Intro to Kubernetes

Table of Contents

[Basic definitions 1](#_Toc80031535)

[Foreword: Containers 1](#_Toc80031536)

[Containers vs virtual machines 2](#_Toc80031537)

[Container runtimes 4](#_Toc80031538)

[What Kubernetes is and is not 4](#_Toc80031539)

[Downsides to using Kubernetes 5](#_Toc80031540)

[How Kubernetes works 5](#_Toc80031541)

[Pods 7](#_Toc80031542)

[Workload objects 7](#_Toc80031543)

[Deployments and ReplicaSets 7](#_Toc80031544)

[StatefulSets 7](#_Toc80031545)

[DaemonSets 7](#_Toc80031546)

[Jobs and CronJobs 7](#_Toc80031547)

[Networking 7](#_Toc80031548)

[Services 8](#_Toc80031549)

[Ingress 8](#_Toc80031550)

[Storage 9](#_Toc80031551)

[Setting up Kubernetes clusters 9](#_Toc80031552)

[Node preparation 9](#_Toc80031553)

[Installing Kubernetes 10](#_Toc80031554)

[Overview of the cluster state 10](#_Toc80031555)

[Configuring client access 10](#_Toc80031556)

[Managing multiple clusters 11](#_Toc80031557)

[Using Kubernetes Clusters 11](#_Toc80031558)

[Overview of a basic application 11](#_Toc80031559)

[Monitoring state 13](#_Toc80031560)

[Liveness, start-up, and readiness probes 15](#_Toc80031561)

[Stateful workloads 15](#_Toc80031562)

[Obtaining and using storage 16](#_Toc80031563)

[Services 16](#_Toc80031564)

[Load balancers 16](#_Toc80031565)

[Headless services 16](#_Toc80031566)

[Secrets and ConfigMaps 16](#_Toc80031567)

[RBAC 17](#_Toc80031568)

[Service accounts 17](#_Toc80031569)

[Using Helm 17](#_Toc80031570)

[Administering Kubernetes clusters and workloads 18](#_Toc80031571)

[Node maintenance 18](#_Toc80031572)

[Cluster upgrades 19](#_Toc80031573)

[Monitoring 19](#_Toc80031574)

[Troubleshooting 20](#_Toc80031575)

[Logging 20](#_Toc80031576)

[Workloads 21](#_Toc80031577)

[Storage 22](#_Toc80031578)

[Network 22](#_Toc80031579)

[Nodes 22](#_Toc80031580)

[Debugging with Telepresence 23](#_Toc80031581)

[Application and cluster backups 23](#_Toc80031582)

[Resources 24](#_Toc80031583)

[Links 25](#_Toc80031584)

# Basic definitions

A “container image” is a pre-packaged program containing all the dependencies and program code in a single unit. This allows the user to run the application directly in all compatible environments without having to install or compile any additional software apart from the container runtime. That simplifies deployment to a large degree.

A “container” is an instance of a container image (be it running or not) with applied metadata, mounted storage and, in the case of currently executing ones, running processes.

A “compatible environment” in the above context is any such environment which is binary-compatible with the prepared image. A developer must create a separate image if they want to target different processor (instruction set) architectures. For example, x86\_64 (AMD64), ARM and other RISC architectures, PowerPC, all require a separate image. Because a container shares the host operating system resources, including the kernel, that must also be compatible. Docker Desktop makes this mostly transparent by running containers inside a Linux virtual machine under Hyper-V on Windows or QEMU on macOS.

A “container runtime” is software which runs containers according to user configuration. It handles user input, pulls images, expands (decompresses) them, mounts storage, attaches network, prepares metadata and calls the container entry point. There are many such runtimes, including Docker, RKT, CRI-O, and containerd, among others.

A “microservices architecture” is any such architecture which breaks up a solution to a business problem into small, loosely coupled, collaborating services. Each of those services does a single specific task and potentially passes on information to others. Such architectures are easier to develop and maintain by a small team, allow for independent scaling of its components, improves the separation of concerns between facets of the overall application, and allows each service to be independently deployable and testable, further improving maintainability.

Kubernetes (or K8s for short) is a container orchestration platform. It allows for declarative configuration and automation of deployments of microservices applications, scaling (including autoscaling), storage provisioning, service discovery, self-healing, and configuration management. This is not an exhaustive list because its catalogue of features is growing rapidly. It also has a very lively ecosystem with many services, support, tools and documentation available online.

# Foreword: Containers

Containers are useful for soft isolating processes into their own environment in order to apply limitations to their resource usage, limit the potential impact of bugs and security problems, and package necessary libraries and other files needed to run the program. They usually only run a single process (it is not recommended to run multiple processes in a single container; if you find yourself needing to do that, consider splitting the service into multiple, smaller services).

Packaging an application into a container simplifies its distribution because all steps to prepare the environment are done upfront. Another upside is that the preparation is described in concrete terms into Dockerfiles, which makes it easy to understand what software it uses and forces developers to think clearly about the implications to deploying their software. Once packaged, the image is uploaded to a registry, from where it can be read across the network.

Storage in containers is provided in layers: several snapshots of a file system (some read-only) are layered on top of each other to provide a full picture of one for a running container.

## Containers vs virtual machines

Although both virtual machines (VMs) and containers are useful for isolating workloads, there are a few major differences between the two approaches.

VMs are “virtual” instances of all hardware needed to run an operating system. This means that they have their own kernels, drivers, memory management, and process scheduling. All processes running inside a VM are fully isolated from the host computer running the hypervisor, and vice versa.

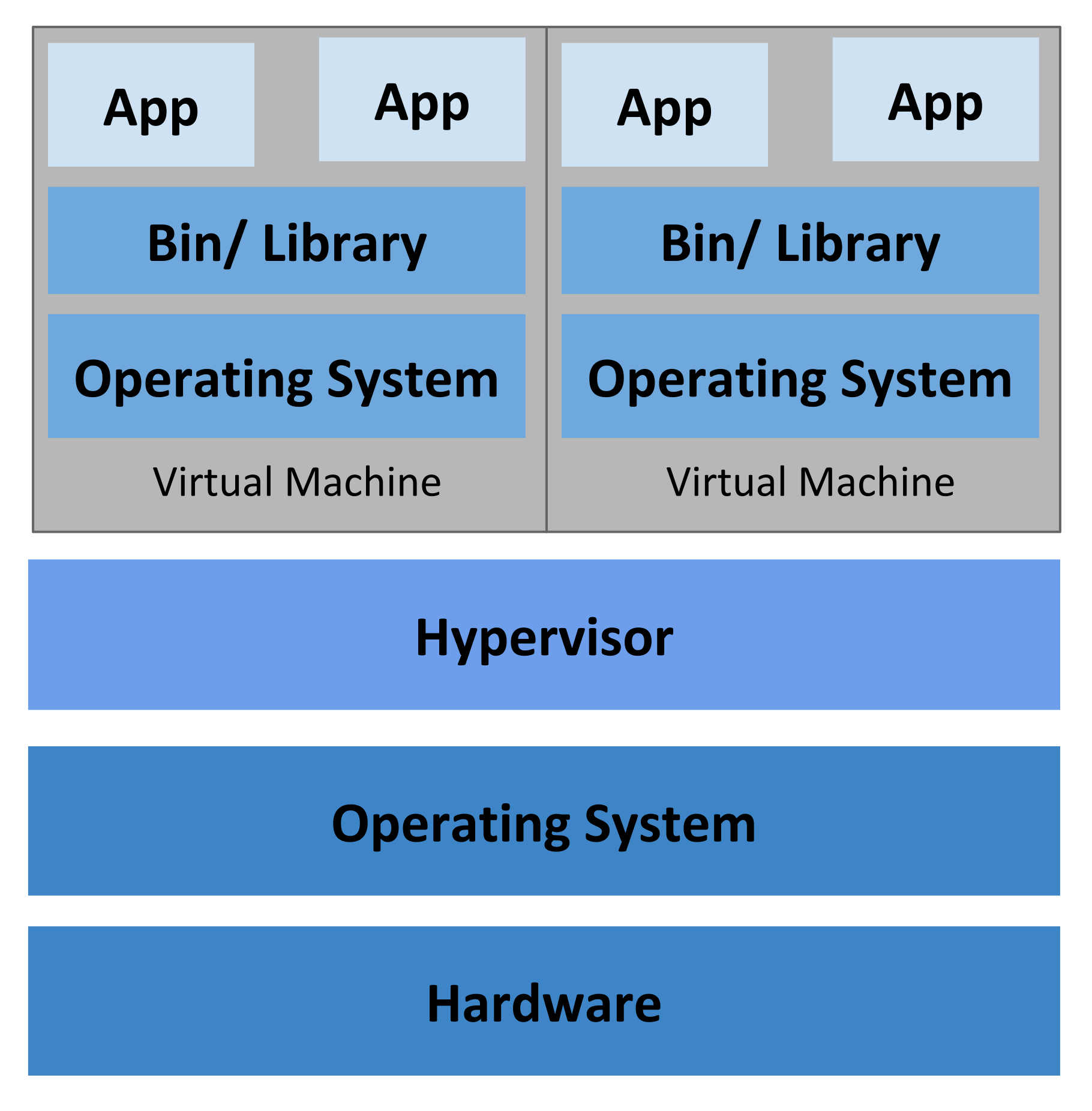
VMs have several distinct advantages over containerisation:

