Kubernetes for Non Technical Readers

Adrian Kosmaczewski

2021-10-01

If you work in the tech field, the word "Kubernetes" is all over the place these days; for those new to the subject, it can be very confusing to understand what it is, what it does, and why it is so important to so many people. In this article I am going to try to go top-down on the whole Kubernetes thing.

Many books and resources about Kubernetes use a bottom-up approach, that is, they start by providing lots of low level definitions, many of which are mostly important for the technically inclined; but here we're going to do things differently.

I'll assume that you have read the word Kubernetes on the press, and you're wondering about what it is, what problems it solves, and how it works. You have some IT background, and maybe you've even worked with backend services and servers in the past. But most importantly, you're curious about this Kubernetes thing, and how it fits in the big picture; hopefully this article will give you some useful pointers. It is, in short, how I would have liked to learn Kubernetes, to have it explained to me.

I've sprinkled the article with plenty of links, so that you can keep exploring after reading it.

TL;DR: sorry, there's none; you have to read the whole article, and it is a long one (~5700 words, ~25 minutes read).

  - Storage
  - More
- Installing Applications in Kubernetes
- Flavors of Kubernetes
- Added Value
- Conformance
- What's Next?

What is Kubernetes?

Simply put, Kubernetes is a platform to run websites and web applications.

And you may ask then, what is a "web application"? It's a program that receives "requests", and returns "responses". You interact with web applications using standard web browsers, such as Firefox, Chrome, Safari, Opera, Brave, or Edge. Users can ideally use any browser to use their preferred web apps, and they should (should) get the same experience no matter which one they choose. This is a common experience in 2021.

For example: Jira, Confluence, WordPress, GitLab, Basecamp, Gmail, or even Galaxus are common and popular examples of web applications, that is, they run on a web server, and you use them through a web browser. There is plenty of such applications, but the ones mentioned above are among the most popular of them.

Of course, Kubernetes is not the only way to run web applications, but it has lots of advantages that make it particularly good for some very common scenarios:

- Need for scalability: what if your online shop becomes a hit overnight?
- Cross-platform and portable: what if you want to move your applications from AWS to a cheaper provider, or just run it in your own laptop?
- "Infrastructure as Code" friendly: what if you need to plan for capacity for future expansions, and you need information about your current infrastructure costs? (I'm going to go over the concept of "infrastructure as code" a bit later)
- *De facto* standard: what if you need new engineers for your team? And can one easily find third-party tools that are compatible with Kubernetes?

For many large applications, Kubernetes provides a very flexible platform for running web applications, and one that has become a standard, for that matter.

But one very important thing to know is that not all web applications need Kubernetes, or even work well with it; for example, some databases systems explicitly aren't Kubernetes-friendly, and engineers must be aware of those limitations.

In those cases, instead of using Kubernetes, you can just run any of these web applications in their own standalone server. By server we mean for example a computer you bought in Digitec, one that you have connected yourself to the

network in your office, and which you make available in your local network. A web server can also be a virtual machine you run on your own laptop, like when you use Oracle VirtualBox, or on AWS EC2, for example.

To summarize, Kubernetes is just one way to run web applications, and one that has become really ubiquitous and popular at that.

A Simple Example

Let's imagine you want to watch movies you've (legally) downloaded in the confort of your home.

To do this, you could buy a new small Mac Mini, or a Synology NAS, and install the Plex media server software in it. Plug a USB disk with your movie collection, and then you can watch those movies from your flatscreen TV in your living room. That's a very simple kind of web application, because your partner can actually access the same Plex application from your laptop in their room, just by using a web browser. The TV is just another "client" of the Plex server; so is the laptop of your partner.

In this simple example, you are running a private service, just for you and your family, in your local network (hopefully you have cabled your home beforehand, or the video quality would suffer a bit). Since there's probably no more than 10 devices (or "clients") in your home accessing the Plex server, it's enough for you to have just one server. Synology NAS units and Mac Minis are quite powerful little machines, as it turns out.

