medium.com

Kubernetes 101: Pods, Nodes, Containers, and Clusters

Daniel Sanche

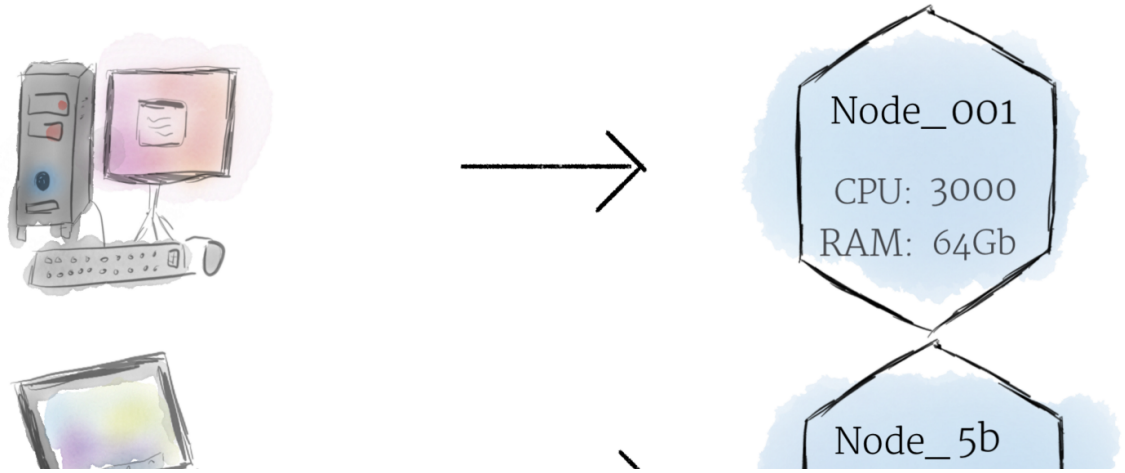
7-9 minutes

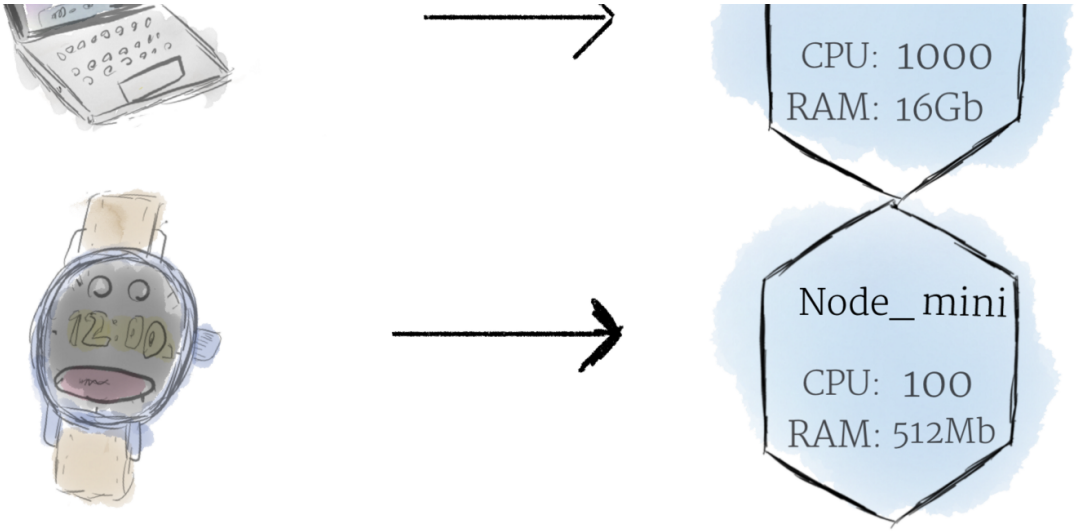
Kubernetes is quickly becoming the new standard for deploying and managing software in the cloud. With all the power Kubernetes provides, however, comes a steep learning curve. As a newcomer, trying to parse the official documentation can be overwhelming. There are many different pieces that make up the system, and it can be hard to tell which ones are relevant for your use case. This blog post will provide a simplified view of Kubernetes, but it will attempt to give a high-level overview of the most important components and how they fit together.

