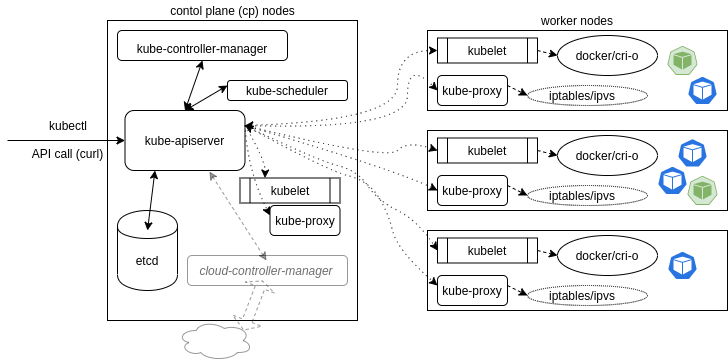
# **Kubernetes Architecture**

To quickly demystify Kubernetes, let's have a look at the Kubernetes Architecture graphic, which shows a high-level architecture diagram of the system components.



**Kubernetes Architecture**

In its simplest form, Kubernetes is made of one or more central managers (aka masters) and worker nodes (we will see in a follow-on chapter how you can actually run everything on a single node for testing purposes). The manager runs an API server, a scheduler, various operators and a datastore to keep the state of the cluster, container settings, and the networking configuration.

Kubernetes exposes an API via the API server: you can communicate with the API using a local client called kubectl or you can write your own client. The kube-scheduler sees the API requests for running a new container and finds a suitable node to run that container. Each node in the cluster runs two containers: kubelet and kube-proxy. The kubelet container receives spec information for container configuration, downloads and manages any necessary resources, and works with the container engine on the local node to ensure the container runs or is restarted upon failure. The Kube-proxy container creates and manages local firewall rules and networking configuration to expose containers on the network.

# **Terminology**

We have learned that Kubernetes is an orchestration system to deploy and manage containers. Containers are not managed individually; instead, they are part of a larger object called a Pod. A Pod consists of one or more containers which share an IP address, access to storage and namespace. Typically, one container in a Pod runs an application, while other containers support the primary application.

Orchestration is managed through a series of watch-loops, also known as operators or controllers. Each operator interrogates the kube-apiserver for a particular object state, modifying the object until the declared state matches the current state. The default, newest, and feature-filled operator for containers is a Deployment. A Deployment deploys and manages a different operator called a ReplicaSet. A replicaSet is an operator which deploys multiple pods, each with the same spec information. These are called replicas. The Kubernetes architecture is made up of many operators such as Jobs and CronJobs to handle single or recurring tasks, or custom resource definitions and purpose-built operators.

There are several other API objects which can be used to deploy pods, other than ensuring a certain number of replicas is running somewhere. A DaemonSet will ensure that a single pod is deployed on every node. These are often used for logging, metrics, and security pods. A StatefulSet can be used to deploy pods in a particular order, such that following pods are only deployed if previous pods report a ready status. This is useful for legacy applications which are not cloud-friendly.

To easily manage thousands of Pods across hundreds of nodes can be difficult. To make management easier, we can use labels, arbitrary strings which become part of the object metadata. These are selectors which can then be used when checking or changing the state of objects without having to know individual names or UIDs. Nodes can have taints, an arbitrary string in the node metadata, to inform the scheduler on Pod assignments used along with toleration in Pod metadata, indicating it should be scheduled on a node with the particular taint.

There is also space in metadata for annotations, which remain with the object, but cannot be used as a selector; however, they could be leveraged by other objects or Pods.

While using lots of smaller, commodity hardware could allow every user to have their very own cluster, often multiple users and teams share access to one or more clusters. This is referred to as multi-tenancy. Some form of isolation is necessary in a multi-tenant cluster, using a combination of the following, which we introduce here but will learn more about in the future:

* namespace  
  A segregation of resources, upon which resource quotas and permissions can be applied. Kubernetes objects may be created in a namespace or cluster-scoped. Users can be limited by the object verbs allowed per namespace. Also the **LimitRange** admission controller constrains resource usage in that namespace. Two objects cannot have the same **Name**: value in the same namespace.
* context  
  A combination of user, cluster name and namespace. A convenient way to switch between combinations of permissions and restrictions. For example you may have a development cluster and a production cluster, or may be part of both the operations and architecture namespaces. This information is referenced from **~/.kube/config**.
* Resource Limits  
  A way to limit the amount of resources consumed by a pod, or to request a minimum amount of resources reserved, but not necessarily consumed, by a pod. Limits can also be set per-namespaces, which have priority over those in the PodSpec.
* Pod Security Policies  
  A policy to limit the ability of pods to elevate permissions or modify the node upon which they are scheduled. This wide-ranging limitation may prevent a pod from operating properly. The use of PSPs may be replaced by **Open Policy Agent** in the future.
* Network Policies  
  The ability to have an inside-the-cluster firewall. Ingress and Egress traffic can be limited according to namespaces and labels as well as typical network traffic characteristics.

