datree.io

The DevOps Engineer's Guide to

Kubernetes Configurations Best Practices

Last updated: July 2020

Introduction

About this guide

Since released as an open source project by Google in 2014, Kubernetes has become the new standard for deploying containers and managing containerized workloads and services.

In this guide, we will cover the top 10 things you should look out for when pushing Kubernetes configurations from development through your deployment process.

Our recommendations come from real-world lessons learned from Kubernetes configurations that made it to production but never should have. Adopting these recommended best practices will reduce risks of production outages, degraded performance, and security breaches.

On preventing Kubernetes misconfigurations

This guide is especially useful for DevOps engineers who are working with highly or increasingly autonomous product development teams, whose developers are tasked with deploying services to production on their own.

While technologies like infrastructure-as-code have made Kubernetes deployments easier for the development teams, sometimes this comes at a cost of misconfiguration mistakes that could bring down or degrade the performance of a service or application.

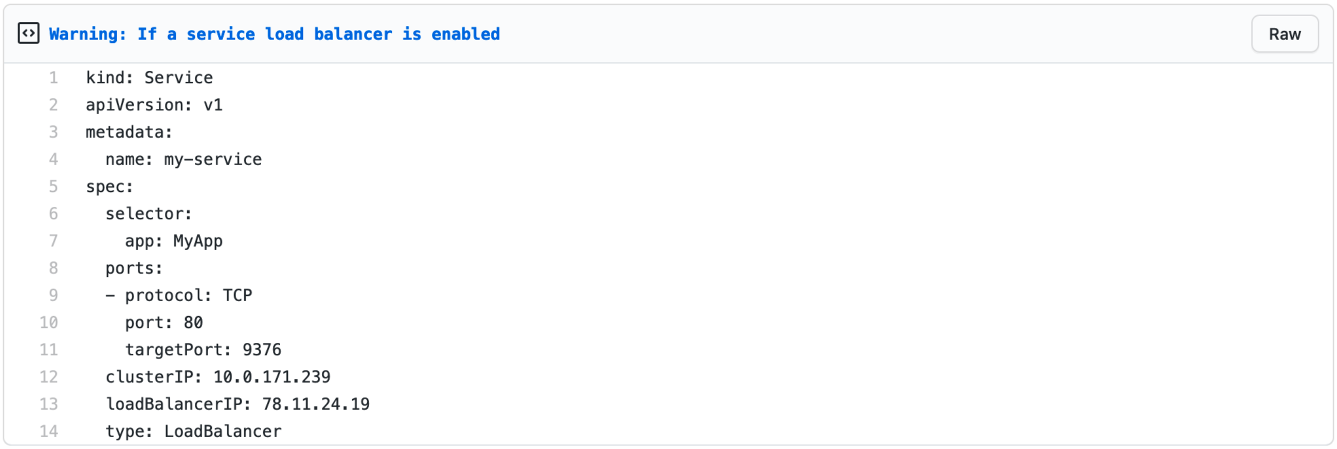
Preventing Kubernetes misconfigurations means preventing potentially costly mistakes. Developers aren't operators, and even operators make mistakes.

We hope you'll be inspired to implement the practices outlined in this guide as policies for your Engineering organization to avoid making such mistakes.

10 best Kubernetes policies

1 Warning: If a service load balancer is enabled

In a service definition if you make it a “type: LoadBalancer,” then the cloud you are on will create a load balancer for you. In AWS this is an ELB (external by default) and in GCP this is a LoadBalancer (external). All too often though, it’s a security risk since you are exposing something onto the internet with few to no security controls. There is at least one external load balancer that handles the services you want to expose to the Internet and everything routes into that.



