# Kubernetes

# What is Kubernetes? Kubernetes is a container orchestration system that lets you deploy, scale, and manage containerized applications. It is also known in its abbreviated form as k8s.

Say you’ve got a bunch of Docker containers running on your computer. Maybe some of these containers need to talk to each other. Or maybe your friend Nancy is trying to access an API endpoint running on one of these containers. How do you ensure that the containers can talk to each other’s? How can you ensure that Nancy can hit that API endpoint securely? What happens when one of your containers dies? What happens if you need to scale your containers because all of a sudden, it’s not just Nancy needing to access that API endpoint…it’s Nancy and all her dev friends who are taking part a hackathon. Then what?

That’s where container orchestration comes in. It can help you with all that, and more. [Container orchestration](https://pvillela.com/2021/containers-kubernetes-and-service-mesh-in-a-nutshell/) is defined as the capability to define, deploy, and operate a compute cluster consisting of multiple virtual machines or physical servers to launch containers and manage their lifecycle.

Kubernetes’ competitors:

* [Docker Swarm](https://docs.docker.com/engine/swarm/)
* [Marathon](https://www.sumologic.com/blog/container-orchestration-mesos-marathon/)
* [Amazon ECS](https://aws.amazon.com/ecs/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc&ecs-blogs.sort-by=item.additionalFields.createdDate&ecs-blogs.sort-order=desc)
* [Azure Service Fabric](https://azure.microsoft.com/en-ca/services/service-fabric/)
* [Hashicorp Nomad](https://www.nomadproject.io/)

# Flavours of Kubernetes

## **Local**

* [KIND](https://kind.sigs.k8s.io/) (stands for **K**ubernetes **in** **D**ocker)
* [Docker Desktop](https://www.docker.com/products/kubernetes) (yes, it comes with Kubernetes, and you can run it locally)
* [Minikube](https://minikube.sigs.k8s.io/docs/start/)

## **Cloud Vendor Solutions**

Most of the major cloud vendors have their own flavours of k8s.

* [Google Kubernetes Engine](https://cloud.google.com/kubernetes-engine) (GKE)
* [Amazon Elastic Kubernetes Service](https://aws.amazon.com/eks/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc&eks-blogs.sort-by=item.additionalFields.createdDate&eks-blogs.sort-order=desc) (EKS)
* [Azure Kubernetes Service](https://docs.microsoft.com/en-us/azure/aks/) (AKS)

## **Enterprise Solutions**

For the enterprise-minded folks who like pretty admin GUIs

These tend to add a management layer on top of Kubernetes, catering to the enterprise crowd. As I mentioned earlier, they tend to have fancy admin consoles.

* [RedHat OpenShift](https://www.redhat.com/en/engage/container-platform-datasheet-20170814?sc_cid=7013a0000026GNBAA2&gclsrc=ds&gclsrc=ds)
* [VMWare Tanzu](https://tanzu.vmware.com/tanzu?utm_source=google&utm_medium=cpc&utm_campaign=amer_gp-b_a2&utm_content=g2_t023&utm_term=vmware%20tanzu&_bt=498180106968&_bk=vmware%20tanzu&_bm=e&_bn=g&_bg=119184092313)
* [Rancher](https://rancher.com/)

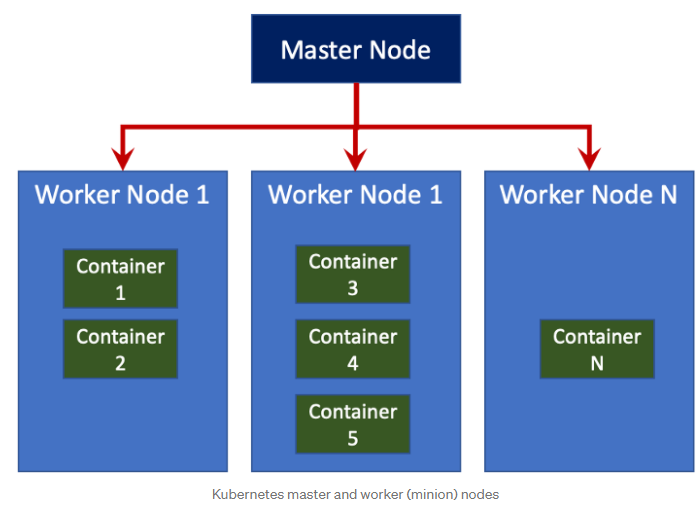
# Kubernetes related stuff

Everything in Kubernetes are called objects. They are all created by Kubernetes API. Lets take a look at the most common objects in Kubernetes to understand and get to know:

# Kubernetes Components:

A Kubernetes cluster consists of nodes divided into two groups:

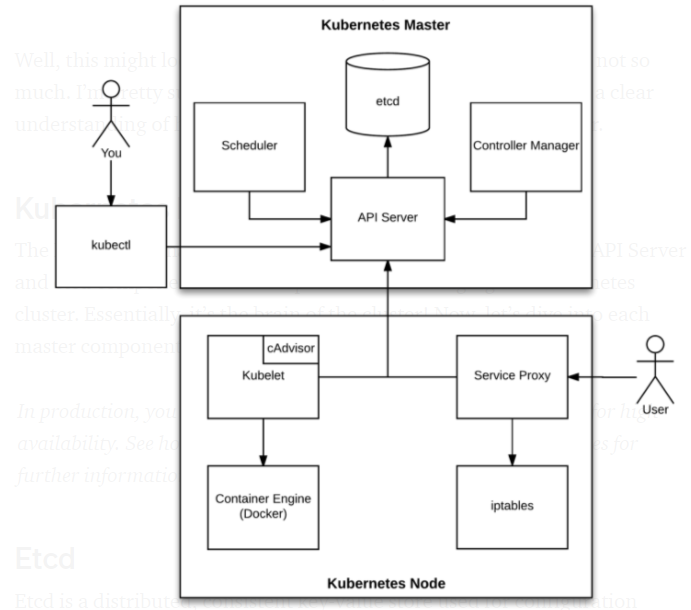
* A set of *master nodes* that host the *Control Plane* components, which are the brains of the system, since they control the entire cluster.
* A set of *worker* *nodes* that form the *Workload Plane*, which is where your workloads (or applications) run.

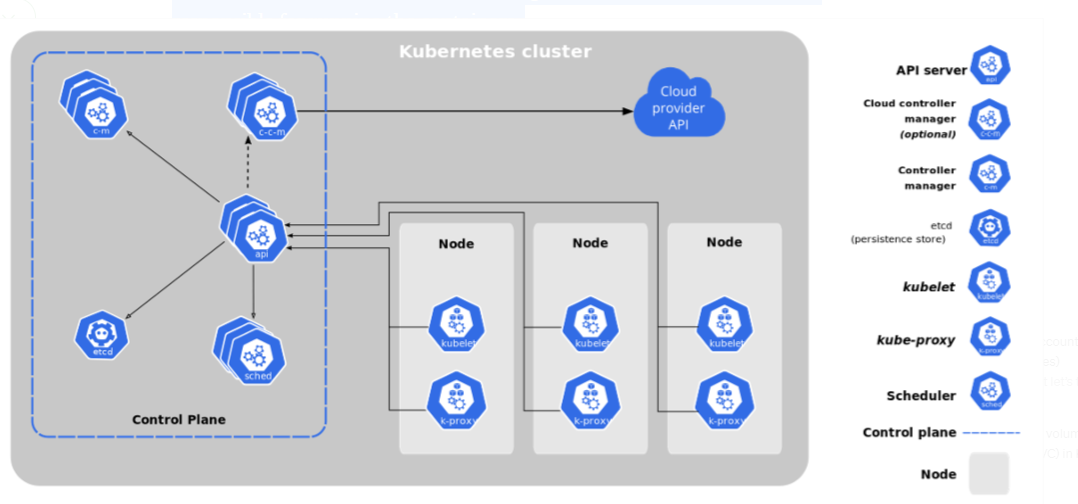


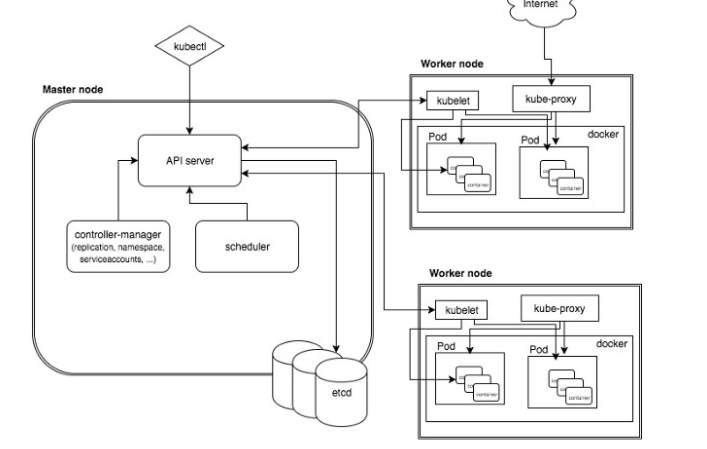
A Kubernetes cluster is made up of nodes. These nodes can be either physical machines, or virtual machines. You can have a cluster of one or more nodes, though ideally, you’ll want at least two nodes in a non-dev scenario.

A cluster typically has a master node, and a bunch of worker (or minion) nodes. The master node is responsible for watching the workers and performing the orchestration (think of what a manager does). The worker nodes are responsible for running the containers.

