# Overview

Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.



This page is an overview of Kubernetes.

The name Kubernetes originates from Greek, meaning helmsman or pilot. K8s as an abbreviation results from counting the eight letters between the "K" and the "s". Google open-sourced the Kubernetes project in 2014. Kubernetes combines over 15 years of Google's experience running production workloads at scale with best-of-breed ideas and practices from the community.

## Why you need Kubernetes and what it can do

Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?

That's how Kubernetes comes to the rescue! Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. For example: Kubernetes can easily manage a canary deployment for your system.

Kubernetes provides you with:

- **Service discovery and load balancing** Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration** Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.

- **Automatic bin packing** You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- **Self-healing** Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- **Secret and configuration management** Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.
- **Batch execution** In addition to services, Kubernetes can manage your batch and CI workloads, replacing containers that fail, if desired.
- **Horizontal scaling** Scale your application up and down with a simple command, with a UI, or automatically based on CPU usage.
- **IPv4/IPv6 dual-stack** Allocation of IPv4 and IPv6 addresses to Pods and Services
- **Designed for extensibility** Add features to your Kubernetes cluster without changing upstream source code.

# What Kubernetes is not

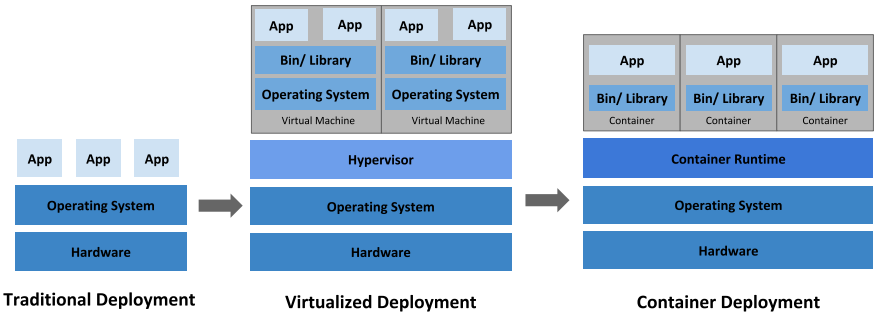
Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system. Since Kubernetes operates at the container level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, and lets users integrate their logging, monitoring, and alerting solutions. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable. Kubernetes provides the building blocks for building developer platforms, but preserves user choice and flexibility where it is important.

Kubernetes:

- Does not limit the types of applications supported. Kubernetes aims to support an extremely diverse variety of workloads, including stateless, stateful, and data-processing workloads. If an application can run in a container, it should run great on Kubernetes.
- Does not deploy source code and does not build your application. Continuous Integration, Delivery, and Deployment (CI/CD) workflows are determined by organization cultures and preferences as well as technical requirements.
- Does not provide application-level services, such as middleware (for example, message buses), data-processing frameworks (for example, Spark), databases (for example, MySQL), caches, nor cluster storage systems (for example, Ceph) as built-in services. Such components can run on Kubernetes, and/or can be accessed by applications running on Kubernetes through portable mechanisms, such as the Open Service Broker.
- Does not dictate logging, monitoring, or alerting solutions. It provides some integrations as proof of concept, and mechanisms to collect and export metrics.
- Does not provide nor mandate a configuration language/system (for example, Jsonnet). It provides a declarative API that may be targeted by arbitrary forms of declarative specifications.
- Does not provide nor adopt any comprehensive machine configuration, maintenance, management, or self-healing systems.
- Additionally, Kubernetes is not a mere orchestration system. In fact, it eliminates the need for orchestration. The technical definition of orchestration is execution of a defined workflow: first do A, then B, then C. In contrast, Kubernetes comprises a set of independent, composable control processes that continuously drive the current state towards the provided desired state. It shouldn't matter how you get from A to C. Centralized control is also not required. This results in a system that is easier to use and more powerful, robust, resilient, and extensible.

# Historical context for Kubernetes

Let's take a look at why Kubernetes is so useful by going back in time.




**Traditional deployment era:**

Early on, organizations ran applications on physical servers. There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues. For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would underperform. A solution for this would be to run each application on a different physical server. But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers.

**Virtualized deployment era:**

As a solution, virtualization was introduced. It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU. Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application.

Virtualization allows better utilization of resources in a physical server and allows better scalability because an application can be added or updated easily, reduces hardware costs, and much more. With virtualization you can present a set of physical resources as a cluster of disposable virtual machines.

Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware.

**Container deployment era:**

Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight. Similar to a VM, a container has its own filesystem, share of CPU, memory, process space, and more. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.

