Detailed Comparison of Kubernetes vs Docker Containers

TABLE OF CONTENTS

Docker and Kubernetes have been the two most commonly used, discussed, and compared phenomena in the recent years of cloud computing. Docker is the most popular container platform, whereas Kubernetes is the de facto container orchestration system. Thus, both tools work on containers but with different focuses and mindsets.

This article will discuss the similarities and differences between Docker and Kubernetes to give you a guideline for choosing the right system for your use case. We'll start with a bit of the history behind microservices architecture and containerization.

Microservices and containerization

Microservices and containerization are the two essential advancements that have changed cloud computing and software development in the last decade. With a microservices architecture, applications become smaller with a narrower scope, namely microservices. Each microservice implements only a single service to be developed, tested, deployed, and updated separately. Using old technologies, you need to install your application inside a virtual machine (VM) and deploy it to a cloud infrastructure, as shown in Figure 1 below. But when you plan hundreds of microservices, it's not scalable to run a separate VM for each service. Containers solve this problem by packaging microservices and running them in the same operating system, as illustrated below in Figure 1.

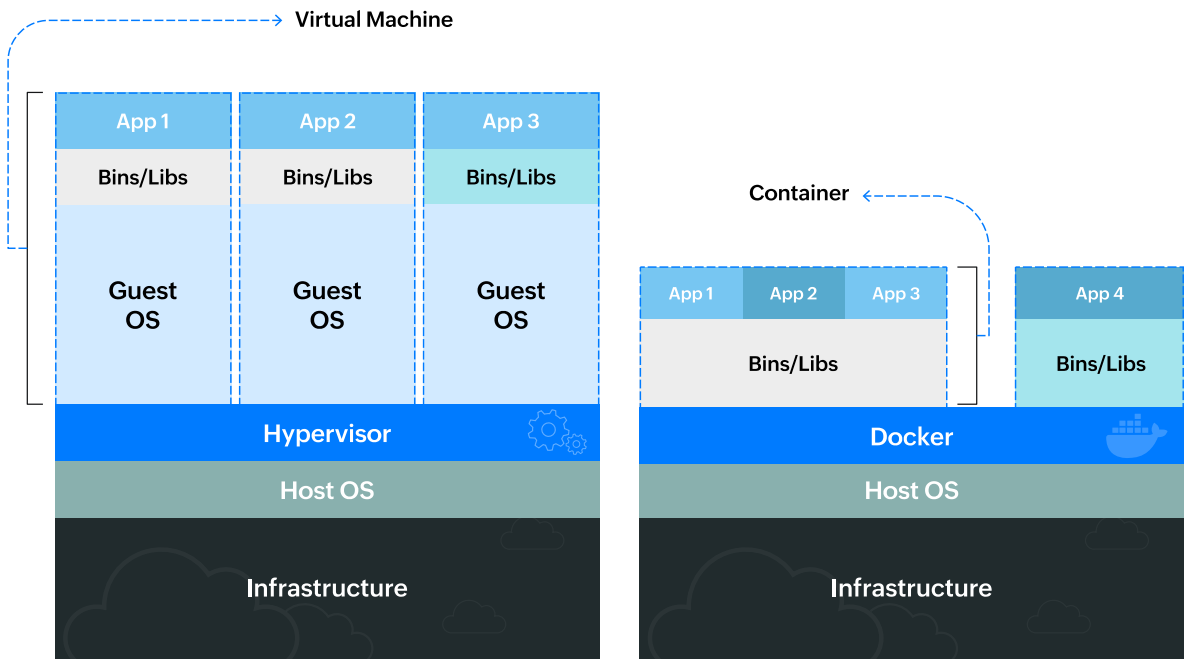

Fig 1. Virtual machines vs. containers

Containers not only solve the scalability issue of microservices but also create a powerful portable system. Developers can code their applications on their laptops, test them locally, and package them with all their requirements into containers that can run on any system, including development laptops, test systems, and production clusters.

Now, we'll explore Docker first, as it is the leading container platform.

Deep dive into Docker

Docker is an open-source container platform that packages applications into

containers and runs them on all operating systems, including macOS, Linux, and Windows. As the leading container platform, Docker is so popular that it is usually used interchangeably with the term containers.

The most common use case for Docker is creating a `Dockerfile` similar to the following with the base image, application copy, build, and run:

```
FROM ubuntu:latest
COPY . /app
RUN make /app
CMD python /app/start.py
```

Copy

You can then create the container image with Docker client commands such as `docker build`, and you will have your application packaged. After, you can either run it locally or upload the container images to a container registry, such as Docker Hub, and redeploy them to any cluster worldwide. This flow is illustrated in the following figure.

