Short Circuit

Kubernetes

The Kubernetes Book
by Nigel Poulton
2025 Edition
Up until Chapter 6: Kubernetes Deployments - Create a Deployment

Summary by Emiel Bos

1 Intro

The word Kubernetes originates from the Greek word for helmsman, a person who steers a ship. You can see this in the logo. You'll often see it shortened to K8s and pronounced as "kates". Kubernetes was developed by a group of Google engineers partly in response to Amazon Web Services (AWS) and Docker. Google built their own cloud, but needed a way to abstract the value of AWS and make it as easy as possible for customers to get off AWS and onto their cloud. They also ran their own production apps on billions of containers per week.[1] At the same time, Docker was taking the world by storm, and users needed help managing explosive container growth. In 2014, Google open-sourced Kubernetes and donated it to the then newly formed Cloud Native Computing Foundation (CNCF). In 2016 and 2017, Kubernetes, Docker Swarm, and Mesosphere DCOS competed to become the industry standard container orchestrator. Kubernetes won. However, Docker Swarm is still being developed and is still used by small companies needing a simple alternative to Kubernetes. The Kubernetes documentation uses Pascal case (aka UpperCamelCase) when referring to API objects, and the same is done here.

Docker Desktop installs Docker, a multi-node Kubernetes cluster, and the `kubectl` and `docker` command-line utilities. Docker Desktop runs your Kubernetes cluster as containers (the user experience is exactly the same though). All major cloud providers have their own Kubernetes service as well, in case you want a multi-node cluster in the cloud. Behind the scenes, `kubectl` reads your *kubeconfig* file to know which cluster to send commands to and which credentials to authenticate with. Your kubeconfig file is called `config` and is in `/Users/<username>/.kube/config` (macOS) or `C:\Users\<username>\.kube\config` (Windows). It defines a list of known Kubernetes clusters, a list of user credentials, contexts (which are pairs of clusters and credentials, e.g. you might have a context called `ops-prod` that combines the `ops` credentials with the `prod` cluster), and the `current-context` (to which commands will be send). If using Docker Desktop, you can easily switch between contexts by clicking the Docker whale and choosing the Kubernetes Context option. You can run a `kubectl config view` command to view your kubeconfig (sensitive data is redacted). You can also run a `kubectl config current-context` to see your current context. An example of a kubeconfig:

```
apiVersion: v1
kind: Config
clusters: # All known clusters are listed in this block
- name: shield # Friendly name for a cluster
 cluster:
  server: https://192.168.1.77:8443 # Cluster's API endpoint/server
  certificate-authority-data: LS0tLS... # Cluster's certificate
users: # Users are in this block
- name: coulson # Friendly name (not used by Kubernetes)
 user:
  client-certificate-data: LS0tLS1CRU... # User certificate/credentials
  client-key-data: LS0tLS1CRUdJTiBFQyB # User private key
contexts: # List of contexts (cluster:user pairs)
- context:
```

[1] Google used their proprietary Borg and Omega container orchestrators for this. Kubernetes is not an open-source version of Borg or Omega, but it shares its DNA and family history with them.

```
  name: director
   cluster: shield # Send commands to this cluster
   user: coulson # Authenticate with these credentials
current-context: director # kubectl will use this context
```

Every time you execute a `kubectl` command it converts the command into an HTTP REST request, sends the request to the Kubernetes cluster defined in the `current-context` of your kubeconfig file, and uses the credentials specified in the `current-context` of your kubeconfig file

