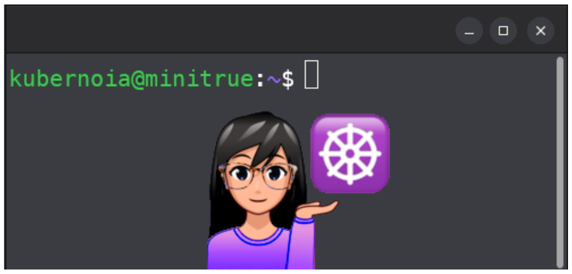
KUBERNETES INTRO

BY PAULOBA



```
kubernoia@minitrue:~$
```

K8s as a Container Orchestration Tool

This doc is a mash up from several sources, all linked within the document.

Main source → Kubernetes as a Container Orchestration Tool

What is kubernetes

Kubernetes, also known as k8s, is an open-source container orchestration tool originally developed by Google Engineers for automating container application deployment, scaling, load balancing and management. Currently, kubernetes is being maintained by the Cloud Native Computing Foundation (CNCF).

The background and ecosystem

The history: kubernetes (κυβερνήτης, Greek for “helmsman” or “pilot”) was founded by Joe Beda, Brendan Burns, and Craig McLuckie. They were joined by other Google engineers including Brian Grant and Tim Hockin and was first announced by Google in mid-2014. There is a significant influence on its development and design by Google’s Borg system, and many of the top contributors to the project previously worked on Borg. The original codename for kubernetes within Google was Project Seven of Nine, a reference to a Star Trek character of the same name that is a “friendlier” Borg.

Here is a paper released by Google at the *Proceedings of the European Conference on Computer Systems* , Bordeaux, France (2015) about large scale cluster management at Google with Borg

https://storage.googleapis.com/pub-tools-public-publication-data/pdf/43438.pdf

The seven spokes on the wheel of the Kubernetes logo are a reference to that codename. The original Borg project was written entirely in `C++`, but the rewritten kubernetes system is implemented in `Go` Programming Language.

The first version of kubernetes (v1.0) was released on July 21, 2015. Along with this release, Google and the Linux Foundation entered into a partnership to form the Cloud Native Computing Foundation (CNCF) and offered kubernetes as a seed technology. On March 6, 2018, Kubernetes Project reached ninth place in commits at GitHub, and second place in authors and issues to the Linux kernel".

Kubernetes ecosystem

Below is the graphical representation of the kubernetes ecosystem in May 2020. It is a recommended path through the cloud-native landscape and provides a comprehensive overview of cloud native technologies and projects over the entire stack, from provisioning and runtimes to orchestration, and app definition and development. Status quo, it comprises 1,390 cards with a total of 2,208,407 stars, market cap of $15.5T and funding of $65.42B.

Kubernetes ecosystem schematics

https://d33wubrfki0l68.cloudfront.net/006ae380ad67eaf5e0fea4da6c37a756bcafe198/f3085/static/landscape-kuber-min-1.jpg

Kubernetes features

Kubernetes has the following features:

- Service discovery and load balancing
- Self-healing (with the use of controllers)
- Storage orchestration
- Automatic rollout and rollback
- Secret and configuration management (no need to include confidential data in the application code)
- Horizontal scaling (automatically deploy more pods if needed)
- Batch execution
  - Jobs
- Automatic Bin packing
  - The bin packing problem is an optimization problem, in which items of different sizes must be packed into a finite number of bins or containers, each of a fixed given capacity, in a way that minimizes the number of bins used.
  - k8s integrates two scoring strategies that support the bin packing of resources
    - MostAllocated → Resource Bin Packing
    - RequestedToCapacityRatio → Resource Bin Packing

K8s Cluster, Master and Worker Nodes

What is a Cluster?

