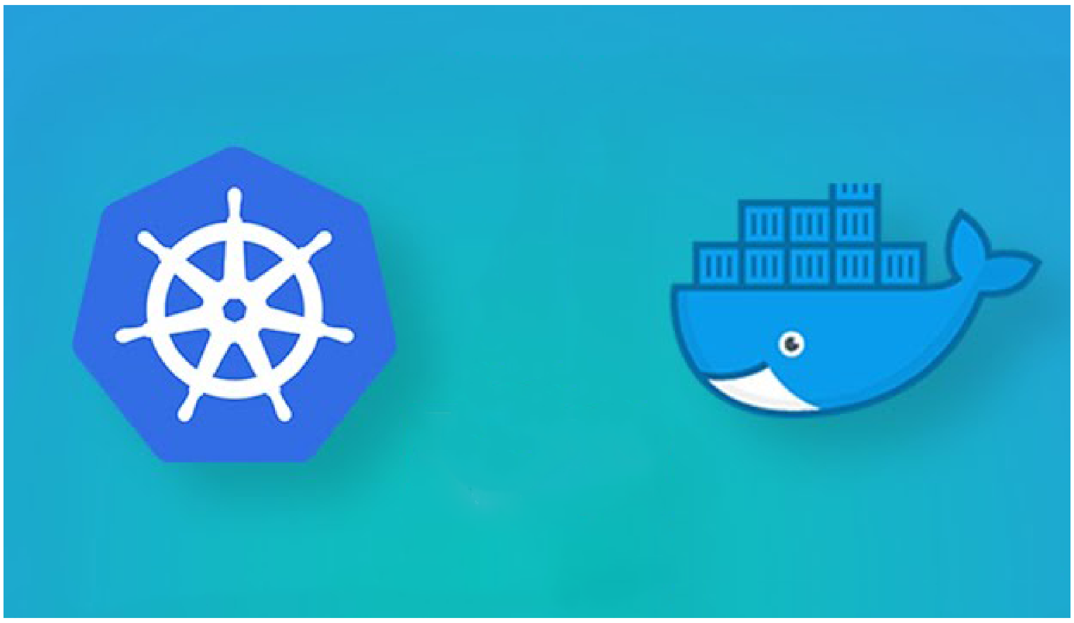
ScoutAPM

Kubernetes vs. Docker Comparison Guide

Table of Contents



Kubernetes vs. Docker

Kubernetes and Docker each play a vital role in modern, microservices-based application development. Since Kubernetes and Docker work in unison to help develop, deploy, and manage large-scale applications – they are not mutually exclusive technologies and they are certainly not in competition with each other. Nevertheless, Kubernetes and Docker are often misunderstood by the non-developer community.

To clear up the confusion around Kubernetes vs. Docker, we've written this guide. After reading each section, you will understand what Kubernetes and Docker are, how they work together, and why are they essential to modern application development.

CHAPTER 1

Containerization, Kubernetes, and Docker Overview

Let's start by defining the core concepts of this discussion.

Containerization: Containerization is a strategy for running applications (and microservices) in a virtual runtime environment that isolates the application from other systems. A container is a server abstraction that allows the application inside it to function as if it were running on a private operating system kernel of its own. Through containerization, you can run multiple apps on the same OS instance, but none of these applications will know about each other.

Docker: Docker is an open-source containerization tool that allows you to create, run, manage, and automate the deployment of containerized apps (and microservices) that run on the same operating system instance. In this way, Docker supports building a microservices-based app architecture that is made up of multiple, independently-running apps and services. A Docker node can only manage the containerized apps and microservices that are running on the operating system instance where the node is installed.

Kubernetes: Kubernetes is an open-source container orchestration tool that allows you to manage an architecture that includes multiple Docker nodes running on different operating system instances. When your container-based architecture includes containers that are running on multiple operating system instances, Kubernetes lets you automate container deployment, load-balancing, networking, and scaling for groups of containers that consist of containers running on different Docker nodes.

Why Use Docker Containers?

Containers are primarily used to build microservices-based applications. In contrast to a traditional monolithic application - where all of an application's services and features are coded into the same piece of programming - a microservices-based architecture breaks a monolith into its component services and features.

The microservices architecture then runs each of these services and features as an autonomous application of its own (a microservice). By loosely connecting these microservices with APIs, the microservices can interact with each other to form a more flexible and "pluggable" architecture that's easier to update and scale.

Docker containers fit into the microservices equation because they are an efficient way to run each microservice in isolation. Compared to running each microservice on its own virtual machine (VMs), for example, containers are faster to start up, use fewer system resources, and don't require a separate OS instance.

Here are some of the benefits of containerization for microservices-based architectures:

- **Saves money:** You can save money on operating system licensing fees because, unlike a VM, multiple containers can run on the same OS instance, which saves money on OS license fees. Containers are also lighter weight and less taxing on server resources than virtual machines, which saves money on server costs.
- **More efficient and faster:** Compared to virtual machines, containers require fewer system resources to operate. They can be as small as 10 megabytes and you can limit their memory and CPU. As lightweight systems, they are extremely fast to start up, destroy, and replicate.

- **A pluggable, flexible architecture:** Compared to the services that form a monolithic application, containerized microservices are less interdependent on the other services that comprise the system. This achieves a “pluggable,” flexible architecture that facilitates updating, adding, or removing services and features without negatively impacting the overall system.

Why Deploy Multiple Docker Nodes on Different Servers?

Using a container orchestration tool like Kubernetes to build a system made up of multiple Docker nodes (with containerized apps running on different OS instances) offers the following advantages:

- **Highly available:** Distributing an architecture across multiple Docker nodes supports a highly available system. With replicated nodes and their associated containers running o different operating systems, if some containers, nodes, or OS kernels go down, the replicated Docker nodes can pick up the slack and keep the system online.

- **Improved scalability:** When the workload on one server instance spikes, container orchestration tools can automatically spin-up and add new Docker nodes and containers (on fresh OS instances) as required. This creates a system that can scale to handle virtually any level of traffic.

CHAPTER 2

What Is Docker? Harnessing the Power of Containers In-Depth

When talking about Docker, it's important to determine whether you're referring to (1) the docker container file format or (2) the open-source platform that helps developers create, run, and automate the deployment of docker files:

1. **The container file format:** As a container file format, a docker image contains only the components – code, libraries, tools, and dependencies – that an application needs to run in isolation. In recent years, docker images have become the de-facto format for running containerized applications and containerized microservices.

2. **The open-source containerization platform:** In addition to being a file format, Docker is also an open-source suite of containerization tools that are accessible via the Windows/Mac dashboard called Docker Desktop. This platform features containerization tools like Docker Engine, a runtime environment that allows you to create, run, and automate the deployment of containers on different operating systems. It also includes Docker Hub, a repository that allows you to find, publish, and share Docker images. Docker Desktop also includes tools like Docker CLI (Command Line Interface) Client, Docker Compose, Notary, Credential Helper, Docker Swarm, and Kubernetes.

How Docker Works

As a containerization tool, Docker's operation depends on a client-server architecture that consist of several fundamental components:

- **Docker Host and Docker Daemon:** Docker Host allows you to run the Docker Engine server instance, i.e., Docker daemon. Docker daemon exposes a REST API that allows operators and clients to interact with and control the daemon. The docker daemon monitors all requests and controls docker and docker objects. It can interact with other daemons if required.
- **Docker Objects:** Docker objects include docker images (container files), containers, volumes, plugins, networks, etc.
- **Docker CLI Client:** The Command Line Client allows you to interact with the Docker daemon by submitting and automating API commands. The REST API allows the client to submit commands to build, run, or distribute Docker containers that are saved (or will be saved) in Docker Registry. CLI Client can interact with more than one Docker daemon.
- **Docker Registry:** Docker Registry allows you to store and distribute Docker container images. Docker Hub is the hosted version of Registry that includes more features like webhooks, organizations, teams, and automated builds.

Here's how these components work together:

1. **Create a Docker image:** Submit a "build" command to the Docker daemon API with Docker CLI Client.
2. **Save the image:** Docker daemon creates the Docker image and saves it to Docker Registry. Docker daemon can save the image locally or remotely via Docker Hub.

3. **Pull an image from the registry (alternatively):** Instead of creating a new Docker image, you can use a "pull" command, and pull an existing container image from Docker Registry or Docker Hub.

4. **Deploy the Docker container:** Run the docker image by submitting a "run" command to Docker daemon with Docker CLI Client. This deploys the container.

What Is Container Orchestration?

When developing an application architecture or IT infrastructure that consists of several containers, you'll need a container orchestration tool for managing resources across the system. The simplest of these tools is Docker Compose. Docker Compose allows you to orchestrate, manage, and schedule the deployment of containers for a multi-container system.