Kubernetes is both

- a cluster, which is one or more *nodes* – physical servers, virtual machines, or cloud instances – providing CPU, memory, and other resources for application use. Kubernetes supports two node types:
  - *Worker nodes* run the (user) applications, aka *workloads*. Workers can be Linux or Windows. It consists of
    - The *kubelet* is the main Kubernetes agent and handles all communication with the cluster. It watches the API server for new tasks, instructs the appropriate runtime to execute tasks, and reports task status to the API server. If a task won't run, the kubelet reports the problem to the API server and lets the control plane decide what actions to take.
    - *Runtimes* for executing tasks, which includes pulling container images and managing lifecycle operations such as starting and stopping containers. All the early versions of Kubernetes shipped with Docker as its container runtime, but Docker got bloated and Docker alternatives were developed. As a result, the Kubernetes project created the *container runtime interface* (CRI) to make the runtime layer pluggable. Some runtimes provide better isolation, some provide better performance, some work with Wasm containers, etc. Kubernetes 1.24 finally removed support for Docker as a runtime as it was bloated and overkill for what Kubernetes needed. Since then, most new Kubernetes clusters ship with containerd (pronounced "container dee") as the default runtime. containerd is a stripped-down version of Docker optimized for Kubernetes, that fully supports applications containerized by Docker. In fact, Docker, containerd, and Kubernetes all work with images and containers that implement the Open Container Initiative (OCI) standards. Different nodes in a single Kubernetes cluster can run different container runtimes, and a single node can even run multiple runtimes. RedHat OpenShift clusters use the CRI-O runtime.
    - A *kube-proxy service* that implements cluster networking and load balances traffic to tasks running on the node.
  - *Control plane nodes* run the system services, and implement the Kubernetes intelligence, or, the *control plane*. Every cluster needs at least one control plane node. However, you should have three or five spread across availability/fault zones for high availability (HA). Every control plane node runs every control plane service, which are
    - The *API server* is the front end of Kubernetes, and all commands and requests go through it. Even internal control plane services communicate with each other via the API server. It's a REST API over HTTPS, and all requests are subject to authentication and authorization.
    - The *cluster store* holds the desired state of all applications and cluster components, and it's the only stateful part of the control plane. It is based on the etcd distributed database, and most Kubernetes clusters run an etcd replica on every control plane node for HA (though this is never a substitute for backup and recovery). However, large clusters that experience a high rate of change may run a separate etcd cluster for better performance. etcd prefers an odd number of replicas to help avoid *split brain conditions*, when replicas experience communication issues and cannot be sure if they have a quorum (majority). If a split-brain occurs, etcd goes into read-only mode preventing updates to the cluster. User applications will continue working, but Kubernetes won't be able to scale or update them. Lastly, etcd uses the RAFT consensus algorithm to prevent inconsistent writes.
    - *Controllers* implement a lot of the cluster intelligence and ensure the cluster runs what you asked it to run. At the highest level, *resources* (API objects) define objects and controllers manage them. Each controller runs as a process on the control plane, and some of the more common ones include the Deployment, StatefulSet, ReplicaSet, and DaemonSet controllers, and many more. They all run as background watch loops, reconciling observed/actual/current state with desired state, in a declarative model[2] (specifying

[2] As opposed to the much more complex imperative model. Kubernetes supports both but prefers the declarative model, because the declarative model integrates with version control systems and enables self-healing, autoscaling, and rolling updates.

the what, not the how). Kubernetes also runs a *controller manager* that is responsible for spawning and managing the individual controllers. The desired state is described in a YAML file, through a process called reconciliation. You post the YAML to the API server – most commonly done with the `kubectl` command-line utility – which authenticates and authorizes it and persists it in the cluster store as a record of intent. You'll almost always deploy Pods via controllers, not manually or individually.

  - The *scheduler* watches the API server for new work tasks and assigns them to healthy worker nodes. Identifying capable nodes involves predicate checks, filtering, and a ranking algorithm. It checks for taints, affinity and anti-affinity rules, network port availability, and available CPU and memory. It ignores nodes incapable of running the tasks and ranks the remaining ones according to factors such as whether it already has the required image, the amount of available CPU and memory, and number of tasks it's currently running. Each is worth points, and the nodes with the most points are selected to run the tasks. The scheduler marks tasks as pending if it can't find a suitable node. If the cluster is configured for node autoscaling, the pending task kicks off a cluster autoscaling event that adds a new node to the cluster and the scheduler assigns the task to the new node.
  - If your cluster is on a public cloud, such as AWS, it will run a *cloud controller manager* that integrates the cluster with cloud services, such as instances, load balancers, and storage. For example, if you're on a cloud and an application requests a load balancer, the cloud controller manager provisions one of the cloud's load balancers and connects it to your app.

Control plane nodes must be Linux.