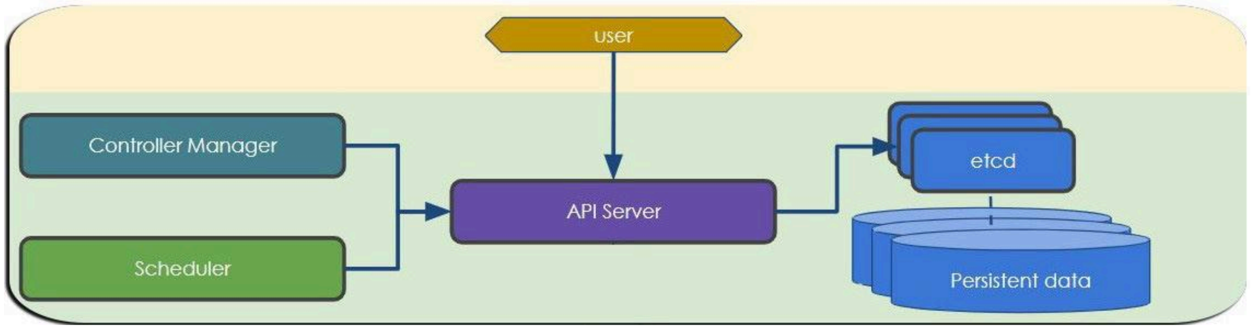
A cluster is a set of nodes grouped and managed by the master's node (a.k.a. control plane). One of the advantages is the ability to have access to applications through the other nodes in case one of the nodes failed. Also, the ability to share load and resources across nodes makes the cluster healthier. There are different tools available to facilitate kubernetes cluster deployment and management including i.a. Google Container Engine for GCP, AWS EKS or Kubermatic Kubernetes Platform that works for any on-prem, cloud, edge, or IoT environment.

What is a Master Node?

* Master Legacy term, synonym for nodes hosting the control plane. → Glossary

From now on this text, take into account that *master node = control plane.*

The master node is the node that controls, monitors, plans and manages worker's nodes, and also performs the administrative tasks. It uses the below components to execute the tasks effectively.



The core of Kubernetes' control plane is the API server. The API server exposes an HTTP API that lets end users, different parts of your cluster, and external components communicate with one another.

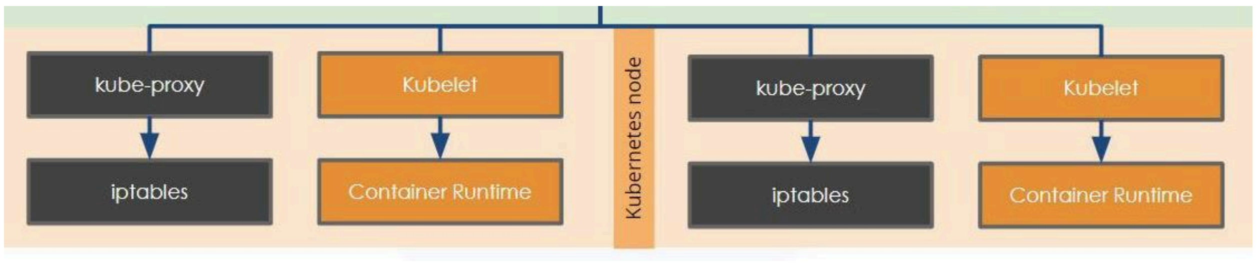
The kubernetes API lets you query and manipulate the state of API objects in kubernetes, for example: Pods, Namespaces, ConfigMaps, and Events.

Most operations can be performed through the kubectl command-line interface or other command-line tools, such as kubeadm, which in turn use the API. However, you can also access the API directly using REST calls.

