Kubernetes Introduction

Nowadays, [according to the 2020 CNCF survey](), 92% of companies are using containers in production, a 300% increase since 2016. Accordingly, Kubernetes is used by 83% in production environments. Kubernetes is an open-source container orchestration engine released by Google in 2014 and (based on [Borg]()). To manage a lot of containers (of applications, microservices, etc.) you use Kubernetes, either running on bare-metal (your own physical servers), or provided by a cloud provider (AWS, Azure, Google Cloud, etc.). Kubernetes makes it easy to deploy, scale, network, and manage multiple (thousands) of containers, automatizing manual tasks that are tedious and error-prone. When you deploy Kubernetes, you get a cluster, which is a set of worker and controller nodes. The figure below shows the main components of the cluster:

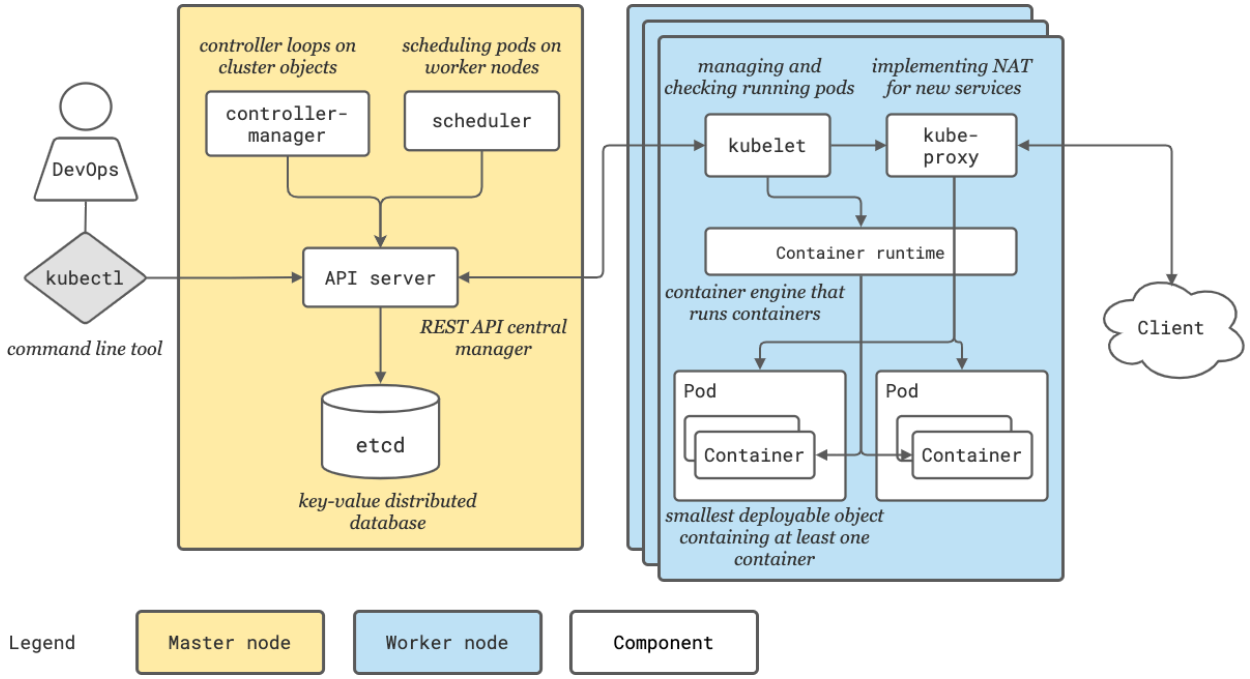

Figure 1: The main components of a Kubernetes cluster and the interactions between the same.

The master node is the brain keeping the cluster up and running, whereas workloads are executed on the worker nodes. The master node is usually replicated (on 2 or 3 nodes) for high availability so that if one master node crashes, the cluster keeps working normally; similarly, also the etcd database can be replicated to provide high availability. Worker nodes are where containers, microservices, and applications are executed: more (powerful) worker nodes are in the cluster, more applications you can deploy on it. Following a more in depth description of each cluster's component:

- *kubectl*: kubectl is a command-line tool that allows users to run commands (e.g. schedule pods) against a Kubernetes cluster.

Master Node

- *API Server*: this component is the front-end of the cluster (i.e. the Kubernetes API) handling requests from the users. In particular, it accepts requests, checks if they are valid (e.g. the user has the right permissions), and executes them.

- *Scheduler*: this component watches for newly created Pods which are not assigned to any worker nodes and selects a node to run the pods (e.g. based on resource requirements).
- *Controller-manager*: this component manages controller processes that make sure the cluster is running and healthy as expected. For example, if there are 3 worker nodes in the cluster, the controller makes sure the nodes are running and healthy.
- *etcd*: key-value database used by Kubernetes to store the configuration data and the cluster status.

Worker Node

- *kubelet*: this component makes sure that containers are running in a Pod.
- *kube-proxy*: this component handles internal (e.g. between pods) and external (e.g. between pods and the Internet) network requests.
- *Container runtime*: the software responsible for running containers (e.g. Docker).
- *Pod*: a pod is the minimum deployable unit in a Kubernetes cluster and contains one or more (similar) containers. The main reason is scalability so that inside each pod you can run as many containers as you need.
- *Container*: a container is a unit of software that contains all resources (i.e. source code, libraries, and other dependencies) required to run an application. In Kubernetes, applications are deployed on containers but containers do not run alone; instead, they are grouped inside pods.

The following sequence diagram shows the steps of deploying a pod (or container) on Kubernetes:

