Top 10 Kubernetes Production Best Practices


Top 10 Kubernetes Production
BEST
PRACTICES
Michael Levan

Kubernetes is an ever-growing platform that is most likely never going to reach a static phase of people saying “yep, it’s done! No more innovation!”. It’s constantly growing, evolving, adapting, and increasing in reliability for many organizations. In a cloud-native world, Kubernetes is becoming more and more popular. In fact, a lot of leaders in the space are starting to call Kubernetes the “operating system of the cloud”.

With a platform that’s always growing and always evolving, there need to be best practices in place that allow engineers to use it as efficiently as possible.

In this e-book, you’ll learn about what the top 10 best practices are for your environment and how they work.

1: Livenessprobes and Readinessprobes

Think of a probe like a way to confirm that everything is working the way it should be. It’s an automation checker of sorts. Within Kubernetes, you have livenessProbes and readinessProbes.

livenessProbes:

- Determine if a Pod is healthy
- Determines if a Pod is running as expected
- Kubelet uses the livenessProbe to know when it should restart a container inside of a Pod

readinessProbes:

- Determine if a Pod should receive a request
- Kubelet uses a readinessProbe to know when it should start accepting traffic

Inside of livenessProbes and readinessProbes, there are a few different probe options:

- HTTP - Run HTTP requests (like a `GET` request)
- Commands - Runs a command inside of your container
- TCP - Attempts to make a TCP connection on a specific port

How It Works

You’ll usually define a livenessProbe or a readinessProbe inside of a Kubernetes manifest to check that the Pod is operating as you expected.

An example is in the below Kubernetes Manifest:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginxdeployment
  replicas: 2
  template:
    metadata:
      labels:
        app: nginxdeployment
    spec:
      containers:
      - name: nginxdeployment
        image: nginx:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 80
        livenessProbe:
          httpGet:
            scheme: HTTP
            path: /index.html
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 5
```

Notice under the `livenessProbe` map there's an `httpGet` map that specifies the `path: /index.html`. This is essentially saying that there's an `index.html` file inside of the Nginx Pod and the livenessProbe should check it to confirm it's working as expected.

2: Namespaces

When you run applications, there may be times that you don't want them on the same virtual machine, right? Maybe one VM is for a frontend app and another VM is for a backend app. You want to separate the two applications.

Namespaces in Kubernetes are the same way. They're used to segregate resources and ensure that the applications aren't in one place. Instead, they can be separated. A great example of why organizations sometimes use namespaces is to separate clients. If you have, for example, a SaaS product that's running on Kubernetes, you're not going to have a different k8s cluster for each client (maybe in some cases, but not regularly). Instead, you would separate client applications into different namespaces.

How It Works

As explained in this section, namespaces allow you to segregate workloads. Let's see what this looks like.

You can create a new namespace by running the following command:

```
kubectl create namespace testnamespace
```

Then you can run a Pod inside of that namespace:

```
kubectl run nginxtest --image=nginx:latest -n testnamespace
```

To confirm that the Pod was created in the `testnamespace` namespace, run the following:

```
kubectl get pods -n testnamespace
```

You'll see the Pod running. If you try running `kubectl get pods` on the default namespace, for example, you won't see the Pod. The reason why is because it's segregated to the `testnamespace` namespace.

3: Set Up To Scale Correctly

Within Kubernetes, there are two primary different ways that you have to think about scaling:

- Scaling control planes and worker nodes
- Scaling applications

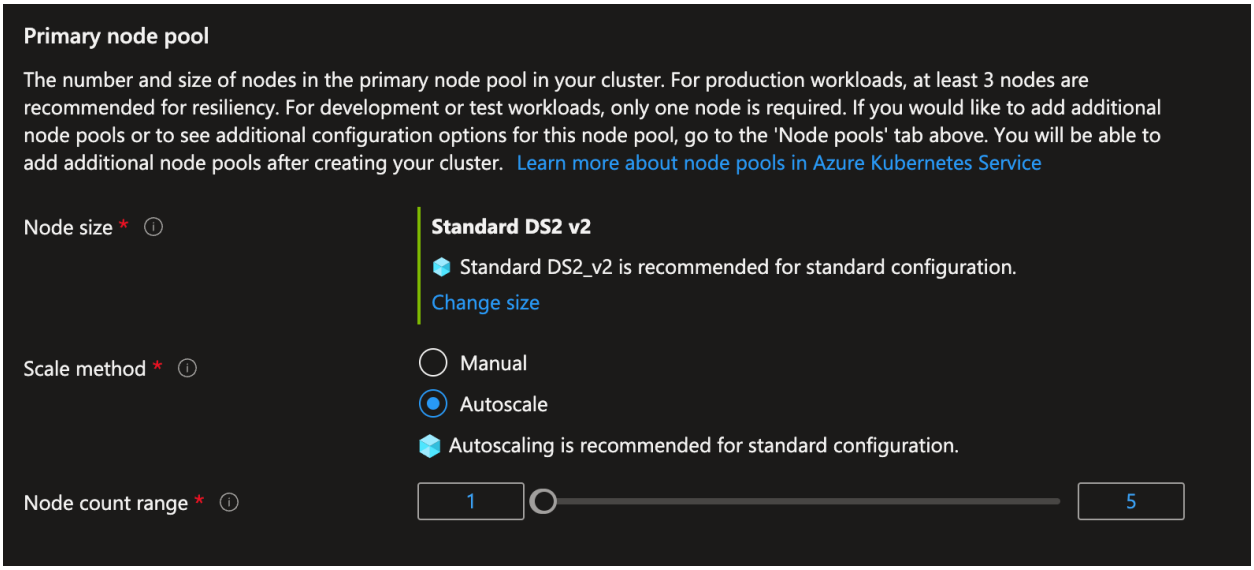
From a control plane/worker node perspective, the idea is to ensure that you have enough servers ready to be used to handle the application load. For example, three worker nodes may work great for the applications that you're already hosting, but if 50 more clients get onboarded to use the application, you may have to scale out depending on the resources available on the first three worker nodes. If that's the case, you have to have the fourth one readily available. Within, for example, Azure or AWS, there are autoscalers available so you don't have to worry about manually adding in another worker node to the cluster.

From an application perspective, it's sort of the same as the control plane/worker nodes. Applications get bogged down after getting slammed by hundreds or thousands of users. Sometimes they simply cannot handle the load, so they need multiple replicas. Luckily with replicas, you can set a `min` count and `max` count so you don't have to worry about manually scaling out Pods. Kubernetes recommends to have at least two Pods (replicas) running for fault tolerance.

How It Works

There are, of course, so many different ways to scale Kubernetes nodes. A short but sweet example is in the screenshot below, which is inside the Azure Kubernetes Service (AKS) portal.