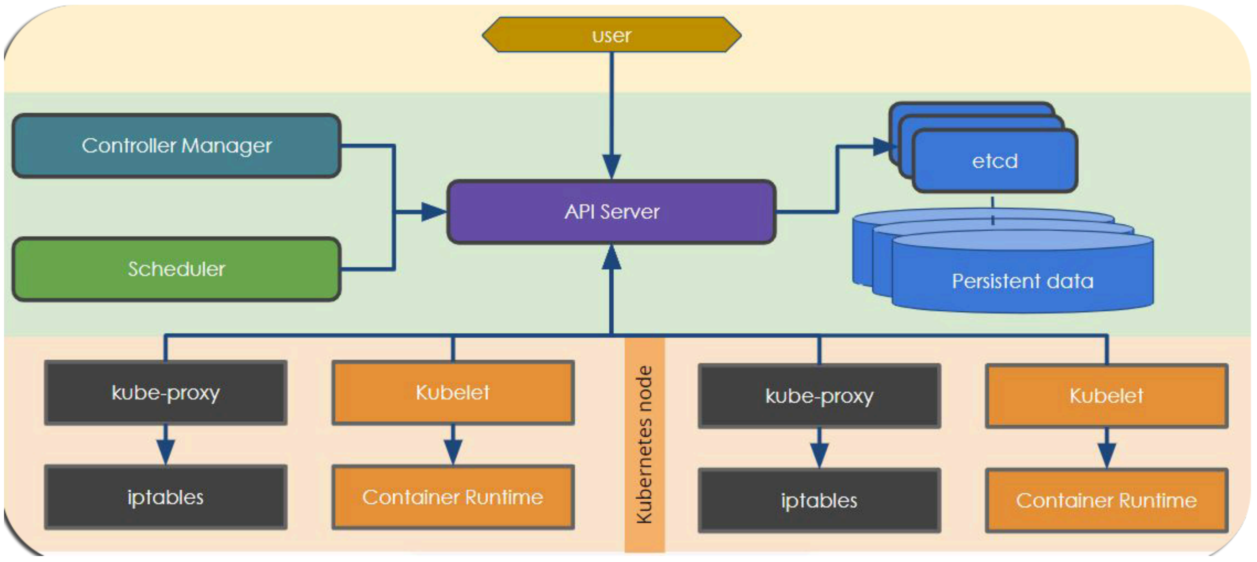
- The Kubernetes API

What is a Worker Node?

A worker node is the node that executes the tasks assigned by the master's node with the help of kubelet. It consists of pods, containers and kube-proxy, which is used to access the external world.



Complete Workflow



Kubernetes Declarative vs Imperative

Kubernetes can be orchestrated in two ways which can either be declarative or imperative. What does this mean?

Imperative mode

This approach which is used in development environments is the easiest and fastest way of deploying to kubernetes using command-line. This type of approach operates on live objects and does not require configuration files but it is used on the command line passed as flags e.g.

`kubectl run <app name>, kubectl create <app name>, kubectl delete <app name>`

Declarative mode

This mode is in contrast to imperative mode, which entails giving step by step instructions declared in a configuration (yaml) file that is specified in a directory and use

`kubectl apply or update <file name>`

command to perform the deployment. This approach is used in production environments.

Hands on k8s lab

Hello k8s application

Now, it is time to put together everything we have been talking about by creating your first kubernetes application. Follow the below steps to deploy your containerised web application.

STEP 0

Install *kubectl*

- Windows → Install and Set Up kubectl on Windows
- Mac → Install and Set Up kubectl on macOS
- Linux → nstall and Set Up kubectl on Linux

Install *cloudSDK*

- Windows → Install the gcloud CLI | Google Cloud
- Mac → Install the gcloud CLI | Google Cloud
- Linux
  - Debian → Install the gcloud CLI | Google Cloud
  - RedHat → Install the gcloud CLI | Google Cloud
  - Other distros → Install the gcloud CLI | Google Cloud

Connect to the lab cluster

```
$ gcloud auth login

$ gcloud config set project testing-lab

$ gcloud container clusters get-credentials lab-cluster --zone
europe-west2-a --project testing-lab
```

You can check the kubectl installed:

```
$ kubectl version
```

and cluster running:

```
$ kubectl cluster-info
```

Create a namespace for this lab with your name

```
$ kubectl create ns example-namespace
```

We are going to deploy an nginx and expose it with a service in a GKE cluster.

nginx /engine x/ is an HTTP and *reverse proxy server*, a *mail proxy server*, and a *generic TCP/UDP proxy server*, originally written by Igor Sysoev.

For a long time, it has been running on many heavily loaded Russian sites including Yandex, Mail.Ru, VK, and Rambler.

According to Netcraft, nginx served or proxied 21.20% busiest sites in January 2023.

Here are some of the success stories: Dropbox, Netflix, Wordpress.com, FastMail.FM.

Source → nginx

STEP 1

Run nginx pod using kubectl

```
$ kubectl run -n example-namespace --image=nginx nginx-app
--port=80 --env="DOMAIN=cluster"
```

STEP 2

Expose the port

```
$ kubectl -n example-namespace expose pod nginx-app --port=80
--name=nginx-http --type=LoadBalancer
```

STEP 3

Check if the container is running

```
$ kubectl get pods -n example-namespace
```

STEP 4

Get the external IP and port by running:

```
$ kubectl get service -n example-namespace
```

The final output should look like the below image if everything works correctly.

```
$ kubectl get service -n example-namespace

NAME       TYPE         CLUSTER-IP     EXTERNAL-IP   PORT(S)       AGE

kubernetes ClusterIP    10.96.0.1      <none>        443/TCP       37m

nginx-http LoadBalancer 10.110.1.201   172.17.0.25   80:30127/TCP  52s
```

