Docker

sematext

Table of contents

Definition: What Is Docker?

Docker is an open source virtualization technology that makes it easy to build, test, and deploy applications. With Docker, you can ship your applications in a container environment that houses everything your application needs to run, from libraries to system tools, configuration files, codes, dependencies, and runtime.

Simply put, Docker allows you to ship, test, and deploy your applications in any environment without worrying about incompatible issues regardless of the machine's configuration settings. Thus, no less work when developing applications coupled with a standardized way to deploy them.

The process of implementing Docker containers is called dockerization or containerization.

When Did Docker Come Out?

Docker started as DotCloud, a platform as a service (PaaS) Y Combinator incubator startup that offers software hosting.

However, in 2013, Solomon Hykes and Sebastien Pahl created Docker when they pivoted and decided to release DotCloud's underlying technology under an open-source license.

DotCloud became Docker, and the underlying technology was called Moby. Docker is available for free as the community edition and commercially as the enterprise edition. However, Mirantis, an open source cloud computing company, acquired Docker Enterprise business in 2019.

How Does Docker Work?

Docker uses virtualization and container technology. Using this technology, Docker acts as an isolated sandbox for building lightweight containers to simplify application development and deployment.

While this sounds quite similar to virtual machines, they're pretty different. Virtual machines use hypervisors to share resources and virtualize hardware components. As a result, virtual machines use resources inefficiently, even though server consolidation is a great benefit. On the other hand, Docker virtualizes the operating system (OS). Containers run on the same kernel and use OS features like control groups (cgroups) to allocate available resources among Docker processes. In addition, each process's access to resources is limited by namespace to share resources within the Docker containers effectively.

Thus, Docker containers are lightweight, carrying just the essential OS processes and dependencies necessary, unlike virtual machines and hypervisors.

Docker uses a client-server model to create, run, and distribute containers. A centralized server—in this case, the Docker daemon server—provides the services that Docker clients request. There are two components to Docker:

- Docker Engine, which is the core product of Docker. It includes:
  - daemon (dockerd), the service that runs on the host operating system.
  - CLI (Docker client), which is the Docker command line.

  They communicate via a network interface or UNIX sockets using Docker REST API.
- Docker Hub, which is a cloud service for hosting, storing, and distributing Docker images.

Why Use Docker Containers: Key Benefits

Because of the simplicity and improvements it brings to the app development lifecycle, Docker has gathered a large community of users. Big companies like Adobe, Netflix, PayPal, and Strips use Docker today for a number of reasons.

Here are some of the main benefits that you'd get from implementing Docker:

- **Cost-saving:** Docker containers use far less memory, especially when compared to their counterparts (virtual machines). Thus, you spend less on IT infrastructure resources.
- **Flexible resource sharing:** Your containerized apps are all running on the same operating system even though your application and its dependencies are isolated from the underlying operating system and other containers by Docker containers.
- **Multi-Cloud Platforms:** Almost all of the main cloud service providers support running Docker, and switching between environments is simple. Thus, you can ship anytime and anywhere.
- **Configuration and consistent delivery of your applications:** Docker offers a faster, more resource-efficient, and standardized way to develop, ship, and run applications. Applications can be distributed on various platforms without worrying about framework, library, and compatibility issues.
- **Pipelines:** Docker allows you to standardize the development and release cycle. This acts as a form of legacy change management for applications and encourages Continuous Integration and Continuous Delivery (CI/CD). Thus, building agile and scalable applications is possible.

When to Use Docker

Docker technology offers cost-effective scalability benefits along with simplified application development and deployment. There are some particular scenarios that might benefit more from these than others. Thus, you should consider using Docker:

- **When optimizing the software development process:** Docker offers its user a reproducible environment. The whole application development workflow has been streamlined, especially since its containers handle the bulk of the configuration, development, and deployment processes.

- **If you want to increase developer productivity and efficiency:** Docker enables you to distribute code and its dependencies among your team members consistently.
- **If you want to encourage CI/CD practices:** Docker containerized applications are created in standardized environments that have been streamlined to save build time and run anywhere. Additionally, Docker has a tool ecosystem and can be integrated with source control and integration platforms like GitHub to help you handle environmental conflicts. Hence, they are excellent for DevOps and Agile work processes.
- **When trying to build a scalable application without downtime:** This can be achieved by creating replicas of your Docker container, monitoring traffic, and directing it based on demand.
- **When trying to run multiple applications on a server:** Docker lets you keep components of each application in separate containers even when they're sharing resources.
- **When testing new technologies:** You can use various applications without installing them on Docker. This is great and prevents any conflict issues with existing software (Docker images) in your environment. All you need to do is stop the container and delete the Docker image when done with it. Docker images can be readily available at cloud-based registry like the Docker Hub.
- **If there is a chance that you may change your hosting environment:** Docker containers work with practically all cloud service providers, and switching between environments is simple.