Docker Compose is only useful for container orchestration when all of the containers are running on a single Docker node (i.e., a single server instance). Managing and deploying containers across multiple Docker nodes that are running on different OS instances requires more sophisticated tools – like Docker Swarm or Kubernetes – for container orchestration (more on this in the next section).

Advantages of Using Docker Containers

To sum up this section on Docker, let's review the most compelling advantages of using Docker containers:

- **Faster startup:** An app or microservice running on its own virtual machine takes minutes to launch because you need to wait for a new operating system instance to come online. In contrast, a containerized microservice spins up in seconds because it's simply another process on an already-running OS.
- **Easier to scale:** Because you can spin up and shut down containerized apps so quickly – without rebooting or shutting down an entire virtual machine – it's fast, easy, and more economical to scale up or scale down applications that consist of containerized apps.
- **More flexible deployment:** You don't have to set up a specific type of OS environment to test, distribute, and share a docker image. Docker lets you simply download the appropriate Docker image, and you can run the container (and the app it contains) on any server and virtually any OS. Docker's available for Mac, Windows, Debian, and other operating systems.
- **Cost savings:** Due to their more efficient use of compute resources, you can run more containerized apps (compared to VMs) on a single server. This saves money on processing and OS licensing fees.

CHAPTER 3

What Is Kubernetes? Enter Container Orchestration

When a microservices-based application (or IT infrastructure) consists of many different containers running on many different servers, managing the system is complicated. This is where the open-source container orchestration tool like Kubernetes can help.

Kubernetes offers an API that developers use to manage and automate requests between containerized apps, deploy or replicate Docker nodes and containers when required, and manage processing power to container instances in response to user/client traffic levels. Kubernetes gives you a single command line or dashboard to set the rules that organize your container-based architecture.

Another way to understand Kubernetes is to see it as a "meta-process" that automates and controls the lifecycle of the containerized applications that form an architecture. Developers code this process as a set of deployment instructions rendered in the human-readable programming language YAML.

Once the process is set, the Kubernetes deployment supervises and controls incoming requests and performance across the architecture according to the instructions.

By following these rules and limits set by developers, Kubernetes knows when to deploy, replicate, restart, and scale containers (and groups of containers) and how to load balance incoming requests and divert processing power to achieve optimum system performance across the network.

How Kubernetes Works

When managing a container-based system with Kubernetes, the entire architecture – or the group of containers that Kubernetes orchestrates – is called a "Kubernetes cluster." A Kubernetes cluster needs the following three components to operate:

Pods: A pod is the basic unit of deployment within the Kubernetes cluster. A pod consists of a single containerized app or a group of containerized applications that need each other to operate. For example, if a web server requires a Redis caching server, both get wrapped into the same pod – and both get spawned when that particular pod is deployed. Kubernetes' ability to manage multi-container pods is one of its advantages as a container orchestration tool; however, if a containerized app can run on its own, it can be assigned to a single-container pod.

Worker Nodes (Docker Nodes): A Worker node refers to a single Docker instance running on its own OS instance. A Docker node – which is sometimes called a worker node or just a node – can run a single or multiple containerized applications.

A worker node consists of the following components:

- **kubelet:** The kubelet transmits information regarding the current status of the node to the Kubernetes Master Node. It also executes master node requests.
- **kube-proxy:** The kube-proxy serves as a network proxy so the containerized microservices on the worker node can interact with one other, with other pods, the Kubernetes cluster, and clients outside the cluster.
- **Docker:** Each worker node is running an instance of Docker Engine, which ultimately manages the containers inside it.

Kubernetes master node: The Kubernetes master node is the core of the operation. This is the OS instance where you install and run Kubernetes. It hosts the Kubernetes process that schedules pods and distributes resources to pods that are running on the worker nodes that make up the architecture. Most Kubernetes clusters have more than one master node running for redundancy purposes.

A Kubernetes master node consists of the following components:

- **kube-apiserver:** The kube-apiserver exposes an API for submitting commands and requests to Kubernetes. Developers use a command-line utility such as kubectl or WebUI dashboards to interact with this API. Developers use the kube-apiserver API to set limits and rules for the Kubernetes cluster.
- **kube-control-manager:** The kube-control-manager keeps track of the entire cluster. It knows how many of each pod are currently running and keeps track of other data that is crucial to orchestrating the cluster.
  Kube-control-manager monitors the state of the cluster by watching activity on the kube-apiserver.
- **kube-scheduler:** This is the "decision-maker" that schedules events and jobs across the Kubernetes cluster based on the limits/rules that operators set, available resources, and the current state of the cluster. Kube-scheduler monitors the state of the Kubernetes cluster by watching activity on kube-apiserver.
- **etcd:** Etcd is where the Kubernetes Master Node stores the limits and rules that govern the orchestration of the Kubernetes cluster. As the "storage stack" for the master node, etcd holds information related to policies, system state, etc.

Advantages of Using Kubernetes for Container Orchestration

Here are some of the most compelling advantages of using Kubernetes for container orchestration:

- Declarative by nature: Developers code the rules and limits that define the state of a Kubernetes cluster in YAML. Since you code these instructions in YAML, you can monitor changes, track version control, encourage collaboration among team members.
- Cloud-agnostic and portable: Kubernetes is compatible with virtually any public cloud, on-premises hardware, or bare-metal server – so you can deploy it virtually anywhere. That being said, some cloud infrastructures are better than others, so make sure that the provider you choose supports load balancing and other features.
- Ideal for large, highly-scalable microservices-based architectures: Kubernetes supports nearly unlimited scalability. This open-source tool is ideal for managing architectures consisting of hundreds of thousands of containerized microservices. When scaling an application is required to manage more traffic, operators can set limits and rules that horizontally scale the number of containers available to the system in response to usage metrics.
- Efficient resource optimization and cost savings: Kubernetes automates whether to run or shut down worker nodes within the cluster according to your specifications. This ensures the efficient use of server resources for faster processing, greater system reliability, and more cost savings.

- High availability: Kubernetes achieves a highly available system through its capacity to spawn new pods and run redundant containers. Moreover, when updating existing containerized apps, you can test new iterations of pods before destroying old ones. This offers redundancy and ensures system stability in the event that the updated containers fail.
- Self-healing: Kubernetes achieves a self-healing system by automatically restarting failed containers and shutting down and replacing frozen containers in accordance with user-defined rules. Kubernetes can also replicate containers that were running on a failed node by spinning them up on a different node.

When to Use Kubernetes vs. Docker Swarm

A comparison of Kubernetes and Docker wouldn't be complete without addressing Docker Swarm, a container orchestration tool that Docker developed for managing multi-node systems. The fact is, both Kubernetes and Docker Swarm serve distinct use-cases.

When to use Docker Swarm: Docker Swarm is a powerful solution capable of managing an architecture of thousands of containers.

Since Swarm is easier to learn and use than Kubernetes, it is most appropriate for teams that are less "tech-savvy." It also offers automated configurations and easier deployment, so it's great for teams that want a faster set-up process. Nevertheless, Swarm has fewer capabilities, so it may not be appropriate when you have an exhaustive list of configuration requirements. Swarm is also lacking when it comes to native monitoring capabilities and the API is limited in functionality.

When to use Kubernetes: Kubernetes offers unmatched capabilities when setting policies for high-availability and auto-scaling in large-scale systems. Therefore, it's ideal when orchestrating complex architecture that consists of hundreds of thousands of containers. The downside of Kubernetes is that it comes with a steep learning curve and set-up takes longer than Swarm.

Ultimately, if you have a skilled team of engineers and you are building a complex infrastructure, Kubernetes is the right choice for your use-case.

CHAPTER 4

Final Thoughts on Kubernetes vs. Docker

After reading this guide, you should have a clear understanding of how Docker and Kubernetes work together to build and orchestrate a large-scale container-based application architecture. As a final review of what we've covered:

Docker serves as the core of any container-based architecture because Docker allows you to create and automate the deployment of multiple containerized apps on a single OS instance.

Kubernetes allows us to manage a more complex architecture that consists of multiple Docker nodes and containers – even hundreds of thousands of containers – running on different operating system instances across a network. As a container orchestration tool, Kubernetes automates the management, scaling, updating, adding, removing, and load-balancing across a cluster of containers running on different operating system instances.

When used in conjunction, Docker and Kubernetes can tackle virtually any kind of scaling and container orchestration challenges faced by a microservices-based application or IT infrastructure.

Docker and Kubernetes are fundamental aspects of modern application design – especially when you're building and maintaining a complex, highly scalable system. However, they are just two arms of a many-armed beast. In addition to these containerization tools, you also need a strategy to monitor the performance of the individual apps and services that make up the system.

This is where Scout APM can help. Scout is an application performance monitoring product that helps developers drill down into the fine-grained details of app performance and stability issues. With Scout, you can get to the bottom of the exact cause of N+1 database queries, sluggish queries, memory bloat, and other performance abnormalities.

Start your free 14-day trial today!