Notice how there's a node count range and you can set it up for autoscaling. The minimum is 1 node and the max is 5. This allows you to have enough nodes to handle all of the traffic, but they won't waste money if there's no load because they'll scale down.



4: Keep An Eye On APIs Used

Any time that you interact with Kubernetes, you're interacting directly with the Kubernetes API. If you create a Pod, create a Deployment, or run a `kubectl get` command against any Kubernetes resource, you're interacting with an API. When you run a `kubectl create` command to create a Pod, Deployment, or any other type of resource, you're technically running a `POST` request against the Kuberentes control plane (API server).

When you're running any type of `get` or `create` command, you're doing it against the Kubernetes API server. The Kubernetes API server is the control plane, sometimes referred to as the master node. The control plane also has other jobs including hosting:

- etcd: Datastore for Kubernetes. It's pretty much the database for Kubernetes.
- Cluster store: Stores the entire state of the Kubernetes cluster.
- Controller manager: Implements controllers that monitor cluster components and responds to events.

- Scheduler: Watches the API server for new work tasks and schedules Pods according to resource utilization.

There are many versions of the Kubernetes API server/control plane. For example, you can find some docs for v1.23 here.

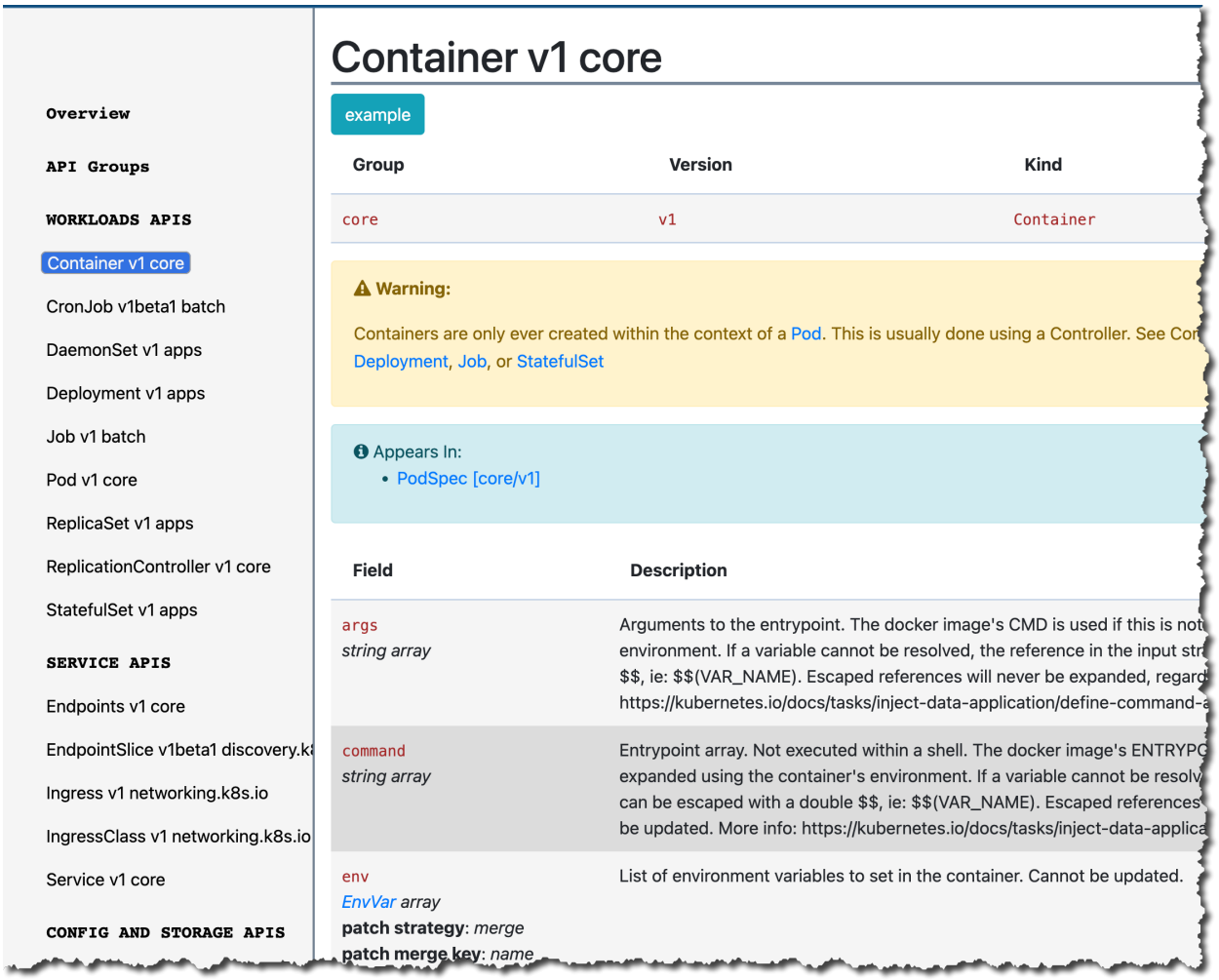
The Kubernetes API has APIs for each resource that you create, delete, update, etc... For example, if you create a new Pod, you're interacting with the v1 core API group. It'll look like the below in your Kubernetes Manifest:

```
apiVersion: apps/v
kind: Deployment
```

Because these are APIs, the APIs of course get updated, replaced, and even removed. There are several APIs that are, for example, beta that are used in production. Sometimes when an API gets updated, certain portions change. As an example, the way you create a Kubernetes Manifest for a specific resource could change as the API moves from beta to stable.

How It Works

The best way to keep up with the various APIs is from the docs. The screenshot below shows a snippet of the much-needed amount of information outlining the APIs and what versions you should use. You can find the website here that'll give you all of the information.



If you want to see similar types of information in terms of showing you what APIs are supported, you can run the following command on a terminal:

```
kubectl api-versions
```

5: Security

There's a ridiculous amount of security needed in Kubernetes. Some of it is on by default and others part you have to really think about. Below is a short list of, what I'll call "pieces of security". There's surely things that are missed from this list because this topic can be an entire book within itself, but this is a good starting point.

- RBAC

    - ClusterRole
    - Role
- Firewall rules
- Service accounts that have roles
- Pod security
- Kubernetes secrets
- Auditing logs
- Read-only containers (set it up in a k8s manifest). Chances are your pod doesn't have to talk directly to the Kube API

As explained in the first paragraph, this topic can be a literal book, so let's keep it brief (but I do recommend diving deeper into these topics).

RBAC in Kubernetes is like standard RBAC. You can create groups that have access to, say, reading Pods. Then you can create users via service accounts that can connect to the RBAC policies.

Because Kubernetes Services are sometimes public, you'll have to think about firewall rules. For example, do you want port `80` open to everyone? Or do you just want port `443` open to the public?

Almost every application requires a secret. Whether it's an API key, a database password, or a username/password, chances are you'll need a Pod to have access to some sort of secret. This is where Kubernetes Secrets come into play and even other types of secret management like HashiCorp Vault.

Pod security is of course huge because that's where an application is running. Much like you have to think about application security when an app is running on a virtual machine, you have to think about it when the app is running in a Pod. There are a million Pod security approaches, but a great one to start with is read-only containers.

As with all security and troubleshooting, logs are available. Standard logs and even audit logs to know what's going on underneath the hood in kubernetes. For example, when a user initiates access to a Kubernetes cluster, whether the request was accepted or rejected, it'll show up in the audit logs.

Because there's a lot on this list, let's think about two pieces of security; RBAC and Kubernetes Secrets

How Kubernetes Secrets work

Because there's a big list of security implementations, let's think about Kubernetes Secrets. First things first, you have to create a secret. You can do this in a few ways:

- The `kubectl` terminal
- A YAML file

To create a secret with `kubectl`, run the following command, which creates a secret called `testsecret` with a key/value pair. The key is called `key1` and the value of the secret is `value1`

```
kubectl create secret generic creds --from-literal user=mikelevan --from-literal pwd=Password123
```

To create a secret in a YAML file, you can use a Secrets Manifest.

```
apiVersion: v1
kind: Secret
metadata:
  name: testsecret
type: Opaque
data:
  username: mikelevan
  password: Password123
```

To use the secret, below is an example of injecting the secret into a Pod. Using this method, the secret essentially becomes available as a file inside of the Pods filesystem.

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: nginx:latest
    secret:
      secretName: testsecret
      optional: false
```

How RBAC Works

First, create a new service account in Kubernetes.

```
kubectl create sa mikeuser
```

To see the new service account, run the following command:

```
kubectl get sa mikeuser
```

After the service account is created, you can create a cluster role. In this case, the cluster role is a readonly permission for Pods.