My first port forward

The official kubernetes documentation it's a great source of knowledge and it's regularly updated, the information on "my first port forward" is based on https://kubernetes.io/docs/tasks/access-application-cluster/port-forward-access-application-cluster/

Creating MongoDB deployment and service

Create a Deployment that runs MongoDB:

```
kubectl apply -f https://k8s.io/examples/application/mongodb/mongo-deployment.yaml
```

The output of a successful command verifies that the deployment was created: deployment.apps/mongo created

View the pod status to check that it is ready:

```
$ kubectl get pods
```

The output displays the pod created:

```
NAME                    READY   STATUS    RESTARTS   AGE
mongo-75f59d57f4-4nd6q  1/1     Running   0          2m4s
```

View the Deployment's status:

```
$ kubectl get deployment
```

The output displays that the Deployment was created:

```
NAME     READY    UP-TO-DATE   AVAILABLE      AGE
mongo    1/1      1            1              2m21s
```

The Deployment automatically manages a ReplicaSet.

View the ReplicaSet status using:

```
$ kubectl get replicaset
```

The output displays that the ReplicaSet was created:
```
NAME                DESIRED     CURRENT     READY     AGE
mongo-75f59d57f4    1           1           1         3m12s
```

Create a Service to expose MongoDB on the network:

```
$ kubectl apply -f https://k8s.io/examples/application/mongodb/mongo-service.yaml
```

The output of a successful command verifies that the Service was created:
service/mongo created

Check the Service created:

```
$ kubectl get service mongo
```

The output displays the service created:
```
NAME    TYPE       CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE
mongo   ClusterIP  10.96.41.183  <none>       27017/TCP  11s
```

Verify that the MongoDB server is running in the Pod, and listening on port 27017:
```
# Change mongo-75f59d57f4-4nd6q to the name of the Pod

$ kubectl get pod mongo-75f59d57f4-4nd6q --template='{{(index
(index .spec.containers 0).ports 0).containerPort}}{{"\n"}}'
```

The output displays the port for MongoDB in that Pod: `27017`

27017 is the TCP port allocated to MongoDB on the internet.

Forward a local port to a port on the Pod

kubectl port-forward allows using a resource name, such as a pod name, to select a matching pod to port forward to.

```
# Change mongo-75f59d57f4-4nd6q to the name of the Pod
$ kubectl port-forward mongo-75f59d57f4-4nd6q 28015:27017
```

Which is the same as any of the commands in the table below:

`$ kubectl port-forward pods/mongo-75f59d57f4-4nd6q 28015:27017`
`$ kubectl port-forward deployment/mongo 28015:27017`
`$ kubectl port-forward replicaset/mongo-75f59d57f4 28015:27017`
`$ kubectl port-forward service/mongo 28015:27017`

Any of the above commands work. The output is similar to this:

```
Forwarding from 127.0.0.1:28015 -> 27017
Forwarding from [::1]:28015 -> 27017
```

Note `$ kubectl port-forward` does not return anything. To continue with the exercises, you will need to open another terminal.

Start the MongoDB command line interface: `$ mongosh --port 28015`

At the MongoDB command line prompt, enter the `ping` command:

```
db.runCommand( { ping: 1 } )
```

A successful ping request returns:

```
{ ok: 1 }
```

K8s package manager: helm

Helm is a package manager for kubernetes, analogous to NPM or YARN.
It is also a deployment management for k8s, instead of having to define various k8s resources to deploy an application, with Helm you just type a few commands in the terminal and enter.

HELM CHART:
is a collection of YAML files; bundle of the k8s resources needed to build a k8s app.
Helm has a Helm Hub where to search and share charts for popular apps.

Here is an example file hierarchy for a Helm chart:

```
chart
├── Chart.yaml
├── charts
├── templates
│   ├── NOTES.txt
│   ├── _helpers.tpl
│   ├── deployment.yaml
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── service.yaml
│   ├── serviceaccount.yaml
│   └── tests
│       └── test-connection.yaml
└── values.yaml

4 directories, 10 files
```

Helm charts are used in the form of a packaged .tgz file and include the Helm chart version in the file name, like this: `chart-0.1.0.tgz`

CONFIGURATION:
a configuration in the `values.yam`l file, which contains configuration explicit to a release of k8s application.
It can be the config for service, ingress, deployment, etc.