It's common to run user applications on control plane nodes in development and test environments. However, many production environments restrict user applications to worker nodes so that control plane nodes can focus their resources on cluster operations. If you're running a multi-node cluster locally, you can use `kubectl get nodes` to list the nodes in the cluster, together with their type, status, age and version of Kubernetes.

- an `orchestrator` of container applications, which is a system that deploys applications across nodes and failure zones for optimal performance and availability, and dynamically responds to changes, i.e. scaling them up and down based on demand, self-healing them when they break, rollouts and rollbacks, etc. The benefits are abstraction of infrastructure and simplified applications portability. Kubernetes is the de facto platform for cloud-native applications, and is therefore sometimes referred to as the operating system (OS) of the cloud. This is because it abstracts cloud resources and schedules application microservices the same way that operating systems like Linux and Windows abstract hardware resources and schedule application processes. Kubernetes makes it easier to deploy to one cloud today and migrate to another cloud tomorrow.

Kubernetes runs containers, VMs, Wasm apps, and more, all of which need wrapping in Pods before they'll run on Kubernetes.[3] Pods can in turn be wrapped in higher-level controllers (e.g. a Deployment) for advanced features. This means there can be three layers of wrapping: the container wraps the app and provides dependencies, the Pod wraps the container so it can run on Kubernetes, and the Deployment wraps the Pod and adds self-healing, scaling, and more. Pods are lightweight and add very little overhead. Pods run on nodes.

2 Pods

Pods are the atomic unit of scheduling in Kubernetes; it doesn't schedule individual containers. Kubernetes schedules containers in the same Pod to a single node (because Pods are a shared execution environment, and you can't easily share resources across different nodes). Starting a Pod is also an atomic operation; Kubernetes only marks a Pod as ready when all its containers are running. You scale an application up by adding Pods (so not by adding containers to existing Pods) and down by deleting Pods.

Pods are *mortal*; once a Pod dies, Kubernetes replaces it with a new one, with a new ID and new IP. When a node fails, or a node evict Pods during node maintenance, and a Deployment controller notices this, it will delete Pods on that node and create new ones on a surviving node, even though it is based on the same Pod spec. The point is, "restarting" Pods actually means deleting and creating Pods, with new UIDs, IPs, and no state.

Pods are *immutable*; you never change them once they're running, but you need to replace them. Kubernetes prevents changes to a running Pod's configuration, but it can't always prevent you from changing the app and/or filesystem inside

[3] Containers and Wasm apps work with standard Pods, workload controllers, and runtimes. However, serverless functions run in standard Pods but require apps like Knative to extend the Kubernetes API with custom resources and controllers. Likewise, VMs need apps like KubeVirt to extend the API and run in a VirtualMachineInstance (VMI) instead of a Pod (but they are modified Pods and very similar and utilize a lot of Pod features). VMs also need custom workload controllers. VMs are the opposite of containers in that they are designed to be mutable and immortal, hence the necessasity of VMIs and custom workload controllers.

of containers; you're responsible for ensuring they're stateless and immutable. If you need to write data to a Pod, you should attach a volume to it and write to that, which will persist after the Pod is gone.
Containers can actually be restarted, which is always done by the local kubelet and governed by the value of the Pod's `spec.restartPolicy`, which can be `Always` (typically for long-living containers/host apps such as web servers, data stores, and message queue), `Never` (when it doesn't matter if a container fails), or `OnFailure` (only attempts a restart if the container fails with an error code, typically for short-living containers that run batch jobs). The policy is Pod-wide, meaning it applies to all containers in the Pod except for init containers.

The `kubectl exec` command lets you execute commands inside running containers. You can use it in two ways:

- `Remote command execution` lets you send commands to a container from your local shell, with output returned to your shell: `kubectl exec <pod> -- <command>`. Use the `--container` option if you want to run the command in a specific container.

- `exec sessions` connect your local shell to the container's shell – specifically your shell's STDIN and STDOUT streams to the STDIN and STDOUT of the container in the Pod – and is the same as being logged on to the container in an SSH session. You do this with the `--it` flag: `kubectl exec -it <pod> -- sh`. The `sh` command starts a new shell process in the session, and your prompt will change to indicate you're now inside the container.

Pod creation and deletion generate *IP churn* and make Pods unreliable; clients cannot make reliable connections to individual Pods as Kubernetes doesn't guarantee they'll exist. Therefore, *Services* come into play by providing a reliable DNS name, IP address and port for a dynamic group of Pods behind it (you can consider this the Service's front-end), and load balances requests to them using labels (the "back-end").

Every Kubernetes cluster runs a Pod network and automatically connects all Pods to it. It's usually a flat layer-2 overlay network that spans every cluster node and allows every Pod to talk directly to every other Pod, even if the remote Pod is on a different cluster node. Your Pod network is implemented by a third-party plugin that interfaces with Kubernetes via the Container Network Interface (CNI). You choose this network plugin at cluster build time, and it configures the Pod network for the entire cluster. Lots of plugins exist, and each one has its pros and cons. Cilium is the most popular at the time of writing and implements a lot of advanced features such as security and observability.

3 Multi-container Pods

The simplest configurations run a single container per Pod (which is why the terms Pod and container are sometimes used interchangeably), but you can have multi-container Pods. In this way, each container is kept simple and focussed on one task, the *single responsibility principle*. The container(s) run in an *execution environment*, which includes a filesystem and volumes (`mnt` namespace), network stack (`net` namespace), memory (`IPC` namespace), process tree (`pid` namespace), and hostname/IP address (`uts` namespace). These are all shared among the containers. IP address is also shared in that case, with different containers having different ports, and they can communicate with each other with the `localhost` interface. You should use multi-container Pods for tightly coupled components that need to share resources, but you should prefer single-container Pods and loosely couple them over the network. Kubernetes has two main patterns for multi-container Pods:

- *init containers* are a special type of container defined in the Kubernetes API that prepare and initialize an environment so it's ready for application containers. You run them in the same Pod as application containers, but Kubernetes guarantees they'll start and complete before the main app container starts and that they'll only run once. You can list multiple init containers per Pod and Kubernetes runs them in the order they appear in the Pod manifest. If any init container fails, Kubernetes attempts to restart it (or fail the Pod if you've set the Pod's `restartPolicy` to `Never`.

- *sidecar containers* run alongside the main app container, and are for adding functionality to an application without having to add it directly to the application container, e.g. telemtry, scraping logs, encryption, etc. They start before the main application container, keep running alongside the main application container, and terminate after the main application container. You define sidecars as init containers (`spec.initContainers`) with the `restartPolicy` set to `Always`.

4 Deployment

You can deploy a Pod manually with the following steps:

1. Define the Pod in a YAML manifest file. An example:

```
kind: Pod
apiVersion: v1 # Build the Pod using the v1 Pod schema
metadata:
  name: hello-pod # Used as HOSTNAME for every container in this Pod (so it should be a valid
      DNS name)
  labels: # To connect to a service for networking
     zone: prod
     version: v1
spec:
   initContainers: # Specifies init containers and sidecar containers
   # Init container
   -  name: init
      image: busybox:1.28.4
      command: ['sh', '-c', 'until nslookup some_service; do echo waiting for
          some_service;\sleep 1; done; echo service found!']
   # Sidecar container
   -  name: syncer
      restartPolicy: Always # This makes this a sidecar. Overrides the Pod's restartPolicy
      image: k8s.gcr.io/git-sync:v3.1.6
      volumeMounts:
      -  name: html
         mountPath: /tmp/git
      env: # Environment variables for use by the container
      -  name: GIT_SYNC_REPO
         value: https://github.com... # This specific container watches this repo...
      -  name: GIT_SYNC_BRANCH
         value: master
      -  name: GIT_SYNC_DEPTH |
         value: "1" |
      -  name: GIT_SYNC_DEST |
         value: "html" # ...and syncs its contents into this shared volume
  containers: # Specifies regular containers (that won't start before all initContainers are
      done)
  -  name: ctr-web
     image: nginx
      volumeMounts:
      -  name: html
         mountPath: /usr/share/nginx/
     ports:
     - containerPort: 8080
     resources:
         requests: # Minima for scheduling
            memory: 256Mi
            cpu: 0.5
       limits: # Maxima for kubelet to cap
         memory: 512Mi
         cpu: 1.0
   volumes:
   -  name: html
      emptyDir: {}
```

Pods have a lot of properties, and anything you don't explicitly define in a YAML file gets populated with defaults.

