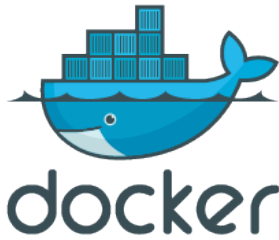# Getting Started with Docker & Docker Engine

- [What is Dockerengine?](#)
- [Docker vs. VM](#)
- [Install Docker Engine on Debian](#)

What is Dockerengine?

Overview

Docker Engine is the heart of Docker, a technology that allows you to create and run small, lightweight packages called **containers**. These containers are like tiny virtual machines but much more efficient. They contain everything an application needs to run, including the code, system libraries, and settings, so it behaves the same on any computer.

Docker 101

https://www.youtube.com/embed/rIrNIzy6U_g

Docker Engine consists of the following major components:

- **Docker Daemon**: This is the background service that handles container management on a Docker host. It listens for Docker API requests and manages the lifecycle of Docker containers, including starting, stopping, and monitoring.
- **REST API**: This API allows external tools and programs to communicate with the Docker Daemon, making it possible to manage Docker resources programmatically.
- **Docker CLI**: The Command-Line Interface (CLI) provides users with the ability to interact with the Docker Daemon using commands. It allows for creating and managing containers, images, networks, and volumes from the terminal.

Docker Engine can run on any Linux-based operating system, including distributions like **Debian**, **Ubuntu**, and **CentOS**, as well as other systems like **Windows** and **macOS** using platform-specific adaptations. On Linux systems, Docker containers share the host's kernel, making them lightweight and highly efficient.

How Does Docker Engine Work?

At its core, Docker Engine uses a **client-server** model. You, the user, interact with Docker by typing commands (using the CLI) or through other software (via the REST API). The **Docker Daemon** (the server part) listens to these requests and manages all the containers on your system.

- **Images and Containers**: Containers are created from something called **images**. Think of an image as a template or blueprint for a container. When you run an image, it becomes a container that can actually perform tasks.
- **Layers and File System**: Docker Engine makes things more efficient by building containers in layers, where each layer represents a change or addition to the image. This way, Docker doesn't need to rebuild everything from scratch each time you make changes.
- **Isolation and Resources**: Docker Engine uses special features in the Linux kernel (the core of the operating system) to isolate containers from each other, ensuring that one container's actions don't affect another. It also controls how much CPU, memory, and other resources each container can use.

Key Features of Docker Engine

- **Lightweight**: Containers don't need their own operating system; they use the host system's resources. This makes them much smaller and faster than virtual machines.
- **Portability**: Once you create a container, it will run the same way on any system that has Docker, no matter where it is. This makes it easy to move your application from your computer to a cloud server or any other environment.
- **Fast**: Containers start up almost instantly because they don't have to load a whole operating system. This makes them ideal for quick testing and development.
- **Isolation**: Each container has its own environment, meaning that your application and its dependencies won't interfere with other applications on the same system.

More Capabilities of Docker Engine

- **Networking**: Docker Engine allows containers to communicate with each other and the outside world through networks. You can connect containers together or expose them to the internet easily.
- **Storage**: Docker Engine can manage data that needs to persist even when containers are restarted or deleted. It does this using **volumes** (for storing data outside of containers) or **bind mounts** (which link folders on the host system to containers).
- **Orchestration Support**: For larger applications, Docker Engine works well with tools like **Docker Swarm** and **Kubernetes**. These tools help manage and automate the running of

many containers at once, often across multiple servers.
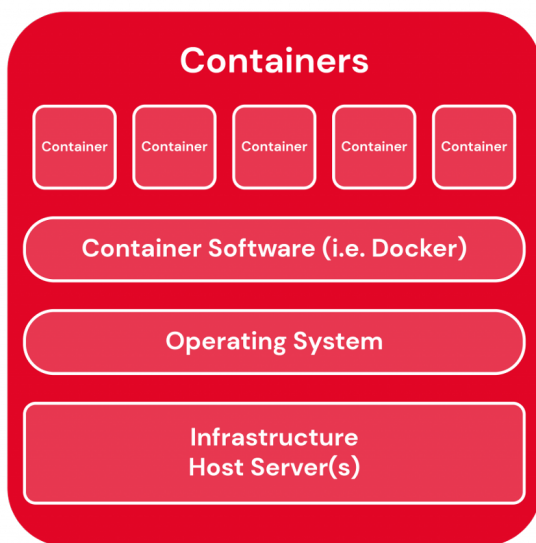
Interested on More?

Check Out "Docker Vs VM's"

Docker vs. VM

Overview

In this article, we'll break down the differences between **Docker** and **Virtual Machines (VMs)**, providing insights to help you decide which technology might be the better fit for your needs. Both Docker and VMs are essential tools for running applications, but they serve different purposes. Before diving into the comparison, let's start with a brief explanation of each.

What is Docker?

In today's rapidly evolving tech world, organizations aim to digitize their businesses, but often face challenges with managing diverse applications across cloud and on-premises infrastructure. Docker addresses this challenge by providing a **container platform** that can host traditional applications and modern microservices, running on both Linux and Windows.
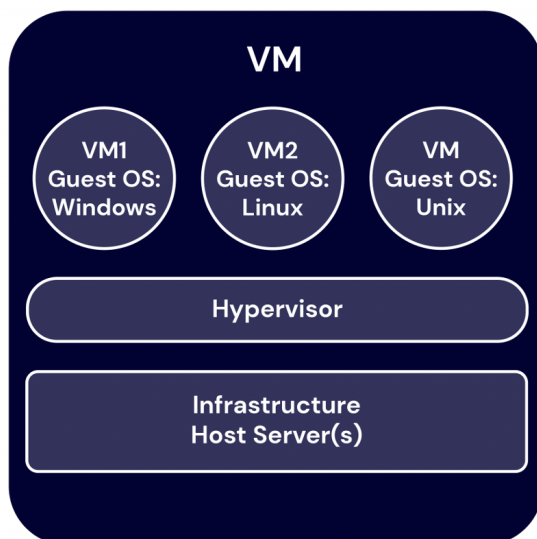
Docker is a tool and a form of **virtualization technology** that simplifies the development, deployment, and management of applications. It achieves this by using **containers**, which are lightweight, self-contained packages that bundle everything needed to run an application, such as

libraries, dependencies, and configuration files.

With Docker, applications run consistently across different systems because the container includes all the necessary elements. Containers are lightweight since they don't need a separate operating system like virtual machines do. Instead, Docker containers share the host system's OS kernel, making them faster and more efficient.

Key benefits of containers include:

- Reduced IT management overhead
- Smaller snapshots of applications
- Faster startup times
- Easier security updates
- Simplified code migration and deployment
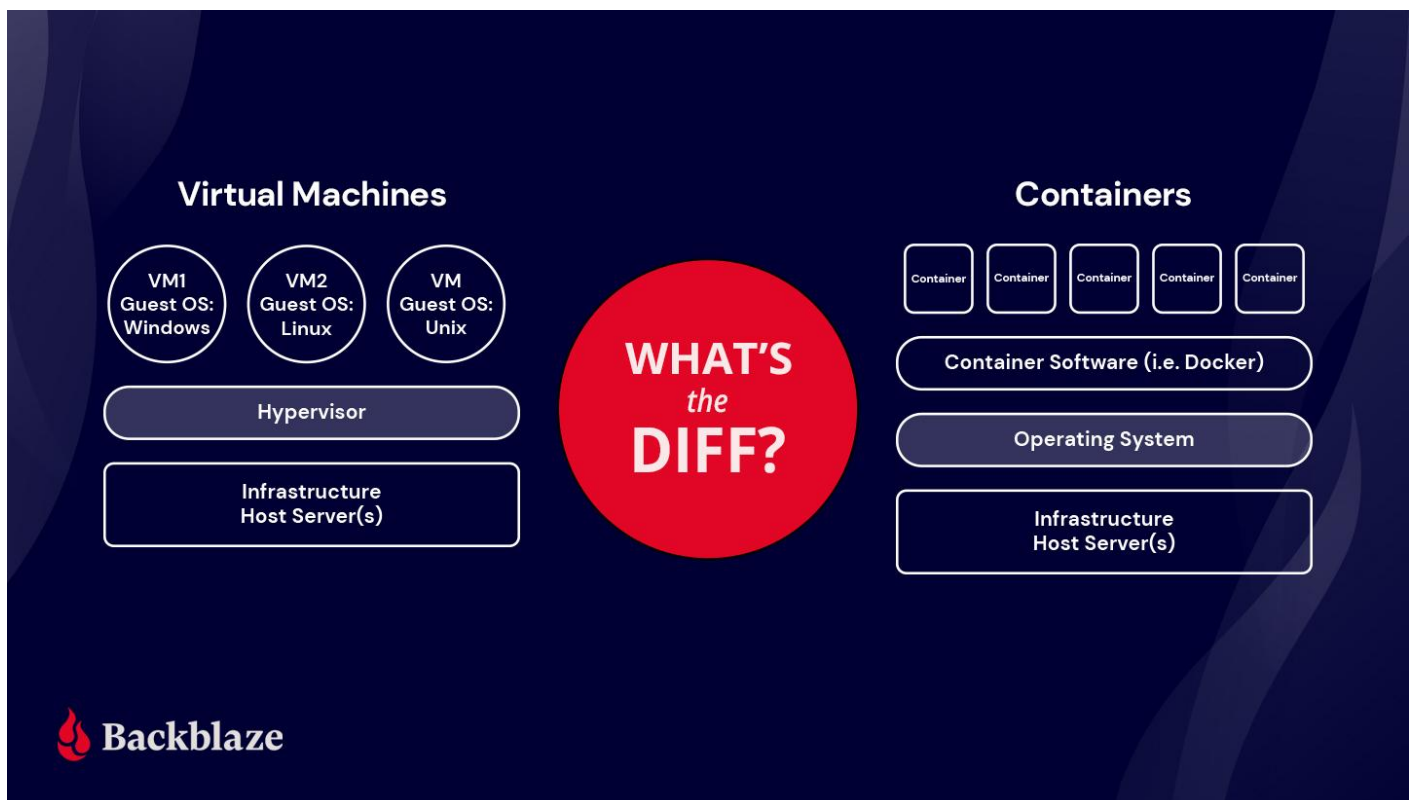