RELEASE:
A chart instance is loaded into Kubernetes.
It can be viewed as a version of the kcs application running based on Chart and associated with a specific config.

REPOSITORIES:
A repository of published Charts.
These can be private repositories that are only used within the company or public through the Helm Hub.

ARCHITECTURE:
Helm has a client-server architecture, including a CLI client and an in-cluster server running in the Kubernetes cluster.

HELM CLI:
Provides the developer a tool to use it, a command-line interface (CLI) to work with charts, config, release, repositories.

DEPLOY AN APPLICATION USING HELM:

```
# Add Incubator repository
$ helm repo add bitnami https://charts.bitnami.com/bitnami

# Update helm repository
$helm repo update

# Install Kafka
$helm install bitnami/kafka --name kafka --namespace queue
```

USE CUSTOM VALUES WHEN DEPLOYING WITH HELM:

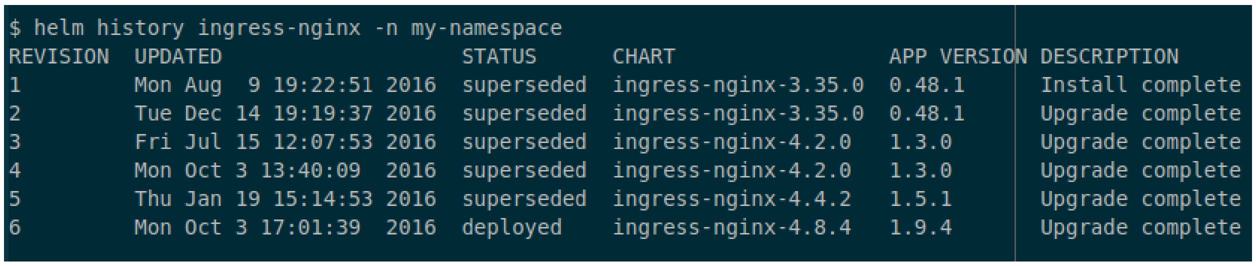
The following command downloads the Kafka helm chart from the Bitnami repo and applies your configuration via values.yaml file:

```
$ helm install bitnami/kafka \
  --name kafka-prod \
  --namespace queue-production \
  -f bitnami/values-prod.yaml
```

DELETE AN APP DEPLOYED WITH HELM FROM A K8S CLUSTER:

```
$ helm delete --purge kafka-prod
```

PRINT HISTORICAL REVISIONS FOR A GIVEN RELEASE:



```
$ helm history ingress-nginx -n my-namespace
REVISION  UPDATED                   STATUS      CHART                 APP VERSION DESCRIPTION
1         Mon Aug  9 19:22:51 2016  superseded  ingress-nginx-3.35.0  0.48.1      Install complete
2         Tue Dec 14 19:19:37 2016  superseded  ingress-nginx-3.35.0  0.48.1      Upgrade complete
3         Fri Jul 15 12:07:53 2016  superseded  ingress-nginx-4.2.0   1.3.0       Upgrade complete
4         Mon Oct 3 13:40:09  2016  superseded  ingress-nginx-4.2.0   1.3.0       Upgrade complete
5         Thu Jan 19 15:14:53 2016  superseded  ingress-nginx-4.4.2   1.5.1       Upgrade complete
6         Mon Oct 3 17:01:39  2016  deployed    ingress-nginx-4.8.4   1.9.4       Upgrade complete
```

CHECK VALUES OF A GIVEN HELM CHART VERSION:

```
$ helm show values ingress-nginx/ingress-nginx --version v4.8.4
```

HELM REPOSITORIES:

Helm chart repositories are remote servers containing a collection of k8s resource files. charts are displayed in directory trees and packaged into Helm chart repositories.

```
$ helm repo add eks-charts https://aws.github.io/eks-charts/
```

ADD HELM REPOSITORIES:

The general syntax for adding a Helm chart repository is:

```
$ helm repo add [NAME] [URL] [flags]
```

To add official stable Helm charts, enter the following command:

```
$ helm repo add stable https://charts.helm.sh/stable
```

