. Use a registry that offers the best uptime

What: Registries are repositories for images, making those images available for download and launch. When you specify your deployment configuration, you'll need to specify where to get the image with a path `<registry>/<remote name>:<tag>` :

```
apiVersion: v1
kind: Deployment
...
spec:
...
  containers
  - name: app
    image: docker.io/app-image:version1
```

Why: Your cluster needs images to run.

How: Most cloud providers offer private image registry services: Google offers the Google Container Registry, AWS provides Amazon ECR, and Microsoft has the Azure Container Registry.

Do your homework, and choose a private registry that offers the best uptime. Since your cluster will rely on your registry to launch newer versions of your software, any downtime will prevent updates to running services.

3. Use ImagePullSecrets to authenticate your registry

What: ImagePullSecrets are Kubernetes objects that let your cluster authenticate with your registry, so the registry can be selective about who is able to download your images.

Why: If your registry is exposed enough for your cluster to pull images from it, then it’s exposed enough to need authentication.

How: The Kubernetes website has a good walkthrough on configuring ImagePullSecrets, which uses Docker as an example registry, here.

Organizing Your Cluster

Microservices by nature are a messy business. A lot of the benefit of using microservices comes from enforcing separation of duties at a service level, effectively creating abstractions for the various components of your backend. Some good examples are running a database separate from business logic, running separate development and production versions of software, or separating out horizontally scalable processes.

The dark side of having different services performing different duties is that they cannot be treated as equals. Thankfully Kubernetes gives you many tools to deal with this problem.

4. Isolate environments by using Namespaces

What: Namespaces are the most basic and most powerful grouping mechanism in Kubernetes. They work almost like virtual clusters. Most objects in Kubernetes are, by default, limited to affecting a single namespace at a time.

Why: Most objects are namespace scoped, so you'll have to use namespaces. Given that they provide strong isolation, they are perfect for isolating environments with different purposes, such as user serving production environments and those used strictly for testing, or to separate different service stacks that support a single application, like for instance keeping your security solution's workloads separate from your own applications. A good rule of thumb is to divide namespaces by resource allocation: If two sets of microservices will require different resource pools, place them in separate namespaces.

How: It's part of the metadata of most object types:

```
apiVersion: v1
kind: Deployment
metadata:
  name: example-pod
  namespace: app-pod
  ...
```

Note that you should always create your own namespaces instead of relying on the ‘default’ namespace. Kubernetes’ defaults typically optimize for the lowest amount of friction for developers, and this often means forgoing even the most basic security measures.

5. Organize your clusters with Labels

What: Labels are the most basic and extensible way to organize your cluster. They allow you to create arbitrary key:value pairs that separate your Kubernetes objects. For instance, you might create a label key which separates services that handle sensitive information from those that do not.

Why: As mentioned, Kubernetes uses labels for organization, but, more specifically, they are used for selection. This means, when you want to give a Kubernetes object a reference to a group of objects in some namespace, like telling a network policy which services are allowed to communicate with each other, you use their labels. Since they represent such an open-ended type of organization, do your best to keep things simple, and only create labels where you require the power of selection.

How: Labels are a simple spec field you can add to your YAML files:

```
apiVersion: v1
kind: Deployment
metadata:
  name: example-pod
  ...
  matchLabels:
    userexposed: true
    storespii: true
```

6. Use Annotations to track important system changes, etc.

What: Annotations are arbitrary key-value metadata you can attach to your pods, much like labels. However, Kubernetes does not read or handle annotations, so the rules around what you can and cannot annotate a pod with are fairly liberal, and they can't be used for selection.

Why: They help you track certain important features of your containerized applications, like version numbers or dates and times of first bring up. Annotations, in the context of Kubernetes alone, are a fairly powerless construct, but they can be an asset to your developers and operations teams when used to track important system changes.

How: Annotation are a spec field similar to labels.