First, lets look at how hardware is represented

Hardware

Nodes

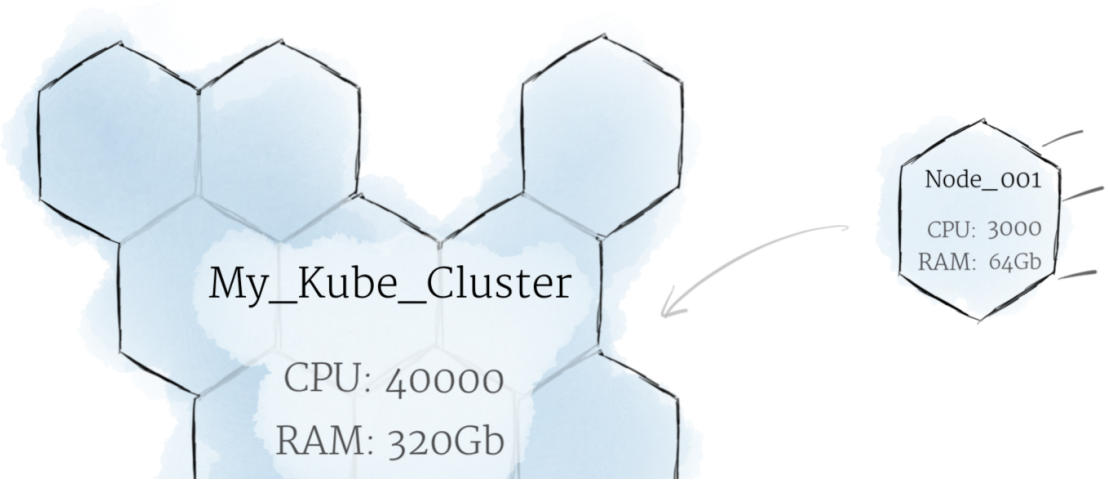


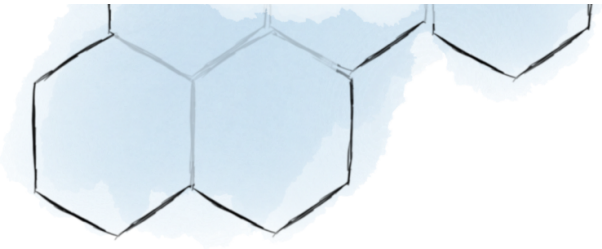


A node is the smallest unit of computing hardware in Kubernetes. It is a representation of a single machine in your cluster. In most production systems, a node will likely be either a physical machine in a datacenter, or virtual machine hosted on a cloud provider like Google Cloud Platform. Don't let conventions limit you, however; in theory, you can make a node out of almost anything.

Thinking of a machine as a "node" allows us to insert a layer of abstraction. Now, instead of worrying about the unique characteristics of any individual machine, we can instead simply view each machine as a set of CPU and RAM resources that can be utilized. In this way, any machine can substitute any other machine in a Kubernetes cluster.

The Cluster





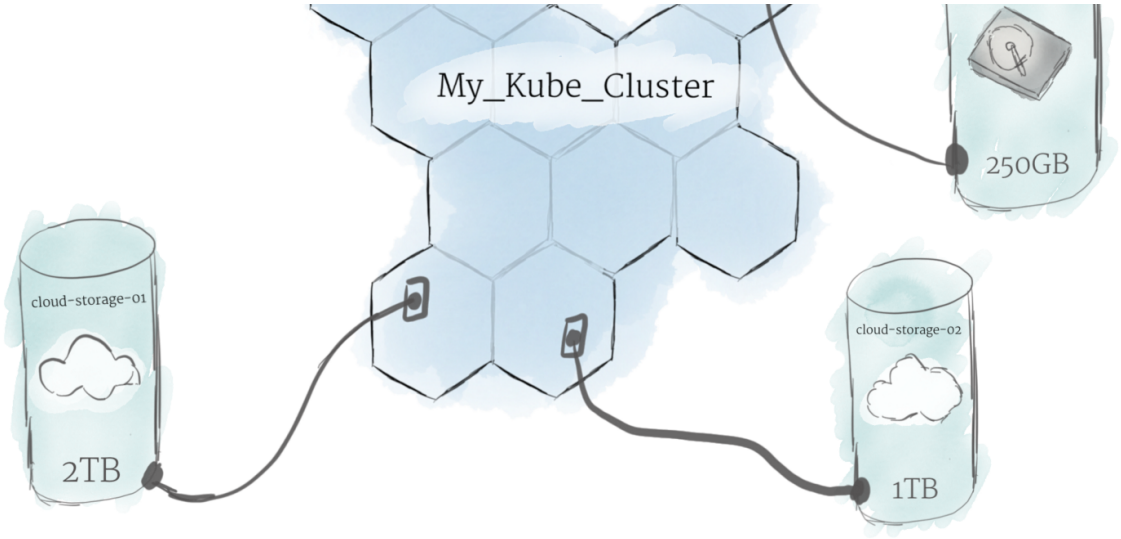
Although working with individual nodes can be useful, it's not the Kubernetes way. In general, you should think about the cluster as a whole, instead of worrying about the state of individual nodes.