LIST THE CONTENTS OF A REPOSITORY:

```
$ helm search repo stable
```

UPDATE HELM REPOSITORIES:

```
$ helm repo update
```

REMOVE HELM REPOSITORIES:

```
$ helm repo remove [REPO1 REPO2 ... REPOn] [flags]
```

```
$ helm repo remove stable
```

More helm tips and tricks helm.sh/docs/howto/charts_tips_and_tricks

Glossary

Nomenclature

- Workload anything that “runs” in GKE, from short-lived jobs to long-running services.
    - Workloads
- Deployment manages deploys, updates, rollbacks, autoscalings and container lifetime. You describe a *desired state* in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.
    - Deployments
- Pod basic building block, runs *at least* one container, but can contain more in the same namespaces.
    - Pods
- CronJobs short-lived workloads, analogy: Crontab. A *CronJob* creates Jobs on a repeating schedule.
    - CronJob
- Job A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete. Deleting a Job will clean up the Pods it created. Suspending a Job will delete its active Pods until the Job is resumed again.
    - Jobs
- ReplicaSet wrapper that tries to have in the cluster the number of pods defined in its spec.
    - ReplicaSet
- DaemonSet A *DaemonSet* ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.
    - DaemonSet
- StatefulSet ensures that the creation of pods follows a stable identity cycle (same naming convention, order to push/pop them).

Like a Deployment, a StatefulSet manages Pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a sticky identity for each of their Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.
If you want to use storage volumes to provide persistence for your workload, you can use a StatefulSet as part of the solution. Although individual Pods in a StatefulSet are susceptible to failure, the persistent Pod identifiers make it easier to match existing volumes to the new Pods that replace any that have failed.
    - StatefulSets
- Services allows you to define a port where the pod will listen to, pointing to an endpoint via an Ingress Load Balancer (LB).
    - Service
- Ingress Load Balancer API object that manages external access to the services in a cluster, typically HTTP. Ingress may provide load balancing, SSL termination and name-based virtual hosting.
    - Ingress

Containers

Containers exist thanks to:

- Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources.
- Cgroups (abbreviated from control groups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.
- Chroot on Unix operating systems is an operation that changes the apparent root directory for the current running process and its children.

kubernetes components

Kubernetes Components

Master (control plane) components

- etcd Distributed key-value store, aka Single Source of Truth.
  - Operating etcd clusters for Kubernetes
- kube-apiserver Checks and stores all API objectsChecks and stores all API objects.
  - kube-apiserver
- kube-scheduler Chooses where a pod should go (new or unscheduled).
  - kube-scheduler
- kube-controller-manager Queries the API server and acts if anything needs action.
  - kube-controller-manager
- cloud-controller-manager Presents the underlying cloud's structure to the cluster.
  - Cloud Controller Manager

Node components

- kubelet This agent ensures that the node is executing the pods that the API server returns the PodSpecs; it makes sure that the containers described in those PodSpecs are running and healthy.
  - kubelet
- kube-proxy Network proxy, handles connections from/to cluster, applying iptables rules. The Kubernetes network proxy runs on each node. This component can do simple TCP, UDP, and SCTP stream forwarding or round robin TCP, UDP, and SCTP forwarding across a set of backends.
  - kube-proxy
- containerd (or ktd) Handles the complete container lifecycle, from image pull to pod kill and delete.
- fluentd/supervisord Data collection software to aid in unifying logging layers.

k8s resources (objects)

- Kind Represents the type of k8s object created. It can be a Pod, DaemonSet, Deployment or Service.
- Version k8s api version used to create the resource, It can be v1, v1beta and v2.
- Metadata Provides information about the resource like name of the pod, namespace under which the pod will be running,labels and annotations.
- Spec Consists of the core information about pod. Here we will tell k8s what would be the expected state of resource, like container image, number of replicas, environment variables and volumes.
- Status Consists of information about the running pod, status of each container. Status field is supplied and updated by Kubernetes after creation.
- We may review the full resources list with the following kubectl command:
- kubectl api-resources -o wide

k8s services

Services act as the unified method of accessing replicated pods. There are 4 major

Service Types:

- ClusterIP Exposes service on a strictly cluster-internal IP.
    - Service ClusterIP allocation
- NodePort Service exposed on each node’s IP on a statically defined port.
    - Service
- LoadBalancer Works in combination with cloud provider to expose service outside the cluster on static external IP.
- ExternalName Reference endpoints outside the cluster by providing a static internally referenced DNS name.