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  ...
  annotations:
    version: four
    launchdate: tuesday
```

Securing Your Cluster

Alright, you've got a cluster set up and organized the way you want - now what? Well, next thing is getting some security in place. You could spend your whole lifetime studying and still not discover all the ways someone can break into your systems. A blog post has a lot less room for content than a lifetime, so you'll have to settle for a couple of strong suggestions.

7. Implement access control using RBAC

What: RBAC (Role Based Access Control) allows you to control who can view or modify different aspects of your cluster.

Why: If you want to follow the principle of least privilege, then you need to have RBAC set up to limit what your cluster users, and your deployments, are able to do.

How: If you're setting up your own cluster (i.e., not using a managed Kube service), make sure you are using ''--authorization-mode=Node,RBAC" to launch your kube apiserver. If you are using a managed Kubernetes instance, you can check that it is set up to use RBAC by querying the command used to start the kube apiserver. The only generic way to check is to look for "--authorization-mode..." in the output of `kubectl cluster-info dump`.

Once RBAC is turned on, you'll need to change the default permissions to suit your needs. The Kubernetes project site provides a walk-through on setting up Roles and RoleBindings here. Managed Kubernetes services require custom steps for enabling RBAC - check out Google's guide for GKE or Amazon's instructions for AKS.

8. Prevent risky behavior using Pod Security Policies

What: Pod Security Policies are a resource, much like a Deployment or a Role, and can be created and updated through kubectl in same way. Each holds a collection of flags you can use to prevent specific unsafe behaviors in your cluster.

Why: If the people who created Kubernetes thought limiting these behaviors was important enough to create a special object to handle it, then they are likely important.

How: Getting them working can be an exercise in frustration. I recommend getting RBAC up and running, then check out the guide from the Kubernetes project here. The most important to use, in my opinion, are preventing privileged containers, and write access to the host file system, as these represent some of the leakier parts of the container abstraction.

9. Implement network control/firewalling using Network Policies

What: Network policies are objects that allow you to explicitly state which traffic is permitted, and Kubernetes will block all other non-conforming traffic.

Why: Limiting network traffic in your cluster is a basic and important security measure. Kubernetes by default enables open communication between all services. Leaving this “default open” configuration in place means an Internet-connected service is just one hop away from a database storing sensitive information.

How: A colleague of mine did a great write up that will get you going here.

10. Use Secrets to store and manage necessary sensitive information

What: Secrets are how you store sensitive data in Kubernetes, including passwords, certificates, and tokens.

Why: Your services may need to authenticate one another, other third-party services, or your users, whether you’re implementing TLS or restricting access.

How: The Kubernetes project offers a guide here. One key piece of advice: avoid loading secrets as environment variables, since having secret data in your environment is a general security no-no. Instead, mount secrets into read only volumes in your container - you can find an example in this Using Secrets write up.

11. Use an image scanner to identify and remediate image vulnerabilities

What: Scanners inspect the components installed in your images. Everything from the OS to your application stack. Scanners are super useful for finding out what vulnerabilities exist in the versions of software your image contains.

Why: Vulnerabilities are discovered in popular open source packages all the time. Some notable examples are Heartbleed and Shellshock. You'll want to know where such vulnerabilities reside in your system, so you know what images may need updating.

How: Scanners are a fairly common bit of infrastructure - most cloud providers have an offering. If you want to host something yourself, the open source Clair project is a popular choice.

Keeping Your Cluster Stable

Kubernetes represents a tall stack. You have your applications, running on baked-in kernels, running in VMs (or on bare metal in some cases), accompanied by Kubernetes' own services sharing hardware. Given all these elements, plenty of things can go wrong, both in the physical and virtual realms, so it is very important to de-risk your development cycle wherever possible. The ecosystem around Kubernetes has developed a great set of best practices to keep things in line as much as possible.

12. Follow CI/CD methodologies

What: Continuous Integration/Continuous Deployment is a process philosophy. It is the belief that every modification committed to your codebase should add incremental value and be production ready. So, if something in your codebase changes, you probably want to launch a new version of your service, either to run tests or to update your exposed instances.

Why: Following CI/CD helps your engineering team keep quality in mind in their day-to-day work. If something breaks, fixing it becomes an immediate priority for the whole team, because every change thereafter, relying on the broken commit, will also be broken.

How: Thanks to the rise of cloud deployed software, CI/CD is in vogue. As a result, you can choose from tons of great offerings, from managed to self-hosted. If you're a small team, I recommend going the managed route, as the time and effort you save is definitely worth the extra cost.

13. Use Canary methodologies for rolling out updates

What: Canary is a way of bringing service changes from a commit in your codebase to your users. You bring up a new instance running your latest version, and you migrate your users to the new instance slowly, gaining confidence in your updates over time, as opposed to swapping over all at once.

Why: No matter how extensive your unit and integration tests are, they can never completely simulate running in production - there's always the chance something will not function as intended. Using canary limits your users' exposure to these issues.

How: Kubernetes, as extensible as it is, provides many routes to incrementally roll out service updates. The most straightforward approach is to create a separate deployment that shares a load balancer with currently running instances. The idea is you scale up the new deployment while scaling down the old until all running instances are of the new version.

14. Implement monitoring and integrate it with SIEM

What: Monitoring means tracking and recording what your services are doing.

Why: Let's face it - no matter how great your developers are, no matter how hard your security gurus furrow their brows and mash keys, things will go wrong. When they do, you're going to want to know what happened to ensure you don't make the same mistake twice.

How: There are two steps to successfully monitor a service - the code needs to be instrumented, and the output of that instrumentation needs to be fed somewhere for storage, retrieval, and analysis. How you perform instrumentation is largely dependent on your toolchain, but a quick web search should give you somewhere to start. As far as storing the output goes, I recommend using a managed SIEM (like Splunk or Sumo Logic) unless you have specialized knowledge or need - in my experience, DIY is always 10X the time and effort you expect when it comes to anything storage related.

Advanced Topics

Once your clusters reach a certain size, you'll find enforcing all of your best practices manually becomes impossible, and the safety and stability of your systems will be challenged as a result. After you cross this threshold, consider the following topics:

15. Manage inter-service communication using a service mesh

What: Services meshes are a way to manage your inter-service communications, effectively creating a virtual network that you use when implementing your services.

Why: Using a service mesh can alleviate some of the more tedious aspects of managing a cluster, such as ensuring communications are properly encrypted.

How: Depending on your choice of service mesh, getting up and running can vary wildly in complexity. Istio seems to be gaining momentum as the most used service mesh, and your configuration process will largely depend on your workloads.

A word of warning: If you expect to need a service mesh down the line, go through the agony of setting it up earlier rather than later - incrementally changing communication styles within a cluster can be a huge pain.

16. Use Admission Controllers to unlock advanced features in Kubernetes

What: Admission controllers are a great catch-all tool for managing what's going into your cluster. They allow you to set up webhooks that Kubernetes will consult during bring up. They come in two flavors: Mutating and Validating. Mutating admission controllers alter the configuration of the deployment before it is launched. Validating admission controllers confer with your webhooks that a given deployment is allowed to be launched.

Why: Their use cases are broad and numerous – they provide a great way to iteratively improve your cluster's stability with home-grown logic and restrictions.

How: Check out this great guide on how to get started with Admission Controllers.

AUDIT YOUR CONTAINER SECURITY AT:
https://www.stackrox.com/assessment/