In this case, your TV "requests" a movie, and the Plex server "responds" with the stream of data that contains the audio and video for your enjoyment.

Of course, you're the "sysadmin" of such server; if you need to update your Plex, because there's a new version with new features you'd like to use, you can just update it, and since your data is stored in a separate USB drive, you don't risk losing anything at all. During the time of the update, of course, nobody can watch movies, so you do that on a Saturday morning, or at any other time when nobody is watching movies.

History

At the beginning of the World Wide Web, most systems ran exactly like the Plex server I've described above. Even Jeff Bezos started Amazon with a single server directly hooked onto an Internet connection. To update software in such a server, developers would just copy the files into the server, verify that everything still worked, and pat themselves on the back when it did.

In my case, when I started working as a software developer in 1997, we had our server (note the singular) in an MCI WorldCom datacenter in upstate New York, south of Albany, in the USA. We did that because Internet connectivity in Buenos Aires was very bad back then, simply because Buenos Aires is far

away south, and not many transatlantic optic cables reached the area back then. Bandwidth was narrow, and hosting the server in the USA was a better option.

At the beginning we had just one server there; it ran two important pieces of software:

- A database server, where all the data of our company was stored (if you're curious, it was a SQL Server 6.5 database server).
- A web server, which was configured to return pages of information to people connecting to it.

The server was using a Pentium II chip, running a version of Windows called Windows NT, with around 256 MB of RAM, which was a crazy expensive amount of memory back then. This simple setup allowed us to serve a whooping 10'000 visitors per day, a record by those standards.

By 2000 we raised some VC money, so we upgraded our installation: we decommissioned the old server, and we bought two new, much more powerful servers; one for the database, and another for the web server, both talking to each other in a simple local network. This allowed us to multiply by 10 the number of visitors we could serve every day, and so we grew.

We uploaded the programs from Buenos Aires for our web application running in the USA using something called FTP (which stands for "File Transfer Protocol"). To manage the system, for example for reboots, backups, and other operations, we used the "Remove Desktop" functionality provided by Microsoft.

We relied on manual operations, and of course, we made mistakes all the time. We had to often roll back operations that went wrong, and restore old backups, all done manually. This was complicated, brittle, and very clumsy.

Problems

At the beginning of the 2000's, the web was becoming more and more popular, and many successful companies had much more traffic than a single server could handle. That is, they could not **scale**.

This is when the concept of a **"load balancer"** appeared; it's a special kind of software that distributes requests among separate, connected, web servers. This meant that you could have lots of web servers, and one load balancer in front of all of them, to distribute requests among all web servers. Above we mentioned nginx, which is a common and popular choice among load balancers.

Load balancers use different algorithms to distribute requests among web servers; the simplest one is called the "round robin algorithm", which simply sends one request per server, server after server, in order. Other algorithms take into account the level of load of each server, so that no server is overloaded with heavy-duty requests at any time. There are other more complicated algorithms, and engineers must choose the one best suited for each particular situation.

But, the thing is, when you have lots of web servers running at the same time with the same application behind a load balancer, one specific problem appears: if a web application allows users to log in (a very common requirement, as it turns out), the same user should be logged at once in all web servers behind the load balancer; you can't pretend the user should login in each server every time! That would not be very convenient, and you would lose customers. You must have one login to rule them all.

Whenever a user "logs in" to an application, a **"session"** is created; that's a fancy name for a small piece of data kept in memory, containing the name and other settings belonging to the current user. In such an application, each user sees their own name and preferences in the application, and cannot access other people's settings. But what happens when you have lots of web servers behind a load balancer?

Well, you need another server just to keep session information separate from the web servers. And now, all of a sudden, you have a very complex architecture:

- One or more web servers.
- A load balancer distributing requests among web servers, for example nginx, to allow the whole service to scale.
- A security information repository; for example, these days companies use systems such as OpenLDAP or Keycloak to store that information, to store the usernames and passwords of their users in a secure server.
- A session store; many web applications use a special database called Redis for that.
- And yes, maybe one or more database servers, maybe also behind a special load balancer, just for them!