Containers have become popular because they provide extra benefits, such as:

- Agile application creation and deployment: increased ease and efficiency of container image creation compared to VM image use.
- Continuous development, integration, and deployment: provides for reliable and frequent container image build and deployment with quick and efficient rollbacks (due to image immutability).
- Dev and Ops separation of concerns: create application container images at build/ release time rather than deployment time, thereby decoupling applications from infrastructure.
- Observability: not only surfaces OS-level information and metrics, but also application health and other signals.
- Environmental consistency across development, testing, and production: runs the same on a laptop as it does in the cloud.
- Cloud and OS distribution portability: runs on Ubuntu, RHEL, CoreOS, on-premises, on major public clouds, and anywhere else.
- Application-centric management: raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.

- Loosely coupled, distributed, elastic, liberated micro-services: applications are broken into smaller, independent pieces and can be deployed and managed dynamically – not a monolithic stack running on one big single-purpose machine.
- Resource isolation: predictable application performance.
- Resource utilization: high efficiency and density.

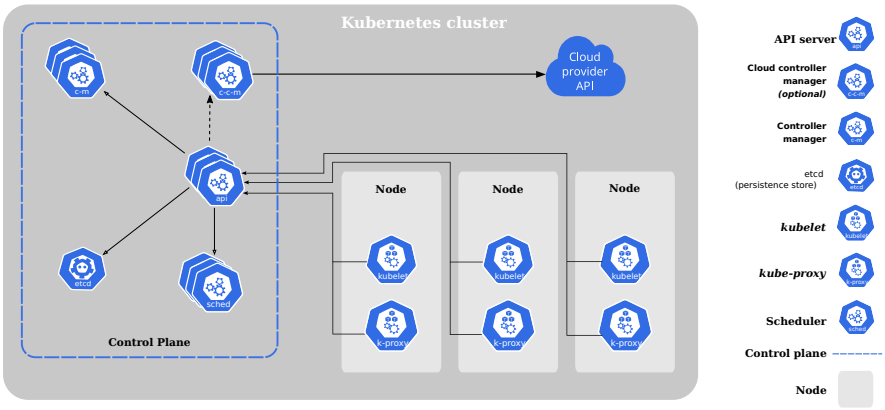
## What's next

- Take a look at the Kubernetes Components
- Take a look at the The Kubernetes API
- Take a look at the Cluster Architecture
- Ready to Get Started?

# 1 - Kubernetes Components

An overview of the key components that make up a Kubernetes cluster.

This page provides a high-level overview of the essential components that make up a Kubernetes cluster.




The components of a Kubernetes cluster

## Core Components

A Kubernetes cluster consists of a control plane and one or more worker nodes. Here's a brief overview of the main components:

### Control Plane Components

Manage the overall state of the cluster:

**kube-apiserver**

The core component server that exposes the Kubernetes HTTP API.

**etcd**

Consistent and highly-available key value store for all API server data.

**kube-scheduler**

Looks for Pods not yet bound to a node, and assigns each Pod to a suitable node.

**kube-controller-manager**

Runs controllers to implement Kubernetes API behavior.

**cloud-controller-manager (optional)**

Integrates with underlying cloud provider(s).

### Node Components

Run on every node, maintaining running pods and providing the Kubernetes runtime environment:

**kubelet**

Ensures that Pods are running, including their containers.

**kube-proxy (optional)**

Maintains network rules on nodes to implement Services.

**Container runtime**

Software responsible for running containers. Read Container Runtimes to learn more.

> ⃠ This item links to a third party project or product that is not part of Kubernetes itself. More information

Your cluster may require additional software on each node; for example, you might also run systemd on a Linux node to supervise local components.

# Addons

Addons extend the functionality of Kubernetes. A few important examples include:

**DNS**

For cluster-wide DNS resolution.

**Web UI (Dashboard)**

For cluster management via a web interface.

**Container Resource Monitoring**

For collecting and storing container metrics.

**Cluster-level Logging**

For saving container logs to a central log store.

# Flexibility in Architecture

Kubernetes allows for flexibility in how these components are deployed and managed. The architecture can be adapted to various needs, from small development environments to large-scale production deployments.

For more detailed information about each component and various ways to configure your cluster architecture, see the Cluster Architecture page.

# 2 - Objects In Kubernetes

Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster. Learn about the Kubernetes object model and how to work with these objects.

This page explains how Kubernetes objects are represented in the Kubernetes API, and how you can express them in `.yaml` format.

## Understanding Kubernetes objects

*Kubernetes objects* are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster. Specifically, they can describe:

- What containerized applications are running (and on which nodes)
- The resources available to those applications
- The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance

A Kubernetes object is a "record of intent"--once you create the object, the Kubernetes system will constantly work to ensure that the object exists. By creating an object, you're effectively telling the Kubernetes system what you want your cluster's workload to look like; this is your cluster's *desired state*.