The diagram below shows us goes on inside the master node and worker nodes:



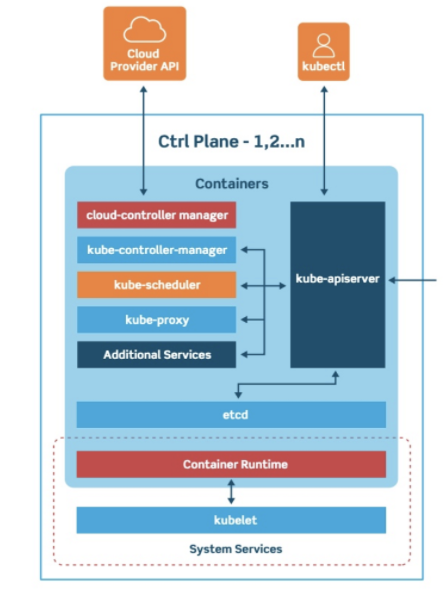




## **Master Node**

The master node is also known as a control plane that is responsible to manage worker/slave nodes efficiently. They interact with the worker node to

* Schedule the pods
* Monitor the worker nodes/Pods
* Start/restart the pods
* Manage the new worker nodes joining the cluster



As the manager of this whole operation, the master node has quite a few components:

* etcd
* API Server
* Controller Manager
* Scheduler

### **etcd**

[etcd](https://etcd.io/) is a distributed key-value database. It is Kubernetes’ source of truth. Every time you make a change to Kubernetes — e.g. by way of sending it YAML or JSON via k8s’ REST API or via the kubectl CLI tool (more on that later) — that change is stored in etcd as JSON. It is also versioned, so you also have some serious version control action going on.

When it comes to Kubernetes, etcd reliably stores the configuration data of the Kubernetes cluster, representing the state of the cluster (what nodes exist in the cluster, what pods should be running, which nodes they are running on, and a whole lot more) at any given point of time.

As all cluster data is stored in etcd, you should always have a backup plan for it. You can easily back up your etcd data using the etcdctl snapshot save command. In case you are running Kubernetes on AWS, you can also back up etcd by taking a snapshot of the EBS volume.

Etcd also implements a **watch** feature, which provides an event-based interface for asynchronously monitoring changes to keys. Once a key is changed, its “watchers” get notified. This is a crucial feature in the context of Kubernetes, as the API Server component heavily relies on this to get notified and call the appropriate business logic components to move the current state towards the desired state.

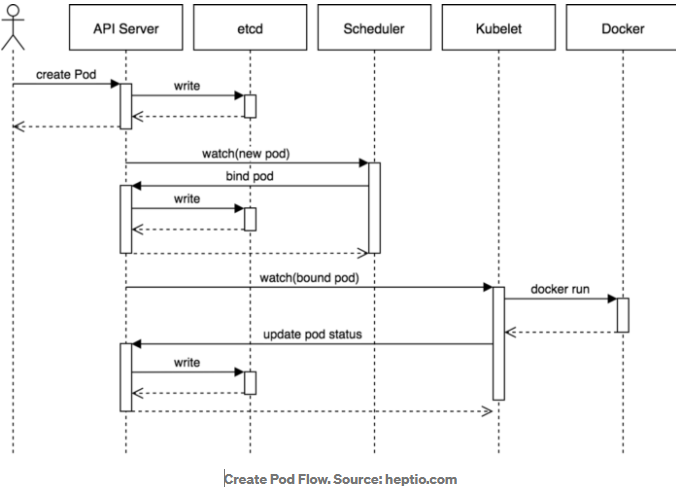
***Nerd alert:*** I personally think that etcd one of the coolest Kubernetes components. You can actually install etcd on your local machine ([OSX](https://brewinstall.org/install-etcd-on-mac-with-brew/) and [Ubuntu](https://computingforgeeks.com/how-to-install-etcd-on-ubuntu-18-04-ubuntu-16-04/), for example), and play around with it by using the [etcdctl](https://etcd.io/docs/v3.4/dev-guide/interacting_v3/) CLI tool. Python even has a few libraries for interacting etcd programmatically. I’ve personally played around with the [Python etcd3 library](https://pypi.org/project/etcd3/), and I highly recommend exploring etcd for yourself!

### **API Server**

The API server is responsible for serving up the Kubernetes API. Wanna talk to Kubernetes and tell it to do things for you? This is is your direct line — whether you’re a user, a program, or kubectl. The API Server is also what’s responsible for sending data to and pulling data from etcd.

The API Server is the only Kubernetes component that connects to etcd; all the other components must go through the API Server to work with the cluster state.

For example, when you create a pod using kubectl, this what happens:



1. kubectl writes to the API Server.
2. API Server validates the request and persists it to etcd.
3. etcd notifies back the API Server.
4. API Server invokes the Scheduler.
5. Scheduler decides where to run the pod on and return that to the API Server.
6. API Server persists it to etcd.
7. etcd notifies back the API Server.
8. API Server invokes the Kubelet in the corresponding node.
9. Kubelet talks to the Docker daemon using the API over the Docker socket to create the container.
10. Kubelet updates the pod status to the API Server.
11. API Server persists the new state in etcd.

### **Controller Manager**

The controller manager is the brain behind the orchestration. Kubernetes has multiple controllers, each responsible for different things. Controllers watch the state of your cluster, then make or request changes where needed. The controller manager makes sure that it tells the right controllers to do the right things.

Some examples of controllers that ship with Kubernetes include the Replication Controller, Endpoints Controller, and Namespace Controller

For example, there are controllers for:

Taking action when a pod goes down

Connecting services to pods

Creating accounts and accessing API tokens

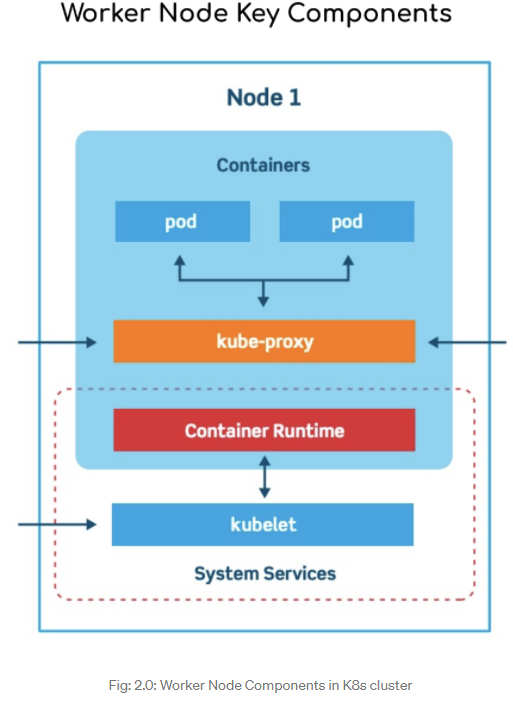
And many others…

**Note:** In case you’re wondering what a pod is, it’s basically a wrapper around one or more containers.

### **Scheduler**

The scheduler distributes work across multiple nodes. Its looks at resource requirements (e.g. CPU and memory requirements) to figure out when to run a pod, and what node to run it on.

## **Worker Nodes**



Worker nodes contain 2 main components:

* **Kubelet**
* **Kube-proxy**

### **Kubelet**

The kubelet is an agent (small app) that runs on each worker node in the cluster. Its main job is to make sure that containers are running in a pod (wrapper of one or more containers). But who tells it to run these containers? That comes from the control plane (where our good friend the controller manager resides). When the control plane needs something to happen in a node, the kubelet makes it happen.

The kubelet runs a container runtime, and, as its name implies, is responsible for actually running containers. More specifically, it manages the complete lifecycle of a container: container image pulling (from a container registry such as [Docker Hub](https://hub.docker.com/)) and storage, container execution, network attachment, etc. [Docker](https://www.docker.com/) is a popular container runtime; however, there are others, such as [containerd](https://containerd.io/), [CRI-O](https://cri-o.io/).

Within a Kubernetes cluster, the **kubelet** watches for **PodSpecs** via the Kubernetes API server.

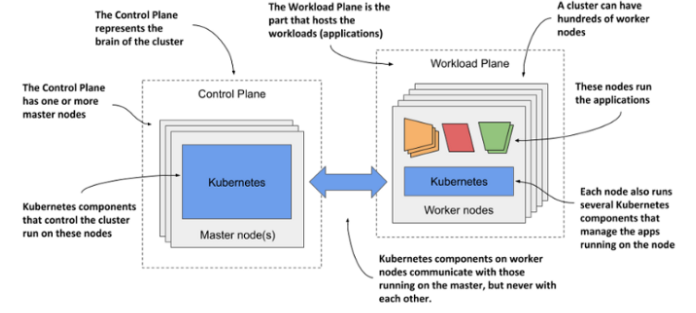
**A PodSpec is a YAML or JSON** object that describes a pod. The **kubelet** takes a set of PodSpecs that are provided through various mechanisms (primarily through the **API server**) and ensures that the containers described in those PodSpecs are running and healthy.

The Kubelet is the primary and most important controller in Kubernetes. It’s responsible for driving the container execution layer, typically Docker.