When Not to Use Docker

While Docker is great for streamlining application development, there are a few instances where you might want to avoid it. Here's when you shouldn't use Docker:

- **When building a portfolio project:** You don't need Docker to build a small project, especially since you don't intend to host them and can easily test them on your system. Additionally, Docker requires a lot of initial component setups like Dockerfiles and docker-compose.yml which can be complex. While you could try it, a simpler alternative would be more appropriate because of the complexity of Docker.
- **If you need to use different operating systems or Kernels:** Although Docker enables you to run and deploy apps across several environments, you must use the operating system for which Docker images were created. This is because the operating system (OS) needs a kernel with the same OS as it. If you try otherwise, you won't be able to use Docker effectively. Thus, an image created on Linux will

only run on Linux. There are, however, a few workarounds. One example is running a Linux container inside a Linux virtual machine (VM) running on Windows.
- **GUI Desktop applications:** Docker wasn’t primarily intended for desktop apps, especially those with a rich graphical user interface (GUI). It was designed with the intent of hosting console-based applications, web apps, and isolated containers. Using Docker for desktop apps with a rich GUI would be a challenging process that requires extra workarounds like integrating X11 forwarding or leveraging tzutalin/py2qt4 with Python and the QT framework in a Linux container.

Understanding Basic Terminology

Here are some Docker-associated terms that you will always run into when using and learning about Docker:

Dockerfile

A Dockerfile is the build instruction that the Docker Engine will run. Docker images are created using the commands specified in this build instruction text document. The commands are, however, standardized. For example, the ‘docker build’ commands will be executed by the Docker daemon in order to create and automate the Docker image creation process in the directory where this command is executed.

Docker Image

A Docker image is a template that provides the specifications required to configure a fully operational container environment. For example, you may create a container and tell it to run an application on the Docker platform thanks to the preconfigured server settings, application code, installations, and dependencies in the Docker images. Docker images can be privately owned or publicly shared with other users on the Docker Hub.

Docker images consist of several layers, each building upon the one before. The top layer, the parent image, is where most Dockerfiles start. This parent also acts as a foundation for your container environments.

However, you need the special FROM scratch directive to control the contents of images completely. This is only accessible from the base image. Think of it as the parent image of the parent image.

Docker Containers

A Docker container is a standardized package that contains everything the application needs to run. In essence, it’s the running instance of a Docker image. This executable running instance package includes the operating system, codes, runtime, system tools, libraries, configuration files, and every dependency required to run the application.

You can think of it as a virtual machine. The main difference between a virtual machine and a Docker container is that it comes with a guest operating system.

Docker Engine

The Docker Engine is the core product of Docker. It includes the Docker daemon (dockerd), APIs, and the Docker client.

Docker Daemon

A Docker Daemon is the service responsible for performing instructions sent to Docker. The daemon and Docker core components run on a physical or virtual server known as the Docker host.

Docker Daemon processes any task sent by the Docker Client after being received over the Docker API. It currently runs only on Linux because it depends on several Linux kernel features. However, there are some workarounds for macOS and Windows too.

Docker Client

The Docker Client is the interface that enables users to interact with the Docker Daemon using the Docker API. Simply put, the Docker Client receives the various Docker tasks, such as creating a Docker Image, and sends them to the appropriate component to be completed.

Docker Registry

A Docker registry is a scalable service for hosting, storing, and distributing Docker images. This registry can host public or private Docker images in the cloud or on-premises. In addition, they contain all the versions of Docker images that have been pushed to the registry.

Docker Hub

Docker Hub is Docker's official cloud-based registry for Docker images. It also features an extensive repository of container images created by members of the container community, independent software companies, and open source projects (Mariadb, Postgres, Nginx, etc.). Think of it as the GitHub of Docker images.

Docker Challenges

The adoption of Docker technology has grown rapidly, especially because it offers a simple, standardized method of creating containers and building applications while enabling continuous software delivery. However, users still experience some challenges, just like with any other technology.

[youtube_video]https://www.youtube.com/watch?v=ec9eOi8-lvY[/youtube_video]

Here are some of the challenges you could encounter when using Docker:

Security and Compliance Concerns

While Docker containers are secure, mainly because they can isolate applications on the host and resources are only used as needed for deployment and demand, there are still some challenges that arise when working with them:

- Technically, anyone using the Docker daemon API runs as root and has access to run any client command.
- Lack of visibility into the Docker environment.
- No facility to enable secret management.
- Because of the technology, applications rely on images and third-party components. Thus, it is important to check these third-party dependencies for vulnerabilities.
- Lastly, container configuration and multi-cloud deployments raise the chance of external vulnerabilities.

Security mitigation is challenging, but here are some suggestions:

- When creating your containers, only use trusted base images.
- TLS security protocol should be used to encrypt communications with Docker registries with valid registry certificates.
- Avoid storing Docker secrets in images or Dockerfiles. Using a secret management solution, such as Sematext Docker agent configuration, is necessary to protect secrets.

Orchestration

Container orchestration is the DevOps practice of automating containerized workload operations. This is important because containerized applications can run into thousands of containers, especially when building large-scale systems. In addition, orchestration simplifies operations and promotes resilience (containers can restart or scale based on demand).

To reap these benefits, you need orchestration tools like Kubernetes and Swarm. Implementing containerized workloads can be complex, which makes orchestration more challenging, especially if your containers are inconsistent. Additionally, there are some scalability, networking, and security concerns. For example, Kubernetes clusters can get complex and vulnerable. If a single cluster is compromised, other clusters can become vulnerable.

However, there are certain ways to help mitigate these concerns, such as enabling RABC (role-based access control) and enhancing security with security modules like AppArmor and SELinux.

[youtube_video]https://youtu.be/DYy4OsbtyiA[/youtube_video]

Data Storage

Container technology is designed to be stateless and lacks data persistence. Data persistence means the data remains even after the containers have been rescheduled or destroyed. The term “stateless” means it doesn’t store data.

This means all internal data gets lost when containers are rescheduled or shut down. While there are fixes for this issue, such as storing the data in an external cloud-native workflow, storage is still tricky and a significant challenge.

Docker Monitoring and Logging

Though your Docker containers are isolated, they use a shared resource—the same kernel in the host operating system.

Monitoring Docker containers is necessary to understand how they are functioning. Docker metrics, Docker logs, and service discovery are required for this. Unfortunately, traditional monitoring technologies are insufficient because of Docker’s unique architecture, dynamic deployment, short life spans, orchestration, and distributed transaction tracing.

Useful Tools for Deploying and Managing Docker Containers in Production

Docker has revolutionized how applications are delivered, yet it still lacks some functionalities that would simplify container management, such as orchestration, security tools, schedulers, and monitoring tools. The good news is that some services and third-party solutions have been developed to support the management of containerized processes.

Here are some helpful tools to assist with deploying and managing Docker containers in production:

- **Orchestration tools:** They help automate containerized workload operations across compute environments and hosts, including application maintenance, deployment, update rollouts, container reconfigurations, resource management, and replacing failed containers. Examples of orchestration tools are Kubernetes, Docker Swarm, and Cloudify.
- **Performance monitoring tools:** Monitoring is critical as it helps troubleshoot and assess your Docker architecture's health. However, while this ensures optimal system performance, Docker monitoring is complex for a number of reasons, like its dynamic and multi-layered nature. Sematext Monitoring, Datadog, and Splunk are a few examples of monitoring tools.
- **Log management solutions:** Container logs are useful for debugging and root cause troubleshooting since they show what is happening in an application. To get the most out of your log files, you need log management solutions. They help simplify Docker log management, from troubleshooting and parsing huge amounts of logs to provide a real-time view of events. Great tools for log management include Sematext Logs, Sumo Logic, and SolarWinds PaperTrail.

[youtube_video]https://www.youtube.com/watch?v=IlfLm3rSD2w[/youtube_video]

Monitoring Docker Performance with Sematext Cloud

Sematext Cloud is a full-stack monitoring solution with support for Docker container monitoring and logging. By combining Sematext Logs and Sematext Monitoring, it brings logs, metrics, events, and traces under one roof to give you end-to-end visibility and insight into your containers.

Sematext Cloud comes with out-of-the-box dashboards that you can easily customize to fit your needs. You get powerful alerting, anomaly detection, scheduling and reporting, and logs and metrics correlation that help speed up troubleshooting. The tool features service auto-discovery that automatically identifies and starts monitoring containers – in just seconds, Sematext provides you with complete visibility into your Docker and overall infrastructure.

Learn more from our tutorial on Docker monitoring with Sematext or start the 14-day free trial to get the full picture of what Sematext can do for your specific use case.

Frequently Asked Questions

- ▶ Why is Docker so popular?
- ▶ What language was used to write Docker?
- ▶ What is the difference between Docker and Kubernetes?

Affordable pricing that grows with you, not ahead of you.

Get started for free See pricing

- ✓ Transparent pricing with no surprises
- ✓ 14-day free trial
- ✓ No credit card needed
- ✓ Start now for free

Production support

Solr, Elasticsearch, OpenSearch and Logging consulting

Advanced training

Product	Services	About
Sematext Cloud	Consulting	Company
Log Monitoring	Support	Pricing
Infrastructure Monitoring	Training	Careers
Distributed Tracing		Customers

User Experience

Status

Synthetic Monitoring

Learn	Compare	Contact
Docs	Alternative to Datadog	+1 347-480-1610
Blog	Alternative to New Relic	info@sematext.com
Guides	Alternative to Pingdom	Brooklyn, NY USA
Glossary	Alternative to Dynatrace	