What is a Virtual Machine (VM)?

A **Virtual Machine (VM)**, on the other hand, is a technology that allows a single physical machine to run multiple independent operating systems, each with its own resources. VMs are typically used when performing tasks that might be risky for the host system, such as running potentially harmful software or testing new operating systems. VMs offer strong isolation, so any issues inside a VM won't affect the host system.

Each VM is a complete system with its own operating system, virtual hardware, and resources like CPU, memory, and storage. A physical host can run multiple VMs, allowing for different environments to run simultaneously. VMs are commonly used in **server virtualization**, where a physical server is divided into several VMs to optimize hardware utilization.

There are two types of VMs:

- **System Virtual Machines**: Allow multiple VMs to run their own operating systems and share the physical resources of the host. These are typically managed by a hypervisor.
- **Process Virtual Machines**: Provide a platform-independent environment for running applications, hiding the underlying hardware details from the application.

While VMs provide strong isolation, they can consume a lot of resources since each VM includes its own operating system. This leads to longer boot times and higher resource usage compared to containers.

Docker vs Virtual Machines: Key Differences

Now that you know what Docker and VMs are, let's explore the key differences:

1. **Architecture**

- **Virtual Machines (VMs)**: VMs require both a **host operating system** and a **guest operating system** for each virtual machine. This guest OS could be any OS (e.g., Linux or Windows), regardless of what the host OS is. Each VM includes a complete instance of the operating system, which makes it resource-intensive.

- **Docker**: Docker containers, on the other hand, run on a single host OS and share that OS's kernel. Because of this, containers are much more lightweight, starting faster and using fewer system resources. Docker is ideal for running multiple applications on a single OS kernel.

2. **Security**

- **Virtual Machines (VMs)**: VMs are more secure by design because they run fully isolated from one another. Each VM has its own OS, kernel, and security features. For applications that require heightened security and isolation, VMs are generally the better choice.
- **Docker**: While Docker containers also offer isolation, they share the host's kernel, which can pose security risks. Running a compromised container with root access could potentially lead to an attack on the host system. It's important to apply additional security measures when using Docker containers in sensitive environments.

3. **Portability**

- **Virtual Machines (VMs)**: VMs are somewhat portable, but moving them between different environments (especially with different hardware) can introduce compatibility issues. VMs are ideal for static applications that don't need to be moved often.
- **Docker**: Docker containers are extremely portable and can run consistently on any system with Docker installed. Since they don't require a guest OS, they can be easily transferred between different platforms and environments (development, testing, production), ensuring seamless portability.

4. **Performance**

- **Virtual Machines (VMs)**: VMs require more system resources because each VM must load its own operating system. This leads to longer boot times and higher resource consumption for memory, CPU, and storage.
- **Docker**: Docker containers are lightweight, allowing them to start and stop quickly with minimal overhead. Since containers share the host OS kernel, they use fewer resources, which leads to better performance and faster scaling.