Here's an important concept: all of these services and servers put together, form what is usually called a **"cluster"**.

And there's plenty more of things that you can have on a cluster: for example, message queues to process purchases; mail servers to send newsletters and forgotten passwords; firewalls to filter requests (usually for security purposes); storage devices connected to the network (like a very expensive Synology NAS) with plenty of disk space available; and much more.

So, remember this: the common name for all of these things is a "cluster". And each of the servers in a cluster is called a **"node"**. Those nodes can be physical or virtual, or a combination thereof, as stated previously.

Complexity

As we just saw, a cluster means having lots of servers to manage, update, backup, and restore. For example, when you have many application servers, and your development team makes an update to your application, you need to roll that new version to all nodes, at once, sharing secrets such as passwords and keys in each node, and without errors!

To make things more difficult, each of the applications we mentioned above is written with different programming languages: for example, GitLab is written in the Ruby programming language; WordPress in PHP, while Jira and Confluence are written in Java. Each of these applications required, thus, separate libraries and configurations, that are absolutely and completely incompatible with each other. This means that the upgrade procedures for each language are also different!

An update might also require database migrations, and if anything goes wrong, you need to roll back to the previous version of the application as soon as possible, because your customers can jump to your competition in a blink of an eye.

To top it off, you need to keep an eye on all of those servers; are they running properly? Are you sure they aren't running out of memory? How about disk space, is there enough? Can you be notified if any of those servers is down?

And what happens if you need an exact replica of this cluster for your development team? And another for your quality assurance team?

Managing all of this complexity by hand is clearly not a solution for big web applications. We need a better way.

Evolution

In 2005, Amazon pioneered the concept of "Infrastructure as a Service" or "IaaS" when it launched Amazon Web Services (AWS); instead of having your own hardware servers on a datacenter, just like we had in MCI WorldCom, you can rent a small portion of their infrastructure, available all over the planet in various datacenters with excellent connectivity, and run as many VMs (virtual machines) as required.

The same idea is now available by various providers: Exoscale, cloudscale.ch, Microsoft Azure, Google Cloud, IBM Cloud, Alibaba Cloud, Oracle Cloud; they all provide the same basic building blocks of infrastructure:

- Compute (that is, virtual machines, like Amazon's EC2 service)
- Storage (in various forms, but usually compatible with Amazon's S3 service)
- Security (users, groups, etc)

Later on, around 2010, there was an explosion of new projects based on the concept of "Infrastructure as Code". In essence, infrastructure as code means writing down a description of your… well, infrastructure, in a machine-readable text file; and then letting a piece of software figure out how to make such infrastructure happen or exist.

You might have heard of Terraform, for example; it is used to setup many virtual machines at once, with one command, and to install software in them.

This is an example of a Terraform file defining an Exoscale SKS (Scalable Kubernetes Service) cluster hosted in their datacenter located in Geneva, Switzerland:

```
locals {
  zone = "ch-gva-2"
}

resource "exoscale_sks_cluster" "demo" {
  zone          = local.zone
  name          = "demo"
  version       = "1.21.1"
  description   = "Webinar demo cluster"
  service_level = "pro"
  cni           = "calico"
  addons        = ["exoscale-cloud-controller"]
}

resource "exoscale_sks_nodepool" "workers" {
  zone               = local.zone
  cluster_id         = exoscale_sks_cluster.demo.id
  name               = "workers"
  instance_type      = "medium"
  size               = 3
  security_group_ids = [exoscale_security_group.sks_nodes.id]
}
```

A "node pool" as specified above is nothing else than… a group of nodes, or virtual machines, running in the Geneva datacenter.