Warning: If a service load balancer is enabled

Raw

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
  clusterIP: 10.0.171.239
  loadBalancerIP: 78.11.24.19
  type: LoadBalancer
```

Anytime the “type: LoadBalancer” line changes, it should be flagged and someone - a DevOps Engineer perhaps - should review for verification and approval.

2 Enable the ‘readiness probe’ on a deployment

A “readiness” probe should be defined in any deployment. This is simply a signal to inform Kubernetes when to put this pod behind the load balancer and when to put this service behind the proxy to serve traffic. If you put an application behind the load balancer before it is ready, then a user can reach this pod but will not get the expected response of a healthy server. This rule is here to give an alert to the pod developer that the readiness probe should be enabled.

3 Warning: If a service exposes a NodePort

“Service: NodePort”, will open a port on all nodes where it can be reached by the network external to the cluster. This exposes the cluster to a security risk. Just like “type: LoadBalancer”, it is relatively easy to inadvertently do this, because many tutorials direct you to do it for ease of use reasons. Even a lot of Helm / stable charts do this by default to make it “easy” for you to reach the application.

Warning: If a service exposes a NodePort Raw

```
apiVersion: v1
kind: Service
metadata:
  name: <my-nodeport-service>
  labels:
    <my-label-key>: <my-label-value>
spec:
  selector:
    <my-selector-key>: <my-selector-value>
  type: NodePort // This sets the “type”
  ports:
   - port: <8081>
     nodePort: <31514>
```

4 Enable ‘liveness probe’ on a deployment

The liveness probe is just as important as the readiness probe. The liveness probe lets Kubernetes know if the pod is in a healthy state and if it isn’t healthy, Kubernetes should restart it. This is done via a simple check, such as getting an HTTP 200ok on some endpoint or a more complex check based on some bash commands. Either way, it is important - and very handy - to let Kubernetes know when the application isn’t working and needs to be restarted.

Enable ‘liveness probe’ on a deployment Raw

```
    livenessProbe:
       httpGet:
         path: /healthz
         port: 8080
       initialDelaySeconds: 3
       periodSeconds: 3
```

5 Enable CPU/Mem requests and limits not set on a deployment

The container(s) in a deployment should automatically requests the CPU and memory resources that it needs and define it for the system. This prevents the pod from being starved of resources while also preventing CPU/Mem from consuming all of the resources on a node.

Enable CPU or Mem requests and limits not set on a deployment

Raw

```yaml
resources:
   requests:
     memory: "64Mi"
     cpu: "250m"
   limits:
     memory: "128Mi"
     cpu: "500m"
```

6 Flag When RBAC rules change

The principle of least privilege is something your security team will be bugging you about. It's a compelling reason to get your RBAC configuration right. This is something that requires an overall review, starting with subjects that can create resources like Deployments or Pods in general or read sensitive resources like Secrets.

The challenging part, is to understand when Role (or ClusterRole) resources does not add privileges over time.

Such change is definitely something that should be flagged for a review.

For example: Changing verbs: ["get"] to verbs: [“*”] is a significant change.

Check out these sample RBAC policies to get started.

7 Control the container images deployed into your cluster

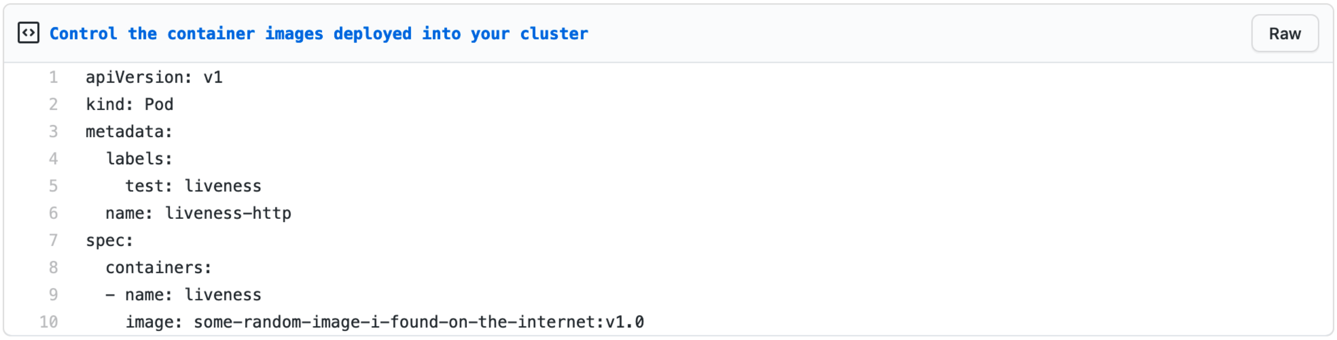
Your company could be more or less stringent on where the binaries come from, depending on their policy on third-party binaries.

If you are pulling common images that organizations use - like the official nginx, MySQL, or Redis - your organization might want to build it from source and/or rehost the image internally instead of pulling from Docker Hub.

The reason is that the images stored in Docker Hub can change if someone pushes the same image and tag to it. That means what you get from pulling the same image and tag may be different from one day to another, causing confusion. Additionally, the difference could be something malicious that could compromise your infrastructure and application.

To mitigate these risks, you can either build the image from the source and host it in your own repository, or push the same images into your repository.

If your organization hosts some or all your container images, you should apply this rule to flag any image not coming from your organization and flag it for someone to approve.



Control the container images deployed into your cluster

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: some-random-image-i-found-on-the-internet:v1.0
```

