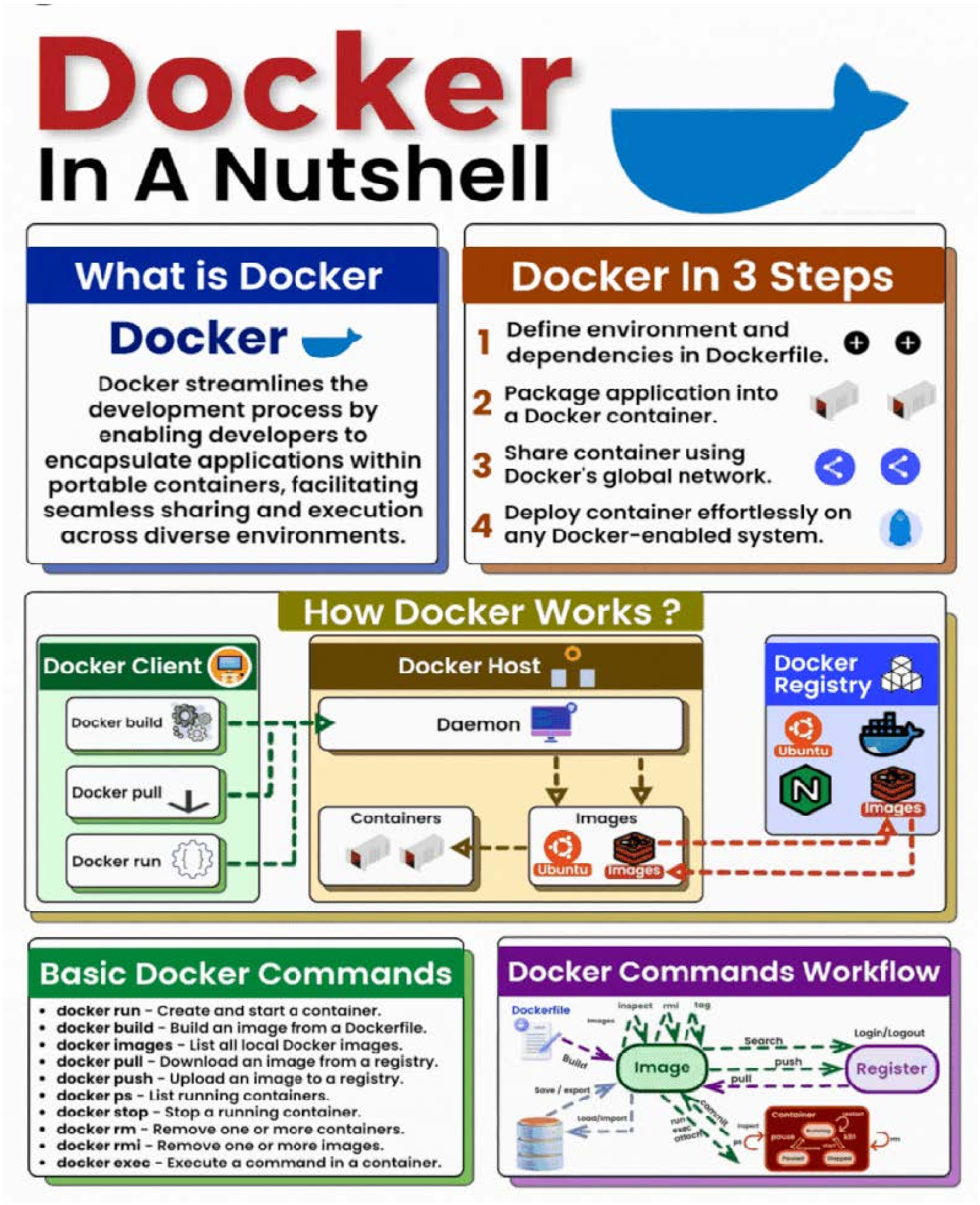
Deploying Docker on Ubuntu

Deployments Simplified – Docker

Cedric Simon D’Souza



What is Docker

Docker streamlines the development process by enabling developers to encapsulate applications within portable containers, facilitating seamless sharing and execution across diverse environments.