2. Post the manifest to the API server. `kubectl apply -f pod.yml` will send the `pod.yml` file to the API server defined in the current context of your kubeconfig file (and authenticates the request using credentials from your kubeconfig file). Run `kubectl get pods` to list Pods and their status and how many of their containers are ready. Adding the `-o wide` option gives a few more columns, and `-o yaml` gets you everything Kubernetes knows about the object; it'll give the desired state in the `spec` block and the current state in the `status` block. The `--watch` flag will make it update. Another great command is `kubectl describe pod <pod>`, which gives a nicely formatted overview of an object, including lifecycle events. You can use the `kubectl logs <pod> --container <container>` command to pull the logs from any container in a Pod; if you don't specify a container, it will pull from the first container in the Pod (useful for single-container Pods).

3. The request is authenticated and authorized

4. The Pod spec is validated

5. The scheduler filters nodes based on

   - *nodeSelectors*; the simplest way of running Pods on specific nodes. You give it a list of labels, and the scheduler will only assign the Pod to a node with all the labels.

   - Affinity rules. *Affinity rules* attract, *anti-affinity rules* repel, *hard affinity rules* must be obeyed, *soft affinity rules* are only suggestions and best effort.

   - *Topology spread constraints*; a flexible way of intelligently spreading Pods across your infrastructure (e.g. availability zones) for availability, performance, locality, etc. across your cloud or data center's underlying. However, you can create custom domains for almost anything, such as scheduling Pods closer to data sources, closer to clients for improved network latency, etc.

   - resource requests (minima) and limits (maxima) tell the scheduler how much CPU and memory a Pod needs, and the scheduler uses them to ensure they run on nodes with enough resources. If you don't specify them, the scheduler may schedule it to a node with insufficient resources. A container never uses more than its limits. For multi-container Pods, the scheduler combines the requests for all containers. Kubernetes automatically sets requests to match limits if you don't specify requests. Every Pod should use request limits.

   - etc.

   The Pod is assigned to a healthy node meeting all requirements. (If the scheduler can't find a suitable node, it marks it as pending.)

6. The kubelet on the node watches the API server and notices the Pod assignment

7. The kubelet downloads the Pod spec and asks the local runtime to start it

8. The kubelet monitors the Pod status and reports status changes to the API server

If it's a short-lived Pod (e.g. a batch job), it enters the *succeeded* state as soon as all containers complete their tasks. If it's a long-lived Pod (e.g. a web server), it remains *running* indefinitely. You can delete Pods with `kubectl delete -f <pod>|<yaml>`.

The manual method is rarely used, because it creates a static Pod that cannot self-heal, scale, perform rolling updates, and if the node fails, the kubelet fails with it and cannot do anything to help the Pod, because they're only managed by the kubelet on the node they're running on, and kubelets are limited to restarting containers on the same node. You'll usually deploy them via higher-level workload controllers that augment them with additional features. The most common of these controllers is a Deployment, which adds self-healing, scaling, rollouts, and rollbacks. An example:

```
kind: Deployment
apiVersion: v1
metadata:
  name: webapp
spec:
  replicas: 4 # No. of Pods, used by the ReplicaSet
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
  template: # Pod template/definition, as exemplified above, but without kind and apiVersion
    ...
```

Posting this Deployment YAML to the cluster will create a Deployment, which in turn creates and manages a ReplicaSet (which is the API object that actually provides the advertised self-healing and scaling features), which in turn creates and manages the Pods. You perform all management via the Deployment and never directly manage the ReplicaSet or Pods. The ReplicaSet controller makes sure the correct number of Pod replicas are always present.

You can scale your apps manually, but Kubernetes has several *autoscalers*:

- The *HorizontalPodAutoscaler* (HPA) adds and removes Pods to meet current demand. Installed on most clusters and widely used.

- The *VerticalPodAutoscaler* (VPA) increases and decreases the CPU and memory allocated to running Pods to meet current demand. It isn't installed by default, has several known limitations, and is less widely used. Current implementations work by deleting the existing Pod and replacing it with a new one every time it scales the Pods resources, which is disruptive and can even result in Kubernetes scheduling the new Pod to a different node. Work is underway to enable in-place updates to live Pods, but it's currently an early alpha feature.