In Kubernetes, nodes pool together their resources to form a more powerful machine. When you deploy programs onto the cluster, it intelligently handles distributing work to the individual nodes for you. If any nodes are added or removed, the cluster will shift around work as necessary. It shouldn't matter to the program, or the programmer, which individual machines are actually running the code.

If this kind of hivemind-like system reminds you of the Borg from Star Trek, you're not alone; "Borg" is the name for the internal Google project Kubernetes was based on.

Persistent Volumes

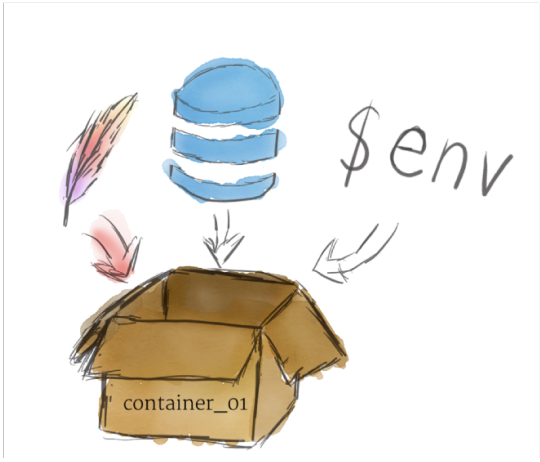
Because programs running on your cluster aren't guaranteed to run on a specific node, data can't be saved to any arbitrary place in the file system. If a program tries to save data to a file for later, but is then relocated onto a new node, the file will no longer be where the program expects it to be. For this reason, the traditional local storage associated to each node is treated as a temporary cache to hold programs, but any data saved locally can not be expected to persist.



To store data permanently, Kubernetes uses Persistent Volumes. While the CPU and RAM resources of all nodes are effectively pooled and managed by the cluster, persistent file storage is not. Instead, local or cloud drives can be attached to the cluster as a Persistent Volume. This can be thought of as plugging an external hard drive in to the cluster. Persistent Volumes provide a file system that can be mounted to the cluster, without being associated with any particular node.

Software

Containers



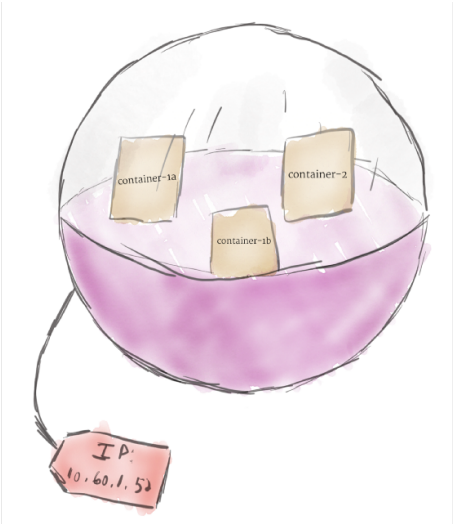
Programs running on Kubernetes are packaged as Linux containers. Containers are a widely accepted standard, so there are already many pre-built images that can be deployed on

Kubernetes.

Containerization allows you to create self-contained Linux execution environments. Any program and all its dependencies can be bundled up into a single file and then shared on the internet. Anyone can download the container and deploy it on their infrastructure with very little setup required. Creating a container can be done programmatically, allowing powerful CI and CD pipelines to be formed.

Multiple programs can be added into a single container, but you should limit yourself to one process per container if at all possible. It's better to have many small containers than one large one. If each container has a tight focus, updates are easier to deploy and issues are easier to diagnose.

Pods



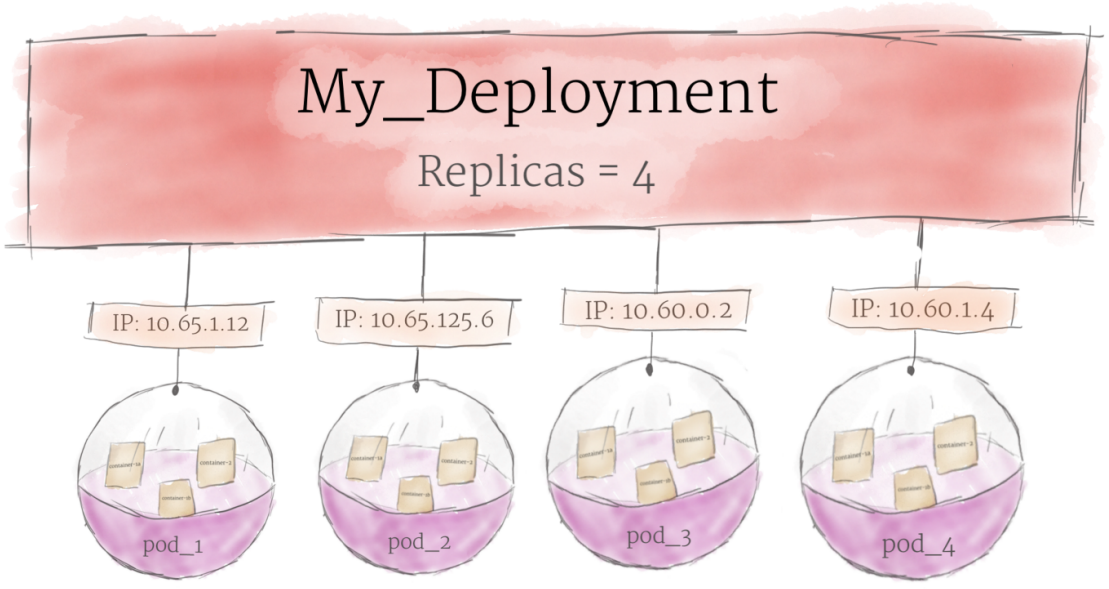
Unlike other systems you may have used in the past, Kubernetes doesn't run containers directly; instead it wraps one or more containers into a higher-level structure called a pod. Any containers in the same pod will share the same resources and local network. Containers can easily communicate with other containers in the same pod as though they were on the same machine while maintaining a degree of isolation from

others.

Pods are used as the unit of replication in Kubernetes. If your application becomes too popular and a single pod instance can't carry the load, Kubernetes can be configured to deploy new replicas of your pod to the cluster as necessary. Even when not under heavy load, it is standard to have multiple copies of a pod running at any time in a production system to allow load balancing and failure resistance.

Pods can hold multiple containers, but you should limit yourself when possible. Because pods are scaled up and down as a unit, all containers in a pod must scale together, regardless of their individual needs. This leads to wasted resources and an expensive bill. To resolve this, pods should remain as small as possible, typically holding only a main process and its tightly-coupled helper containers (these helper containers are typically referred to as "side-cars").