5. **Resource Efficiency**

- **Virtual Machines (VMs)**: VMs need more system resources as they load an entire OS for each instance. Running multiple VMs can quickly consume a large portion of the host's CPU, memory, and storage, making them less efficient when compared to Docker containers.
- **Docker**: Docker containers don't need a full OS, which makes them highly efficient in terms of memory and CPU usage. Since containers share resources based on demand, they are well-suited for applications that need to scale quickly

Docker vs Virtual Machine Comparison Table

Feature	Docker	Virtual Machines (VMs)
Boot Time	Starts in seconds	Takes minutes to boot
Architecture	Shares host OS kernel	Each VM has its own guest OS
Memory Efficiency	Lightweight, no need to virtualize	Requires full OS for each VM
Isolation	Limited isolation, shares host OS	Full OS isolation
Deployment	Quick and easy deployment	Slower and more resource-intensive
Usage	Best for containerized apps	Better for full OS and high security

Should You Choose Docker or Virtual Machines?

Choosing between Docker and VMs depends on your use case:

- **When to use Docker**: If you need to quickly develop, test, and deploy applications, Docker is a great choice. Containers are portable, lightweight, and work well with modern development workflows like **microservices** and **CI/CD pipelines**. Docker is also ideal for running applications across different environments without worrying about compatibility issues.
- **When to use Virtual Machines (VMs)**: For applications that require full OS isolation, increased security, or the ability to run multiple operating systems on the same host, VMs are the better option. VMs are commonly used in **production environments**, especially when security is a primary concern, or when running **legacy applications** that require a specific operating system.
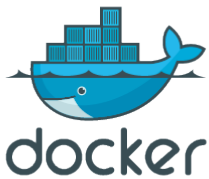
Conclusion: Complementary Tools

Docker and virtual machines are not competing technologies, but rather complementary tools that serve different purposes. VMs provide strong isolation and are ideal for running applications that need their own OS, while Docker containers are lightweight, flexible, and designed for quickly deploying modern applications. Many organizations use both Docker and VMs in a hybrid approach, depending on the specific needs of their applications and infrastructure.

Both technologies have their strengths, and understanding the differences will help you make the right choice for your project.

Install Docker Engine on Debian

This Wikipage hase been integrated by aeoneros from the Original Source: [Docker.Docks](#)

> To get started with Docker Engine on Debian, make sure you [meet the prerequisites](#), and then follow the [installation steps](#).

Prerequisites

OS requirements

OS Requirements
To install Docker Engine, you need the 64-bit version of one of these Debian versions: • Debian Bookworm 12 (stable)   • Debian Bullseye 11 (oldstable)   Docker Engine for Debian is compatible with x86_64 (or amd64), armhf, arm64, and ppc64le (ppc64el) architectures.

Uninstall old versions

Uninstall Old Versions

fore you can install Docker Engine, you need to uninstall any conflicting packages.

Distro maintainers provide unofficial distributions of Docker packages in their repositories. You must uninstall these packages before you can install the official version of Docker Engine.

The unofficial packages to uninstall are:

- `docker.io`
- `docker-compose`
- `docker-doc`
- `podman-docker`

Moreover, Docker Engine depends on `containerd` and `runc`. Docker Engine bundles these dependencies as one bundle: `containerd.io`. If you have installed the `containerd` or `runc` previously, uninstall them to avoid conflicts with the versions bundled with Docker Engine.

Run the following command to uninstall all conflicting packages:

```
for pkg in docker.io docker-doc docker-compose podman-docker containerd runc; do sudo apt-get remove $pkg; done
```

`apt-get` might report that you have none of these packages installed.

> Images, containers, volumes, and networks stored in `/var/lib/docker/` aren't automatically removed when you uninstall Docker. If you want to start with a clean installation, and prefer to clean up any existing data, read the uninstall Docker Engine section.

Installation for Linux

> Before you install Docker Engine for the first time on a new host machine, you need to set up the Docker `apt` repository. Afterward, you can install and update Docker from the repository.

1. Set up Docker's `apt` repository

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/debian/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/debian \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

> If you use a derivative distro, such as Kali Linux, you may need to substitute the part of this command that's expected to print the version codename:
> `$(. /etc/os-release && echo "$VERSION_CODENAME")`
> Replace this part with the codename of the corresponding Debian release, such as `bookworm` .

2. Install the Docker packages

To install the latest version, run:

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

3. Verify that the installation is successful by running the `hello-world` image:

```
sudo docker run hello-world
```

This command downloads a test image and runs it in a container. When the container runs, it prints a confirmation message and exits.

> You have now successfully installed and started Docker Engine.