There's many other similar systems; suffice to mention Vagrant, Puppet, Ansible, and Chef, all of which can be used for similar purposes in different contexts.

Kubernetes provides a form of Infrastructure as Code as well, but specifically not for hardware (physical or virtual, which is something Terraform does very well), but rather for software.

With Kubernetes configuration files you can specify the exact name and version of the applications you want to run, the number of copies running in production, and you can even specify load balancers for it all.

And since everything is written down in a text file, you can create an exact replica of your production environment for development, testing, redundancy, or any other purpose. Just re-run the same software on a different environment, and build a copy of your infrastructure somewhere else. Boom, done.

Now we're talking. We can use Terraform to bootstrap a new cluster in Exoscale in less than 5 minutes, commissioning some virtual machines and installing all the required components, and then install and run lots of software on it.

Anatomy of a Kubernetes Cluster

As said previously, Kubernetes is a platform to run web applications. And it does that by managing a cluster, just like the ones we defined above.

A Kubernetes cluster requires nodes, as expected. These nodes are either hardware machines (for example, expensive Dell or Hewlett Packard enterprise servers, or cheaper and smaller Raspberry Pi machines) or virtual machines (like those you can create with VirtualBox on your laptop). Those nodes usually run Linux, but since not so long ago they can also run Windows.

Of all of those nodes you have, Kubernetes takes one (usually the more powerful one) and gives it a special name: the **"Control Plane Node"**. The other nodes, usually cheaper and smaller, are called **"Worker Nodes"**. You usually need at least one control plane node and two worker nodes to have a decent cluster, although just one of each also might be a good choice depending on your needs.

(Turns out you can also load balance the control plane, to create what is usually referred to as an "HA" or "High Availability" cluster, but that's a subject for another time.)

The role of the control plane is to distribute "work" among the worker nodes. The control plane watches the worker nodes continuously, and can decide at any time to "drain" a node and move work to another node.

Hence, the more worker nodes, the better, because each node brings memory and CPU power to the whole cluster. But that also means higher costs, both financial and human, to get everything wired together properly. This is a common trade off.

Developers and system administrators "talk" to the control plane (that is, send requests to it) so that work is distributed, launched, and monitored on all worker nodes. This is done through what is called the Kubernetes API (or Application Programming Interface), that is highly standardized. Developers do not need to know all the details of that API, and just use a tool called `kubectl` to talk to the control plane.

There are many other components inside of the control node:

- The API server, mentioned above.
- The scheduler, selecting the nodes in which to run work.
- The controller manager, in charge of managing work and its lifecycle.
- A database called etcd to keep track of the whole inventory of stuff inside of the cluster: work, nodes, configuration, etc, so that if the whole cluster must be rebooted for some reason, everything comes back alive in a similar state.

This means that the control plane is monitoring continuously the worker nodes, so that all apps are running smoothly, and notifies cluster administrators of outstanding events. It can also scale applications, if needed, when the demand

grows substantially, or even restart applications if they are not behaving properly. In short, it does a lot of work on behalf of human operators.

Pods, Containers, Deployments, Services, Storage...

We mentioned previously that a control plane distributes "work" among the nodes of your cluster. But what how does that "work" look like?

Simply put, a unit of work in Kubernetes is one or more small containers running your applications inside. Those units of work composed of containers are called **"pods"**.

For example, in the case of GitLab, you can have a single pod running the application, and another pod running a database server. GitLab is written in the Ruby programming language, and has a certain number of "dependencies" it requires to run: configuration files, libraries of code, and more. A pod is a lightweight way to encapsulate the app together with all of its dependencies in a single portable unit. Kubernetes can "schedule" pods to run on a node or on another, depending of many factors, most of which are configurable by the operators of the cluster.