8 Apply network policy to your deployments

Applying Security Groups policies to your VMs or your Kubernetes worker nodes are considered essential to security. We should do the same with Kubernetes workloads.

The best practice is to limit inbound and outbound traffic to only what you need so you don't accidentally expose unwanted services on the outbound. Kubernetes has Network Policy functionality that's equivalent to Security Groups. All resources should have Network Policy rules associated with their deployments.

For every deployment set, there should be a network policy file or the following resource:

Apply network policy to your deployments - part I Raw

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
```

If you want to get super fancy with it, you can match up the ports list to those outlined in the deployments pods exposed list and/or the service port list.

Apply network policy to your deployments - part II Raw

```
ports:
    - protocol: TCP
      port: 6379
```

Ideally, these would all match so that the developers know that everything reconciles and the network policy doesn't list a port that is not used by the service or pod.

Also make sure that you provisioned your cluster with network plugin (CNI) that supports network policies..

9 Flag any service account changes

Service accounts provide an identity mapped to some set of Kubernetes API server access permissions for a pod to use. Changes to these are typically minor and easily overlooked, but have big ramifications on security and API server access. Whenever a change is made, the right person(s) should be notified to review the changes.

If you see this type of change, flag it:

Flag any service account changes – part I

```
apiVersion: v1
kind: ServiceAccount
```

If a service account name changes, flag it:

Flag any service account changes – part II

```
apiVersion: v1
kind: Pod | Deployment | Daemonset | Statefulset | rs | rc
metadata:
 name: my-pod
spec:
 serviceAccountName: build-robot
```

10 Flag when Pod ‘toleration’ to run on the master nodes is added

The Kubernetes master nodes are the control nodes of the entire cluster. Only certain items should be permitted to run on these nodes. To effectively limit what can run on these nodes, taints are placed on the nodes to specify items that tolerate the taint can run on them. However, this does not preclude anyone from using these taints on their pods to run on the master nodes.

If you encounter the toleration below on a Pod specification in one of your deployment resources, and your cluster is self-managed, it should be flagged for review:

Flag when Pod ‘toleration’ to run on the master nodes is added

```
node-role.kubernetes.io/master:NoSchedule
```

Conclusion

The 10 best practices you just read are just a small subset of our entire library of Kubernetes configuration policies including for Terraform, AWS CloudFormation, and more. They are among the most common ones however and could represent “quick wins” for your team.

Final Thoughts

DevOps empowers developers to not just write code, but also deploy them to production. Developers are making more infrastructure changes. Thanks to infrastructure-as-code, this is easier than ever.

But developers aren't operators, and even operators make mistakes. How do you prevent misconfigurations that could negatively impact production?

Consistently preventing misconfigurations could be very challenging, especially in fast-growing or large teams.

The ways people try to solve this problem, like writing down policies in a shared document or wiki, sending mass emails from time to time to entire teams, Slacking them in the team channel, or hoping code reviews will catch everything, are nowhere near consistently effective – let alone scalable.

Whether you decide to build something in-house or use a purpose-built commercial tool, investing in automation of configuration checks is a good idea and will pay off through prevention of costly mistakes that affect production stability.

datree.io

About Datree

Datree is an automated testing solution for Infrastructure-as-code. It helps DevOps teams and developers prevent out-of-control infrastructure and remove blockers to deployments.

Any infrastructure that can be misconfigured through code can be protected by Datree: Kubernetes, Helm, Terraform, AWS CloudFormation, Docker, and more. Custom policies can be created and enforced company-wide to meet specific requirements.

Datree integrates with GitHub, with Atlassian BitBucket and GitLab support coming soon. It runs in the cloud with optional support for on-prem.

Talk to us for a free trial