* The resources allocated to a VM are usually guaranteed, meaning that processes inside it do not compete with others outside of it for access to memory and CPU time (there are exceptions).
* Since a hypervisor simply provides the (virtual) hardware, a user can install a different operating system inside a VM from that of the host computer.
* A VM provides full, hard isolation of the processes running inside it.

They also come with a few downsides:

* As stated above, resources assigned to a VM are pre-allocated and cannot be shared with other programs running on the host machine. That usually results in their under-utilisation and makes running a VM a lot “heavier”.
* Start-up times for VMs are typically a lot longer than that of a container.
* The requirement to run a full operating system inside the VM means there is a lot of memory being used to run essentially the same software (the OS) over and over. With scale, this can mean a lot of resources being wasted.

Below is a diagram of the layers associated with running virtual machines.



In contrast, containers share the same operating system with the host and between each other. This makes them a lot more lightweight and results in faster start-up times. To achieve isolation, container runtimes leverage the kernel’s namespaces feature. This feature allows to silo processes: they can only “see” resources and other processes in the same namespace.

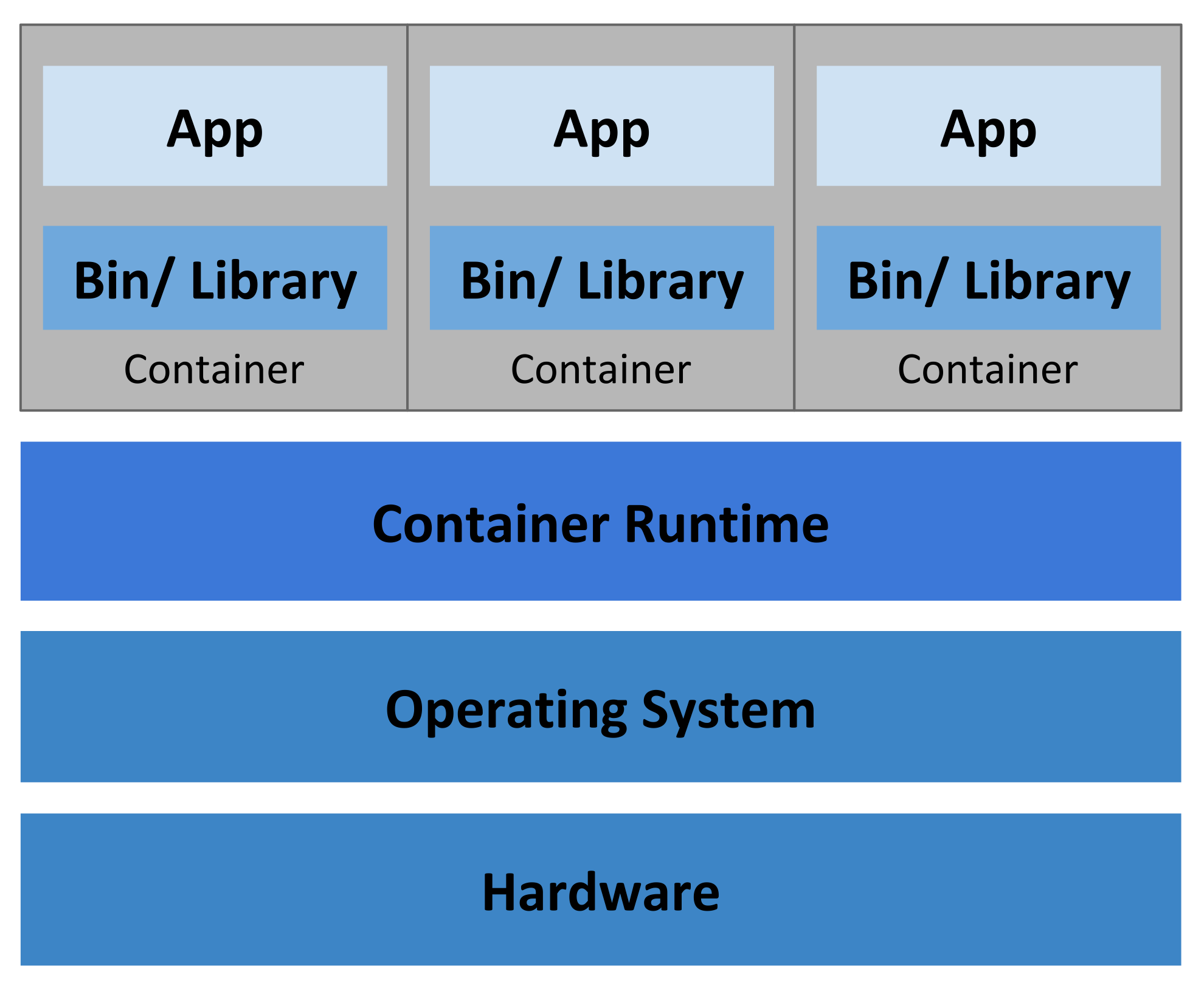
The upsides to containerisation include:

* Sharing the memory space of the host results in a lot less CPU time and memory needed to run a container. This also helps for improved resource utilisation, or “bit packing” (something that Kubernetes does automatically across multiple hosts).
* Due to the layered structure of the storage, containers can share a large proportion of that with other instances of an image as read-only layers, which means less disk space usage.
* The lightweight nature of containers means that more copies of a program can run on the same hardware as opposed to a VM, which can lead to large savings with scale.

Of course, there are downsides as well:

* Isolation of a container is not perfect: vulnerabilities in the host operating system can sometimes be exploited by malicious containerised software. Having said that, modern versions of container runtimes, sane security practices, and careful configuration planning can largely mitigate these effects.
* They are unsuitable for cases where control of the lower levels of the operating system are required. For example, if you need a different kernel or an altogether different OS, your best bet is a VM.
* Because resources are shared with the host OS and other containers, a runaway process on the host, for example, can render most or all containers running on it unstable or unusable.

Here is a diagram of the logical layout of a host computer running containers:



There is a middle-of-the-road option which is becoming increasingly popular: unikernels. Those act as an additional layer sitting just above the operating system kernel which exposes only a subset of the system calls provided by it to running applications (those being read, write, exit\_group, clock\_gettime, ppoll, pwrite64, and pread64). It does this by way of mandatory seccomp (secure computing mode). Another feature is that once the process starts, it is only allowed to work with file descriptors opened prior to the enablement of the option. Unikernels can usually work with the same CRI interface that Kubernetes employs to talk to container runtimes. Some examples of unikernel systems are Kata containers, gVisor by Google, and Nabla Containers.

## Container runtimes

The layer which configures and runs containers on a host computer is called a container runtime. Kubernetes interfaces with an instance of that runtime on each host (or “node” in Kubernetes terminology) via its Container Runtime Interface (CRI). There are many container runtimes available, and which one you choose to use depends on your operating system and personal preference. The most popular one at the time of writing is Docker, however Kubernetes will soon drop support for that because it does not directly implement the Kubernetes CRI; instead, K8s relies on a shim for which the maintainers have chosen to drop support.

The container runtime is the program which interprets the user input, pulls and extracts the container image, allocates resources (kernel namespace, storage, cgroup) to a container, applies metadata and environment to it, and calls its entry point.