# **Control Plane Node**

The Kubernetes master runs various server and manager processes for the cluster. Among the components of the master node are the kube-apiserver, the kube-scheduler, and the etcd database. As the software has matured, new components have been created to handle dedicated needs, such as the cloud-controller-manager; it handles tasks once handled by the kube-controller-manager to interact with other tools, such as Rancher or DigitalOcean for third-party cluster management and reporting.

There are several add-ons which have become essential to a typical production cluster, such as DNS services. Others are third-party solutions where Kubernetes has not yet developed a local component, such as cluster-level logging and resource monitoring.

**kube-apiserver**

The kube-apiserver is central to the operation of the Kubernetes cluster.

All calls, both internal and external traffic, are handled via this agent. All actions are accepted and validated by this agent, and it is the only agent which connects to the etcd database. As a result, it acts as a master process for the entire cluster, and acts as a frontend of the cluster's shared state. Each API call goes through three steps: authentication, authorization, and several admission controllers.

kube-scheduler

The kube-scheduler uses an algorithm to determine which node will host a Pod of containers. The scheduler will try to view available resources (such as available CPU) to bind, and then assign the Pod based on availability and success. The scheduler uses pod-count by default, but complex configuration is often done if cluster-wide metrics are collected.

There are several ways you can affect the algorithm, or a custom scheduler could be used simultaneously instead. A Pod can also be assigned bind to a particular node in the pod spec, though the Pod may remain in a pending state if the node or other declared resource is unavailable.

One of the first configurations referenced during creation is if the Pod can be deployed within the current quota restrictions. If so, then the taints and tolerations, and labels of the Pods are used along with those of the nodes to determine the proper placement. Some is done as an admission controller in the kube-apiserver, the rest is done by the chosen scheduler.

You can find more [details about the scheduler on GitHub.](https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/scheduler.go)

**etcd Database**

The state of the cluster, networking, and other persistent information is kept in an etcd database, or, more accurately, a *b+tree* key-value store. Rather than finding and changing an entry, values are always appended to the end. Previous copies of the data are then marked for future removal by a compaction process. It works with curl and other HTTP libraries, and provides reliable watch queries.

Simultaneous requests to update a particular value all travel via the kube-apiserver, which then passes along the request to etcd in a series. The first request would update the database. The second request would no longer have the same version number as found in the object, in which case the kube-apiserver would reply with an error 409 to the requester. There is no logic past that response on the server side, meaning the client needs to expect this and act upon the denial to update.

There is a cp database along with possible followers. They communicate with each other on an ongoing basis to determine which will be master, and determine another in the event of failure. While very fast and potentially durable, there have been some hiccups with some features like whole cluster upgrades. The kubeadm cluster creation tool allows easy deployment of a multi-master cluster with stacked etcd or an external database cluster.

**Other Agents**

The kube-controller-manager is a core control loop daemon which interacts with the kube-apiserver to determine the state of the cluster. If the state does not match, the manager will contact the necessary controller to match the desired state. There are several controllers in use, such as endpoints, namespace, and replication. The full list has expanded as Kubernetes has matured.

Remaining in beta as of v1.16, the cloud-controller-manager interacts with agents outside of the cloud. It handles tasks once handled by kube-controller-manager. This allows faster changes without altering the core Kubernetes control process. Each kubelet must use the **--cloud-provider-external** settings passed to the binary.

# **Worker Nodes**

All worker nodes run the kubelet and kube-proxy, as well as the container engine, such as Docker or [cri-o](https://cri-o.io/). Other management daemons are deployed to watch these agents or provide services not yet included with Kubernetes.

The kubelet interacts with the underlying Docker Engine also installed on all the nodes, and makes sure that the containers that need to run are actually running. The kubelet agent is the heavy lifter for changes and configuration on worker nodes. It accepts the API calls for Pod specifications (a PodSpec is a JSON or YAML file that describes a Pod). It will work to configure the local node until the specification has been met.

Should a Pod require access to **storage**, **Secrets** or **ConfigMaps**, the kubelet will ensure access or creation. It also sends back status to the kube-apiserver for eventual persistence.

The kube-proxy is in charge of managing the network connectivity to the containers. It does so through the use of iptables entries. It also has the userspace mode, in which it monitors Services and Endpoints using a random high-number port to proxy traffic. Use of ipvs can be enabled, with the expectation it will become the default, replacing iptables.

Kubernetes does not have cluster-wide logging yet. Instead, another CNCF project is used, called [Fluentd](https://www.fluentd.org/). When implemented, it provides a unified logging layer for the cluster, which filters, buffers, and routes messages.

Cluster-wide metrics is not quite fully mature, so [Prometheus](https://prometheus.io/) is also often deployed to gather metrics from nodes and perhaps some applications.