To work with Kubernetes objects—whether to create, modify, or delete them—you'll need to use the Kubernetes API. When you use the `kubectl` command-line interface, for example, the CLI makes the necessary Kubernetes API calls for you. You can also use the Kubernetes API directly in your own programs using one of the Client Libraries.

### Object spec and status

Almost every Kubernetes object includes two nested object fields that govern the object's configuration: the object `spec` and the object `status`. For objects that have a `spec`, you have to set this when you create the object, providing a description of the characteristics you want the resource to have: its *desired state*.

The `status` describes the *current state* of the object, supplied and updated by the Kubernetes system and its components. The Kubernetes control plane continually and actively manages every object's actual state to match the desired state you supplied.

For example: in Kubernetes, a Deployment is an object that can represent an application running on your cluster. When you create the Deployment, you might set the Deployment `spec` to specify that you want three replicas of the application to be running. The Kubernetes system reads the Deployment spec and starts three instances of your desired application--updating the status to match your spec. If any of those instances should fail (a status change), the Kubernetes system responds to the difference between spec and status by making a correction--in this case, starting a replacement instance.

For more information on the object spec, status, and metadata, see the Kubernetes API Conventions.

## Describing a Kubernetes object

When you create an object in Kubernetes, you must provide the object spec that describes its desired state, as well as some basic information about the object (such as a name). When you use the Kubernetes API to create the object (either directly or via `kubectl`), that API request must include that information as JSON in the request body. Most often, you provide the information to `kubectl` in a file known as a *manifest*. By convention, manifests are YAML (you could also use JSON format). Tools such as `kubectl` convert the information from a manifest into JSON or another supported serialization format when making the API request over HTTP.

Here's an example manifest that shows the required fields and object spec for a Kubernetes Deployment:

application/deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the tem
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

One way to create a Deployment using a manifest file like the one above is to use the `kubectl apply` command in the `kubectl` command-line interface, passing the `.yaml` file as an argument. Here's an example:

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

The output is similar to this:

```
deployment.apps/nginx-deployment created
```

## Required fields

In the manifest (YAML or JSON file) for the Kubernetes object you want to create, you'll need to set values for the following fields:

- `apiVersion` - Which version of the Kubernetes API you're using to create this object
- `kind` - What kind of object you want to create
- `metadata` - Data that helps uniquely identify the object, including a `name` string, `UID`, and optional `namespace`

- `spec` - What state you desire for the object

The precise format of the object `spec` is different for every Kubernetes object, and contains nested fields specific to that object. The Kubernetes API Reference can help you find the spec format for all of the objects you can create using Kubernetes.

For example, see the `spec` field for the Pod API reference. For each Pod, the `.spec` field specifies the pod and its desired state (such as the container image name for each container within that pod). Another example of an object specification is the `spec` field for the StatefulSet API. For StatefulSet, the `.spec` field specifies the StatefulSet and its desired state. Within the `.spec` of a StatefulSet is a template for Pod objects. That template describes Pods that the StatefulSet controller will create in order to satisfy the StatefulSet specification. Different kinds of objects can also have different `.status` ; again, the API reference pages detail the structure of that `.status` field, and its content for each different type of object.

> **Note:**
> See Configuration Best Practices for additional information on writing YAML configuration files.

# Server side field validation

Starting with Kubernetes v1.25, the API server offers server side field validation that detects unrecognized or duplicate fields in an object. It provides all the functionality of `kubectl --validate` on the server side.

The `kubectl` tool uses the `--validate` flag to set the level of field validation. It accepts the values `ignore` , `warn` , and `strict` while also accepting the values `true` (equivalent to `strict` ) and `false` (equivalent to `ignore` ). The default validation setting for `kubectl` is `--validate=true` .

**`Strict`**

Strict field validation, errors on validation failure

**`Warn`**

Field validation is performed, but errors are exposed as warnings rather than failing the request

**`Ignore`**

No server side field validation is performed

When `kubectl` cannot connect to an API server that supports field validation it will fall back to using client-side validation. Kubernetes 1.27 and later versions always offer field validation; older Kubernetes releases might not. If your cluster is older than v1.27, check the documentation for your version of Kubernetes.

# What's next

If you're new to Kubernetes, read more about the following:

- Pods which are the most important basic Kubernetes objects.
- Deployment objects.

- Controllers in Kubernetes.
- kubectl and kubectl commands.

Kubernetes Object Management explains how to use `kubectl` to manage objects. You might need to install kubectl if you don't already have it available.

To learn about the Kubernetes API in general, visit:

- Kubernetes API overview

To learn about objects in Kubernetes in more depth, read other pages in this section: