What's Kubernetes?

The post you've all been waiting for and struggling to pronounce

Justin
Nov 10, 2020

The TL;DR

Developers use Kubernetes to **turn their individual containers and virtual machines into** full fledged, **working applications**.

- Modern apps will often consist of a bunch of containers doing separate things that **need to work together**
- Kubernetes **orchestrates** these containers and takes care of **exposing them** to people like you and me
- You can generally split the "point" of Kubernetes into three buckets: **container health**, **networking**, and **configuration**
- The whole Kubernetes ecosystem is **complicated and intense**, and it's not for everyone or every app

For better or worse, K8s (the world's acronym of choice) is becoming the system of choice for deploying even remotely complex software these days.

Why this is hard to understand

As with most software concepts, people struggle to understand Kubernetes because they're missing *background knowledge* - there's some other stuff you need to know first before K8s will click for you. And those things are (in no particular order) - the cloud, microservices, and containers.

> 🧠 **Jog your memory** 🧠
>
> If you haven't read yet about Docker and containers, you should!
>
> 🧠 **Jog your memory** 🧠

The basic idea here is that developers are now pretty consistently deploying their apps via containers on public cloud infrastructure, and organizing those containers to work together is challenging.

→ The cloud

Instead of buying their own servers, companies now pay AWS and co. to rent compute and storage, paying monthly or yearly for how much they're actually using. The most common *unit* is called a virtual machine (in AWS lingo, this is usually an EC2 instance), which is sort of like a single computer or server. Most apps will be made up of multiple of these instances, all doing different things.

→ Microservices

Over the past 10-15 years, developers have been leaning towards splitting up their applications into small, understandable units instead of one giant piece of code. You might have a single "service" for sending emails to customers, one for processing analytics events from your app, and one for your backend apis.

→ Containers

To quote myself from the original post explaining Docker and containers:

> *Containers are a way to **run code in isolated boxes** within an operating system; they help avoid disastrous failures and add some consistency to how you set up applications.*
>
> - *Special software called a **hypervisor** allows multiple isolated operating systems to run on one computer, and containers do the same for operating systems (🤯)*
> - *Containers also provide a **consistent environment** for deploying software, so developers don't need to configure everything 1000 times*
> - *The most popular container engine is **Docker**: it's open source and used by ~20% of all developers*

> by >30% of all developers

> *Containers (and Docker) have spawned an entire new ecosystem of exciting developer tools. You may have heard of Kubernetes, the *checks notes* greek god of container infrastructure.*

Keep in mind that applications are usually made up of *multiple* containers, a la the microservices philosophy. And that's where things get interesting.

Why scaling is hard, and where Kubernetes came from

So let's say you're a developer, and you've been working real hard on this new application, which you've containerized and packaged inside a very nice Docker container. You log into your powerful AWS server, type `docker run` into the command line, and boom, after configuring domains, your application is running at `coolsite.com`. Your Google Analytics tells you that 50 people are using it every day, and you're very satisfied. But what happens when it grows?

At some point, your app will be too successful for its own good, and too many people are accessing it for that one little Docker container to handle. So you log back into your AWS server, and do two things:

1. You spin up another Docker container (so now there are two)
2. You add something called a load balancer that distributes traffic between your two containers

Great, now you're good to go. But what happens when something goes wrong? Let's say you forgot to add an error handler into your code, and so one of your containers stops working. As a very prudent developer, you've set up alerts, so you get a Slack message telling you things aren't quite right. So you log back into your server, restart container #1 (the second one was fine), and you're good to go.

But what happens when you need to upgrade your app? And what happens when you need 10 containers, not 2? What happens when there's so much load on your load balancer that you need *another* load balancer? The problems go on and on - such is the nature of big, scaled software. So you could have a dedicated engineer focused on just sitting around, waiting for problems to happen - but that doesn't really work very well when you've got an application with 15,000 containers. What you really *need* is a set of tools that make it easy to *automate* the work of creating, managing, and scaling containers.

This is exactly the problem that Google had internally back in the mid to late 2000s. They were operating at a truly massive scale (billions of users!), and needed to make sure that things worked, really fast, all the time. So they developed a system called Borg that handled all of this nasty operational work *automatically*. Eventually, they open sourced a newer version of it in 2014 and dubbed it Kubernetes. It's open source, which means you can literally read and use the code on Github.