### **Kube-proxy**

The kube-proxy handles network communications inside and outside your cluster. This means that if pods need to talk to each other, or if some external service needs to talk to a pod, kube-proxy helps make that happen.

kube-proxy has an intelligent algorithm to forward network traffics required for pod access which minimized the overhead and makes service communication more performant



*The two planes together make up the Kubernetes cluster.*

# Resources, Controllers, and Operators

let’s get into some other key Kubernetes concepts and terminology:

## **Resources**

A Kubernetes [resource](https://kubernetes.io/docs/reference/using-api/api-concepts/) refers to either an object or operation in Kubernetes, accessed via the Kubernetes API. A resource type is known as a kind, and is represented as JSON object. This JSON object is stored (and versioned) in our good friend, etcd.

There are two categories of resources: primitive resources, and custom resources. Primitive resources come “out of the box” with Kubernetes. Primitive resources include Pod, Service, Deployment, ServiceAccount, PersistentVolumeClaim, RoleBinding…I could go on.

## **Controllers**

As discussed earlier, a controller watches the state of your cluster, then makes or requests changes where needed, to achieve the desired state.

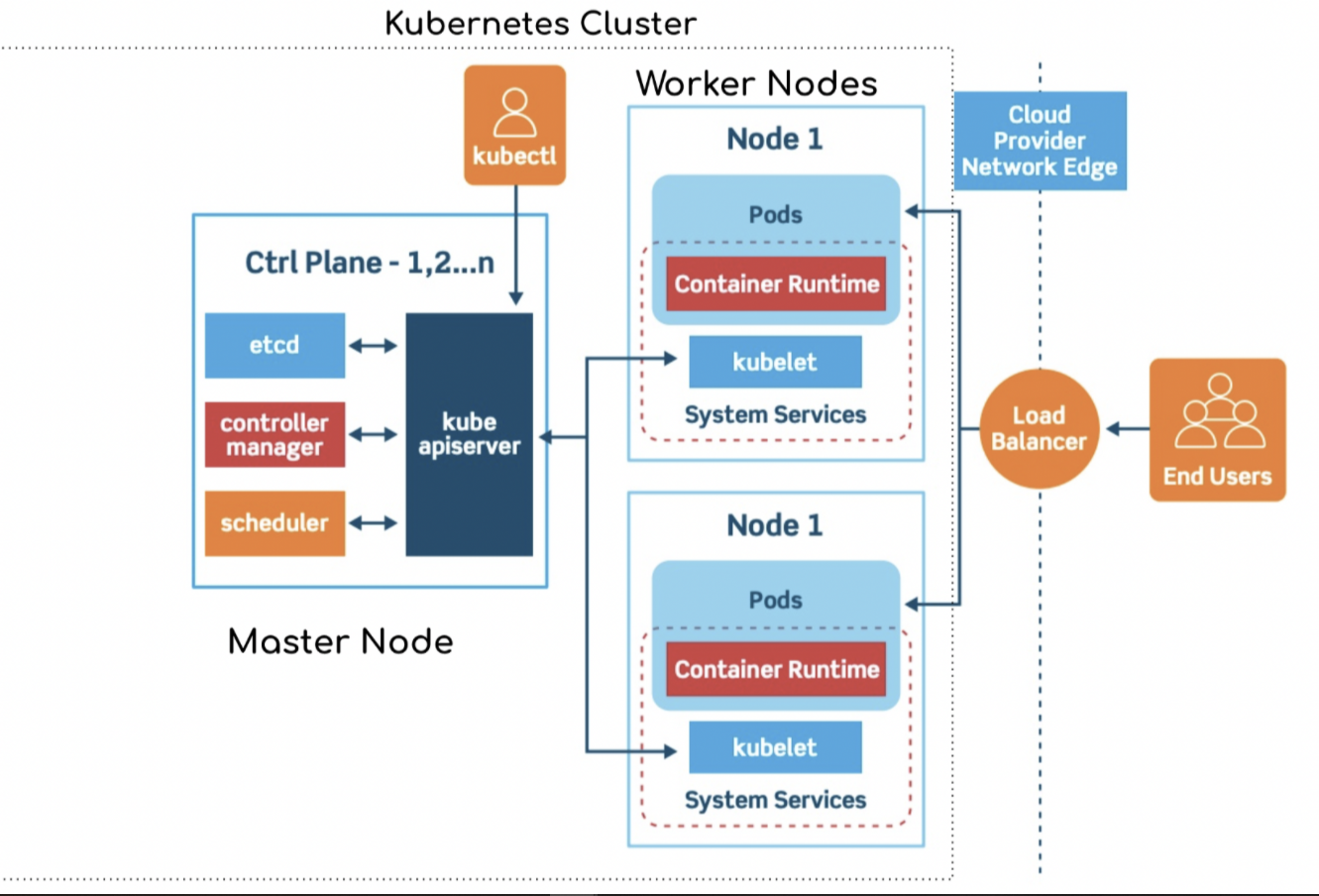
## **Operators**

An operator is a type of controller. Remember those custom resources that I talked about earlier? It’s all well and good to define a custom resource, but at the end of the day, how do you get Kubernetes to do something useful with it? That’s where operators come in — they’re the code behind the scenes that make those custom resources *do*that useful something.

***Fun fact:*** Operators can be written in any language, and there are a few frameworks out there that set up some boilerplate code to help you write your own operator.

# kubectl

kubectl is a command-line interface (CLI) for managing operations on Kubernetes clusters. That’s it! It communicates with our good friend, the API Server, to get information about our cluster, and to tell Kubernetes to do stuff for us, like create a new resource, or modify an existing one. As I’ve mentioned before, when you extend the Kubernetes API using the magical combination of operators and custom resource definitions (CRDs), you can use kubectl to access/update those resources too!



# Anatomy of POD

* **Pods** are designed to run multiple cooperative processes that should act as a cohesive unit. Those processes are wrapped in containers.
* All the containers that form a Pod are running on the same machine. A Pod cannot be split across multiple nodes.
* All the processes (containers) inside a Pod share the same set of resources, and they can communicate with each other through localhost. One of those shared resources is storage.
* A volume (think of it as a directory with shareable data) defined in a Pod can be accessed by all the containers thus allowing them all to share the same data.

|  |
| --- |
| There are three things which are common across all yml files, those are namely: **apiVersion, kind, metadata** |

# Labels in Kubernetes

Kubernetes labels allow DevOps teams to identify Kubernetes objects and organize them into groups. One good use-case of this is grouping pods based on the application they belong to. Teams can also develop any number of labeling conventions including ones to group pods by environment, customer, team/owner or release version.

To get the labels applied to a created pod :

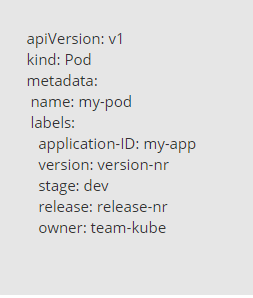
kubectl get pod <my-pod-name> --show-labels

Add a label to a pod using Kubectl:

kubectl label pod <my-pod-name> versionID=ver0.9

Update a label for a pod using Kubectl:

kubectl label --overwrite pods my-pod team=ops



## recommended Kubernetes labels

app.kubernetes.io/name

app.kubernetes.io/instance

app.kubernetes.io/version

app.kubernetes.io/component

app.kubernetes.io/part-of

app.kubernetes.io/managed-by

## Differentiate between Kubernetes labels vs annotations

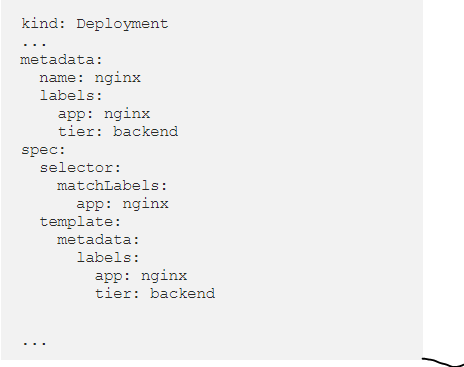
Kubernetes labels and annotations are both ways of adding metadata to Kubernetes objects. The similarities end there, however. Kubernetes labels allow you to identify, select and operate on Kubernetes objects. Annotations are non-identifying metadata and do none of these things.

Annotations allow you to add non-identifying metadata to Kubernetes objects. Examples include phone numbers of persons responsible for the object or tool information for debugging purposes. In short, annotations can hold any kind of information that is useful and can provide context to DevOps teams.

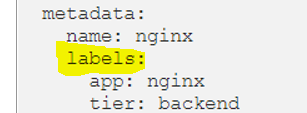
Example:



# matchLabels, labels, and selectors explained in detail



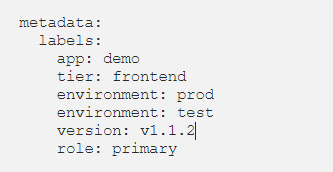
this is kind of confusing. What does the selector, label, and matchLabel do? And ***why are there multiple nested labels?***

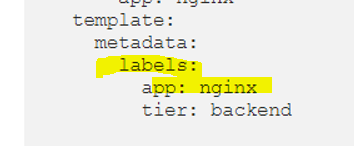


*The first metadata describes the deployment itself.* It gives a label *for that actual deployment*. So, if you want to delete that deployment, you would say kubectl delete -l app=nginx,tier=backend

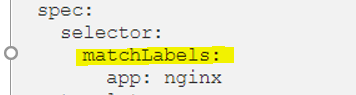
This describes the deployment itself. It gives a label for that actual deployment. So, if you want to delete that deployment, you would say kubectl delete -l app=nginx,tier=backend.

A label can have any as many key and value specifications as you want.



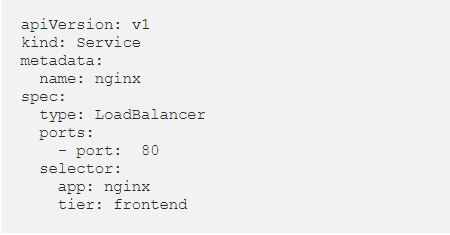


From the above, the template is actually a podTemplate. It describes a pod that is launched. One of the fields for the pod templates is replicas. If we set replicas to 2, it would make 2 pods for that deployment, and the deployment would entail both of those pods. So, that template for the pod has a label. So, this isn’t a label for the deployment anymore, it’s a label for the pod that the deployment is deploying. That’s a subtle, but important distinction.



Here is you can see **selector** tag, which is used by deployment to talk with its pods. The selector field defines how the Deployment finds which Pods to manage.

But anyway, the selector: matchLabels tells the resource, whatever it may be, service, deployment, etc, to match the pod, according to that label. So, maybe a potential reason is for uniformity. For instance, when you want to apply a service (to expose the pod to the web or something), you need to apply that service to the pod with a match label, like this:



So, matchLabels are not supported by Service, but only certain new resources like Deployment. If you see in above yaml for service resource there is no match labels , this is another confusing part.

why sometimes it’s matchLabels, and somtimes it’s just selector with a map. Only [**Job**](https://kubernetes.io/docs/concepts/jobs/run-to-completion-finite-workloads/), [**Deployment**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/), [**Replica Set**](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/), and [**Daemon Set**](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/)support matchLabels.

# Health Checks (Kubernetes Probes):

Kubernetes provides a **health checking**mechanism to verify if a container in a pod is working or not working.

## **Liveness, Readiness and Startup Probes:**

The [kubelet](https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/) uses liveness probes to know when to restart a container. For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a container in such a state can help to make the application more available despite bugs.

The kubelet uses readiness probes to know when a container is ready to start accepting traffic. A Pod is considered ready when all of its containers are ready. One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.

The kubelet uses startup probes to know when a container application has started. If such a probe is configured, it disables liveness and readiness checks until it succeeds, making sure those probes don't interfere with the application startup. This can be used to adopt liveness checks on slow starting containers, avoiding them getting killed by the kubelet before they are up and running.

## **When should application developers use Liveness Probes?**

— If the application starts up in a few seconds or less, then a liveness probe is probably unnecessary. If it takes more than a few seconds, it is recommended to put in a liveness probe to ensure that the container initializes without error rather than crashing.

— If the process in your container is able to crash on its own whenever it encounters an issue or becomes unhealthy, it is okay to not have a liveness probe. The kubelet will automatically perform the correct action in accordance with the Pod’s restartPolicy.

— **To handle cases in which container is not responsive/hung and needs to be killed + restarted, a liveness probe is highly recommended** along with the Pod’s ‘restartPolicy’ set to Always or OnFailure.

## **When should application vendors use Readiness Probes?**

When a new pod is created, its services are **not**immediately registered in cluster’s load balancer and so they’re not allowed to serve external traffic (services running inside the same cluster still can access them). This way Kubernetes can prevent an instance of a service to receive requests when it is still loading or too busy to process them properly. **Readiness probes**are used to check when the containers running in that pod are (guess what?) ready and able to receive external requests.

Probes are defined inside container’s definition in the .yaml resource file used to create the pod:

apiVersion: v1  
kind: Pod  
metadata:  
 (...)  
spec:  
 containers:  
 - name: my-awesome-container  
 image: some-awesome-image  
 (...)  
 ***<probe-type>*:  
 *<probe-action>*: (...)  
 initialDelaySeconds: 5  
 periodSeconds: 10  
 timeoutSeconds: 3  
 successThreshold: 2  
 failureThreshold: 4**

**initialDelaySeconds:**

Amount of time in seconds to wait before executing the probe for the first time. This interval is particularly good when we have applications with lots of dependencies and/or large loading time. If this property is not set, the probe will be executed as soon as the container is loaded.

**periodSeconds** :

We can also set the interval between one execution and another, in seconds, using the **periodSeconds** property. If not set, the probe will be executed every 10 seconds.

**timeoutSeconds:**

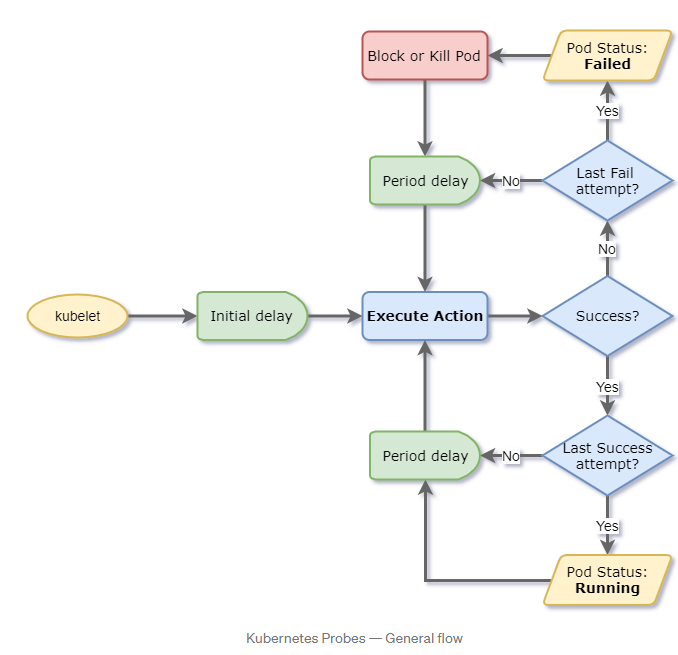
After this initial delay, the probe is executed and *kubelet* waits for a certain amount of time for a result before assuming a fail for timeout

**failureThreshold**:

If the probe fails, *kubelet* tries again how many times as set here. this way any temporary unavailability won’t cause the probe to put the container in a failed state.

**successThreshold** :

In some situations, a single successful result may not be enough to really ensure the health of our container. In this case, the **successThreshold** property sets how many consecutive times the action needs to be successful to change the state of a container from failed to successful.



# Deployment:

Many times while working on your production server, you may encounter an instance where you would like to push a new update of your web application, while doing so it is desired not to break the running web app and to avoid any downtime. You may be required to deploy many instances of your web application to manage user workloads and to ensure the application is always available for the user to access.

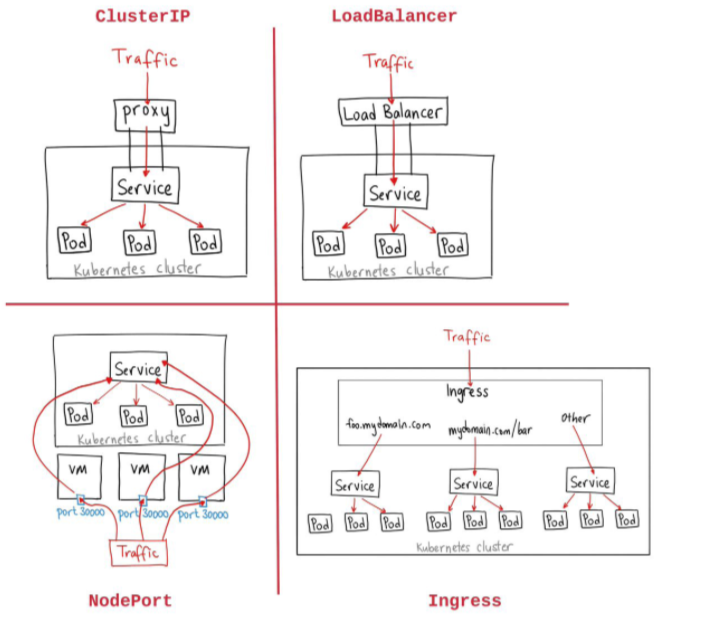
Kubernetes **Deployment** here is your true mate which helps you to seamlessly perform your application rolling update. It also helps you to scale the web app horizontally by helping you create multiple replicas of the web application.

**So in the Kubernetes ecosystem, A Deployment provides declarative updates for Pods and ReplicaSets without breaking the user side experience.**

Deployment act as a controller in Kubernetes which helps Kubernetes developers and administrator achieve the desired state for pods, replica sets in the manner decided by the controller.

# Deployment vs service:

# Services:



## **Getting Started with Communication:**

Pods are the smallest unit in Kubernetes and have a relatively short life-span. They are born, and they are destroyed. They are never healed. The system heals itself by creating new Pods (cells) and by terminating those that are unhealthy or those that are surplus. The system is long-living, Pods are not.

*Controllers*, together with other components like the *scheduler*, are making sure that the Pods are doing the right thing. They control the scheduler. We used only one of them so far.

ReplicaSet is in charge of making sure that the desired number of Pods is always running. If there’s too few of them, new ones will be created. If there’s too many of them, some will be destroyed. Pods that become unhealthy are terminated as well. All that, and a bit more, is controlled by ReplicaSet.