Containers and Image Registries

And what are pods made of? Well, internally pods consist of one or many **"containers"** (well, usually just one, but you could have many if needed). And what are containers? They could be thought of as very lightweight, very fast small virtual machines. And how do you make them? Developers can create and run container images (think "templates") in their machines, without the need for a full Kubernetes cluster, using any of the following applications:

- Docker
- Podman

Once developers create container images, they can share them with other developers, either in the open, or with their colleagues inside an enterprise. For that, developers store containers images inside a **container image repository**, of which there are many examples: Docker Hub, Quay, AWS ECR, GitHub packages (ghcr.io), the Google Container Registry, and ttl.sh. Some companies even choose to run their own container image registry internally, using tools like the OpenShift image registry, the GitLab Container Registry or Harbor.

As I said, developers do not need Kubernetes to run containers; but we do need containers to do useful things with Kubernetes, because pods are made out of containers, and Kubernetes runs pods. This is why Kubernetes is usually referred to as a **"container orchestrator"** system, because it helps running complex apps made with various containers all wrapped into pods, and connected with one another, like the complex system we described earlier in this document.

We are going to talk about containers in another article (and, if you are technically inclined, please forgive my assertion about containers being "small virtual machines" above). For the moment, suffice to say that in the case of Kubernetes, pods "wrap" one or more containers. This allows Kubernetes to move work from node to node more easily. And it is helpful, because some applications require various containers to work together, tightly bound one to the other, so you need to have them always together. A pod provides precisely this advantage.

Deployments

Some applications are more complex and require not just one pod, but many pods at once; for that matter, Kubernetes has the concept of **"deployments"**; a deployment is a set of pods that must always run together.

Kubernetes makes sure that if, by ill luck, one of the pods of a deployment dies (because it crashes or some other problem) another one replaces it as soon as possible. This is done automatically by Kubernetes, and operators can use **"liveness probes"** to make sure that Kubernetes checks pods correctly.

There is another kind of probes used by Kubernetes, called **"readiness probes"** which is used to make sure that pods only receive requests when they are ready for them. They are used when the pods start.

We mentioned above that Kubernetes has a form of "infrastructure as code" built in; well, below you can see an example of a Kubernetes deployment defined in a text file using a format called YAML, or "Yet Another Markup Language", which defines a hierarchy of key and value pairs. Think "ini" files, like those common in old versions of Windows, but with a tree hierarchy.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-deployment
spec:
  replicas: 3
  template:
    spec:
      containers:
      - image: quay.io/username/container-name:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 9090
          name: sample-port
        resources:
          limits:
            cpu: 150m
          requests:
            cpu: 100m
```

```
        livenessProbe:
          httpGet:
            path: /healthz
            port: sample-port
          periodSeconds: 10
          initialDelaySeconds: 10
```

In the example above, the deployment specifies 3 pods, each containing one container inside, based on a container image called `quay.io/username/container-name:latest`, that is, stored in the Quay image registry. The `latest` word indicates the version of the container image; which means that, if needed, we could be using a previous version, in case the newest available has bugs.

The deployment above also shows some interesting limits for the deployment, so that pods don't consume more CPU time than allocated. This is very important for operators, to be able to control their usage costs, and to plan for future upgrades.

Scaling

Another thing that Kubernetes provides is **automatic vertical scaling**, that is, launching new pods if the demand of a service grows; for example, if your web application store is very popular, and you have lots of new customers online, Kubernetes spins new pods automatically to cope for the demand. Those pods could (should) run in a separate node, if there are any available.

The YAML below defines a simple autoscaling strategy:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: sample-autoscaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: sample-deployment
  minReplicas: 1
  maxReplicas: 30
  targetCPUUtilizationPercentage: 20
```

Of course, this might need new nodes to be added, which is something that Kubernetes cannot help you with, because Kubernetes is about software, not hardware; the nodes must be added by an engineer, either manually, or using a tool like Puppet, for example. Kubernetes can, however, once a new node has been added to the cluster, distribute work on it automatically without needing to restart.

Services

Kubernetes does not make applications available to the Internet by default. This might sound counterintuitive, but it's for security reasons; nothing is visible by default. To expose a deployment to the outer world, you must create an object called a **"service"**. There are many kinds of services available in Kubernetes, but the most common one is the "LoadBalancer" kind, which you already now what it is for.

This is how you define a LoadBalancer service in YAML:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: fortune-service
spec:
  type: LoadBalancer
  ports:
    - port: 3000
      targetPort: fortune-port
```

Storage

Finally, another thing that Kubernetes allows to specify in code is the kind and amount of storage for applications. This allows operation teams to plug and unplug storage providers, and to configure applications to use them accordingly. This way you can allocate 10 GiB of disk space to this application on a fast SSD drive, or 50 GiB of space in a slower but cheaper SATA disk. Some common providers of storage for Kubernetes include Rook, Gluster, and Longhorn.

More

There are many more things that Kubernetes can manage off-the-box: for example, **secrets** (passwords, keys, etc); **configuration values** (which you can change after deployment, if needed); **environment variables** (used to tweak the behaviour of an application in subtle ways); **namespaces** (to separate different objects belonging to different applications running in the same cluster); and much, much, much more.

Installing Applications in Kubernetes

There are many ways to install applications in a Kubernetes cluster:

- The usual way is to write files in YAML format, with all the required options, and then "applying" those files to the cluster using the `kubectl` tool specified above.
- Another way is to combine many YAML files together in a "kustomization file" (yes, with a K), so that they are all applied together in one operation.

- Finally, most apps these days are bundled with what are called "Helm Charts" which are lots of YAML files in a standard structure, specifying all of the pods, deployments, services and storage options required for an application to run.

Lately there is another option to install applications on Kubernetes, using what is called an "Operator", allowing Kubernetes to extend the number of objects available in the cluster. For example, K8up is a backup system bundled as an operator, which allows developers to create YAML files that explicitly mention the "Backup" or "Restore" objects.

Flavors of Kubernetes

Kubernetes is an open source project originally created by Google. They based it on a previous system they used internally called "Borg", in production since at least the 2000s. Google donated Kubernetes to the Cloud Native Computing Foundation, the CNCF, which is a non-profit that raises money so that developers can keep on working on various projects related to Kubernetes.

Kubernetes is written in the Go programming language, which has many characteristics that are specially useful for cloud projects. This is the reason why many of the most popular applications hosted by the CNCF or used around Kubernetes are written in Go, such as Kubernetes, Docker, Prometheus, Istio, and many, many others.

By the way, the word "Kubernetes" is of Greek origin, and is actually pronounced "Kivernitis"; it means captain or driver of a ship, which is why the logo of the Kubernetes project is the steering wheel of a ship. Kubernetes is the Greek root of the words "cybernetics" and "government", which should sound very appropriate to you by now.

Another way to write "Kubernetes" is to just write "K8s", where the "8" is the number of letters between the "K" and the "s" in "Kubernetes". This is serious, please don't laugh.

Because it's open source, Kubernetes is available to everyone and all, so companies are free to grab the source code, apply their own modifications on top, and sell it if they wish; this is how the following projects came to existence:

- Red Hat OpenShift
- Rancher RKE
- Amazon EKS and EKS Anywhere
- Google GKE
- Microsoft Azure AKS
- Exoscale SKS
- Katacoda
- DigitalOcean Managed Kubernetes
- IBM Cloud Kubernetes Service

- And the smaller members of the family, meant to run in your laptop or on small computers: Minikube, MicroK8s, Kind, and K3s

To avoid confusion, I should add that the "Rancher Enterprise Management System" (or "Rancher" for short) is a different thing than the Rancher RKE project mentioned above; it is a web application that allows you to manage Kubernetes installations, but it is not a flavor of Kubernetes per se. But yes, Rancher can run on a Kubernetes cluster! There's a Helm chart for it, of course (see? All of the concepts are falling in place now). The Rancher Enterprise Management System is extremely popular and convenient, and lots of companies use it to manage their clusters from a central location every day.