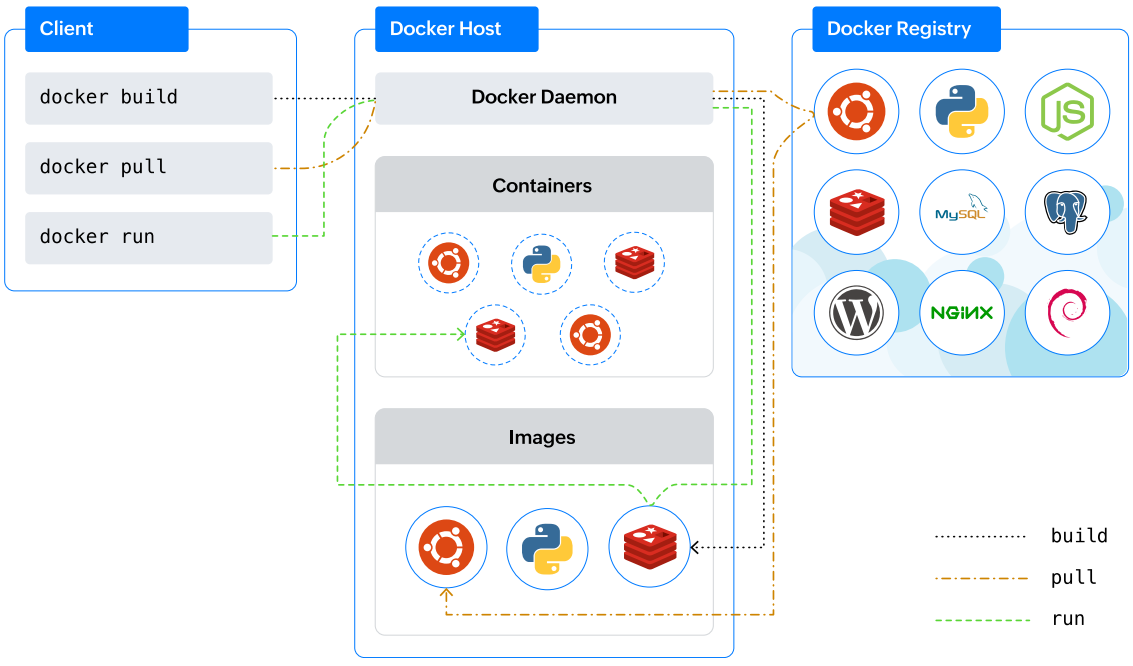

Fig 2. The flow of using Docker

Docker provides all the client and runtime capabilities to create containerized applications and execute them as a container platform. However, as an application gets more complicated, the operational burden of managing containers becomes a real problem.

Orchestration systems for containers

Docker is the perfect tool to create containers, store them in registries, and run them on servers. But there are a few issues for which Docker does not offer an out-of-the-box solution:

- Distribution of containers to servers, clusters, and data centers
- Keeping applications up and running with the required number of instances
- Upgrading applications without downtime

These issues are also known as cloud-native characteristics of modern applications. Therefore, a need for container orchestration systems has arisen. There are three leading container orchestrators on the market: Docker Swarm (https://docs.docker.com/engine/swarm/), Apache Mesos (http://mesos.apache.org/), and Kubernetes (https://kubernetes.io/). Docker Swarm is the best solution if you want to run a couple of containers distributed to servers. If you're looking for more flexibility, you can check out Apache Mesos. But if you need a battle-tested container orchestrator, you should use Kubernetes, which we'll cover next.

Deep dive into Kubernetes

Kubernetes is the de facto open-source container orchestration system. It was

originally developed by Google, where every essential Google service runs in containers in the cloud, so Kubernetes is essentially the packaged version of Google's expertise in managing containers.

You can create and run cloud-native distributed applications on Kubernetes clusters and benefit from the following crucial features:

- Distribution of containers across multiple servers, data centers, and even geographical regions
- Automation of health checks, restarts, and placement of applications
- Hardware and resource management, as well as optimized resource usage over servers

In order to understand Kubernetes better, there are two things you need to learn: **the overall architecture of Kubernetes and the mindset of Kubernetes API objects.**

A Kubernetes cluster consists of worker servers, namely nodes, to run containerized applications and a control plane to manage the entire cluster. In every node, two Kubernetes components are required:

- `kubelet` : A Kubernetes agent used to run containers grouped as Pods
- `kube-proxy` : A daemon for network connectivity between nodes within a Kubernetes cluster

In the control plane, there are five main components:

- `kube-apiserver` : The API server that exposes Kubernetes APIs
- `etcd` : The Key/value store to keep the Kubernetes resource objects
- `kube-scheduler` : The scheduler for assigning pods to nodes
- `kube-controller-manager` : The set of Kubernetes controllers to watch for native Kubernetes resources and take action according to their specification and status
- `cloud-controller-manager` : The set of Kubernetes controllers specific to the cloud provider in which the cluster runs to handle infrastructure-related operations

Components of Kubernetes and their interaction can be summarized as seen in Figure 3 below.

