appvia

Your Guide to Kubernetes

Kubernetes (often seen abbreviated as K8s) is an open-source platform built for the automated deployment, scaling and managing of containerised applications.

Google originally created Kubernetes as a small-scale project called 'Borg' around 2003, where they managed their own workloads. It was made Open Source in 2014 and presented to the world as 'Kubernetes'. Kubernetes provides a way to scale your applications and infrastructure without having to scale your operations teams. You're abstracted away from cloud providers, which gives the ability to easily manage applications across on-premises, hybrid, or public cloud infrastructure.

This guide is a comprehensive overview of important elements, considerations and functionalities of Kubernetes, plus real-world experience on how to start utilising it.

Table of Contents

appvia

Why should you use Kubernetes?

You might not need Kubernetes to successfully run your applications, but it's going to make your life a whole lot easier. There are reasons why it's remained the 'golden child' of container management, namely due to...

Security

Kubernetes simplifies the **'defence in depth'** approach by defining best practice at all levels of your application stack: Cloud, cluster, container and code. Together with the **'immutable infrastructure'** concept made popular by containers, the security posture of your stack can be hard to navigate. But through automation and securing your clusters in line with best practice for each cloud provider, you can significantly improve your security footprint.

In Kubernetes, there is a segregation of workloads running on a single instance, and the attack surface is reduced because only a single node or container is able to be attacked at any given time. Securing your Kubernetes cluster also looks a little different on each cloud provider, which can be an involved process if you don't automate it.

Cost

There's an enormous cost benefit to Kubernetes, which isn't always realised. Because the same host is reused, you're able to determine the amount of free resources within a given node and automatically schedule something else on that node. In a non-orchestrated environment, you would undoubtedly have a lot of free (read: wasted) space on the host, whereas a Kubernetes environment optimises that for you.

The elasticity of Kubernetes applications allow you to schedule workloads up and down, so that you can minimise your costs and scale only according to demand.

Scalability

Scalability is a cornerstone of Kubernetes usability. Your infrastructure is able to **automatically scale in line with your application** without you needing to manually do something about it. Although it's an add-on to Kubernetes, it's considered to be a 'straight out-of-the-box' solution for managing your applications at scale.

Availability

Best practices with Kubernetes based applications push you to architect applications for failure, which in turn improves the resilience and availability. This can be done at many levels ... from the applications failing, the underlying infrastructure, to the availability zone within the cloud provider. If they're designed well, your applications can recover from these without any loss of service to your users.

Portability

Because your application is built within a container, it can be moved around easily. There's also a lot of freedom when it comes to choosing operating systems, container runtimes, process or architectures, cloud platforms and PaaS in Kubernetes. And with great freedom, comes great portability.

When not to use Kubernetes

Is there ever a wrong time to utilise Kubernetes?

Yes ... and no. There are situations where you might not be able to realise the full value that Kubernetes can bring, at least not yet, and there are also potential road-blocks you could run into along the way.

Kubernetes is complicated

No doubt, there's a steep learning curve to Kubernetes. If you don't have an expert on your team who possesses an in-depth knowledge of K8s or a solution that can help manage the complexity, you could end up spending a lot of time and resources getting up to speed.

Kubernetes is expensive

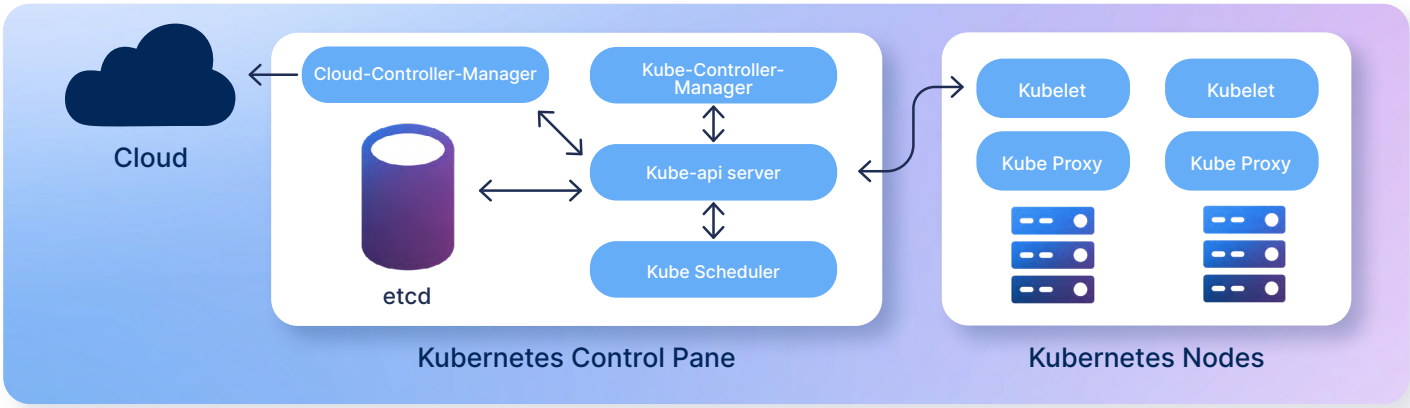
And having that expertise as a resource on your team comes at a hefty cost, despite Kubernetes itself being cost-efficient. Costs can also vary significantly depending on which cloud providers and managed Kubernetes services you use, for better or for worse.

But, is it worth it?

We'd say so. Kubernetes was designed to boost performance and reduce the operational effort of distributed systems. You can make decisions that will lower the cost of your Kubernetes ecosystem, and it's complex usability doesn't mean that your team has to be Kubernetes experts to realise its benefits - there are solutions for that.

Kuberneres architecture

As with most things, the first step to getting on board is understanding how Kubernetes is set up. At a base level, a K8s environment consists of a control plane (master), a distributed storage system for keeping the cluster state consistent (etcd) and a group of cluster nodes (kubelets).



Control plane (master)

The Control plane is made up of the kube-api server, kube scheduler, cloud-controller-manager and kubecontroller-manager. Kube proxies and kubelets live on each node, talking to the API and managing the workload of each node.

Cloud-controller-manager

The cloud-controller-manager runs in the control plane as a replicated set of processes (typically, these would be containers in Pods).

Kube-API Server

The Kube-API server validates and configures data for API objects, including pods and services. The API Server provides the frontend to the cluster's shared state, which is where all of the other components interact.

etcd

etcd is a distributed key-value store and the primary datastore of Kubernetes. It stores and replicates the Kubernetes cluster state.

Kube-controller-manager

The Kubernetes controller manager embeds the core control loops shipped with Kubernetes. In applications of robotics and automation, a control loop is a non-terminating loop that regulates the state of the system.

Kubelet

The kubelet maintains a set of pods, composed of at least one container, in entirety. Its functionality is watching for pod specs via the Kubernetes API server.

Kube-proxy

Kube-proxy is a network proxy that runs on each node. It maintains network rules, which allow for network communication to Pods from network sessions inside or outside of a cluster.

Kube-scheduler

Kube-scheduler is the default scheduler for Kubernetes, in charge of scheduling pods onto nodes. It runs as part of the control plane.

The basics of Kubernetes terminology

Desired state in Kubernetes (vs actual state)

Desired state is a core concept of Kubernetes. It means that, through a declarative or an imperative API, you describe the state of the objects that will run your containers.

Kubernetes will continuously attempt to make the actual state match your desired state, which is how your containers are actually running. The actual state may never reach the desired state (for example, you may have defined scale limits that prevent a workload from being scheduled at the desired size), but the controllers will be constantly running and working to fix issues, remediate errors, and schedule workloads as soon as possible.

Namespace

A virtual 'slice' of that cluster where you can provision resources, organise objects and deploy applications inside the cluster.

Cluster

A cluster is a group of nodes that run your containerised applications. You manage the cluster and everything it includes with Kubernetes.

Node

A node is a worker machine that contains the services necessary to run Pods. You'll typically have several nodes in a cluster, and each node is managed by the Master.

Single-tenancy vs Multi-tenancy Kubernetes

Single-tenancy in terms of Kubernetes refers to an instance where a single workload, application, team or environment is associated with a single Kubernetes cluster.

Multi-tenancy, on the other hand, is when you have multiple applications, teams or environments all running side-by-side on one, large cluster.

The two options are distinctly opposite and, although both are valid approaches, there are a few **reasons why you would choose one** over the other. You should base your choice around these five key areas: Security, Reliability, Cost and Operational Overhead.

Cloud security

Kubernetes has measures in place to prevent security breaches — like Pod Security Standards and NetworkPolicies — but, as is the running trend, it takes experience to tweak these tools in the right way. And even then, they can't prevent every security breach.

The way you configure your clusters has a hefty impact on your security. While we've established that no system can ever be completely secure, single-tenant deployments are optimised for increased security, because each customer's data is completely separate from the others. Hence, there's almost no chance that one customer will accidentally access another customer's data.

In a multi-tenant environment, where multiple applications are running side-by-side, apps share the hardware, network and operating system on the nodes of the cluster so there is a heightened risk of a breach.

Reliability of the architecture

Much like the security benefit to single tenancy architecture, the reliability risks are lessened when there is a separation of customer information. If you have everything running on a single cluster and a misconfigured application workload causes a cluster-wide outage, then all your workloads will be down.

And, there are many simple mistakes that could cause a catastrophic failure. We get deeper into some of the most common errors below, but these could include: insufficient resources, failing liveliness readiness probes etc.

Operational overhead

In a single tenant environment, you have more control over backups and recovery, because one system is backed up to a dedicated part of a SaaS server. If an app has specific requirements, these requirements can be installed in its cluster without affecting any of the other clusters.

Every cluster can be equipped with exactly the configuration that the corresponding app needs — no more and no less — which improves the efficiency of both the development and operation of your applications.

Cost comparison

At first glance, operating fewer clusters is the cheaper route - there's more resource overhead cost when you have more clusters. More clusters, more money. But, managing the costs of storage and network bandwidth are far easier to manage with just one account per cluster.

Working with Kubernetes for the first time

There are several steps you'll want to take when first **diving into Kubernetes.** Your first step should be to have a Kubernetes cluster to play with, which could be in the form of Minicube or Kind. Once you've done that, you'll deploy an application through two K8s resources: deployment and service.

Common Kubernetes errors

When you're getting started with Kubernetes, there are many reasons your deployments might fail, given the large amount of unfamiliar jargon to come to terms with and large margin for error.

Here's an at-a-glance look at some of the **most common errors in Kubernetes** and how to overcome them.

1. You have insufficient cluster resources

When creating a pod, Kubernetes will schedule the pod to a node which has enough free CPU, memory and ephemeral storage to satisfy the pod's requests. When the cluster doesn't have sufficient resources available, the pod's state will be 'pending' and when describing the pod, its events will contain "Failed Scheduling" messages.

```
Events:
Type      Reason            Age                From               Message
----      ----              ----               ----               ----
Warning   FailedScheduling  29s (x2 over 29s)  default-scheduler  0/1 nodes are available: Insufficient memory
```

You can resolve this by reducing the pod's requests or by increasing the size of the cluster, either manually or by installing **autoscaler** into the cluster.

---

2. You've specified the wrong container image

After deploying a pod, you may see that the state is 'ErrImagePull' or 'ImagePullBackoff'. There are two potential causes of this. Firstly, if the image you have specified in your pod manifest does not exist. Alternatively, if the image is hosted in a private repository, you might need to configure **image pull secrets.**

```
[➜  ~  kubectl get all
NAME                                READY   STATUS         RESTARTS   AGE
pod/nginx-6d56989578-pgb95          0/1     ErrImagePull   0          12s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx               0/1     1            0           12s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-6d56989578              1         1         0       12s
```

When the pod first fails to start, you'll see the 'ErrImagePull' status, indicating that the image did not pull. After a few failed attempts, the status will change to ImagePullBackOff and kubernetes will add a delay between attempts to pull the image.

```
[➜ ~ kubectl get all
NAME                                READY   STATUS             RESTARTS   AGE
pod/nginx-6d56989578-pgb95          0/1     ImagePullBackoff   0          48s
NAME                                READY   UP-TO-DATE         AVAILABLE  AGE
deployment.apps/nginx               0/1     1                  0          48s
NAME                                        DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-6d56989578            1         1         0       48s
```

3. Your app is crashing post-launch

If your app has crashed, there could be a few things to blame. Check your container logs using the kubectl logs command for any application errors. You should also check the exit code of the failed container; an exit code of 137 or 143 usually indicates that the container was shut down by Kubernetes due to insufficient resource requests, a failing probe or another reason. After a number of failures in a short space of time, the status will change to CrashLoopBackOff.

```
[➜ ~ kubectl get all
NAME                                READY   STATUS             RESTARTS   AGE
pod/nginx-7459d6bcc8-tt84p          0/1     CrashLoopBackOff   2          4m6s
NAME                                READY   UP-TO-DATE         AVAILABLE  AGE
deployment.apps/nginx               0/1     1                  0          4m6s
NAME                                        DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-7459d6bcc8            1         1         0       4m6s
```

4. Your liveliness/readiness probes are failing

Kubernetes provides two kinds of probes: readiness probes and liveness probes. These probes make sure that Kubernetes is able to detect when your application is unhealthy or unable to serve traffic, even when the application process has not exited.

A readiness probe is used when a container is first started and is used to wait for a container to become ready to handle requests. A liveness probe is used to ensure that a container continues to serve traffic once it has started. These probes can be configured to restart the container if it fails to start in time or an error causes it to stop handling requests.

If your probes are failing, check your container logs for any errors that could be causing the failure. Also, make sure it's possible for your probe to run ... for example, an http based probe will fail if the container is not listening on the port being checked or is only listening on local host or an exec based probe will fail if the target script/binary does not have execute permissions.

This error is caused by the file “healthcheck.sh” which is specified as the liveness probe not existing.

```
Events:
Type Reason Age From Message
---- ------ ---- ---- -------
Normal Scheduled 2m2s default-scheduler Successfully assigned appvia/nginx-685f6b867dj5jhf
to docker-desktop
Normal Pulled 79s (x3 over 2m) kubelet, docker-desktop Successfully pulled image “nginx”
Normal Created 79s (x3 over 2m) kubelet, docker-desktop Created container nginx
Normal Started 79s (x3 over 2m) kubelet, docker-desktop Started container nginx
Normal Pulling 61s (x4 over 2m2s) kubelet, docker-desktop Pulling image “nginx”
Warning Unhealthy 61s (x4 over 111s) kubelet, docker-desktop Liveliness probe failed: OCI runtime exec
failed: exec failed: container_linux.go:346 starting container process caused “exec: \” . /healthcheck.
sh\”: stat.healthcheck.sh: no such file or directory”: unknown
Normal Killing 61s (x3 over 101s) kubelet, docker-desktop Container nginx failed liveness probe, will be
restarted
```

---

5. Your Pods are getting OOMKilled and restarting when creating a Pod

You can specify both a request and a limit for cpu and memory resources. Kubernetes will always schedule a pod to a node that has enough cpu and memory available to satisfy your request. A container that exceeds its memory request is in danger of being restarted or rescheduled, and a container that exceeds its memory limit will be restarted immediately. A pod that exceeds its cpu requests will not be killed, but may be throttled if the node does not have enough available cpu, and will always be throttled if it exceeds its limit, which may significantly reduce its performance.