**The problem with our current setup is that there are no communication paths. Our Pods cannot speak with each other. So far, only containers inside a Pod can talk with each other through localhost.** That led us to the design where both the API and the database needed to be inside the same Pod. That was a lousy solution for quite a few reasons.

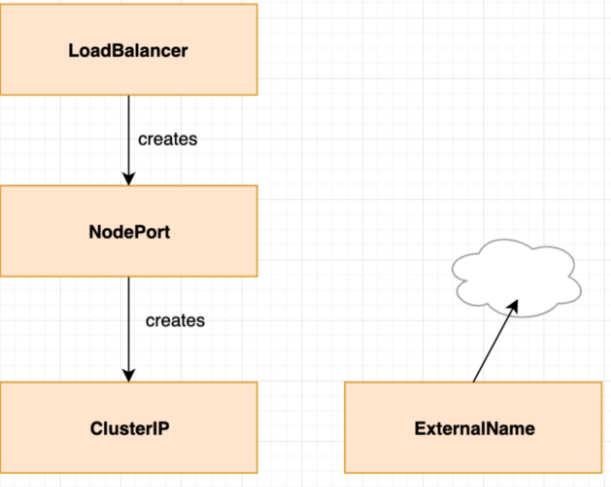
The main problem is that we cannot scale one without the other. We could not design the setup in a way that there are, for example, three replicas of the API and one replica of the database. The primary obstacle was communication.

Truth be told, each Pod does get its own address. We could have split the API and the database into different Pods and configure the API Pods to communicate with the database through the address of the Pod it lives in.

However, since Pods are unreliable, short-lived, and volatile, we cannot assume that the database would always be accessible through the IP of a Pod. When that Pod gets destroyed (or fails), the ReplicaSet would create a new one and assign it a new address.

We need a stable, never-to-be-changed address that will forward requests to whichever Pod is currently running.

## **Kubernetes Services simply visually explained**



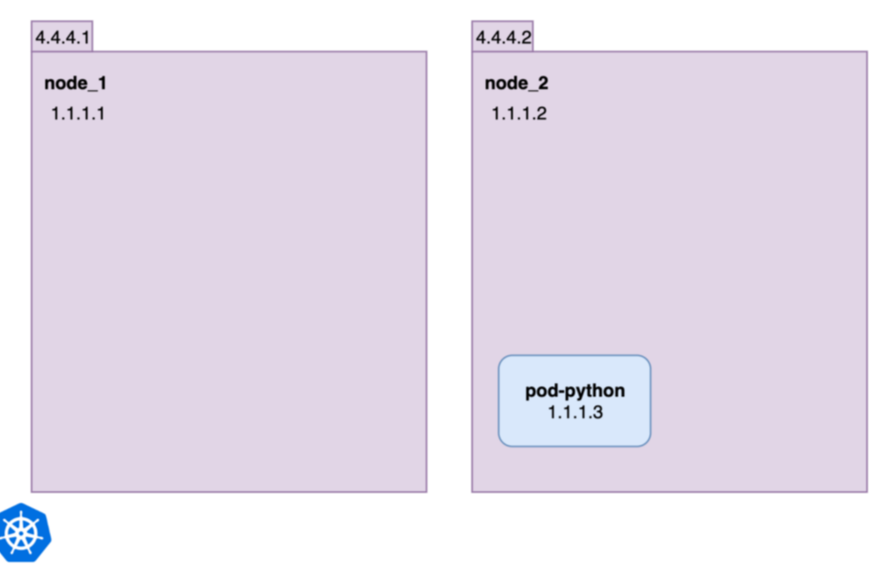
I would like you to imagine that if you create a NodePort service it also creates a ClusterIP one. And if you create a LoadBalancer it creates a NodePort which then creates a ClusterIP.

### **Services and Pods:**

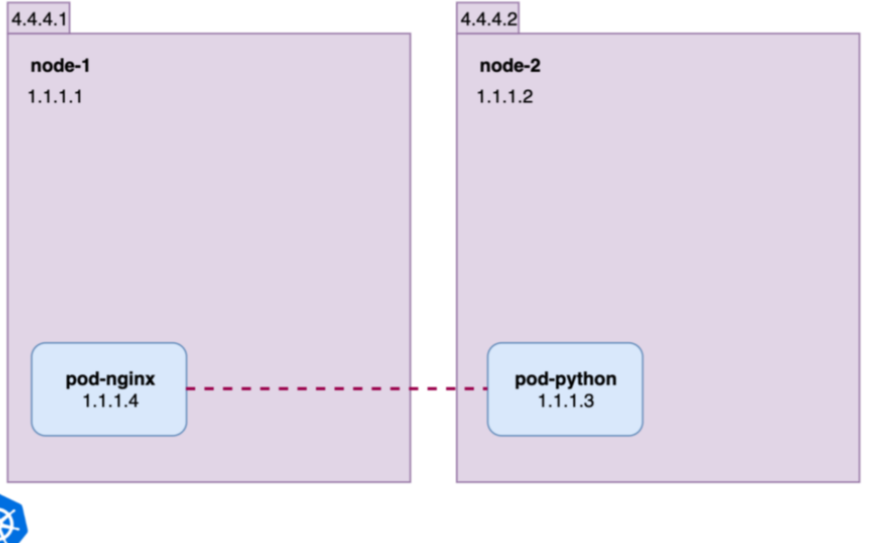
Services point to pods. Services do **not** point to deployments or replicasets. Services point to pods directly using labels. This gives great flexibility because it doesn’t matter through which various (maybe even customized) ways pods have been created.

Step1:

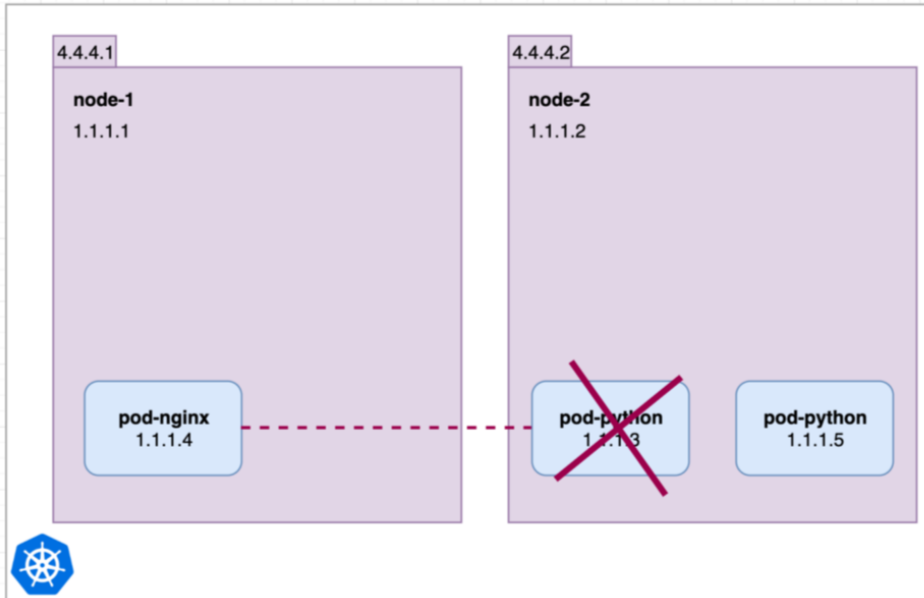
We start without any services.



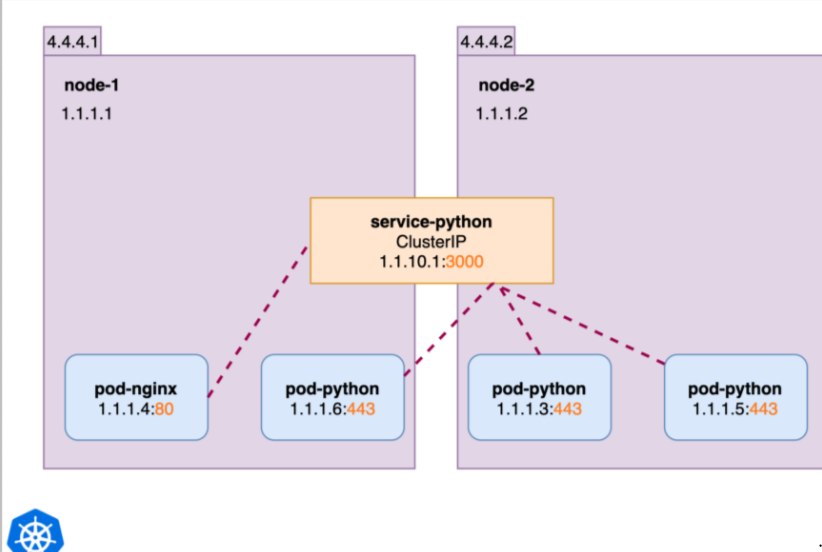
We have two nodes, one pod. Nodes have external (4.4.4.1, 4.4.4.2) and internal (1.1.1.1, 1.1.1.2) IP addresses. The pod **pod-python** has only an internal one.



Now we add a second pod **pod-nginx** which got scheduled on **node-1**. This wouldn’t have to be the case and doesn’t matter for connectivity. In Kubernetes, all pods can reach all pods on their internal IP addresses, no matter on which nodes they are running. This means **pod-nginx** can ping and connect to **pod-python** using its internal IP 1.1.1.3.



Now let’s consider the **pod-python** dies and a new one is created. (We don’t handle how pods might be managed and controlled in this article.) Suddenly **pod-nginx** cannot reach 1.1.1.3 any longer, and suddenly the world bursts into horrific flames… but to prevent this we create our first service!



We extend the example, spin up 3 instances of python and we now display the ports of the internal IP addresses of all pods and services.

All pods inside the cluster can reach the python pods on their port 443 via http://1.1.10.1:3000 or http://service-python:3000. The ClusterIP **service-python** distributes the requests based on a random or round-robin approach. That’s what a ClusterIP service does, it makes pods available inside the cluster via a name and an IP.

The **service-python** in the above image could for have this yaml:

**apiVersion:** v1  
**kind:** Service  
**metadata:**  
 **name:** service-python  
**spec:  
 ports:** - **port:** 3000  
 **protocol:** TCP  
 **targetPort:** 443  
 **selector:  
 run:** pod-python  
 **type:** ClusterIP

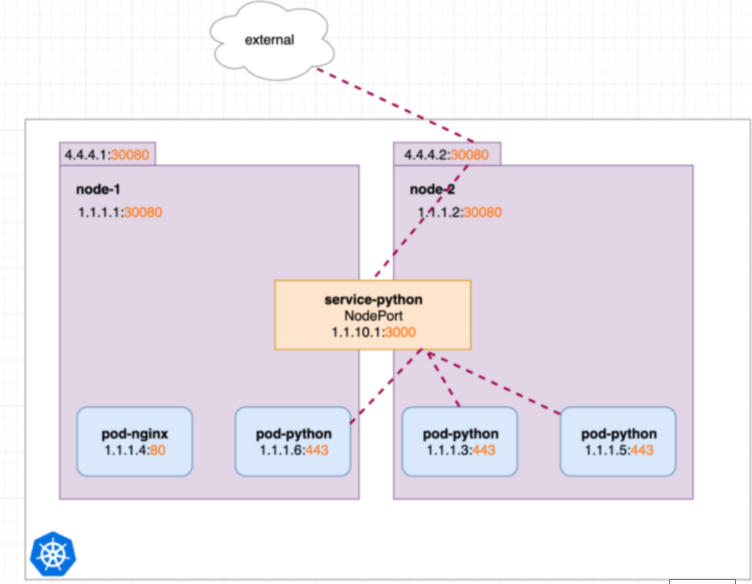
Running kubectl get svc :



### **NodePort:**

Now we would like to make the ClusterIP service available from the outside and for this we convert it into a NodePort one. In our example we convert the **service-python** with just two simple yaml changes:

apiVersion: v1  
kind: Service  
metadata:  
 name: service-python  
spec:  
 ports:  
 - port: 3000  
 protocol: TCP  
 targetPort: 443  
 **nodePort: 30080**  
 selector:  
 run: pod-python  
 **type: NodePort**



This means our internal **service-python** will now also be reachable from every nodes internal and external IP address on port 30080.

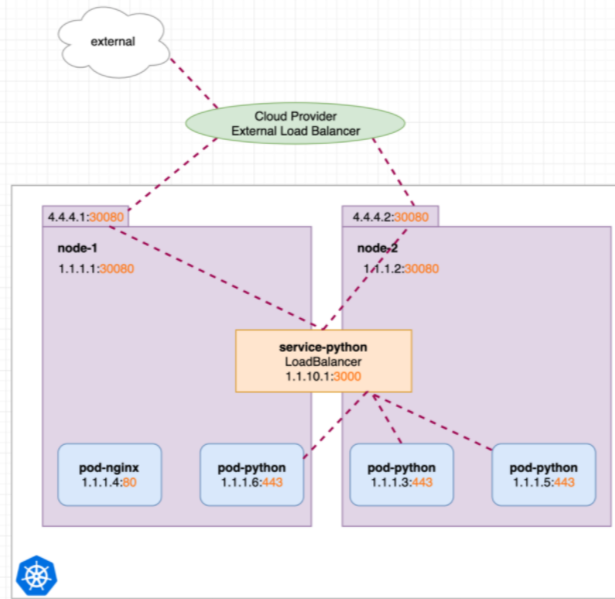
A pod inside the cluster could also connect to an internal node IP on port 30080.

Running kubectl get svc shows the same cluster ip. Just the different type and additional node port:



### **LoadBalancer**

We use a LoadBalancer service if we would like to have a single IP which distributes requests (using some method like round robin) to all our external nodes IPs. So it is built on top of a NodePort service:



Imagine that a LoadBalancer service creates a NodePort service which creates a ClusterIP service. The changed yaml for LoadBalancer as opposed to the NodePort before is simply:

apiVersion: v1  
kind: Service  
metadata:  
 name: service-python  
spec:  
 ports:  
 - port: 3000  
 protocol: TCP  
 targetPort: 443  
 nodePort: 30080  
 selector:  
 run: pod-python  
 **type: LoadBalancer**

All a LoadBalancer service does is it creates a NodePort service. In addition it sends a message to the provider (AWS, Azure, GCP) who hosts the Kubernetes cluster asking for a loadbalancer to be setup pointing to all external node IPs and specific nodePort. If the provider doesn’t support the request message, well then nothing happens and the LoadBalancer would be equal to a NodePort service.

Running kubectl get svc shows just the addition of the EXTERNAL-IP and different type:

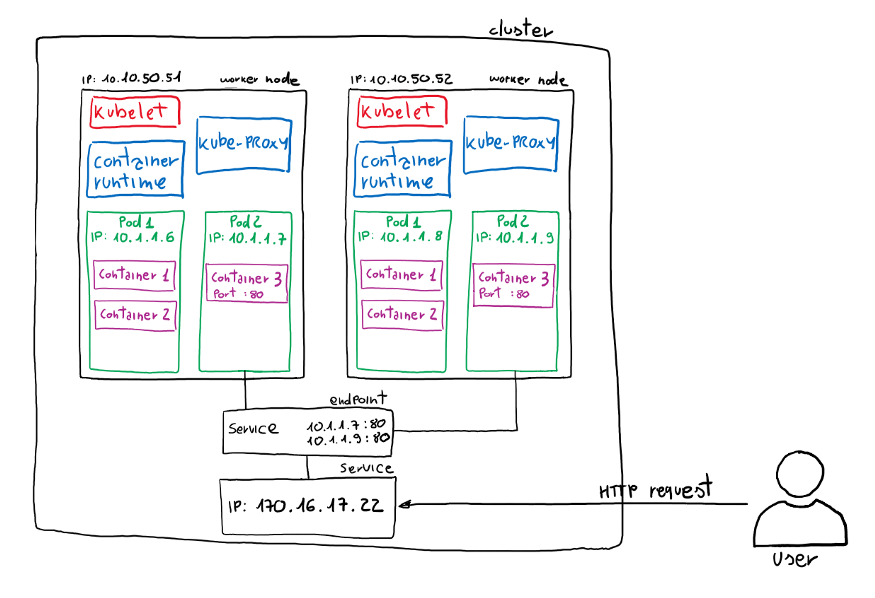


The LoadBalancer service still opens port 30080 on the nodes internal and external IPs as before. And it still acts like a ClusterIP service.

## **Kubernetes Endpoints**

Services as objects with a static IP that forwards the requests to pods whose selector contains a label described in its YAML description.

However, a service does not link straight to pods. When Kubernetes processes a service description, and if the service selector matches a pod label, Kubernetes will automatically create an **Endpoints** object with the same name as the service, which stores the pod’s IP address and port. Consequently, when the service receives a request, its proxy will redirect it to one of those IPs and ports.



In this example, I will deploy three pod replicas and a service in which the selector will contain one of the pod’s labels.

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: training-deployment  
spec:  
 **selector:  
 matchLabels:  
 app: training  
 version: v1** replicas: 3  
 template:  
 metadata:  
 labels:  
 app: training  
 version: v1  
 spec:  
 containers:  
 - name: training-container  
 image: training:1.0  
 ports:  
 - containerPort: 80  
---  
apiVersion: v1  
kind: Service  
apiVersion: v1  
metadata:  
 name: training-service  
spec:  
 **selector:  
 app: training** type: ClusterIP  
 ports:  
 - name: training-port  
 port: 8080  
 targetPort: 80

Execute the file:

kubectl create -f deployment.yaml

Find the endpoints:

kubectl get endpoints

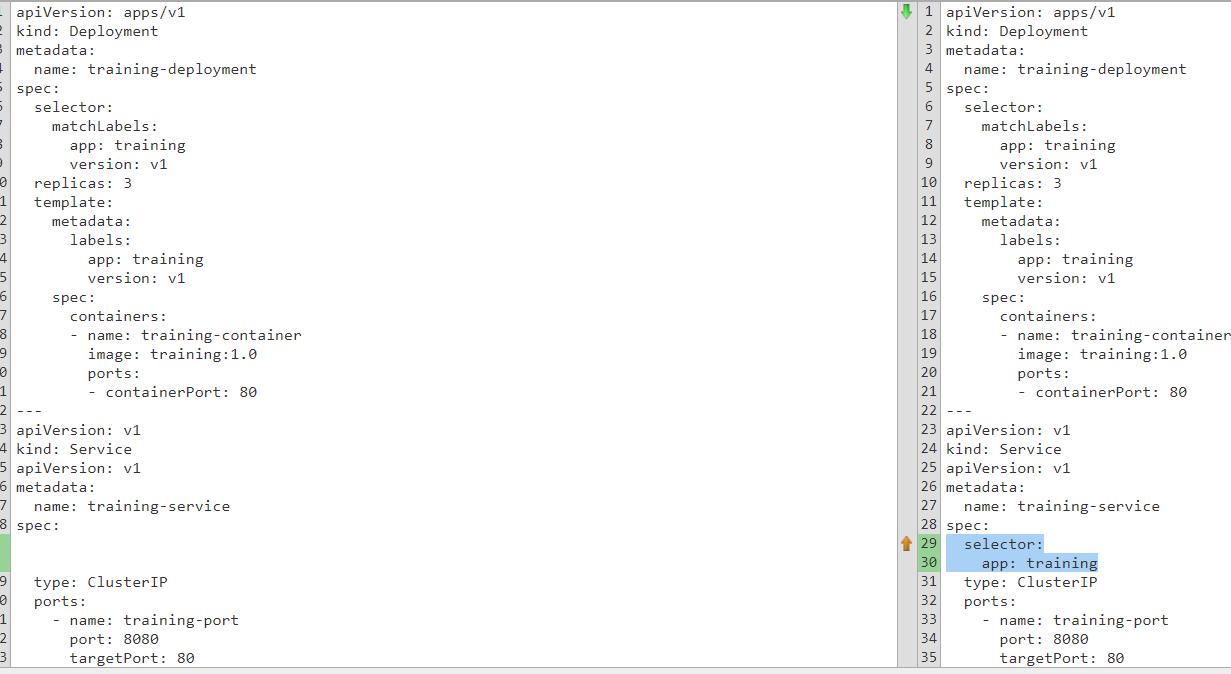
NAME ENDPOINTS AGE  
kubernetes 192.168.65.3:6443 12d  
training-service 10.1.0.52:80,10.1.0.53:80,10.1.0.54:80 22s

## **Decouple services from endpoints**

if you create a service without specifying a pod selector, Kubernetes will not generate the endpoint resource and it will be up to you to create one with the endpoints to link to the service. As mentioned before, this endpoint might have the same name as the service.

For example:

Lets rewrite the yaml file above and don’t specify the selector app.

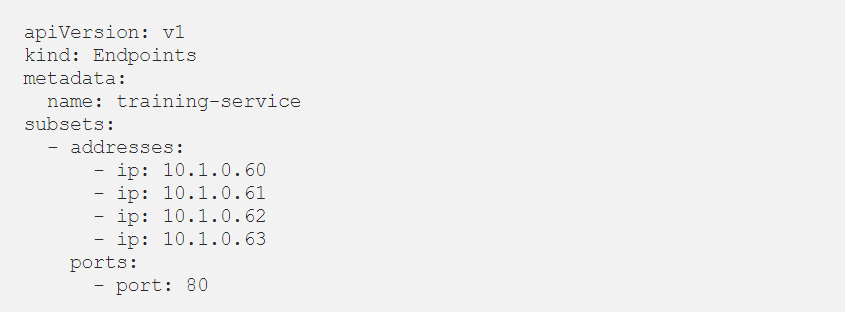


And now if you do:

kubectl get endpoints

NAME ENDPOINTS AGE  
kubernetes 192.168.65.3:6443 13d

Now in order to create the endpoint we must create another YAML file with the endpoint definition as follows:



# Ingress:

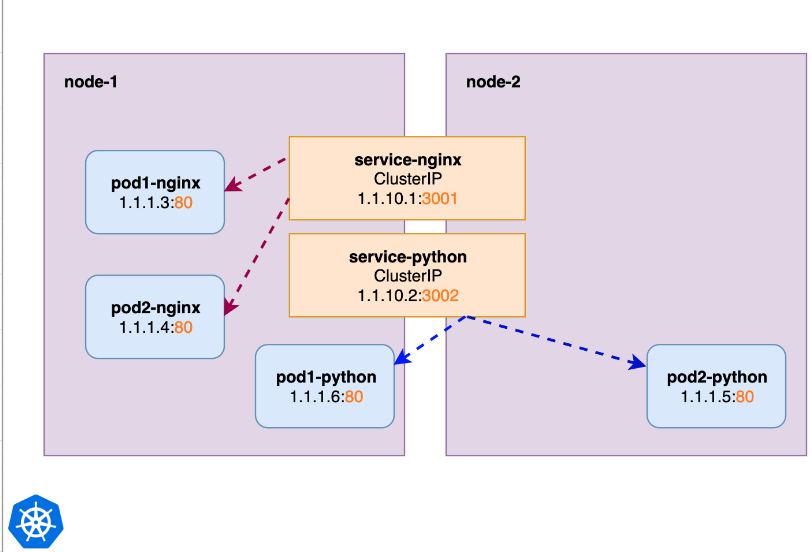
We have seen in the Kubernetes services sections on how to expose our application to the outside world using the NodePort and LoadBalancer. If we only have to have a single service port we can use NodePort. In the case of multiple instances of the same service, we have to use the LoadBalancer.

*But what if we have to add one more service to our node and access it from another URL. In this case, we will have to add another load balancer to our cluster. This means that each service exposed with a LoadBalancer will get its own IP address and we will have to pay for each of these load balancers which can be quite expensive.*

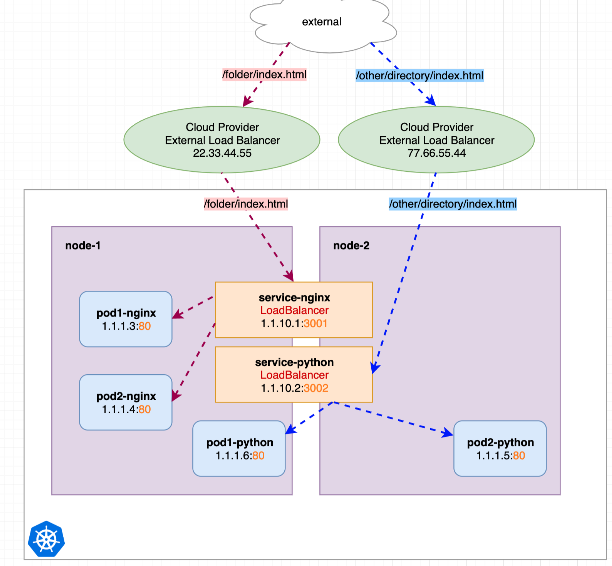
Lets understand the above with a simple Kubernetes example:

We have two worker nodes, we ignore master nodes here. We have two services **service-nginx** and **service-python** which point to various pods.

Services are not scheduled on any specific Node, lets just say they are “available everywhere in the cluster”.



You should understand whats happening here. Internally in our cluster we can reach the Nginx pods and the Python pods through their services. Next we would like to make those available from outside the cluster as well. So we convert those into LoadBalancer services:

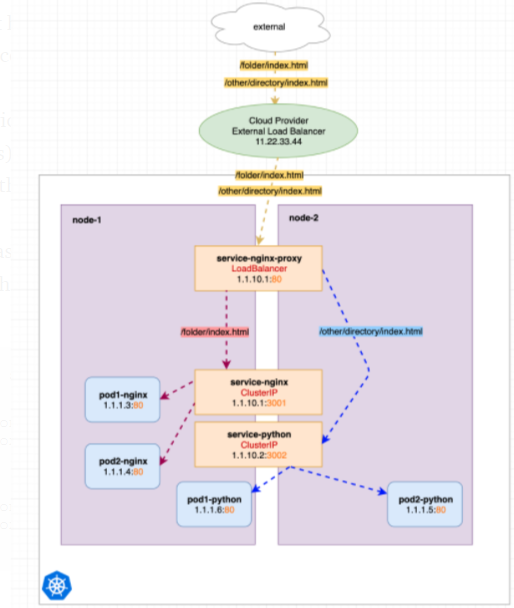


We see two LoadBalancers, each having its own IP. If we send a request to LoadBalancer 22.33.44.55 it gets redirected to our internal **service-nginx**. If we send the request to 77.66.55.44 it gets redirected to our internal **service-python**.

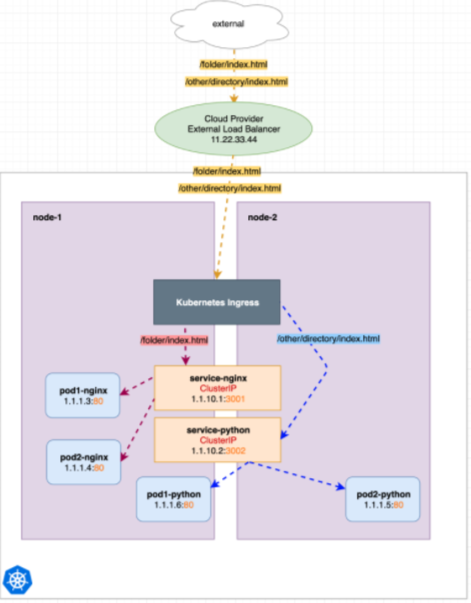
**This works great!** But IP addresses are rare and LoadBalancer pricing depends on the cloud providers. Now imagine we don’t have just two but many more internal services for which we would like to create LoadBalancers, costs would scale up.

## **Manually configure a Nginx Service as proxy**

Nginx can act as a proxy. In the following image we see a new service called **service-nginx-proxy** which is actually our only LoadBalancer service. The **service-nginx-proxy** would still point to one or multiple Nginx-pod-endpoints, but for simplicity I didn’t include this in the graphic. The other two services from before are converted back to simple ClusterIP ones:



We can see that we only hit one LoadBalancer (11.22.33.44) but with different http urls, the requests are displayed in yellow as its the same target and just contains different content (request urls). Really not much changed. We just used a pre-configured Nginx (Kubernetes Ingress) which does already all proxy redirection for us which saves us a lot of manually configuration work:



Kubernetes Ingress is not a Kubernetes Service. Very simplified its just a Nginx Pod which redirects requests to other internal (ClusterIP) services. This Pod itself is made reachable through a Kubernetes Service, most commonly a LoadBalancer.

# Deployments vs StatefulSets vs DaemonSets

Below are 3 different resources that Kubernetes provides for deploying pods:

1. Deployments
2. StatefulSets
3. DaemonSets

There is one other type ReplicationController but Kubernetes now favors Deployments as Deployments configure ReplicaSets to support replication.

Codes for this discussion is captured here : https://github.com/MeSabya/Kubernetes/tree/main/counter-app

## Deployments:

Deployment is the easiest and most used resource for deploying your application. It is a Kubernetes controller that matches the current state of your cluster to the desired state mentioned in the Deployment manifest. e.g. If you create a deployment with 1 replica, it will check that the desired state of ReplicaSet is 1 and current state is 0, so it will create a ReplicaSet, which will further create the pod. If you create a deployment with name **counter**, it will create a ReplicaSet with name **counter-<replica-set-id>**, which will further create a Pod with name **counter-<replica-set->-<pod-id>.**

### **Important Point to note:**

*Deployments are usually used for stateless applications. However, you can save the state of deployment by attaching a Persistent Volume to it and make it stateful, but all the pods of a deployment will be sharing the same Volume and data across all of them will be same.*

We can deploy using the command below:

1. Kubectl apply -f deployment.yaml
2. Kubectl get pods # To get the pod info
3. Kubectl logs <podname> # To check the logs
4. Kubectl get pv #To check the persistent volume info
5. Kubectl exect -it <container name> --<space should be here>bash #To get into container shell

Then cd /app

Check the content of the counter.txt

At step#3: We can see that the second pod will log from where the first pod had left.

At step#4: Only one PVC will be created that both the pods will be sharing so it can cause inconsistency.

So At line#5 if you see they both share the same file.

### **Deployment Strategy:**

Deployments, as discussed, creates a ReplicaSet which then creates a Pod so whenever you update the deployment using RollingUpdate(default) strategy, a new ReplicaSet is created and the Deployment moves the Pods from the old ReplicaSet to the new one at a controlled rate.

*Rolling Update means that the previous ReplicaSet doesn’t scale to 0 unless the new ReplicaSet is up & running ensuring 100% uptime.*

In Deployments, you can also manually roll back to a previous ReplicaSet, if needed in case if your new feature is not working as expected.

## Statefulsets:

StatefulSet(stable-GA in k8s v1.9) is a Kubernetes resource used to manage stateful applications. It manages the deployment and scaling of a set of Pods, and provides guarantee about the ordering and uniqueness of these Pods.

StatefulSet is also a Controller but unlike Deployments, it doesn’t create ReplicaSet rather itself creates the Pod with a unique naming convention. e.g. If you create a StatefulSet with name **counter,**it will create a pod with name **counter-0,**and for multiple replicas of a statefulset, their names will increment like **counter-0**, **counter-1**, **counter-2, etc.**

Every replica of a stateful set will have its own state, and each of the pods will be creating its own PVC(Persistent Volume Claim). *So a statefulset with 3 replicas will create 3 pods, each having its own Volume, so total 3 PVCs.*

The 5 steps mentioned in the “Deployment” section , repeat the same here. And the observation is:

At step#3: We can see that the second pod will start logging from 1 as every pod has its own volume, so it does not read the file of first pod.

At step#4: 2 PVC will be created as there are 2 replicas.

### **Deployment Strategy:**

If you update a StatefulSet, it also performs RollingUpdate i.e. one replica pod will go down and the updated pod will come up, then the next replica pod will go down in same manner.

StatefulSets don’t create ReplicaSet or anything of that sort, so you cant rollback a StatefulSet to a previous version.

### **Usecases:**

StatefulSets are useful in case of Databases especially when we need Highly Available Databases in production as we create a cluster of Database replicas with one being the primary replica and others being the secondary replicas.

# DaemonSet:

A DaemonSet is a controller that ensures that the pod runs on all the nodes of the cluster. If a node is added/removed from a cluster, DaemonSet automatically adds/deletes the pod.

### **Deployment Strategy:**

When you deploy the daemonset, it will create pods equal to the number of nodes. In terms of behavior, it will behave the same as Deployments i.e. all pods will share the same Persistent Volume. only one PVC will be created that all pods will be sharing.

If you update a DaemonSet, it also performs RollingUpdate.

### **Usecases:**

### Monitoring Exporters: You would want to monitor all the nodes of your cluster so you will need to run a monitor on all the nodes of the cluster like NodeExporter.

Logs Collection Daemon: You would want to export logs from all nodes so you would need a DaemonSet of log collector like Fluentd to export logs from all your nodes.

# Some Important Commands:

## [Is there anyway to get the external ports of the kubernetes cluster](https://stackoverflow.com/questions/37648553/is-there-anyway-to-get-the-external-ports-of-the-kubernetes-cluster)?

**Answer**: <https://stackoverflow.com/a/54718303>

# Revise what you have read:

## **Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use what?**

<https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0>

They are all different ways to get external traffic into your cluster, and they all do it in different ways.

### **ClusterIP**

A ClusterIP service is the default Kubernetes service. It gives you a service inside your cluster that other apps inside your cluster can access. There is no external access.

The YAML for a ClusterIP service looks like this:

apiVersion: v1  
kind: Service  
metadata:   
 name: my-internal-service  
spec:  
 selector:   
 app: my-app  
 type: ClusterIP  
 ports:   
 - name: http  
 port: 80  
 targetPort: 80  
 protocol: TCP

## [**Difference between targetPort and port in Kubernetes Service definition**](https://stackoverflow.com/questions/49981601/difference-between-targetport-and-port-in-kubernetes-service-definition)

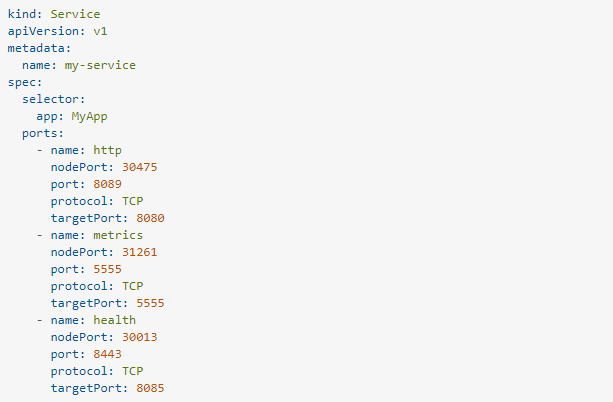
If we see the above service yaml file , What is the difference between the port and targetPort?

***Service***: This directs the traffic to a pod.

***TargetPort***: This is the actual port on which your application is running inside the container.

***Port***: Some times your application inside container serves different services on a different port.

Example: The actual application can run 8080 and health checks for this application can run on 8089 port of the container. So if you hit the service without port it doesn't know to which port of the container it should redirect the request. Service needs to have a mapping so that it can hit the specific port of the container.



if you hit the my-service:8089 the traffic is routed to 8080 of the container(targetPort). Similarly, if you hit my-service:8443 then it is redirected to 8085 of the container(targetPort). But this myservice:8089 is internal to the kubernetes cluster and can be used when one application wants to communicate with another application. So to hit the service from outside the cluster someone needs to expose the port on the host machine on which kubernetes is running so that the traffic is redirected to a port of the container. This is node port(port exposed on the host machine). From the above example, you can hit the service from outside the cluster(Postman or any rest-client) by host\_ip:nodePort

Say your host machine ip is 10.10.20.20 you can hit the http, metrics, health services by 10.10.20.20:30475, 10.10.20.20:31261, 10.10.20.20:30013.

# Deploying python application on Kubernetes cluster:

**Step1**: Create the container and build the image using:

*Docker build -t <image name>: <version>*

**Step2**: Push the image to the docker hub using the command:

*Docker push <image\_name>:version*

**Step3:** Define the deployment yaml file, create the deployment using:

*Kubectl create -f <yaml>*

**Step4**: Define the service yaml file. Create service using:

*Kubectl create -f <yaml>*

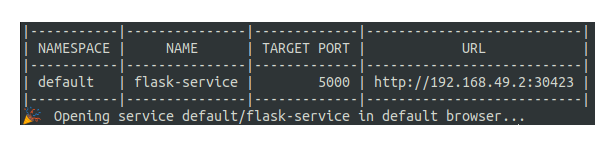
**Step5:** check the details of service using:

*Kubectl get svc <svcname>*

**Step6**: Expose the API service to the outside world via terminal:

*minikube service <service name>*

We should see something like this:



Go to the provided URL and you will see the **your jsonify** message. We can also do

curl <the above url> to check how it works.