- The *ClusterAutoscaler* (CA) adds and removes cluster nodes so you always have enough to run all scheduled Pods. Installed on most clusters and widely used.

- Community projects like karmada take things further by allowing you to scale apps across multiple clusters.

Modern cloud-native microservices should be loosely coupled and communicate only via well-defined via APIs, to allow updating without impacting other microservices. They should also be backward and forward compatible, to allow updating without caring which versions of clients are consuming your service. These are requirements for zero-downtime rolling updates (*rollouts*), which Kubernetes does by creating a new replica running the new version and terminates one running the old version until all replicas are on the new version. What happens under the hood, when you re-post an updated Deployment YAML file with a new Pod spec to the API server, the Deployment controller creates a new ReplicaSet with the same number of Pods (assuming you haven't changed that) but running the newer version and systematically increments the number of Pods in the new ReplicaSet as it decrements the number in the old ReplicaSet. The old ReplicaSet is kept for *rollbacks* (to an earlier version), and this is just the opposite process of a rollout. Kubernetes gives you fine-grained control over rollouts and rollbacks, e.g. through insert delays, controlling the pace and cadence of releases, and even probing the health and status of updated replicas.

Multi-dimensional autoscaling is jargon for combining multiple scaling methods

5 Namespaces

You can divide a Kubernetes cluster into multiple *virtual clusters* with *namespaces*.[4] The `kubectl api-resources` command prints the API resources/objects (Pods, services, deployments, etc.) supported on the server, and whether they are namespaced, i.e., whether you can deploy to specific namespaces with custom policies and quotas. Each namespace can have its own users, permissions, resource quotas, and policies
A *tenant* is a loose term that can refer to individual applications, different teams or departments, or even external customers. Namespaces are most commonly used to divide clusters for use by tenants within the same organization, such that multiple tenants can share the same cluster. Namespaces are easy to setup and manage, but only provide soft isolation and cannot prevent compromised workloads from escaping the namespace and impacting workloads in others. [5] For example, a production cluster can be divided into the finance, hr, corporate-ops, etc. namespaces.

You can list namespaces with the `kubectl get namespaces` or `kubectl get ns` command.[6] Every Kubernetes cluster has a set of pre-created namespaces: `default` is where objects are deployed if you don't specy a namespace, `kube-system` is where control plane components such as the internal DNS service and the metrics server run, `kube-public` is for objects that need to be readable by anyone, and `kube-node-lease` is used for node heartbeats and managing node leases. You can inspect a namespaces on your cluster with `kubectl describe ns <namespace>`.

Namespaces are first-class resources in the core v1 API group, meaning they're stable, well-understood, and have been around for a long time. You can work with them imperatively via the CLI and declaratively via YAML config files. You can create a new namespace imperatively with `kubectl create ns <namespace>` or declaratively with:

```
kind: Namespace
apiVersion: v1
metadata:
   name: <namespace>
   labels:
      env: <some_variable>
```

and running `kubectl apply -f <file>.yml`. You delete a namespace with `kubectl delete ns <namespace>`.
You can refer to namespaces imperatively with the aforementioned `-n` or `--namespace` flags, or declaratively:

```
apiVersion: v1
kind: ServiceAccount
metadata:
   namespace: <namespace>
   name: default
```

[4] Not to be confused with Linux kernel namespaces, which partition operating systems into virtual operating systems called containers.

[5] This is why using namespaces to divide a cluster among external tenants isn't as common. At the time of writing, the most common way of strongly isolating tenants is to run them on their own clusters and their own hardware.

[6] You can substitute `namespace` with `ns` when working with kubectl. You can also add `-n` or `--namespace` to kubectl commands to filter results against a specific namespace. When working with a specific namespace, it can be annoying having to add the `-n` or `--namespace` flag on all your kubectl commands. For this reason, you can set your kubeconfig to run all future kubectl command against a specific namespace with `kubectl config set-context --current --namespace <namespace>`. Make sure you set it to the `default` namespace again when you're done.