To avoid this, refine your containers requests to ensure that they are not exceeded by your container, and make use of a HorizontalPodAutoscaler to ensure that increases in application load are handled by an increased number of pod replicas.

```
[➜ ~ kubectl get all
NAME                                READY   STATUS      RESTARTS   AGE
pod/nginx-7459d6bcc8-tt84p          0/1     OOMKilled   1          2m49s
NAME                                READY   UP-TO-DATE  AVAILABLE  AGE
deployment.apps/nginx               0/1     1           0          2m49s
NAME                                        DESIRED  CURRENT  READY  AGE
replicaset.apps/nginx-7459d6bcc8            1        1        0      m49s
```

Often you'll see a CrashLoopBackoff error rather than an OOMKilled error. After too many OOMKilled errors in a short time, Kubernetes will add a delay between retries and the status will change to CrashLoopBackoff as above. In order to diagnose the problem in that case, you can check the output of 'kubectl describe pod' where you should be able to see the reason for the pods termination.

```
Containers:
Nginx:
Container ID: docker://3b0a27e56f49d25cf14394d648bf943f5f163d8d4e6a6e4532e42dcc70ae4169
Image: nginx
Image ID: docker-pullable://nginx@sha256:a93c8a0b0974c967aebe868a186e5c205f4d3bcb5423a56559f2f9599074bbcd
Port: <none>
Host Port: <none>
State: Running
Started: Mon, 20 Jul 2020 12:34:16 +0100
Last State: Terminated
Reason: OOMKILLED
Exit Code: 137
Started: Mon, 20 Jul 2020 12:33:18 +0100
Finished: Mon, 20 Jul 2020 12:33:59 +0100
Ready: True
Restart Count: 5
Limits:
memory: 5000Ki
Requests:
memory: 5000Ki
Environment: <none>
Mounts:
/var/run/secrets/kubernetes.io/serviceaccount from default-token-jb79s (ro)
```

Comparing clouds - AWS vs Google cloud

You have the 'pick of the litter' of cloud providers so to speak. There has been a long-standing battle between AWS, Azure and Google Cloud, and you want to compare them all to make sure your team is utilising the one(s) that work for you.

Spoiler: They're all used for the same purpose and provide nearly the same services. But you should still take a good look at each to determine which might be the best for you, or which aspects to use of which - if you're utilising a Kubernetes Managed Service. Keep in mind when you look at each cloud provider that they each have varied price models, and you might also consider using a provider closer to your geographical region, or the region of your users.

Amazon Web Services

AWS has been in the game the longest and, because of that, is often considered the leading platform. With a vast tool set that continues to grow exponentially, the **capabilities of Amazon Web Services (AWS) are unmatched.**

Microsoft Azure

A close competitor to AWS with an extremely capable cloud infrastructure, **Microsoft Azure** has the benefit of integrating with other Microsoft tools. It should be a strong candidate for enterprises with an existing Microsoft footprint with cloud services and SaaS licenses being managed in a single place.

Google Cloud

A well-funded potential underdog in the competition, Google entered the cloud market after Amazon Web Service and doesn't have the enterprise focus that helps draw corporate customers. **But Google is Google**, with strong expertise in AI and analytics that present as significant advantages, and having created Kubernetes their managed Kubernetes service is unrivalled.

Avoiding day 2 Kubernetes problems

Nothing worth having comes easy, right? Complex functionalities in Kubernetes like networking, managing authentication, connectivity between services, role based access controls etc. are hard to get your head around if you're completely new to the game.