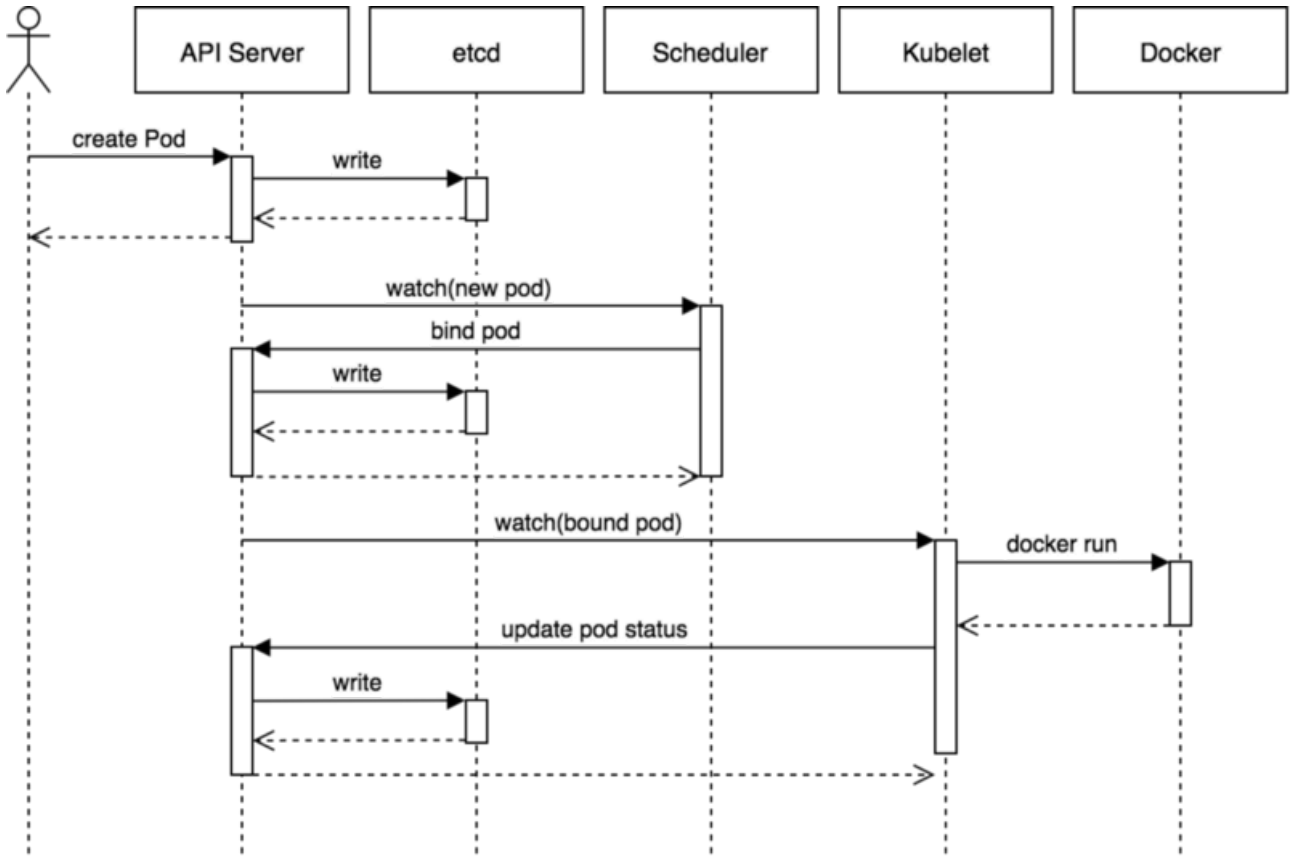

Figure 2: The sequence diagram of deploying a Pod on Kubernetes.

Brief Description of Figure 2

1. (Step 1) The user, using the kubectl tool, sends a request to the API server to run a new pod.
2. (Step 2-8) The API server saves the request in the etcd database; then, the new pod is assigned to a worker node by the scheduler, which is continuously watching for new requests. Finally, the worker node assigned to the pod is saved in the pod's configuration stored in etcd.
3. (Step 9-12) The kubelet, continuously watching for pods to be deployed, notices the new pod request and sends the pod configuration to the container engine (in this case CRI) to run the pod's containers; finally, the container engine runs the pod, and the kubelet notifies the API server.
4. (Step 13-15) The status of the new pod is saved into etcd and the kubelet is notified.