# What Kubernetes is and is not

As mentioned in the definitions section, Kubernetes is an orchestration platform: it plans and coordinates the elements of a system to achieve a desired state. That is also expressed in the way that state is configured: it allows its users to define the configuration declaratively, meaning that it expresses what the result of it should be without specifying the exact procedure to achieve it. As such, it abstracts away a lot of the leg work usually required to achieve the same result manually, allowing developers to take full control of an application’s lifecycle without needing decades of experience administering systems, networks and storage. This is useful not only to allow the developer to understand the implications of running the application they are authoring in a real-world context but also to speed up development times (no more waiting for the sysadmin to begin work on your ticket for a new VM) and expand the scale with which one can operate (run a database, message queue and a search engine alongside you application with a single command). It is difficult to overstate the impact of the plethora of features which Kubernetes provides.

It is important to understand where the responsibilities of the orchestration platform end and those of an administrator begin. Kubernetes does not manage the host computer itself beyond allocating its resources. System updates (including those to Kubernetes itself), low-level hardware problems, OS and container runtime bugs or issues, attaching and removing hardware, and managing system storage are all things still firmly in the domain of a system administrator. Due to the complexity of modern computer networks, it can sometimes be difficult to determine why you are observing problems with them; for that, sometimes you need the involvement of your network administrator (although keep in mind that Kubernetes’ own CNI and the associated plugins are plenty complex themselves and can be the source of your problems, too).

## Downsides to using Kubernetes

With the sheer number of features Kubernetes provides comes complexity. In smaller environments and especially during development that can become stifling to a developer’s progress. If you are writing a small application which consists of a couple of services, you may elect to use a simpler alternative like Docker Compose. If it starts growing but you want to retain the deployment configuration, and you don’t require fancy features like automatic service discovery, you may be better off using Docker Swarm. When developing complicated systems, your main task becomes managing complexity above everything else, so make sure you pick the right tool for the job.

Porting applications to Kubernetes can also consume a considerable amount of time. Applications whose design is such that they expect stable network identifiers may require additional configuration to adapt to the scalable and impermanent nature of most Kubernetes objects. In such cases one can use external services to bridge ported parts of the applications to un-ported ones which reside outside of the cluster, whilst retaining the naming and access patterns.

# How Kubernetes works

True to its paradigm, Kubernetes consists of several loosely coupled and collaborating layers. They can be separated into three groups: the control plane, node components and auxiliary processes.

The control plane is the central nervous system of Kubernetes. It receives and validates user input, stores cluster state, and directs the state of every other part of the system. It consists of the following pieces:

* kube-apiserver – the API server is the Kubernetes front-end. It does relatively little: it receives and interprets user input, checks RBAC rules, potentially changes payloads according to mutating webhooks, stores changes into the data store and alerts other components of those changes. It is essentially a CRUD application, mainly storing and serving data.
* etcd – this is a strongly consistent, distributed key-value store which holds all cluster information in Kubernetes. Its key features also include consistent performance and change notifications. It is the only stateful part of the Kubernetes control plane.
* kube-scheduler – this component watches for newly created Pods and schedules them onto nodes. It considers resource requirements and availability, affinity, hardware and software constraints, data locality, among other criteria determining where a Pod can and should run.
* kube-controller-manager – runs control loop processes for various object types that Kubernetes supports, continuously directing cluster state towards the desired outcome.

The control plane is, by default, isolated on a node that does not run other workloads (although that can be disabled). It can be configured to be highly available by running multiple copies of its components on several machines.

Cluster member machines, onto which user workloads are assigned, are called nodes. To facilitate control over those, Kubernetes uses a few other services:

* kubelet – this daemon run containers for Pods according to their specification. It acts as the node-bound link between the control plane and each node’s container runtime.
* kube-proxy – this implements the Kubernetes service concept by controlling network rules on nodes. It will either use the packet filtering mechanisms provided by the OS, if those are available, or forward the traffic itself.
* Container runtime – as discussed previously, this runs the actual containers on each node.

Auxiliary processes are components which provide cluster-wide services by using Kubernetes resources (Deployments, DaemonSets, etc). All of them are optional and must be installed separately from Kubernetes. They include:

* DNS – the usual source of any network problem. Jokes aside, this component provides in-cluster resolution and is extremely important to the service concept implemented in Kubernetes. It is included in the default installation.
* Web UI – if you prefer to point and click, this is for you. It must be installed separately.
* Monitoring – there are several different implementations for this functionality, the most popular by far being Prometheus. It requires some co-operation by the workloads (like exposing metric counters and gauges) for its full functionality. Must be installed separately.
* Cluster-level logging – facilitates centralised log collection. Those are usually implemented as sidecar containers running alongside your application which forward log messages to a central location running software like fluentd.

Anything outside of those three loose categories is considered user workloads. Those themselves can comprise anything from a single Pod containing a single container to fleets of Deployments, StatefulSets, ReplicaSets and so on, working in synchrony to provide a complex service or solve a business problem.

It is important to familiarise oneself with the terminology Kubernetes uses. Terms describing each aspect are used consistently throughout the documentation. They are treated as proper nouns (they are TitleCased) to potentially distinguish them from a more common usage of the term. There are quite a few to list all of them here; a full list can be found in the Kubernetes Standardised Glossary. A few of the more common ones are described below.

## Pods

This is the smallest, simplest Kubernetes object. It represents a set of running containers. They are commonly set up to run a single container, however they can also implement sidecar containers to ship logs or monitor, and init containers to prepare some facet of the environment before start-up.

Please note that a Pod is different from a container. Not only can it contain multiple containers, but it has additional metadata attached, like its spec (or desired state). All containers running in a Pod share a lot of its resources.

## Workload objects

Pods do not usually exist in isolation and are controlled by Deployments, StatefulSets or some other higher-level object.

### Deployments and ReplicaSets

Deployments are a Kubernetes API object which controls a replicated service, most commonly by running Pods. They have no local state. Each replica is typically a Pod, distributed among nodes in the cluster.

ReplicaSets are used by Deployments (among others) to ensure the required number of Pods are running in the cluster.

### StatefulSets

Like Deployments, StatefulSets manage a set of Pods based on a single container spec. Unlike a Deployment, they provide uniqueness and ordering of those Pods. That means that each Pod has a sticky identity which is not interchangeable: each has a persistent identifier which it maintains during (re-)scheduling.

StatefulSets are useful when parts of the application require storage volumes to keep state. Pods retain that via the persistent identifiers they receive across restarts.

Please note that using state can have wider implications, like data locality, which may limit the flexibility of the application scheduling. Also note that the requirement for ordering means that a rolling restart of the whole StatefulSet is done serially (one after the other) rather than in a parallel fashion, as is with Deployments. Therefore, take careful consideration whether a certain application strictly requires the keeping of state and whether it can be served by a Deployment instead.

### DaemonSets

This is a special type of workload which runs a copy of a service once per cluster node. These are mainly used for system daemons, like log or metric collectors, although they can be used for message queues or other services which match this requirement.

### Jobs and CronJobs

These objects are used for finite tasks which run to completion. They spawn one or more Pods which are expected to exit once they are finished with their task. CronJobs are such tasks which run on a schedule.

## Networking

Networking in Kubernetes can be divided into four categories:

* Tightly coupled container-to-container communication: this the case where multiple containers are running inside a single Pod. In this case, they can communicate between themselves on the loopback interface. They share the Pod’s IP address and MAC address.
* Pod-to-Pod communication.
* Pod-to-Service communication (see Services below).
* External-to-Service communication (see Services below).

Kubernetes’ approach to networking aims to alleviate a lot of the problems of running a (sometimes large) number of containers in a common environment, chief of which is port allocation. To achieve that, each Pod gets its own IP address, much like a VM. With this approach a user does not need to worry about cluster-level issues like port conflicts.

IP addresses exist in the Pod scope; this is called the IP-per-Pod model.

There are a few rules to the network implementation (barring any intentional measures):

* Pods on a node can communicate with all other Pods on all other nodes without NAT.
* Host daemons and processes (including the kubelet) can communicate with all Pods on that node.
* On host operating systems which support this (e.g., Linux), Pods which use the host network of a node can communicate with all Pods on all nodes without NAT.

There are many implementations of the Kubernetes networking model. Which one you choose is determined by the container runtime in use, other environment (cloud vs self-managed), desired functionality and personal preference. Those implementations are plugged into Kubernetes through its Container Networking Interface (CNI). Refer to the Kubernetes documentation for more information.

### Services

Services are an abstract Kubernetes object used to expose an application running on a set of Pods as a single entity, commonly called micro-service. Since the nature of Pods is to be (sometimes very) impermanent, the list of network destinations for a certain application can change very rapidly from one moment to the next. That means that any list of such destinations may be invalid the moment it is obtained. A useful solution to this problem is to define a Service with a Pod selector (typically based on Pod labels). Every Pod which fulfils the selector’s criteria is then registered as an Endpoint for that service (back-end). Front-end users of the service can then use the Service’s name to connect to an instance of the back-end application. By default, Kubernetes will load-balance requests to the Service across all Endpoints; session affinity is also supported but requires additional configuration.

A special type of Service is the ExternalName one. Those do not use selectors; instead, they have defined endpoints outside of the cluster (either FQDNs or IP addresses). Those are useful when using external resources (e.g., databases or APIs) and want to retain naming consistency or when transitioning applications into Kubernetes (part of the application is in-cluster, another is out-of-cluster).

### Ingress

Typically, end-users will need to access and interact with the application somehow. This is usually achieved by using an Ingress. That is an API object which manages external access to a service running in a cluster, typically via HTTP. Ingresses can also terminate SSL, load balance and provide name-based virtual hosting. Ingresses usually target Services.

The functionality of an Ingress is provided by an ingress controller. Unlike most other controllers, that is not included in the kube-controller-manager and must be installed separately. There is a whole host of ingress controllers, including ones based on NGINX, HAProxy, Istio, among many others, each of which come with their own feature set and documentation.

## Storage

Data written on a container’s file system is typically ephemeral – it only lasts the lifetime of the container. If that crashes or otherwise exits, all changes and files written are lost. If an application must keep data across restarts of its Pods, it uses volumes.

Kubernetes provides two elements which fulfil this requirement:

* PersistentVolume – this is a piece of storage connected to the cluster and is typically provisioned by an administrator or dynamically using Storage Classes. Think of it as the “server-side” part of the equation.
* PersistentVolumeClaim – this is a request for storage by a user. It either binds to an existing available PersistentVolume or causes one to be dynamically allocated (if configured). This is the “client-side” of the equation: if Pods consume node resources, like memory and CPU, PVCs consume PV resources.

There are many types of PersistentVolumes: NFS, local, iSCSI, and so on. Many of them are provided by external plugins, and the list is ever-growing. For a full list refer to the Kubernetes documentation.

# Setting up Kubernetes clusters

It is not difficult to install Kubernetes. All you need is a computer which can run the container runtime and enough resources (CPU, memory, disk space, and perhaps other resources if your workload requires them, like GPUs). Due to its modular nature, you don’t have to have an exact replica of the production environment in order to replicate the exact conditions under which your workload will execute for its end users. This is extremely useful during development, as well.

## Node preparation

Because Kubernetes’ design calls for tightly packing workloads across nodes according to their resources, one of the long-standing requirements for installing it is disabling the system swap space. According to the documentation, this allows for better predictability of the performance each node can supply: a workload partly or wholly swapped out will perform worse than one that is in system memory. There is some discussion whether this is a good approach, and some workloads benefit from using the system swap, but this discussion is outside of scope of this document. If your resource constraints permit it, disable the system swap before installing the Kubernetes tools. Otherwise, use the appropriate flags to circumvent this limitation. Recent versions of Kubernetes (>=1.22) have acknowledged this and include more support for running on swapping nodes.

Another thing to note is that clusters must have stable identities. Those are based on IP addresses (the server certificates are bound to them at installation time). You can probably see the problem already: multihomed hosts or ones whose address changes regularly (e.g., laptops and other DHCP clients) will emit errors if the IP address onto which the cluster’s certificates are bound changes, or if they are called on a network interface different from the one picked at install time. There are a few methods to work around this: bind to virtual interfaces with predictable and stable addresses or circumvent these identity checks altogether (aka, the nuclear option) all of which are beyond the scope of this document.

It is important to ensure that nodes in a cluster can communicate with each other unimpeded. Firewalling can make this difficult, especially if you are running additional daemons which require obscure ports to coalesce. For example, MetalLB requires TCP/UDP ports 7946 for this.

Link quality can also be a problem. It is not recommended to run clusters across WAN links with unpredictable quality: those whose latency, jitter and loss can change dramatically over short periods of time.

## Installing Kubernetes

The specific steps to installing Kubernetes are beyond the scope of this document. Please consult the documentation to consider all variables of your environment.

An example is included which uses Vagrant to bring up a cluster in a set of virtual machines. It uses Ansible to provision the master and nodes. Please consult the playbook for the exact method of installation in this case.

All examples from this point onward imply that they are run against this Vagrant/Ansible installation, however they are universal to any full-fledged Kubernetes installation. Other types of installations (e.g., MicroK8s) will differ in some ways.

### Overview of the cluster state

Once installed, you can interrogate the Kubernetes cluster with kubectl. A few examples follow.

#### Node status

$ kubectl get nodes

NAME STATUS ROLES AGE VERSION

k8s-master Ready control-plane,master 22h v1.21.3

node-1 Ready <none> 22h v1.21.3

node-2 Ready <none> 22h v1.21.3

#### Pod status

$ kubectl get pods --all-namespaces

NAMESPACE NAME READY STATUS RESTARTS AGE

ingress-nginx ingress-nginx-controller-5b45b668f5-2pq7q 1/1 Running 0 22h

kube-system calico-kube-controllers-58497c65d5-jspj7 1/1 Running 0 22h

kube-system calico-node-fgfg7 1/1 Running 0 22h

kube-system calico-node-k8lcb 1/1 Running 0 22h

kube-system calico-node-t2qwq 1/1 Running 0 22h

kube-system coredns-558bd4d5db-fw79t 1/1 Running 0 22h

kube-system coredns-558bd4d5db-t6l48 1/1 Running 0 22h

kube-system etcd-k8s-master 1/1 Running 0 22h

kube-system kube-apiserver-k8s-master 1/1 Running 0 22h

kube-system kube-controller-manager-k8s-master 1/1 Running 3 22h

kube-system kube-proxy-j9p8h 1/1 Running 0 22h

kube-system kube-proxy-mlkhr 1/1 Running 0 22h

kube-system kube-proxy-pbd8l 1/1 Running 0 22h

kube-system kube-scheduler-k8s-master 1/1 Running 3 22h

#### Deployment status

$ kubectl get deployments --all-namespaces

NAMESPACE NAME READY UP-TO-DATE AVAILABLE AGE

ingress-nginx ingress-nginx-controller 1/1 1 1 22h

kube-system calico-kube-controllers 1/1 1 1 22h

kube-system coredns 2/2 2 2 22h

## Configuring client access

Every user of the Kubernetes API, human or otherwise, must authenticate to the Kubernetes API before receiving any responses from it. For people this is usually done with kubeconfig files. Those are YAML files which include the server and client identities in the form of base64-encoded payloads, as well as the address of the API for the cluster and any additional configuration (e.g., default namespace).

The default location of the kubeconfig is $HOME/.kube/config (in \*NIX-land) or %USERPROFILE%\.kube\config (on Windows). Other than that, you can specify that in the environment as KUBECONFIG:

$ KUBECONFIG=path/to/kubeconfig kubectl get cm -n ingress-nginx

NAME DATA AGE

ingress-controller-leader-nginx 0 22h

ingress-nginx-controller 0 22h

kube-root-ca.crt 1 22h

### Managing multiple clusters

The kubeconfig is structured as to allow using multiple contexts. If you find yourself managing or otherwise needing to access multiple clusters, you can include the information for each in one file (according to YAML’s formatting rules) and then switch around contexts with kubectl:

$ kubectl config get-contexts

CURRENT NAME CLUSTER AUTHINFO NAMESPACE

anton-laptop anton-laptop anton-laptop-admin nolan

anton-thinkpad anton-thinkpad anton-thinkpad pine

\* live live live-admin pratt

mini mini mini-admin walken

monster monster monster-admin hemsworth

physical physical physical-admin evans

$ kubectl config use-context physical

Switched to context "physical".

# Using Kubernetes Clusters

There are several methods of using clusters:

* The kubectl command. This is the standard tool that is distributed alongside the server components. It is available for most (all?) platforms and supports all functionality which the respective server components using the same version expose. It is highly recommended that you familiarise yourself with this tool.
* The Kubernetes dashboard. This is a web application which can be installed inside the cluster. It is an optional component. It is supported by the same team which authors Kubernetes.
* Lens and other GUI tools. Lens is the client-side equivalent of the Kubernetes dashboard. It uses the same kubeconfig as kubectl.
* Various IDE integrations. For example, JetBrains IDEs have a Services tab which can show Kubernetes workloads. Those use the standard kubeconfig configuration.

This document will use kubectl throughout, although most of the examples here can be achieved with other tools as well.

## Overview of a basic application

The standard way of configuring workloads for Kubernetes deployment is via YAML files. However, the API converts that to JSON behind the scenes, so this format is supported too, naturally.

Because the average application can comprise many different objects (e.g., a Deployment, a Service, and an Ingress), the number of templates can grow, sometimes substantially. In order to maintain a serviceable deployment, it is common to use templating tools like Helm which can generate the configuration from templates and on the fly. It can also be used for versioning, centralised template storage, among other features. It is recommended to use such a tool for anything beyond a simple single-Pod scenario. Helm is introduced in more detail later because it is important to understand the underlying YAML syntax and idiosyncrasies before attempting to template them.

This example describes a Deployment with two Pods, each of which containing a single container running NGINX. It is lifted verbatim from the Kubernetes documentation with the addition of a liveness and readiness probes, so I won’t go into too much detail, apart from explaining some key elements. The YAML goes as follows.

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

selector:

matchLabels:

app: nginx

replicas: 2 # tells deployment to run 2 pods matching the template

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

livenessProbe:

httpGet:

path: /

port: 80

readinessProbe:

httpGet:

path: /

port: 80

Every object must refer to the API version which supports it. The naming reflects whether the object is in alpha, beta or GA state. In the above example, the object Deployment is part of the apps API group, and is in GA, or stable state (i.e., the version of the API is v1). Other potential states can be alpha, or very early feature release candidates, and beta, which denotes a slightly more mature feature which has seen some testing, although it may still differ from its final, stable form. It is important to note that, if you use non-stable APIs, there will certainly come a time where you will need to re-work the templates as the feature in question matures through these stages. Messages emitted by the API, and in turn from kubectl (and, finally, by extension, from templating tools like Helm) will alert you as that is about to happen. Pay close attention to those messages and test your applications regularly against potential new versions of Kubernetes which may be installed in your environment.

Object also have certain metadata attached to them which comprises labels and annotations. Labels are used for targeting. For example, a Service will pick its backends by using the labels of the Pods which run it. Versions and other data are also included in the labels section. In this example, the Deployment matches all Pods which have a label named “app” which has a value of “nginx”. That’s how it knows which Pods belong to it.

Likewise, annotations also can include such metadata, however it is usually used to denote non-identifying data to objects. The form annotations can take is more liberal than labels: it can be small or large, structured or unstructured, and include characters not permitted in labels.

The rest of the example is specific to Deployment objects. For more about that please refer to the Kubernetes API documentation.

### Monitoring state

Applying this YAML to the cluster will create several objects: a Deployment, a ReplicaSet, and two Pods. The root of this hierarchy is the Deployment. It has the name specified in its metadata. The other objects will also inherit this but also add additional random ASCII characters and digits to it. An example of the ReplicaSet name is “nginx-deployment-66b6c48dd5”. A Pod which is a member of this deployment can be “nginx-deployment-66b6c48dd5-5lvqh”. Those will be generated at runtime and cannot be predicted because a Deployment is a state-less type of workload. Therefore, if you would like to fetch a Pod’s state, you must first find its name. This can be achieved with the same criteria the Deployment uses, a label:

$ kubectl get pods -l app=nginx

NAME READY STATUS RESTARTS AGE

nginx-deployment-66b6c48dd5-5lvqh 1/1 Running 0 30m

nginx-deployment-66b6c48dd5-dcgsj 1/1 Running 0 30m

This list tells you that there are two Pods matching this description, each of which has one container defined inside them (the second number), with one container running (the first number). Both have started successfully (state is Running), have not had any restarts, and were defined 30 minutes ago. Please note that the time describes when the Pod was defined in Kubernetes by the ReplicaSet and is not the container start-up time.

A lot more information can be obtained by “describing” the Pod; please see the output below. Please note that I killed the NGINX process manually to make it more interesting.

$ kubectl describe pod nginx-deployment-679c5d56bf-8gffv

Name: nginx-deployment-679c5d56bf-8gffv

Namespace: default

Priority: 0

Node: node-1/192.168.50.11

Start Time: Sun, 08 Aug 2021 20:26:33 +0400

Labels: app=nginx

pod-template-hash=679c5d56bf

Annotations: cni.projectcalico.org/containerID: 003697b99dfa64b498f50c3b84bbddaeb3ae100a6f8b9ca98f328737568f78a0

cni.projectcalico.org/podIP: 192.168.84.132/32

cni.projectcalico.org/podIPs: 192.168.84.132/32

Status: Running

IP: 192.168.84.132

IPs:

IP: 192.168.84.132

Controlled By: ReplicaSet/nginx-deployment-679c5d56bf

Containers:

nginx:

Container ID: containerd://18b4aa85b8381db0b1264eea10e56c10c278e7eafad00afcc01e701633d57e37

Image: nginx:1.14.2

Image ID: docker.io/library/nginx@sha256:f7988fb6c02e0ce69257d9bd9cf37ae20a60f1df7563c3a2a6abe24160306b8d

Port: 80/TCP

Host Port: 0/TCP

State: Running

Started: Sun, 08 Aug 2021 20:28:00 +0400

Last State: Terminated

Reason: Completed

Exit Code: 0

Started: Sun, 08 Aug 2021 20:26:34 +0400

Finished: Sun, 08 Aug 2021 20:28:00 +0400

Ready: True

Restart Count: 1

Liveness: http-get http://:80/ delay=0s timeout=1s period=10s #success=1 #failure=3

Readiness: http-get http://:80/ delay=0s timeout=1s period=10s #success=1 #failure=3

Environment: <none>

Mounts:

/var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-kqxkc (ro)

Conditions:

Type Status

Initialized True

Ready True

ContainersReady True

PodScheduled True

Volumes:

kube-api-access-kqxkc:

Type: Projected (a volume that contains injected data from multiple sources)

TokenExpirationSeconds: 3607

ConfigMapName: kube-root-ca.crt

ConfigMapOptional: <nil>

DownwardAPI: true

QoS Class: BestEffort

Node-Selectors: <none>

Tolerations: node.kubernetes.io/not-ready:NoExecute op=Exists for 300s

node.kubernetes.io/unreachable:NoExecute op=Exists for 300s

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal Scheduled 99s default-scheduler Successfully assigned default/nginx-deployment-679c5d56bf-8gffv to node-1

Normal Pulled 12s (x2 over 98s) kubelet Container image "nginx:1.14.2" already present on machine

Normal Created 12s (x2 over 98s) kubelet Created container nginx

Normal Started 12s (x2 over 98s) kubelet Started container nginx

Besides all the obvious information, some key elements are:

* Under containers, nginx, Last State you will see why the container has restarted most recently, alongside the most recent exit code. As a developer, it can be useful to employ meaningful exit codes which can tell you at a glance what kind of problem occurred. Anything different from zero is usually considered an error in \*NIX-land.
* The list of Events can contain recent happenings with the Pod, along with the reason why a Pod won’t start. Although the messages can seem cryptic sometimes, always read them if they are listed. Doing that will a) make you accustomed to them, and b) will reveal key details about errors. Please note that this section is not permanent and old information is replaced and expires regularly. For long-running Pods it will be empty, assuming they have encountered no recent errors.
* The Liveness and Readiness lines will tell you about the method with which Kubernetes checks if the Pod is operating correctly.

### Liveness, start-up, and readiness probes

It is highly recommended to define meaningful ways of keeping track of application state. This is key to Kubernetes’ self-healing mechanism. It is the responsibility of the developer to determine what method is best for this. If there are failures of these probes, there will be accompanying information in the Events section (if that has happened recently). From Kubernetes’ viewpoint there are three different methods to determine the state: HTTP request, TCP socket connect or a command which is executed inside the container.

## Stateful workloads

It is practically inevitable that a Kubernetes user will have to deal with workloads which keep state. Those can include databases and their replicas, higher-level file storage APIs (e.g., MinIO), search engines, and generally any workloads which depend on state which is kept on disk or employ the notion of service identity. The Kubernetes object used in such cases is called StatefulSet. Instances of a StatefulSet enjoy the following features:

* Stable and unique network identifiers. The names of the Pods will be predictable, and you can use those to address members of the StatefulSet (in conjunction with DNS; the IP addresses can and will change).
* Stable, persistent storage. Pods in a StatefulSet will receive the same storage volume according to their ordinals on every start-up. Also, volumes are not automatically deleted with the removal of the StatefulSet to ensure data safety.
* Ordered and graceful deployment and scaling. Instances will be brought up or down in order, whilst waiting for each instance to become ready before starting the next (in the case of scaling up; see the documentation on probes on how service state should be determined). It is important to note that if a service takes a long time to reach such a state, that will cause operations like rolling restarts or upgrades to also take a while.
* Ordered, automated rolling updates.

The main consideration that must be taken with stateful services is the increased context each instance and the service as a whole drags with itself.

For example, when using storage, the Pods become practically bound to the volume. That can have implications on where the Pod can start due to data locality – some data may exist only on certain cluster nodes but not on others. In case the nodes which hold the data do not have sufficient resources to run the Pods, that can spell problems starting the stateful service.

That is only a single example, but you can see how easily the expanded size of the context can have far-reaching implications. Carefully consider whether your service requires stable identifiers and ordered deployment and deletion. If it doesn’t, use a Deployment or ReplicaSet instead.

### Obtaining and using storage

There are various methods of storing on-disk data in Kubernetes. Which one you choose depends heavily on your workload and environment.

Storage can be obtained by either pre-provisioning it and then adding it to Kubernetes (e.g., create an NFS share, then a PersistentVolume, and finally a claim inside a Pod template), or automatically provisioning it.

Automatically provisioned volumes are only supported for StatefulSets via their volume claim templates. First, a StorageClass must be defined. That holds all information describing the backend storage (e.g., for NFS that would be the server address and path). Then, in the volume claim template section of the StatefulSet you refer to this StorageClass with some additional information (typically the access more – read/write or read-only – and the required capacity). Once deployed, PVs and PVCs are created automatically.

Storage can be used with stateless workloads as well; it just cannot be automatically provisioned with volume claim templates. For example, you can define your own PersistentVolume and the requisite claim to connect to an NFS server and access the data there.

To access some types of storage you will need additional software and configuration. Please consult the Kubernetes documentation for more details.

## Services

When designing and using microservices, especially in an environment where instances of those can pop into existence just as fast as they disappear, one must contend with the problem of service discovery – how can service A reach service B? Kubernetes’ solution to this is Services: an abstraction for a group of Pods and a policy by which to access them.

Services typically define a selector with which to find matching Pods (although that is not universal; see the documentation on Services without selectors). For each Pod which matches the selector, an Endpoint is created which contains the Pod’s IP address along with some metadata. That in combination with the policy (the protocol and port) comprises all back-ends for a given service. Then, if a user would like to connect to a service, they use its DNS name.

### Load balancers

Services also provide basic load balancing. The algorithm used is round-robin, with some additional footnotes like configurable session stickiness, avoidance of busy back-ends, and rules for already established connections. For more advanced methods of load balancing (e.g., least-connection, destination/source hashing, delay-based) please consult the documentation on Services.

### Headless services

To avoid load balancing one can use a so-called “headless service”: one that does not have an IP of its own but instead returns the address of the backend Pods (for services that define a selector).

In the case of StatefulSets you must create a headless service which groups the network identity of the pods. It does that by defining their DNS names in the form of <pod-name>.<service-name>.<namespace>.svc.cluster.local (assuming that the cluster domain is cluster.local).

## Secrets and ConfigMaps

Best practice dictates that container images are built for a generic case and then configured to match the appropriate environment by passing configuration data during runtime. A developer’s life is greatly simplified by supporting a single image that they can deploy during testing, staging, user acceptance and production. However, each of these scenarios will likely require different addresses and credentials for a database, for example. It is also a terrible mistake to include passwords, tokens or other confidential information inside images. To help with this Kubernetes provides two types of objects: ConfigMaps and Secrets.

ConfigMaps store non-confidential data in the form of keys and values. These can then be referred to by environment variables or mounted as a tree inside the file system of a container.

Secrets are very similar in their implementation and use but they are specifically intended for storing small amounts of sensitive data like passwords, private keys and the like. Please note that, by default, Secrets are stored unencrypted in the Kubernetes data store (etcd) and anyone with API access can read or write them. To provide true protection you must use role-based access control (RBAC, discussed below) rules and perhaps enable encryption at rest for the cluster. Please consult the documentation for more information.

## RBAC

To control access to resources (authorisation) by users of its API Kubernetes uses role-based access controls: roles are defined with their permissions, then role bindings are used to link them to principals (users, groups or service accounts).

Roles exist in two varieties: Roles and ClusterRoles. A Role is simply a set of permissions ANDed together (there is no “deny” permission). Those are defined with their namespace, and all the included rules apply only to this namespace.

A ClusterRole is much the same with one important difference: it applies to the whole cluster and all its namespaces. Use those sparingly, if ever.

To apply these permissions to a certain principal, you define a RoleBinding or ClusterRoleBinding, respectively for each type of role mentioned above.

Users in Kubernetes can take many forms: certificates, tokens, externally defined, among others. The documentation describes this in much more detail.

### Service accounts

Sometimes services running inside the cluster will need to interrogate or make modifications to Kubernetes API objects. In such cases one can employ service accounts: a special type of user in Kubernetes which can be used to obtain an identity from inside a Pod. Those accounts are created and removed automatically with the workload which defines them. They use the same method of binding to roles as normal users (see above).

## Using Helm

As you can see, there are many different types of objects in Kubernetes. This is not even an exhaustive list, just the basics. Now imagine that you need to deploy a solution which comprises a dozen microservices along with their volumes, configuration, roles and bindings, and everything in between. If using flat YAMLs that may mean many dozens of files, all of which must refer to each other correctly and have coherent configuration (is this ingress targeting the correct port in the Pod template?). Complexity explodes.

A popular solution to this problem is Helm: a templating tool for Kubernetes with support for versioning and publishing its so-called “charts”, upgrades and a basic interpreted language.

Helm organises all variable definitions for a chart in a “values” file. This holds all user-supplied information, like port numbers, replica counts, start-up arguments and everything else. This values file is then applied on several templates to replace placeholders for the data and arrive at a set of valid YAML definitions for everything a microservice requires to deploy correctly.

These charts can be published and shared from a central location, making distribution simple. Many projects already provide official Helm charts, and there are many such created by third parties.

A basic starter chart can be generated with a single Helm command, which can serve as a basis for your own. It is good practice to keep the chart inside the source code of your service. In doing so you have both the application code and its deployment semantics in one place. This is also a requirement by many CI/CD automation systems.

# Administering Kubernetes clusters and workloads

The task of administering Kubernetes clusters is sometimes precarious, however it is becoming more structured as the project matures. In the early days it wasn’t difficult to sometimes draw yourself in a corner during upgrades, with the only exit being full reset and reinstall of the cluster. This means disruption to workloads and annoyed developers and users. A lot of water has gone under the proverbial bridge since then, thankfully.

With such a lively project, it is good practice to keep up with versions because developers like to use bleeding edge and, let’s be honest, there are some rough edges that are constantly being worked out. Being stuck with an old version means you may have to deal with problems which have already been solved upstream. Another reason is that upgrading from 2+ versions back needs to be done in stages because of built-in limitations to the upgrade procedure.

The flipside to this is that with each new version come new deprecations and graduation of features, which is more than likely to require developer involvement to correct. The administrator and the cluster users need to co-ordinate before every upgrade to ensure smooth operation for the end users. As an administrator, it is useful to provide a “staging” or some other non-production cluster which is upgraded first (with appropriate notice), then let the developers work out the details.

## Node maintenance

Computers need constant babysitting to continue working as intended. Disk space runs out, hardware expires, dust cakes on the components, oxidation on the contacts, DNS problems, even sometimes the alignment of the stars seems to influence events. Add to that upgrades, and additions and removals of machines, and the administrators have their work cut out for them.

Thankfully, Kubernetes’ self-healing mechanism reduces the impact of such disruptions to the underlying infrastructure. As nodes come and go, workloads can adjust, sometimes even automatically, although that takes careful planning and configuration from the side of developers as well. With a properly configured Pod disruption budget, services can remain available during upheaval of the cluster member nodes.

If a node must go out of service while operational, the administrator needs to “drain” it of its workloads. This will cause all services which can leave to move over to the other remaining nodes. Some will not have this opportunity: DaemonSets, for example, cannot due to their design (one instance per node), some because of extraneous factors, like data locality (a service which depends on some files which reside solely on the node being taken out cannot function elsewhere) or (anti-)affinity. Therefore, it is important for the developers to also understand the points being described here.

Please refer to the section on cluster administration in the Kubernetes documentation for more information. I cannot hope to cover even one per cent of its information here: storage, DNS, data encryption, CPU management policies and a lot more.

## Cluster upgrades

As previously mentioned, it is good practice to “keep up with the Joneses” regarding Kubernetes versions. There are three major steps to perform an upgrade:

1. Upgrade the control plane, aka the master node, or nodes if you are using a highly available setup. This is usually done with the kubeadm command in two stages: plan and execute.
2. Upgrade the cluster nodes. The steps are identical as with the previous stage. The safest approach is to run the upgrade sequentially while waiting for each node to settle and operations to resume on it.
3. Upgrade the clients (kubectl). This is the easy part because it will not cause any outages.

You may also need to poke around the manifests in the /etc/kubernetes directory in case you have some eccentricities to your configuration (e.g., extra feature flags).

Be careful with regular operating system updates! If one is done to the container runtime this will likely mean a restart to it which, in turn, means a restart to all containerised workloads.

Consider pinning the Kubernetes and container runtime versions in your package manager to exclude them from automatic all-package upgrades. In fact, that is exactly what the documentation lists as one of the installation and upgrade steps. Then you would un-pin them before you intend to upgrade.

## Monitoring

This section can be a novel by itself. The task can be most simply split into two sections: monitoring the cluster and monitoring the workloads.

Monitoring of the cluster is somewhat more straightforward because the components involved (the OS and the hardware) usually provide the necessary sensors, gauges and counters to construct graphs and alarms. Most monitoring software, of which the most popular by a country mile is Prometheus, comes ready with data collection for some of the headline numbers: CPU and memory utilisation, network bandwidth, I/O, uptime and a few others. Some extra work may be involved in constructing alarms from those and adding other, more specific measurements, like temperature sensors.

Monitoring the workloads is a different kettle of fish. Because there can be a large variety of technologies being employed which can also come and go regularly, to have a meaningful view of events can mean a lot of work. Consider an application which uses a database, message queue, search engine, and has a web front-end. Those already imply dozens of measurements that must be taken: how quickly does the database execute queries, how many reads and writes happen there? How many topics are there on the message queue and how many messages are waiting in each? Mean response time of the search engine? Time to first byte on the web server? And those are just scratching the surface. Now add building graphs and alarms to merely collecting the numbers and you should have an idea of what a Sisyphean task this can sometimes be.

Keep in mind that Kubernetes workloads have unpredictable IP addresses and sometimes names. It is a good idea to use monitoring software built specifically with autoscaling or “cloud” environments in mind, such that can discover and monitor containers automatically.

Developers have a major role to play. Bespoke software should integrate counters and gauges as an integral part of its function, placed in key positions in the application logic. This will help diagnose performance problems during runtime. Outside of that, it is best for them to be involved in the metric collection of all services their software uses, as well as picking the best numbers to present on a graph or alarm on.

# Troubleshooting

Anyone using Kubernetes for anything more serious than listing existing objects will inevitably encounter problems. If you ask ten people you will get ten methods for troubleshooting, but whatever the method, it will always boil down to:

1. Isolate: find where the problem occurs.
2. Replicate and investigate: find why the problem occurs.
3. Fix: address the underlying issue.

More often than not, the operator of a Kubernetes workload will deal with the exact same instance that a potential end user will. If that is the case, then step two would simply be “investigate”. That makes the situation a lot easier to address because replicating problems is sometimes extremely difficult because re-enacting the same environment might be nigh impossible. Of course, this is not always the case: the deployment may be air-gapped, for example, and the only way to debug a problem may be to drive over and sit in front of it.

Whatever the case may be, remember the mantra: “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it”. It is therefore extremely important to adhere to the microservices architecture paradigm, more specifically that each component should do one thing, and one thing only, and do it in the simplest way possible. If you do this your task of troubleshooting it will be greatly simplified because once you find where the problem occurs, you only must deal with a very small and simple chunk of code.

As with any complex system, problems may vary from straightforward to fix (a typo, say) to borderline opaque to their causes due to layers upon layers of abstraction being involved. Keep in mind that whatever you are seeing may very well be a side effect and not the cause itself and that treating symptoms is not nearly as efficient as treating the cause (as any doctor will tell you). You need to dig until you find where the bodies are buried.

With those general points out of the way, troubleshooting will take different forms depending on what part of the equation is not producing the result you were expecting.

## Logging

I thought a lot about where to place this point in the structure of this document because logging is something fundamental to the development of manageable applications but decided to put this in the Troubleshooting section, because this is where the pain will be.

I cannot stress enough how important it is to have a structured logging approach in your application. It will make the difference between glancing a message in a container, flipping a switch and then returning to drink your cocktail of choice on your day off, or fumbling around for hours trying to find out why a certain part of the application is not working. During development, it is easy to overlook the necessity of log messages. After all, you have the process executing right in front of your eyes and you can see its state in the debug section of your IDE. This is not how things will turn out in production. Remember Murphy’s law: anything that can happen, will. You run no risks of being too liberal with your log messages. If you spread them correctly through the levels (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL, FATAL), you can raise or lower the number of messages being printed out. My rule of thumb is, add logging until you think you have enough, then add some more.

You may have noticed I used the word “structured” above when referring to the approach. Besides using the correct levels, this also means the format of the messages printed must be predictable: field order, timestamps, field separators, multi-line separators, and so on. Not only that, but they also need to be consistent. This is to ensure that an automated logging collection mechanism can parse them with little configuration. Those are usually configured with regular expressions, so think of it this way: what is the smallest number of formats and simplest regex I can use to parse each and every message printed?

Use the standard library of your language for logging, if it has one. If it doesn’t, use existing established logging frameworks (Log4J, Log4Cxx, etc.). This will all but guarantee easier maintenance and configuration.

## Workloads

Provided that Kubernetes has accepted your deployment manifests, the first place to look for problems is where the action happens – Pods. I tend to keep a console open with the following command running whilst I investigate:

# This will work for Linux which has the `watch` command:

watch --interval=1 kubectl get pods -o wide --all-namespaces

# Kubernetes also provides a “watch” mechanism built into its API:

kubectl get pods -o wide --all-namespaces -w

The first method will give you a clear view of what Pods (in this case) are *currently* running; however, you will not see what has happened in the interim, only right at this moment (well, up to about one second in the past). Due to the fleeting nature of most objects in Kubernetes this may mean you are missing certain things even if you are staring at the output.

The second method will list everything that exists and then what happens: every state transition, every IP allocation. The downside here is that, to get a total picture of the situation right now, in your mind you need to accumulate all lines that have been printed since you started the command. Another problem here is that Kubernetes does not print when a workload disappears, only its last state before doing so (Terminating, say). It may be sitting there in that state, for all you know, until you restart the command.

The “watch” method works for all listable API objects, not just Pods.

Once you have an eye on things, it is time to isolate. First, find if the problem is in the application itself or Kubernetes. Pod statuses will tell you a lot of the story: Error and CrashLoopBackoff means the application is exiting and doing so with an inappropriate exit code. Describing the Pod will tell you the exit code (see the Last State section) and what Kubernetes sees on its side of the table but in these cases, it will be more illuminating to look at the container logs. Remember that, to look at the previously exited instance’s logs, you can use the --previous flag to kubectl logs (provided it didn’t happen too long ago; logs are not kept forever).

In contrast, some of the problems that you may see which lay on the platform side of things are:

* ErrImagePull and ImagePullBackOff – for some reason the container runtime cannot pull the container image. Describe the pod to get the reason (e.g., SSL certificate mismatch, authentication, no such tag).
* CreateContainerError – the container cannot start. Describing the Pod will give more information (e.g., entrypoint is missing).
* Pending – less of an error than a normal stage of the Pod lifecycle in normal circumstances but if this does not change relatively quickly into another stage, then you may have other problems. Again, describing the Pod will tell you of any errors. There may be insufficient resources to run the Pod, for example.

I am sure I am missing some scenarios here but those are the major ones. In any case, what I have found to work most of the time is finding where the problem exhibits itself and then describing the resources outwards (e.g., Pod, storage, PVC, PV, StorageClass, etc. in the case of volumes).

## Storage

Since storage in Kubernetes involves a few moving parts, investigating problems there means looking in a few places. Similar to the case with Pods, I like to start looking from the point of where the problem exhibits itself (the Pod description says that storage cannot be allocated) and then work out from there. Since we know that connecting a volume flows through a PersistentVolumeClaim and then to a PersistentVolume, those are the same steps to take when troubleshooting. List and then describe each in turn.

If you find that a PVC is stuck in the Pending state, then it is likely that either an auto-provisioner is either missing or misconfigured, or that there is no existing PV for the claim to “take”. In the former case, unless you are explicitly naming a StorageClass to use, then the PVC will use the default SC, and sometimes there is no class marked as default. Please refer to the documentation for information about how to mark a StorageClass as default or change the PVC definition to explicitly name an existing one.

Because the actual storage-providing backend is likely to reside outside of the cluster (a SAN, say) you may be faced with a problem which requires a storage administrator’s involvement. Another reason to keep all your administrators happy. Remember, last Friday of July is the international sysadmin appreciation day.

## Network

I have always dreaded dealing with network problems, but perhaps that’s just me. Just reading the CNI plugins’ documentation is daunting: tons of features, tunnels, encryption, BGP and whatnot. Again, it is prudent to isolate the problem as much as possible. This may mean executing a shell inside the container and using ping or netcat or curl or dig to determine what the application is seeing. These are not usually all included into an image; in such cases you may have to install them (if you have Internet or a local package mirror) or keep an image with all your debugging tools pre-installed sitting on the image registry.

Plan ahead and, once again, it is probably DNS.

## Nodes

Resources run out and disk space fills up, those are just the inevitable facts of life. For those, Kubernetes has “watermarks” for when to start worrying. Describing a node will tell you if you breached any of them. They are:

* MemoryPressure
* DiskPressure
* PIDPressure
* NetworkUnavailable (this appears in the Calico CNI plugin, it may not exist or be named differently with others).

Each of them has a different value for the watermark; please refer to the documentation. Whenever each of the “pressure” watermarks is breached, Kubernetes will take corrective action: start evicting resources or deleting ephemeral data. Ensure you keep a keen eye on those!

An important note on DiskPressure: as mentioned above, breaching this watermark may cause data to be deleted which is probably not what you want or can live with. Make sure you know what the watermark is (I believe it is 80% by default, although that may have changed) and set it according to your environment. Remember that, for very large disks, 20% may mean terabytes of unused space, so you may have to tweak this for the case at hand.

Another important point for disks filling up regarding Docker: it has its own watermark, and if that is breached it will start deleting unused container images (those which are not associated with a running container). This can lead to race conditions where, during deployment, images may continuously be deleted during start-up because a disk has become full according to Docker.

## Debugging with Telepresence

Sometimes even with logging and monitoring in place it is difficult to pinpoint the source of problems. This just goes to show that nobody can even predict every corner case that can pop up. For such times there is Telepresence: a tool which allows you to replace workloads in a functioning cluster with one that will tunnel all requests to and from it to your machine. This is extremely useful because it allows you to run the process in your IDE while using the same environment as the process exhibiting the problems, use breakpoints, as well as change code on the fly and restart the process. Please see the project website for more information: <https://www.telepresence.io>.

## Application and cluster backups

As applications run, they accumulate data. In today’s world of Big Data and data scientists, the data are rightfully becoming the point to any application (as opposed to the code). They say, “data is the new oil” (data is plural of datum, so it would be more accurate to say “data are the new oil” but that’s how the saying goes). As such, it is very important to have regular backups and adequate retention of the application data. You must also test your backups by actually restoring from them on a regular basis. If you don’t, you may have a false sense of security which will evaporate the moment you notice that your precious backup is of 40 bytes size or otherwise corrupted. It is good to practice drills in order to reduce the Mean Time to Recovery (MTTR).

There are a few tools which operate on the platform level which can help with backups, Velero. This will take a snapshot of the PVC. It makes life simpler but there are some major downsides: applications will have their files open and the results of the snapshots may be inconsistent. Another point is that some applications, like Elasticsearch, explicitly advise against copying the file system level data to produce backups but instead using the built-in snapshot mechanism. The bottom line is, read the documentation for your stateful component to determine what the best course of action may be whilst considering your risk tolerance.

As far as backing up Kubernetes itself goes, since the only stateful component in the equation is etcd, you only have that to worry about. Please refer to the Kubernetes documentation; the section called “Operating etcd clusters for Kubernetes” describes the steps needed to snapshot this component.

# Resources

Any project is only as good as its documentation: it may cure cancer but if that is achieved by some mystical means that nobody knows about, is that useful to anyone? If a tree falls in a forest… Luckily, Kubernetes’ documentation is great. It does require you to familiarise yourself with the terminology and how things fit together but that can be surmised by reading its introductory parts first. Hopefully I have managed to introduce some of the main ideas with this document.

Because of the (growing) complexity of the API, you will likely never memorise its every nook and cranny, and that is okay, if you know where to look when you need to recall some information. Here is a list of a few places (Stack Overflow notwithstanding):

* The main documentation entry point is <https://kubernetes.io/docs>.
* Kubernetes exposes the full version-specific API documentation via the API itself. This can be invoked this with kubectl (you need to have a connection to a functioning cluster):

$ kubectl explain pod.spec.tolerations

KIND: Pod

VERSION: v1

RESOURCE: tolerations <[]Object>

DESCRIPTION:

If specified, the pod's tolerations.

The pod this Toleration is attached to tolerates any taint that matches the

triple <key,value,effect> using the matching operator <operator>.

FIELDS:

effect <string>

Effect indicates the taint effect to match. Empty means match all taint

effects. When specified, allowed values are NoSchedule, PreferNoSchedule

and NoExecute.

key <string>

Key is the taint key that the toleration applies to. Empty means match all

taint keys. If the key is empty, operator must be Exists; this combination

means to match all values and all keys.

operator <string>

Operator represents a key's relationship to the value. Valid operators are

Exists and Equal. Defaults to Equal. Exists is equivalent to wildcard for

value, so that a pod can tolerate all taints of a particular category.

tolerationSeconds <integer>

TolerationSeconds represents the period of time the toleration (which must

be of effect NoExecute, otherwise this field is ignored) tolerates the

taint. By default, it is not set, which means tolerate the taint forever

(do not evict). Zero and negative values will be treated as 0 (evict

immediately) by the system.

value <string>

Value is the taint value the toleration matches to. If the operator is

Exists, the value should be empty, otherwise just a regular string.

* The Kubernetes GitHub issue tracker, because sometimes you may encounter an actual bug: <https://github.com/kubernetes/kubernetes/issues>. This goes for many of the additional controllers, ingresses and plugins.
* The large number of blogs and websites of its users and proselytising companies, e.g., <https://learnk8s.io>.
* Your favourite search engine.

As a Kubernetes user you are now unwittingly subscribed to its changelog documents. Keep a keen eye on all messages during execution of kubectl (and by extension, helm) because they will alert you to deprecations to APIs you are using in your templates. Features mature rapidly (from alpha, to beta, to general availability, or GA). As that happens, their namespace and, more importantly, their syntax changes. Don’t let such changes blindside you on some idle Tuesday, rendering your deployments obsolete, and read the Kubernetes release notes regularly.

Establish and maintain a healthy relationship with your Kubernetes administrators. Ask them to alert you well in advance of any version updates they are planning and take preventive steps to avoid any outages, including backups and template modernisations. Where possible, use the included virtualised Kubernetes configuration to test and practice deployment to new versions. If that is not possible due to resource requirements, for example, ask the appropriate people to institute a staging environment, onto which to test the changed configuration.

## Links

* Kubernetes documentation: <https://kubernetes.io/docs>
* The Cloud Native Computing Foundation Landscape: <https://landscape.cncf.io>
* The Dockerfile reference: <https://docs.docker.com/engine/reference/builder/>
* Telepresence: <https://www.telepresence.io>
* LearnK8s: <https://learnk8s.io>
* Kubernetes issue tracker: <https://github.com/kubernetes/kubernetes/issues>
* Kubernetes release notes: <https://kubernetes.io/releases/notes/>

FIN