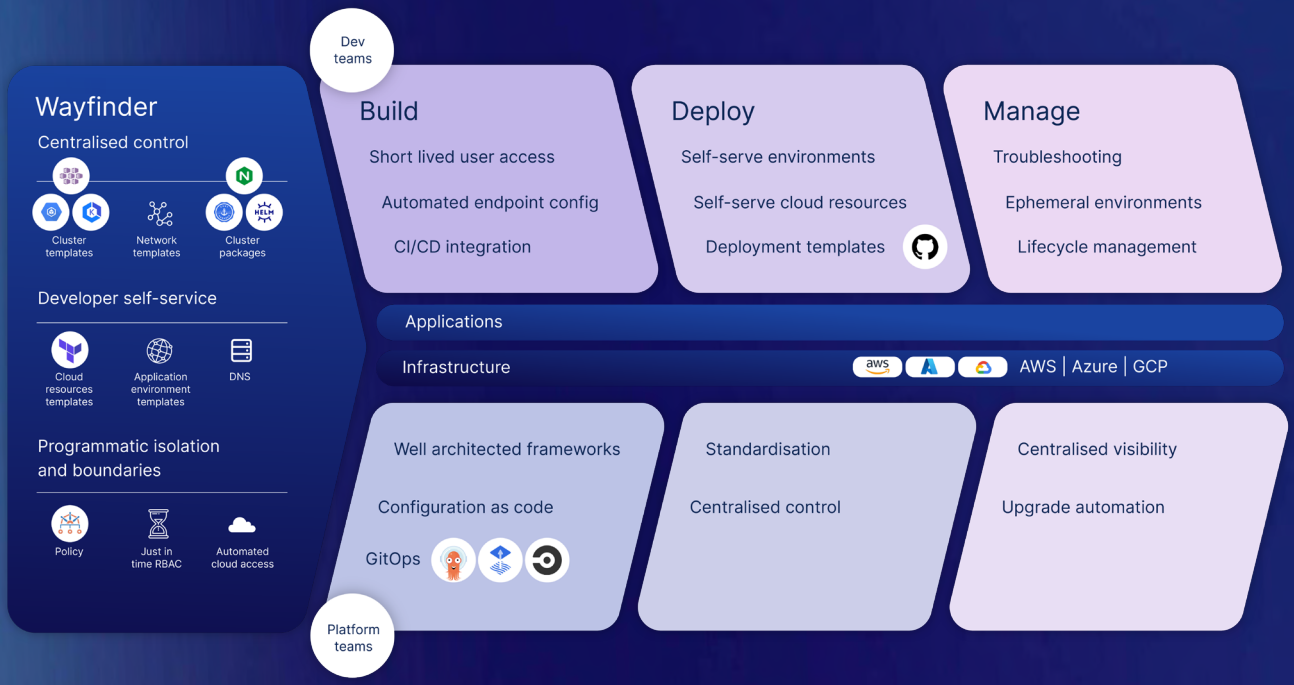
But utilising Kubernetes **doesn't need to be an absolute nightmare**. Appvia Wayfinder is a cloud-based platform that enables the automation of development environments and security standards for teams using Kubernetes. Designed to **make Kubernetes a commodity** for any organisation by removing the need for specialists.

appvia | Wayfinder

Appvia Wayfinder

Simplifying Cloud Management, Empowering Developers

Appvia Wayfinder transforms the way that teams deliver containerised applications to the cloud. By enabling self-service provisioning of cloud resources and environments Wayfinder delivers efficiency, scalability, and security for both application and platform teams.



Key Capabilities

Scalable Infrastructure Management

Appvia Wayfinder enables self-service provisioning of cloud resources, centralises control for platform teams and enhances developer productivity with seamless CI/CD integration.

Streamlined Testing and Security

By automating ephemeral namespaces for testing and applying best-practice configurations, Wayfinder ensures optimal security and efficiency in your infrastructure.

Kubernetes Simplified

Unleashing the robust capabilities of Kubernetes without the associated complexities, Wayfinder makes container orchestration accessible and cost-effective.

We Solve

Delivery Bottlenecks

Appvia Wayfinder eliminates roadblocks for application teams and eases pressure on platform teams, paving the way for efficient and smooth app development.

Escalating Cost and Complexity

With its simplified Kubernetes management and cloud cost optimisation, Wayfinder reins in disproportionate cloud spending and reduces the need for deep in-house expertise.

Unmanageable Security Overheads

By employing best-practice configurations and offering built-in security features, Wayfinder significantly mitigates the risk of infrastructure-related security breaches.

Learn more at appvia.io