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

Once the cluster role is created, you can bind the cluster role to a group or even a user. In the example below, you'll see that the cluster role is binding to a service account.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-pod-global
subjects:
- kind: ServiceAccount
  name: mikeuser
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: reader
  apiGroup: rbac.authorization.k8s.io
```

6: Monitoring and Observability

There are of course several ways and components to monitor inside of Kubernetes, but the core of it is:

- The control plane(s)
- The worker node(s)

- The application(s)

Monitoring the control plane(s) can vary based on where you're running Kubernetes. If you're running in a Kubernetes cloud service, like AKS or EKS, you don't have to monitor the control plane(s). If you're running an on-prem Kubernetes cluster, you'll have to monitor it the same way that you would worker nodes. The idea is to monitor:

- Infrastructure components (RAM, CPU, etc.)
- Operating System components
- Core Kubernetes pieces (Kubelet, API server, Etcd, etc.)

For the worker nodes, you need to monitor the same infrastructure and OS components.

At the application level, you'll need to monitor not only the containers inside of a Pod and the Pod themselves, but the binary/app that's running inside of the container needs to be monitored. A Pod/container can be running just fine, but the application inside of the Pod/container may not be.

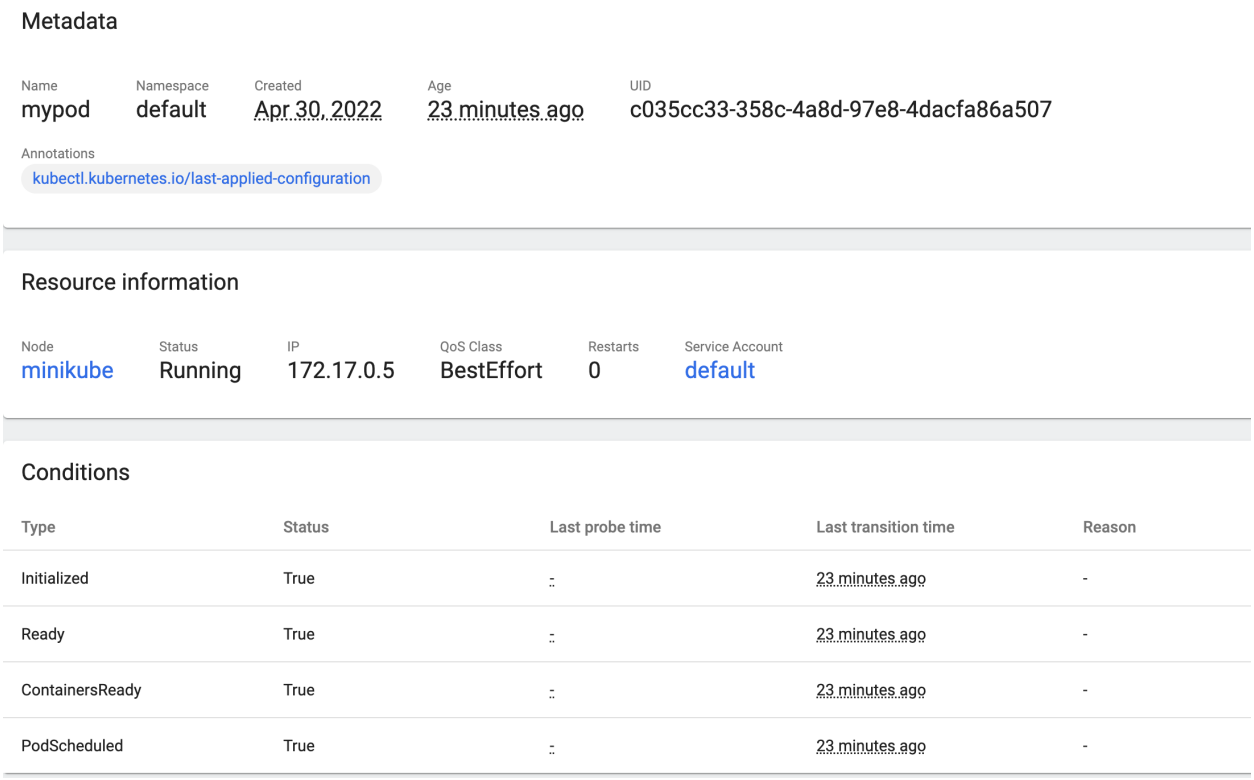
How It Works

There are several ways to monitor, so we won't see all of the options in this blog post, but a few tools you should think about are:

- Monitors in the cloud service you're using. For example, Azure Monitor or AWS CloudWatch
- Prometheus, a popular container monitoring tool
- Utilizing Grafana and Prometheus together for a better view. Grafana sort of sits in front of Prometheus and makes a better visualization experience

Regardless of which option you choose, you always want to ensure you're monitoring both the infrastructure layer and the application layer.

An example screenshot below shows an Nginx pod running on a Kubernetes cluster using Kubernetes Dashboard. This is just one of the many ways to monitor Pod activity.



7: Proper Networking

As engineers dive deeper and deeper into Kubernetes, many are surprised at just how important networking is in the k8s space. The truth is, applications (backend, frontend, etc.) won't work out-of-the-box without some sort of emphasis on networking.

Let's say you're installing Kubernetes on a couple of Ubuntu virtual machines using Kubeadm. Although Kubeadm will install just fine, it doesn't automatically install any networking components. You have to install a Kubernetes networking framework, and there are a lot to choose from. Some are more robust and complicated. Others are more straightforward and work out of the box.

A lot of engineers will either use:

- Weave
- Flannel

You can see a lot of network frameworks here: https://kubernetes.io/docs/concepts/cluster-administration/networking/#how-to-implement-the-kubernetes-networking-model

Once the network is up, you have to think about how Kubernetes Pods are going to talk to each other and allow you to talk to them from the outside world.

There are a few different ways to work with Pod communication:

- Services
- Ingress Controllers
- Service Mesh

A Kubernetes service allows you to expose apps that are running inside of a Pod as a network service. This is great if you need, for example, a public facing application. If you're in the cloud, say Azure or AWS, you can expose a Kubernetes Service with a cloud load balancer.

Ingress controllers take cloud and on-prem load balancers to the next level. Sort of think about it like a reverse proxy if you have a network background. You can take a bunch of Kubernetes Services and tie them to one load balancer. This is the preferred method vs having a bunch of load balancers to manage. Not only can it be time consuming, but cloud load balancers aren't cheap.

Once Kubernetes applications that are public facing are working properly, you have to think about how applications will communicate with each other. That's where a Service Mesh comes in. A Service Mesh, inside and outside of Kubernetes, has one primary purpose; control how different parts of an application communicate with one another. Although a service mesh is specifically for applications, it's technically considered part of the infrastructure layer. The reason why is because a lot of what a service mesh is doing is sending traffic between services, which is primarily a networking component.

Although the primary functionality of a service mesh for many organizations is the service communication, it can perform several other tasks including:

- Load balancing
- Observability
- Security including authorization policies, TLS encryption, and access control
- Helps troubleshoot network latency

This section may seem like less of a tip and more of an explanation, but the goal is to help you understand that networking is a huge part of Kubernetes. Much like when you're working on-prem or in the cloud, networking architecture should always be at the front of your mind.

How It Works

Let’s take a look at an example of a Kubernetes Ingress Controller using Nginx Ingress. The example will have an assumption that you’re running Kubernetes on `localhost`, so something like Minikube. However, you’ll see in the code that you can easily change it.

First things first, you need a Kubernetes Deployment and Service, which you’ll see in the Kubernetes Manifest below:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginxdeployment
  replicas: 2
  template:
    metadata:
      labels:
        app: nginxdeployment
    spec:
      containers:
      - name: nginxdeployment
        image: nginx:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginxservice
spec:
  selector:
    app: nginxdeployment
  ports:
    - protocol: TCP
      port: 80
```