Deployments



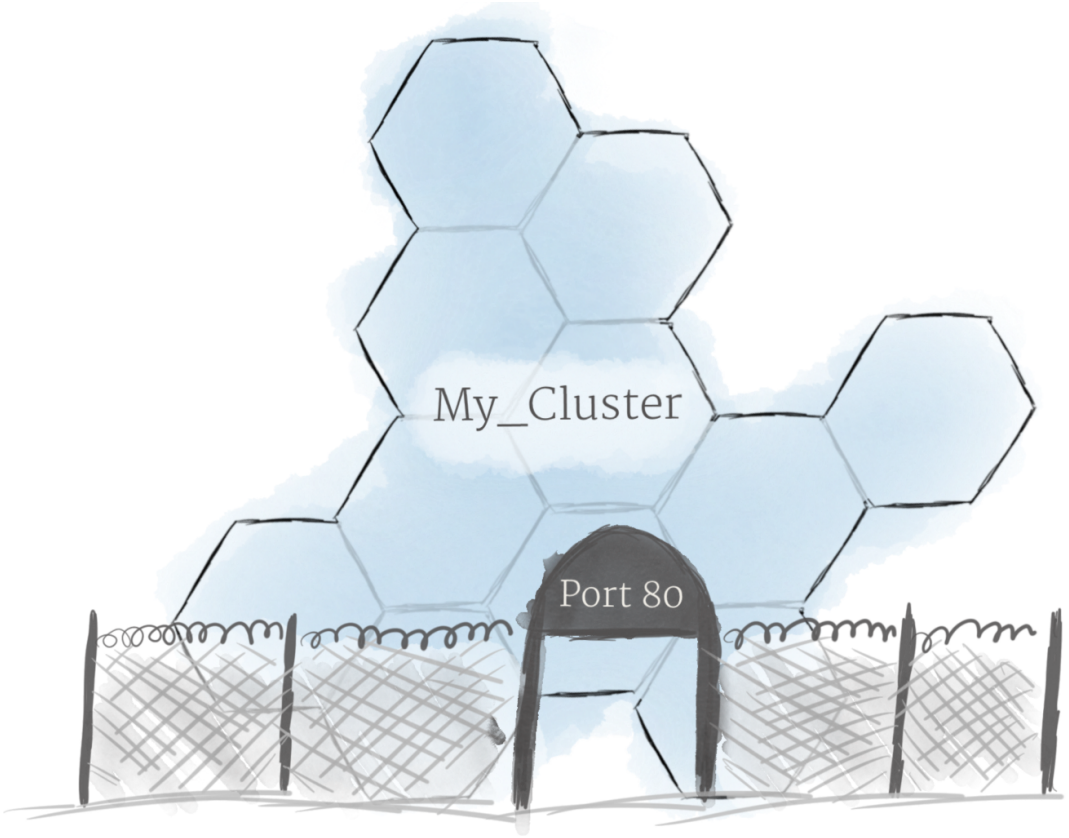
Although pods are the basic unit of computation in Kubernetes, they are not typically directly launched on a cluster. Instead, pods are usually managed by one more layer of abstraction: the

deployment.

A deployment's primary purpose is to declare how many replicas of a pod should be running at a time. When a deployment is added to the cluster, it will automatically spin up the requested number of pods, and then monitor them. If a pod dies, the deployment will automatically re-create it.

Using a deployment, you don't have to deal with pods manually. You can just declare the desired state of the system, and it will be managed for you automatically.

Ingress



Using the concepts described above, you can create a cluster of nodes, and launch deployments of pods onto the cluster. There is one last problem to solve, however: allowing external traffic to your application.

By default, Kubernetes provides isolation between pods and the outside world. If you want to communicate with a service

running in a pod, you have to open up a channel for communication. This is referred to as ingress.

There are multiple ways to add ingress to your cluster. The most common ways are by adding either an Ingress controller, or a LoadBalancer. The exact tradeoffs between these two options are out of scope for this post, but you must be aware that ingress is something you need to handle before you can experiment with Kubernetes.

What's Next

What's described above is an oversimplified version of Kubernetes, but it should give you the basics you need to start experimenting. Now that you understand the pieces that make up the system, it's time to use them to deploy a real app. Check out Kubernetes 110: Your First Deployment to get started.

To experiment with Kubernetes locally, Minikube will create a virtual cluster on your personal hardware. If you're ready to try out a cloud service ,Google Kubernetes Engine has a collection of tutorials to get you started.

If you are new to the world of containers and web infrastructure, I suggest reading up on the 12 Factor App methodology. This describes some of the best practices to keep in mind when designing software to run in an environment like Kubernetes.

Finally, for more content like this, make sure to follow me here on Medium and on Twitter (@DanSanche21).