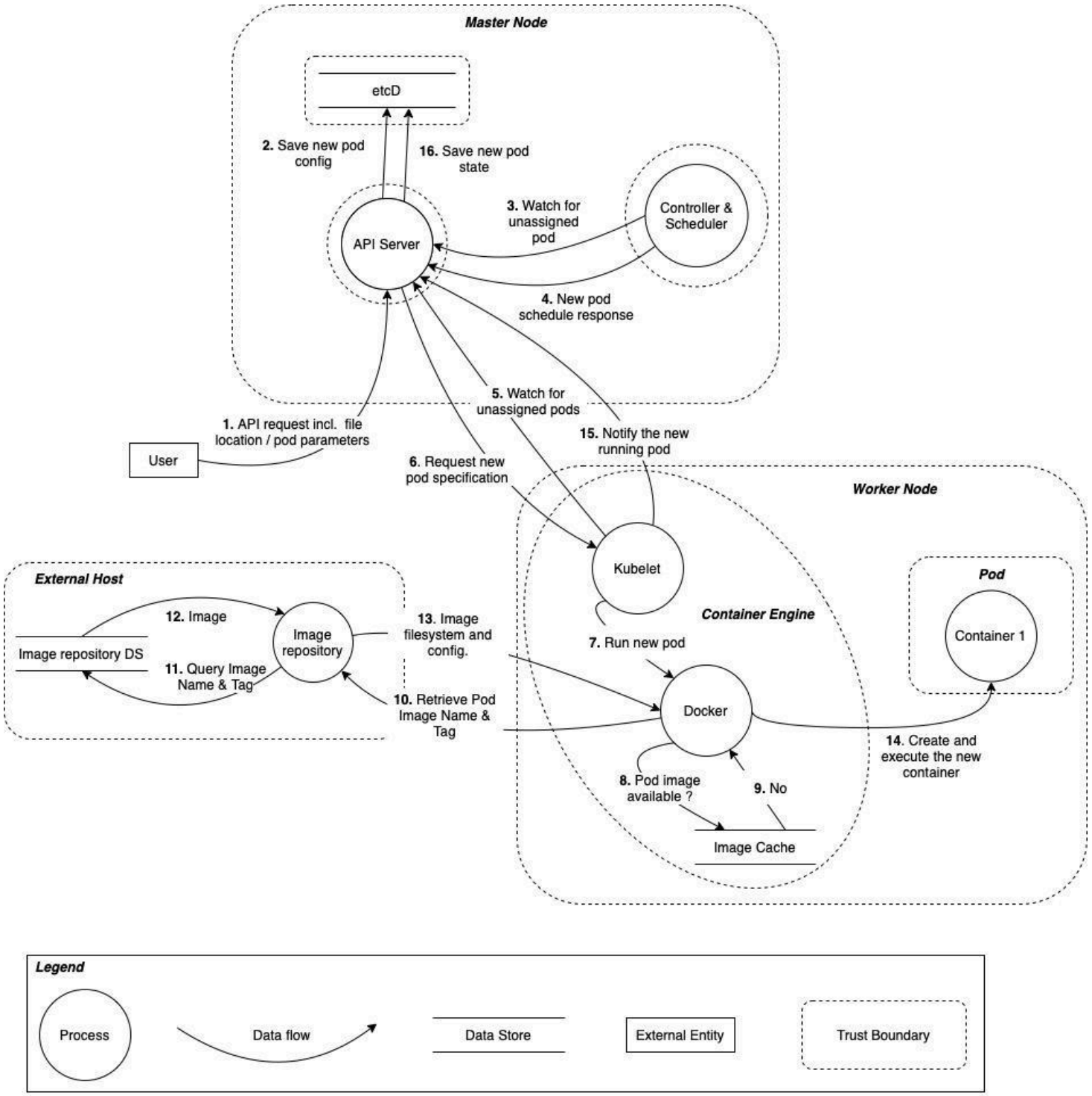

Figure 3: The Data Flow Diagram (with legend) of deploying a Pod on Kubernetes.

How a container image is retrieved from a registry

This procedure may be different based on the container engine being used. The following refers to the Docker engine.
By default, Docker retrieves images from the Docker Hub registry, which is a container images repository where users can upload and download images as they will. Also, Docker, on the local

machine where it is installed, creates a local registry mirror (or image cache) in which it stores container images downloaded by the user. Every time a user requests a container image (e.g. using docker pull) the Docker daemon checks whether the image is already present in the local registry mirror. If the image is present, the daemon retrieves the image from the local storage, otherwise, the image is retrieved from the remote Docker Hub repository and stored locally. The main reason for using a remote and local repository is related to performance. Indeed, different images usually share several layers, which, if already downloaded, make the image-building process much faster.

Kubernetes Networking

Kubernetes networking is quite a complex topic on its own, involving communications between (the many) internal components (e.g. pod-to-pod) and between internal and external components (e.g. pod-to-Internet). In a Kubernetes cluster, the network is managed by an additional component, the Container Network Interface (CNI) plugin, which runs inside the cluster; the plugin, for example, assigns IP addresses to pods, allows to define and enforce network security policies, and so on. There are several plugins available for Kubernetes (e.g. Calico, Cilium, and Weave Net) which can be categorized based on the network layer they work at, according to the 7 layers of the OSI model.

- *Layer 2 (Data Link Layer) Plugins*: plugins working at layer 2 use a (virtual) bridge and (virtual) ethernet interfaces to connect pods within the same worker node. Plugins working at this layer are subject to common Layer 2 attacks, such as ARP poisoning, DNS and IP spoofing or stealing other pods' identities. Despite being less secure, these plugins are faster.

- *Layer 3 (Network Layer) Plugins*: plugins working at this layer do not use a layer 2 (virtual) device, like a network bridge, instead layer 3 solutions, such as IP in IP or BGP protocols. Depending on the environment, such plugins are usually more secure and more scalable.