What Kubernetes actually does

Most places on the internet call Kubernetes a "container orchestrator," like we're in Radio City Hall or something. The best way to understand things is to take a look at the **specific tasks** that Kubernetes does for developers, and you can generally split them into 3 categories.

→ Health

Containers run into issues all the time - your database might get stuck in a transaction, you might have forgotten to add a handler for a new error type, or your code might be stuck in an infinite loop. When that happens, you need to **restart** your container, or deploy a new one - Kubernetes can do that automatically for you.

You can also run **custom health checks** with K8s, by having it ping your containers every hour, or check for specific responses and do something if they're wrong. You might send a request to your `/users` endpoint, and if the response is null, notify your team and replace a container.

→ Networking and a "master layer"

Kubernetes helps take your containers and **expose them to the public web via DNS** (think: `yourwebsite.com/api` maps to the `api` container), and also load

balance if there's a lot of traffic coming in and you want to distribute that among multiple containers.

K8s also helps your containers access shared resources like cloud storage, so you don't need to connect each container individually.

> 🔍 **Deeper Look**🔍
>
> A good example of a shared resource is **secrets**, or developer-speak for passwords and API keys. Teams generally store them in a secure place (one of the popular ones is Hashicorp Vault) where they can be accessed programmatically by services and applications, and Kubernetes helps connect those to your containers.

🔍 **Deeper Look**🔍

→ Configuration

The final - and perhaps the most complex and exciting - thing that Kubernetes does is automatic configuration of your infrastructure. The way it works is basically this:

1. You describe to Kubernetes what your app should *look like* - a service for this, that, etc. - called the "desired state"
2. Kubernetes does the work - spinning up containers, networking them, running commands, upgrading - to actually get to that state

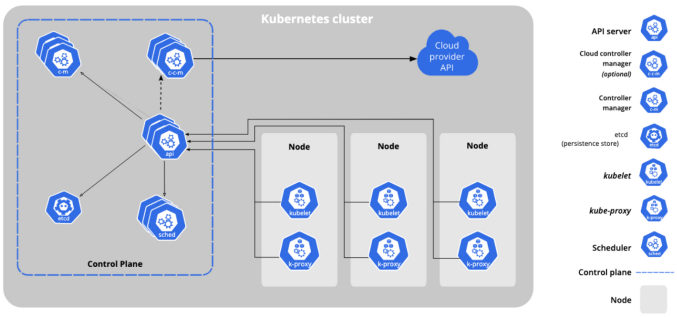
In the K8s ecosystem, these "desired states" are organized using Kubernetes Objects, which is a fancy term for a specific configuration language and system of organization. Basically, you need to speak the Kubernetes language to take advantage of what it has to offer.

If any of these individual sets of features don't make sense, that's OK - just keep in mind that K8s is taking care of the frustrating *operational* elements of deploying containerized applications - things that developers would normally have to do manually, or write complex scripts for.

Kubernetes 101

It's worth diving into what these Kubernetes Objects actually are, and giving a basic overview of how the system works. Feel free to skip this section if you're feeling overwhelmed, or if this is too in depth for you. These concepts aren't very important if you don't intend on working with K8s in some capacity.

If you're a developer, and you waltzed over to the Kubernetes documentation, you'd find this diagram staring back at you:



I didn't understand this the first (2)7 times I looked at it, so here's the simpler version. There are 3 big pieces to the Kubernetes system: **nodes**, **pods**, and the **control plane**. They sort of form a little army of infrastructure.

> 😇 **Don't sweat the details** 😇
>
> Each one of these "master groups" has a bunch of Kubernetes specific components in it with even more names and acronyms. Just focus on what the group is trying to accomplish, and you'll figure out the names down the road if you ever go deeper.

😇 **Don't sweat the details** 😇

1. **Nodes**

This one is easy - a node is just **a single piece of infrastructure**, usually a VM. You can think of it as one server, and since Kubernetes is built to manage *distributed systems*, it has the capability to manage *multiple* servers. You install a few Kubernetes components (kubelets and kube-proxies) on each node so that the system can control it.

A node is like a single enlisted person in the army. This person is just a resource (well not literally but like for this example) - they haven't been assigned to the army, the navy, or the air force yet - and they haven't been put in a specific place.

2. **Pods**

A Kubernetes pod is a **group of containers** (it can also just be one container, which is pretty common). This brings us back to the "desired state" concept - let's say your "application" pod is supposed to have 3 containers. If the pod currently has 3 and one fails, Kubernetes will take care of replacing it. If the pod currently has 4 and one fails, it might *not* replace it, because it's still in the desired state.

A pod is like an army unit, say the 101st Airborne Division, the subject of the critically acclaimed HBO miniseries *Band of Brothers*. It's a group of specific personnel with a specific purpose.

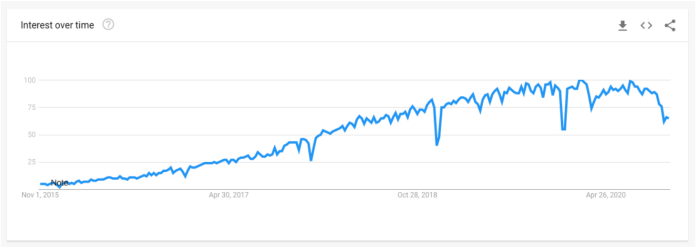
3. **Control Plane**

The control plane is the mastermind that **orchestrates** all of the cool things that Kubernetes is doing, like scheduling containers to pop up and disappear, allocating different pods to different nodes (imagine we want our "application" pod to be on our most powerful server), and such.

The control plane is like military leadership, both strategic (where should we be allocating our resources?) and practical (let's go and allocate those resources).

Do you need Kubernetes? Does anyone?

Kubernetes has skyrocketed in popularity over the past few years. We're now at the point where most developers are at least (probably) aware of it, and 13% of professional developers say they've used it.



But if you've been following along, you've probably grasped that K8s is very complicated, and seems oriented towards big, giant, scary applications that need to maintain the highest levels of performance. And you're right - that's *exactly* the circumstances that Kubernetes was built for, internally at Google.

There's an interesting paragraph in the K8s documentation that compares the system to PaaS like Heroku:

> "*Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system. Since Kubernetes operates at the container level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, and lets users integrate their logging, monitoring, and alerting solutions. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable.*"

The long short here is that while Kubernetes makes it easier to deploy complex containerized applications, it's not the simplest way to do that. Heroku or DigitalOcean's new App Platform also let you deploy Docker containers behind a load balancer. As with all software, there are tradeoffs to everything, and with the power of K8s comes complexity and a steep learning curve.

This is a tried and true story in open source software - a company like Google develops an internal system under very specific constraints, and then releases it

to the world (2014, in Kubernetes's case). And people ignore those specific constraints, think that they need to use it, and end up complicating everything because of it. So do you really need Kubernetes?

The answer, generally is no. If you're running a blog, you don't need Kubernetes. If you are working on a side project with a few thousands users, you don't need Kubernetes. If you are deploying a simple website, you don't need Kubernetes. Most of the world's largest and most complex applications are doing just fine without Kubernetes.

Over time, expect clearer social guidelines to develop around when apps need something like this and when they don't. In the meanwhile, an entire ecosystem is developing around the technology:

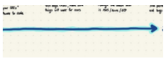
- Managed Kubernetes services from major cloud providers: EKS (AWS), GKE (GCP), AKS (Azure), and DigitalOcean even
- Managed services for databases like MySQL running on Kubernetes (e.g. Planetscale)
- Monitoring and observability for Kubernetes (e.g. Grafana and Datadog)

And finally - in case you were wondering - here's the story behind the Kubernetes name.

Further reading

- Writers across the web have struggled to explain Kubernetes in an accessible way, and this post's approach is to aggregate explanations from different engineers in the industry
- The New Stack has an e-book that talks about the K8s ecosystem and some key open source projects

(Thanks to Jake Cooper for feedback and additions)



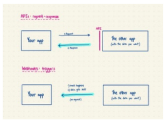
one giant tool for literally anything you'd want to do relating to building and...

Justin Jan 4 8

What's DevOps?

IT has a cool new name

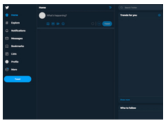
Justin Jan 5, 2021 34



What are webhooks?

Triggered

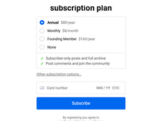
Justin Sep 13, 2021 28 5



What's Headless E-Commerce?

We may be running out of names

Justin Nov 2, 2021 20 7



I built a (basic) Substack clone in a month

This was probably a waste of my time

Justin Sep 2, 2020 50 4

See all ›