Fig 3. Overall Kubernetes architecture

Kubernetes objects are the declarative definition of the requested state. For instance, the following a deployment (https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) definition is a Kubernetes object that will run four instances of the nginx container in a cluster:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-zoho
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 4
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.21.1
        ports:
        - containerPort: 80
```

Copy

When you send the `nginx-zoo` to the `kube-api-server`, it will check its schema and save it to the `etcd`. Then, controllers in the `kube-controller-manager` will create further Kubernetes resources, such as replication sets and four pods. The `kube-scheduler` will assign each pod to an available node, and the `kubelet` on the nodes will communicate with the container runtime to create containers. Finally, you will have your containers running on the nodes.

When you then query the `kube-apiserver` with a client such as `kubectl`, it will show the status as follows:

```
$ kubectl get deployments
NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
nginx-zoho              4/4     4            4                15s
```

Copy

The application running on the cluster will benefit from all Kubernetes features like self-healing, scalability, and easy out-of-the-box upgrades.

From an infrastructure point of view, you can create Kubernetes clusters in your on-premises data centers or consume the services from the three main cloud providers:

- Google Cloud Kubernetes Engine (GKE)
- Azure Kubernetes Service (AKS)
- Amazon Elastic Kubernetes Service (EKS)

Kubernetes orchestrates the containers and assigns them to run on nodes, which means it needs a container runtime to be available on the nodes. The following section discusses how Kubernetes and the container runtime Docker are similar and how they differ.

Docker vs. Kubernetes

Kubernetes and Docker are both robust solutions to manage containerized applications. Docker allows you to build, upload, and run containers, while Kubernetes lets you distribute and manage them over clusters. That means you need to have both in your toolbox to create and manage cloud-native microservices.

So, what are the similarities and differences between these two legends of the container world?

Similarities between Kubernetes and Docker

Kubernetes and Docker both focus on running containerized applications and have the following common features:

- You can use both in cluster mode by running on multiple servers.
- Both can schedule and distribute the workload over a cluster.
- Both systems handle the networking, storage, and computation of containerized applications.

Differences between Kubernetes and Docker

Despite the shared features listed above, Kubernetes and Docker have some significant differences, listed as follows:

- Docker is a platform to build, run, and store container images, whereas Kubernetes is a container orchestration system.
- Kubernetes provides out-of-box features for the cloud-native world, while you would need various Docker services—such as Docker Swarm—to achieve the same functionality.
- Kubernetes focuses on abstraction layers and API extendibility, whereas Docker focuses on containers directly.

The following table provides a more comprehensive comparison of the two systems:

		Kubernetes
Operations	Installation	Complex
	Upgrades	Complex
	Logging and monitoring	External
	Learning curve	Difficult
Application Management	Container image building	Not available
	Cloud-native application types	All modeled in Kubernetes A
	Automated rollouts and rollbacks	Available as first-class operation

When to choose Kubernetes over Docker (and Vice Versa)

Choosing between Kubernetes and Docker is not always straightforward, and there are certain scenarios for which you will need to select Docker over Kubernetes or vice versa:

- If your cluster is small-scale or a single node, you should choose Docker, as it will make it easier to manage your applications. Kubernetes, with its strong abstraction, is more beneficial for a larger number of nodes.
- If your application stack is small—like a couple of microservices—it's better to use Docker Compose instead of Kubernetes.
- For development and testing, Docker is a better option with less overhead and quick setup time.
- Kubernetes needs a container runtime to run containers on nodes, while with Docker, you can use it as a container runtime or select any other compatible runtime.

Conclusion

To conclude, according to where you stand in your containerization journey, you may need to opt for Docker or Kubernetes or decide to use them both together. Before making a choice, it is essential to first evaluate your requirements, estimated scalability levels, and budget. Luckily, you can move your applications from Docker to Kubernetes or vice versa without changing a single line of code in your applications. You only need to redesign your deployment architecture and cluster setup. This level of flexibility is one of the benefits of working with containers.

Docker and Kubernetes are the past, present, and future of containerized microservice architecture. They have enabled the development, testing, and deployment of cloud-native applications in a scalable way, so it is inevitable for both to have increased adoption in the industry.

Analyze your use cases, and don't be late to the containerization trend!

Related Articles

AWS ECS vs EKS – Ultimate Showdown

Compare Amazon ECS vs EKS. Get a detailed comparison of both the Amazon ECS & EKS which helps you decide on the right AWS container service for you.

Containers vs Virtual Machines

Comparison between containers and virtual machines. What are the major differences between containers and virtual machines. Read to understand.

Building Docker Images Compatible with Azure Container Registry

This article provides a hands-on demonstration of how to build a Docker image and push it to the Azure Container Registry