After you define your Deployment and Service, you can define your Ingress:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-nginxservice-a
spec:
  ingressClassName: nginx-servicea
  rules:
  - host: localhost
    http:
```

```
      paths:
      - path: /nginxappa
        pathType: Prefix
        backend:
          service:
            name: nginxservice
            port:
              number: 8080
```

Notice how the Ingress is pointing to the `nginxservice` that's in the Kubernetes Manifest above the Ingress Manifest. In the Ingress Manifest, you're also creating a path that you want the application to be reached on and the port.

8: Make The Process Repeatable

As with everything in the tech world today, the goal is to make everything as repeatable as possible. Although there are many pieces of Kubernetes that are already automated for you, there are still two pieces that you'll have to think about automating:

- Automate cluster creation
- Automate app deployment creation and updates

For cluster creation, it's all Infrastructure-as-Code related typically. If you're creating an on-prem Kubernetes cluster, chances are you'll have a combination of Infrastructure-as-Code and Configuration Management. For example, Terraform to deploy the cluster and Ansible to configure the operating systems, install Kubernetes, etc...

For app deployment, you'll most likely start with a Kubernetes Manifest. Although the app creation is automated with a k8s Manifest, it's not automated, as in, the deployment itself. To automate the deployment, you don't want to constantly have to do it locally, so something like CICD would be efficient.

How It Works

Let's take a look at an infrastructure repeatable process using Terraform.

Below you'll see a Terraform `main.tf` that allows you to create an Azure Kubernetes Service (AKS) cluster. Notice that there are four variables:

- `name` = name of the cluster
- `location` = Azure region
- `resource_group_name` = Where the AKS cluster will live

- `node_count` = How many worker nodes will exist for the AKS cluster

Because of those four variables, this `main.tf` is pretty much repeatable for any environment. You can create the cluster, spin it down, and re-create it whenever you'd like.

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "=3.0.0"
    }
  }
}

provider "azurerm" {
  features {}
}

resource "azurerm_kubernetes_cluster" "k8squickstart" {
  name                = var.name
  location            = var.location
  resource_group_name = var.resource_group_name
  dns_prefix          = "${var.name}-dns01"

  default_node_pool {
    name       = "default"
    node_count = var.node_count
    vm_size    = "Standard_A2_v2"
  }

  identity {
    type = "SystemAssigned"
  }

  tags = {
    Environment = "Production"
  }
}
```

9: Testing

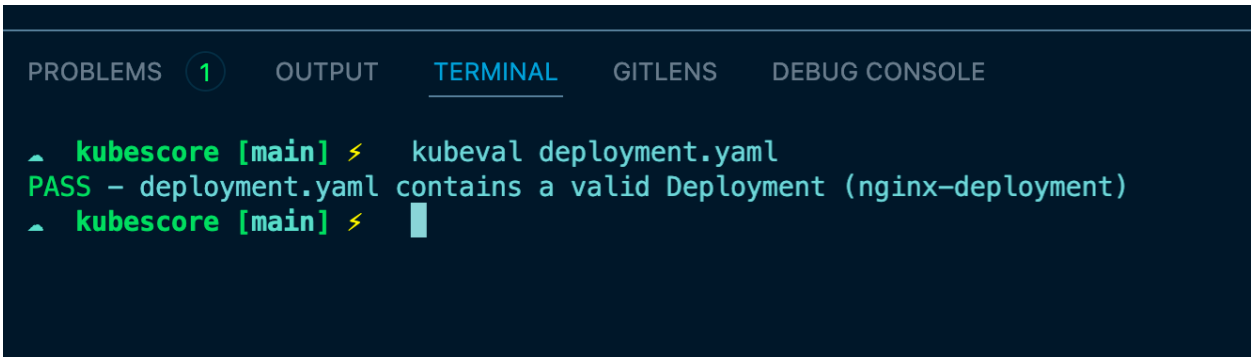
Code is code and apps are apps. It doesn't matter where they're running. Whether it's on a virtual machine, bare metal, or a container. You need to test functionality of an application. You need to unit test it. You need to confirm it works the way you're expecting and QA it. It's not any different for Kubernetes.

From an application/developer perspective, there's a few things to test:

- Integration testing to confirm the app is working as you're expecting

- Probes (livenessProbes, readinessProbdes) which you saw in a previous section
- Linting (static code analysis)
- Policy-as-Code
- Performance testing (benchmarking an app)

To validate Kubernetes Manifests for example, there's a great tool called Kubeval.



From an infrastructure standpoint, there's the standard ways you would test infrastructure with various scenarios:

- Perform a benchmark to see if, for example, the cluster autoscales
- Confirm the load on the servers is what you're expecting
- Test fault tolerance and HA

Although sort of out-of-scope, there's also the piece of testing the tools you'll use for deploying Kubernetes. For example, if you're using Terraform to deploy Kubernetes clusters, you should use a testing tool like Terratest to test the Terraform code.

How It Works

Although there are many types of tests, let's focus on linting. A linter is a tool that scans your code to confirm there are no errors. Some engineers call it Static Code Analysis.

Let's take this Kubernetes manifest as an example. Save it in a location of your choosing and name it `deployment.yaml`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
```

```
  selector:
    matchLabels:
      app: nginxdeployment
  replicas: 2
  template:
    metadata:
      labels:
        app: nginxdeployment
    spec:
      containers:
      - name: nginxdeployment
        image: nginx:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 80
```

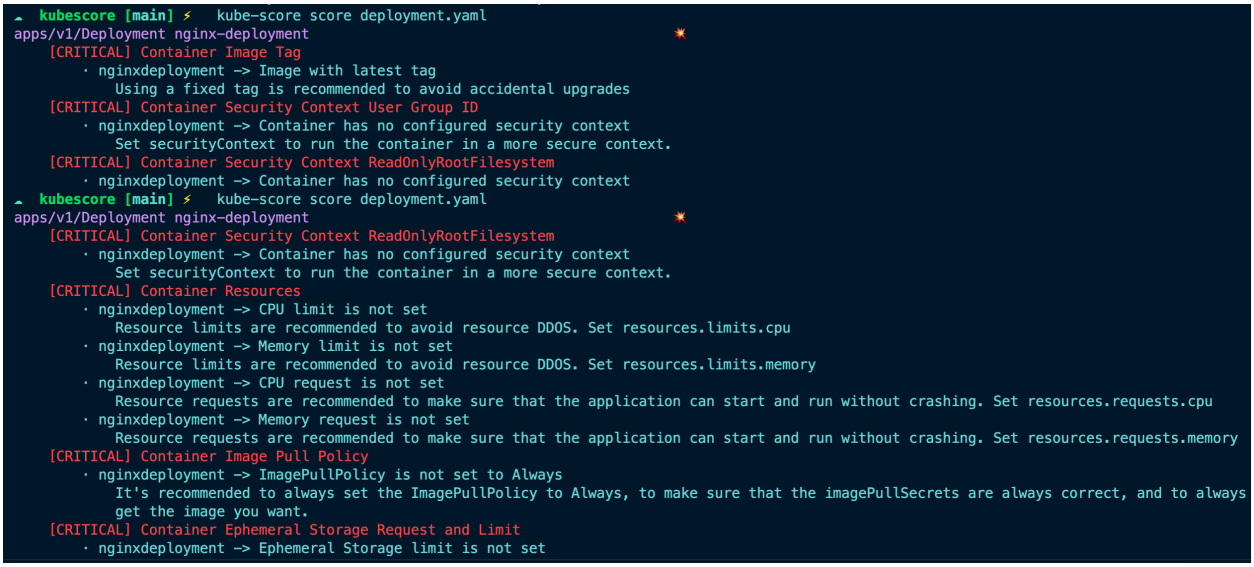
Next, install kube-score. The Mac example is below, but there are a few other options which you can find here.

```
brew install kube-score
```

After installing kube-score, you can run kube-score by using the following command:

```
kube-score score deployment.yaml
```

You'll see an output similar to the screenshot below.



10: Backups and Storing Data

The final tip is to think about backups and how to store data. Backups in Kubernetes are strange. For example, if you have to back something up, that means it's Stateful data. Some engineers steer clear of using Kubernetes for Stateful applications while others embrace it. If you have a Stateful app, that means it needs to be backed up and redundant in one way, shape, or form.

Outside of the application itself, there's also the infrastructure. If you're running a cloud service like AKS or EKS, backing up infrastructure won't do much for you. If you're running a bare metal Kubernetes cluster, it will mean everything. For example, Etcd, the k8s data store/database holds pretty much every piece of critical information when it comes to the Kubernetes cluster. If Etcd goes down and you don't have a backup, you're going to be in a world of tech hurt.

How It Works

Backing up a cluster itself or putting etcd on another server is going to depend on an environment and what you're using. As an example, if you're using AKS or EKS, you probably won't be doing any actual cluster backups. However, you do need to ensure that your code is in a location that you can pull from and everything should be able to be re-deployed from code.

What you can do in this case is:

- Ensure that all Kubernetes Manifests are stored in source control, like GitHub
- Use GitOps to automatically deploy and control the state of your environment