The usual analogy to explain the various flavors of Kubernetes is to compare it with the various flavors of Linux distributions. In Linux, there is a project for the Linux Kernel on one side, and then there are "Linux distributions" such as Ubuntu, openSUSE, Red Hat Enterprise Linux (aka RHEL), each targeting different use cases, on the other. For example, Ubuntu is great for home users, while RHEL is suitable for big enterprises with thousands of employees. Installing the raw Linux kernel can be cumbersome, but Ubuntu and RHEL all come with nice graphical installers that make it easy to install.

Similarly, each Kubernetes flavor has a target audience: K3s is great for hobbyists, while OpenShift is for big enterprises. RKE is great for teams who want to run and manage their own Kubernetes cluster in their own premises, and Minikube is great for developers. They are all much easier to install and launch than the raw Kubernetes project. And so on.

Added Value

Many providers add value to Kubernetes in different ways, usually as follows:

- Pre-configured storage options, ready to use.
- Security: users, groups, roles, etc.
- Built-in CI/CD functionality.
- Simplicity of installation and management.
- And more…

The reason they do this is because by default, Kubernetes does not provide any of these things; no user interface, no user management, no CI/CD, nothing at all. And to make things worse, it can be complicated to install it from scratch!

The canonical example of such added value is Red Hat OpenShift, which is touted as an enterprise-ready Kubernetes platform. It is 100% compatible with Kubernetes, but provides much more, like integrated user management, a CI/CD engine, "image streams", serverless features, etc.

Both OpenShift and the Rancher Management System provide a visual GUI, which is a full replacement for the YAML files mentioned above; this way, administrators can quickly operate on one or many clusters using a point-and-click

user interface. You don't need to write the files; just point and click. Some companies prefer that approach.

The simplest, smallest possible Kubernetes distribution is K3s, also created by Rancher, which is targeted for "Internet of Things" and "Edge" deployments, that is, small clusters running in small machines like Raspberry Pi.

There are other Kubernetes distributions that can run on a laptop: Minikube, Kind, Docker Desktop, MicroK8s, K3s / K3d, and AWS EKS Anywhere are examples. With a single command, a few seconds later you have a small cluster running in your laptop, where the nodes are simulated using virtual machines or containers running locally. This helps developers create applications for Kubernetes, as they can just run them locally without having to pay for an external cluster in a cloud provider.

Conformance

One of the jobs of the CNCF is to verify the conformance of all those Kubernetes flavors, so that they are certified; this is done automatically, through scripts, that verify each and every standard feature of the Kubernetes specification in each flavor of Kubernetes. For example, you can find the conformance results for K3s in GitHub, and be sure that it passes all tests.

This ensures that an application that could run on Minikube can also run on EKS; of course, this is not true the other way around. First, because your laptop running Minikube is much less powerful than the servers at AWS. And second, because many distributions extend Kubernetes with proprietary extensions. This means that an application specifically built with OpenShift in mind might not (and usually does not) work properly in K3s. Of course, doing so might provide economic benefits, but it also reduces portability; but this is another common tradeoff in our industry.

By using a certified Kubernetes distribution, organizations can be sure to find certified engineers, compatible tooling, and thus reduce the risk of lock-in. Theoretically, you are able to move your application from one provider to another with minimum effort, and that in itself is a strong incentive for businesses to use Kubernetes.

What's Next?

If you read until here, congratulations and thanks! There is so much material about Kubernetes that it is hard to decide where to go.

The links in the article might serve as guides for you to learn more about it, but if what you want is to get your hands dirty and "make your own cluster", you should watch this video: "i built a Raspberry Pi SUPER COMPUTER!! // ft. Kubernetes (k3s cluster w/ Rancher)" by NetworkChuck (published July 2021). It is very funny, very well explained, and it actually teaches you how to create

a cluster using the K3s Kubernetes distribution and one or many Raspberry Pi computers.