Docker In 3 Steps

1. Define environment and dependencies in Dockerfile.
2. Package application into a Docker container.
3. Share container using Docker's global network.
4. Deploy container effortlessly on any Docker-enabled system.

Basic Docker Commands

- **docker run** – Create and start a container.
- **docker build** – Build an image from a Dockerfile.
- **docker images** – List all local Docker images.
- **docker pull** – Download an image from a registry.
- **docker push** – Upload an image to a registry.
- **docker ps** – List running containers.
- **docker stop** – Stop a running container.
- **docker rm** – Remove one or more containers.
- **docker rmi** – Remove one or more images.
- **docker exec** – Execute a command in a container.

Table of Contents

What is Docker

Docker is an open-source platform **based on Linux containers** for developing, shipping, and running applications inside **containers**. we can deploy many containers simultaneously on a given host. Containers are very fast and lightweight because they don't need the extra load of a hypervisor in comparison to the virtual machines because they run directly within the host machine's kernel.

Docker is a popular containerization platform that enables developers to package applications and their dependencies into lightweight, portable containers. It provides a consistent runtime environment, ensuring that applications run seamlessly across different systems and environments. Docker simplifies the process of creating, distributing, and running containerized applications, making it a favourite choice for local development and single-host deployments.

In the old days of software development, getting an application from code to production was slow and painful. Developers struggled with dependency hell as test and production environments differ in subtle ways, leading to code mysteriously working on one environment but not the other. Then along came Docker in 2013, originally created within dotCloud as an experiment with container technology to simplify deployment. Docker was open-sourced that March, and over the next 15 months it emerged as a leading container platform.

Containers are used for deploying Microservices applications in an easy way. It's difficult to talk

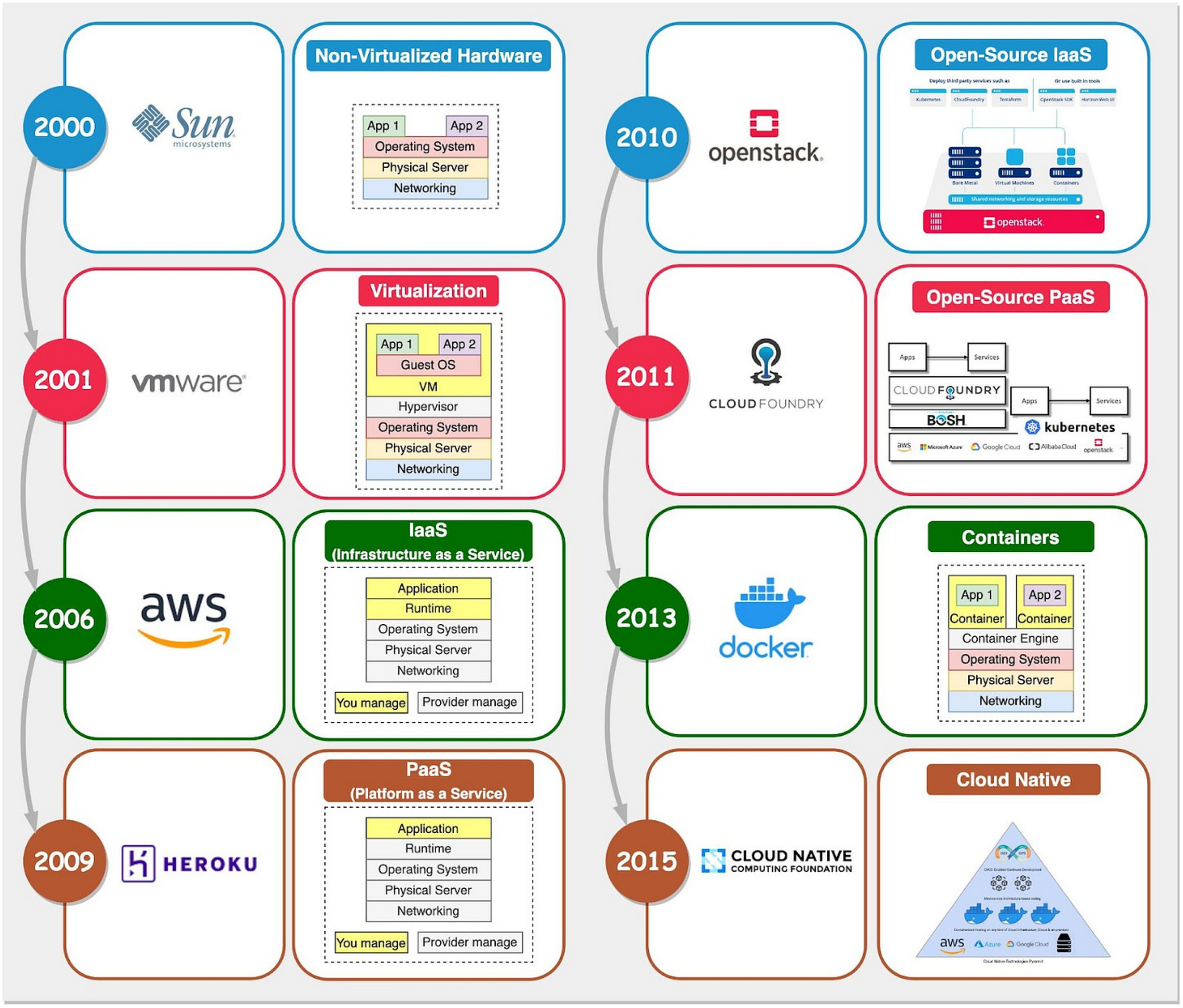
about microservices without talking about containers. Before Docker's release in 2013, Cloud Foundry was a widely used open-source PaaS platform. Many companies adopted Cloud Foundry to build their own PaaS offerings. Compared to **IaaS**, **PaaS** improves developer experience by handling deployment and application runtimes. Cloud Foundry provided these key advantages:

- Avoiding vendor lock-in - applications built on it were portable across PaaS implementations.
- Support for diverse infrastructure environments and scaling needs.
- Comprehensive support for major languages like Java, Ruby, and Javascript, and databases like MySQL and PostgreSQL.
- A set of packaging and distribution tools for deploying application.

Architecture of Docker

Just as people use Xerox as shorthand for paper copies and say "Google" instead of internet search, Docker has become synonymous with containers. Docker is more than containers, though. Docker is a suite of tools for developers to build, share, run and orchestrate containerized apps.

In the past two decades, backend infrastructure evolved rapidly, as illustrated in the timeline below:



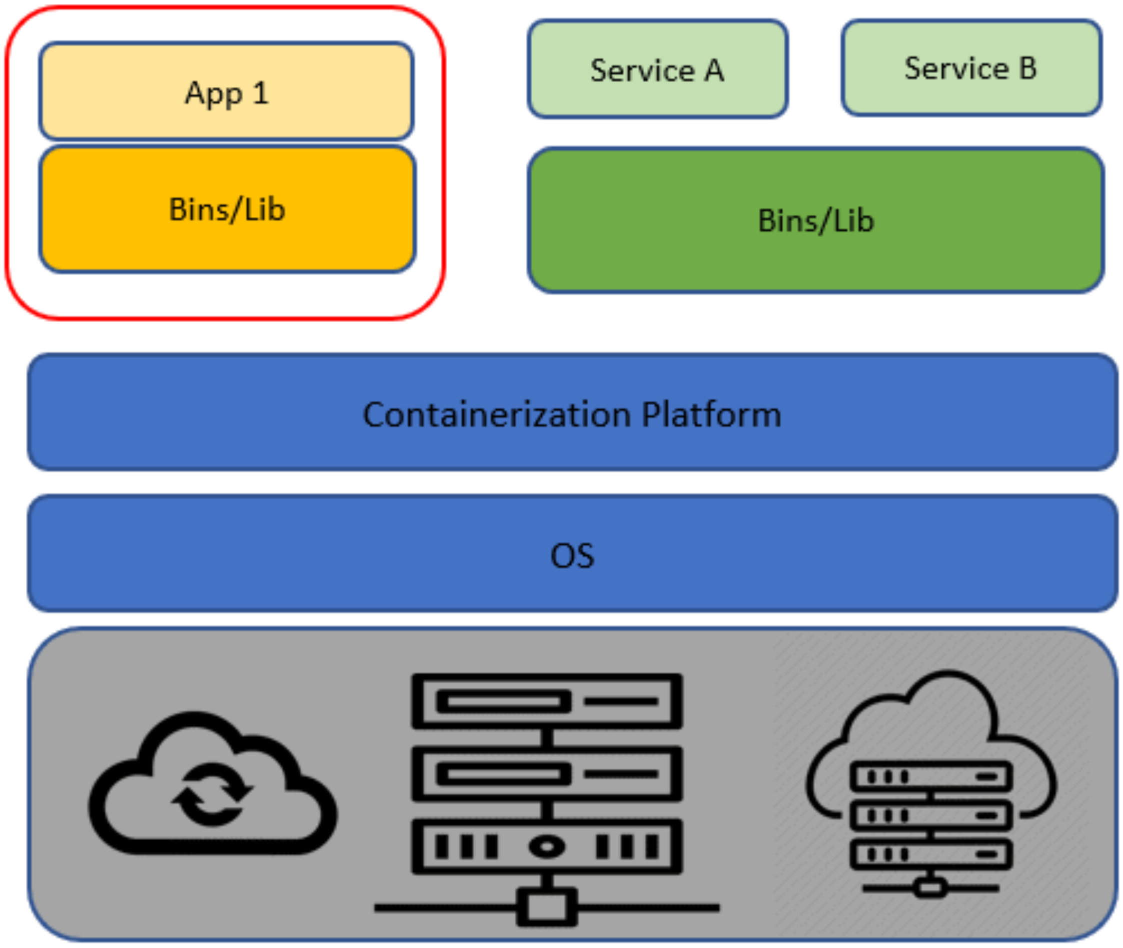
In the early days of computing, applications ran directly on physical servers ("bare metal"). Teams purchased, racked, stacked, powered on, and configured every new machine. This was very time-consuming just to get started.

Then came hardware virtualization. It allowed multiple virtual machines to run on a single powerful physical server. This enabled more efficient utilization of resources. But provisioning and managing VMs still required heavy lifting.

Next was infrastructure-as-a-service (IaaS) like Amazon EC2. IaaS removed the need to set up physical hardware and provided on-demand virtual resources. But developers still had to manually configure VMs with libraries, dependencies, etc.

Platform-as-a-service (PaaS) like Cloud Foundry and Heroku was the next big shift. PaaS provides a managed development platform to simplify deployment. But inconsistencies across environments led to "works on my machine" issues.

This brought us to Docker in 2013. Docker improved upon PaaS through two key innovations.



Container technology is often compared to virtual machines, but they use very different approaches.

A VM hypervisor emulates underlying server hardware such as CPU, memory, and disk, to allow multiple virtual machines to share the same physical resources. It installs guest operating systems on this virtualized hardware. Processes running on the guest OS can't see the host hardware resources or other VMs.

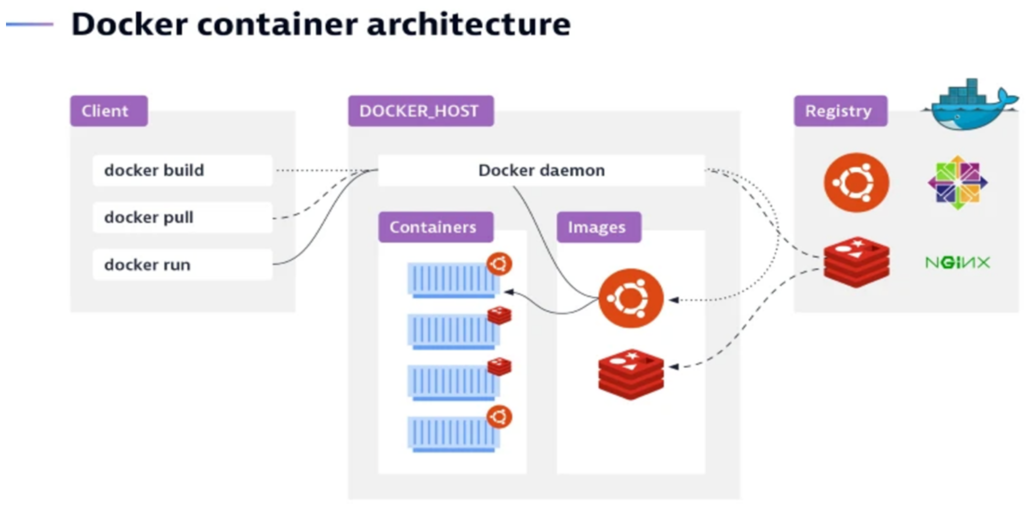
In contrast, Docker containers share the host operating system kernel. The Docker engine does not virtualize OS resources. Instead, containers achieve isolation through Linux namespaces and control groups (cgroups).

Namespaces provide separation of processes, networking, mounts, and other resources. cgroups limit and meter usage of resources like CPU, memory, and disk I/O for containers. We'll visit this in more depth later.

This makes containers more lightweight and portable than VMs. Multiple containers can share a host and its resources. They also start much faster since there is no bootup of a full VM OS.

Docker is not "lightweight virtualization" as some would describe it. It uses Linux primitives to isolate processes, not virtualize hardware like a hypervisor. This OS-level isolation is what enables lightweight Docker containers.

How does Docker work?

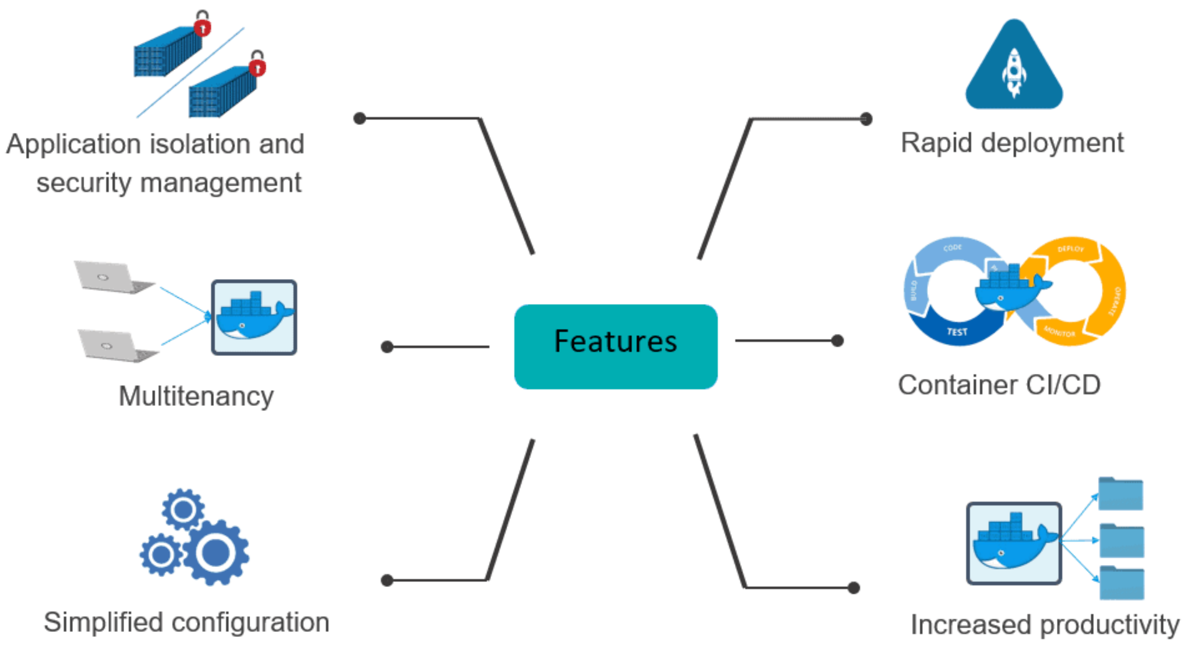


Docker's client-server architecture, the client talks to the daemon, which is responsible for building, running, and distributing Docker containers. While the Docker client and daemon can run on the same system, users can also connect a Docker client to a remote Docker daemon.

Why Do We Need Containers?

The main advantage of containers is that they are lightweight and portable and thus helps the developer a lot in configuring and deploying their application. There are many reasons for using Containers but only some of them are listed below:

- **Lightweight:** Containers share the machine OS kernel and therefore they don't need a full OS instance per application. This makes the container files smaller and This is the reason why Containers are smaller in size, especially compared to virtual machines. As they are lightweight, thus they can spin up quickly and can be easily scaled horizontally.
- **Portable:** Containers are a package having all their dependencies with them, this means that we have to write the software once and the same software can be run across different laptops, cloud, and on-premises computing environments without the need of configuring the whole software again.
- **Supports CI/CD:** Due to a combination of their deployment portability/consistency across platforms and their small size, containers are an ideal fit for modern development and application patterns—such as DevOps, serverless, and microservices.
- **Improves utilization**: Containers enable developers and operators to improve CPU and memory utilization of physical machines.



Different Types of Containers

The very growth and expansion in container technology bring a large set of choices to choose from. Docker is the best known and most widely used container platform by far. But there are some more technologies on the container landscape, each with their own individual use cases and advantages.

Docker

Docker is one of the most popular and widely used container platforms. It enables the creation and use of Linux containers. Docker is a tool which makes the creation, deployment and running of applications easier by using containers. Not only the Linux powers like Red Hat and Canonical have embraced Docker, but the companies like Microsoft, Amazon, and Oracle have also done it. Today, almost all IT and cloud companies have adopted Docker.

Know more about **Docker Architecture & it’s components**.

LXC

LXC is an open-source project of **LinuxContainers.org**. The aim of LXC is to provide isolated application environments that closely resemble virtual machines (VMs) but without the overhead of running their own kernel.

LXC follows the Unix process model, in which there is no central daemon. So, instead of being managed by one central program, each container behaves as if it’s managed by a separate program. LXC works in a number of different ways from Docker. For example, we can run more than one process in an LXC container, whereas Docker is designed in such a way that running a single process in each container is better.

CRI-O

CRI-O is an open-source tool which is an implementation of the Kubernetes CRI (Container Runtime Interface) to enable using OCI (Open Container Initiative) compatible runtimes. Its goal is to replace Docker as the Container engine for Kubernetes. It allows Kubernetes to use any OCI-compliant runtime as the container runtime for running pods. Today, it supports runc and Kata Containers as the container runtimes but any OCI-conformant runtime can be used.

rkt

The rkt has a set of supported tools and community to rival Docker. rkt containers also known as Rocket, turn up from CoreOS to address security vulnerabilities in early versions of Docker. In 2014 CoreOS published the App Container Specification in an effort to drive innovation in the container space which produced a number of open-source projects.

Like LXC, rkt doesn't use a central daemon and thereby provides more fine-grained control over your containers—at the individual container level. However, unlike Docker, they're not complete end-to-end solutions. But they are used with other technologies or in place of specific components of the Docker system.

Podman

Podman is an open-source container engine, which performs much of the same role as the Docker engine. But the difference between them is the way in which they work. Like rkt and LXC, Podman also does not have a central daemon but Docker follows the client/server model which is, using a daemon to manage all containers.

In Docker, if the daemon goes down, we also lose control over the containers. But in Podman, containers are self-sufficient, fully isolated environments, which we can manage independently of one another. In addition, Docker gives root permission to the container user by default, whereas non-root access is standard in Podman. Altogether, this isolation and user privilege features make Podman more secure by design.

runC

runC is a lightweight universal OS container runtime. It was originally a low-level Docker component, which worked under the hood embedded within the platform architecture. However, it has since been rolled out as a standalone modular tool. The idea behind the release was to improve the portability of containers by providing a standardized interoperable container runtime that can work both as part of Docker and independently of Docker in alternative container systems.

As a result, runC can help you avoid being strongly tied to specific technologies, hardware or cloud service providers.

containerd

containerd is basically a daemon, supported by both Linux and Windows, that acts as an interface between your container engine and container runtimes. It provides an abstracted layer that makes it easier to manage container lifecycles, such as image transfer, container execution, snapshot functionality and certain storage operations, using simple API requests.

Similar to runC, containerd is another core building block of the Docker system that has been separated off as an independent open-source project.

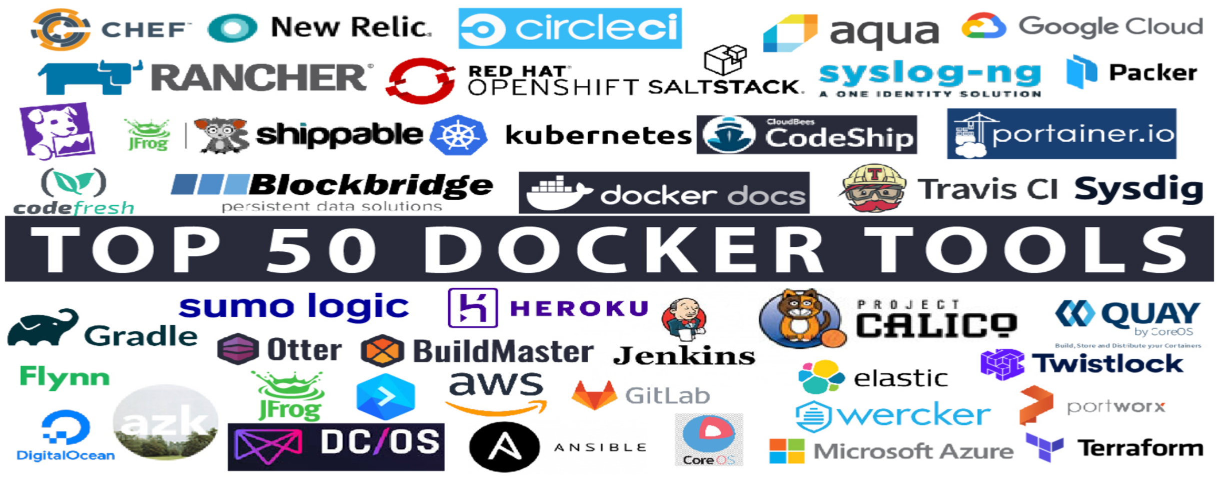
Difference Between Docker And Containers

Docker has become the synonym of containers because it is the most popular and widely used container platform. But container technology is not new, it has been built into Linux in the form of LXC for over 10 years, and similar operating-system-level virtualization has also been offered by FreeBSD jails, AIX Workload Partitions and Solaris Containers.

Developer tools for building container images

Docker Build creates a container image, the blueprint for a container, including everything needed to run an application – the application code, binaries, scripts, dependencies, configuration, environment variables, and so on. Docker Compose is a tool for defining and running multi-container applications. These tools integrate tightly with code repositories (such as GitHub) and continuous integration and continuous delivery (CI/CD) pipeline tools (such as Jenkins).

https://blog.inedo.com/devops/top-50-docker-tools/



Sharing images

Docker Hub is a registry service provided by Docker for finding and sharing container images with your team or the public. Docker Hub is similar in functionality to GitHub.

Running containers

Docker Engine is a container runtime that runs in almost any environment: Mac and Windows PCs, Linux, and Windows servers, the cloud, and on edge devices. Docker Engine is built on top containerd, the leading open source container runtime, a project of the Cloud Native Computing Foundation (CNCF).

Built-in container orchestration

Docker Swarm manages a cluster of Docker Engines (typically on different nodes) called a swarm. Here the overlap with Kubernetes begins.

Container Orchestration Challenges?

Although Docker Swarm and Kubernetes both approach container orchestration a little differently, they face the same challenges. A modern application can consist of dozens to hundreds of containerized microservices that need to work together smoothly. They run on multiple host machines, called nodes. Connected nodes are known as a cluster.
Hold this thought for a minute and visualize all these containers and nodes in your mind. It becomes immediately clear there must be a number of mechanisms in place to coordinate such a distributed system. These mechanisms are often compared to a conductor directing an orchestra to perform elaborate symphonies and juicy operas for our enjoyment. Trust me, orchestrating containers is more like herding cats than working with disciplined musicians (some claim it's like herding Schrödinger's cats). Here are some of the tasks orchestration platforms are challenged to perform.

Container deployment

In the simplest terms, this means to retrieve a container image from the repository and deploy it on a node. However, an orchestration platform does much more than this: it enables automatic re-creation of failed containers, rolling deployments to avoid downtime for the end-users, as well as managing the entire container lifecycle.

Scaling

This is one of the most important tasks an orchestration platform performs. The "scheduler" determines the placement of new containers so compute resources are used most efficiently. Containers can be replicated or deleted on the fly to meet varying end-user traffic.

Networking

The containerized services need to find and talk to each other in a secure manner, which isn't a trivial task given the dynamic nature of containers. In addition, some services, like the front-end, need to be exposed to end-users, and a load balancer is required to distribute traffic across multiple nodes.

Observability

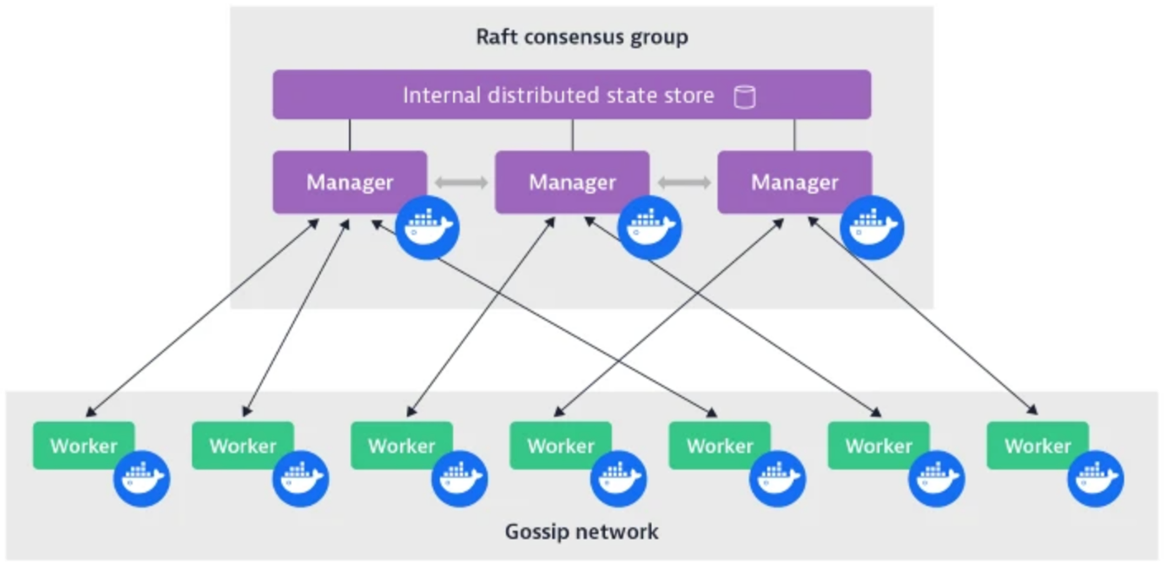
An orchestration platform needs to expose data about its internal states and activities in the form of logs, events, metrics, or transaction traces. This is essential for operators to understand the health and behaviour of the container infrastructure as well as the applications running in it.

Security

Security is a growing area of concern for managing containers. An orchestration platform has various mechanisms built in to prevent vulnerabilities such as secure container deployment pipelines, encrypted network traffic, secret stores and more. However, these mechanisms alone are not sufficient, but require a comprehensive DevSecOps approach.

With these challenges in mind, let's take a closer look at Docker Swarm.

Docker Swarm architecture



A swarm is made up of one or more nodes, which are physical or virtual machines running in Docker Engine.

Swarm seamlessly integrates with the rest of the Docker tool suite, such as Docker Compose and Docker CLI, providing a familiar user experience with a flat learning curve. As you would expect from a Docker tool, Swarm runs anywhere Docker does and it's considered secure by default and

Installation

Installation on Windows with WSL2

Corporates now can ONLY utilized a licensed version of Docker Desktop Client on Windows.

Pre-requisites

1. Windows 10 version 2004 or higher and specific build

- Specific build details are: **Windows 11 or Windows 10, Version 1903, Build 18362 or later**.
- To check which version is installed press Windows logo key + R and press Enter to view Windows build details.

NOTE: Windows upgrade would be required if it does not meet these version and build details for WSL2 to work,.

2. Disconnect VPN if u are inside a corporate firewall

Installation on Ubuntu Virtual Machine

OS requirements

To install Docker Engine, you need the 64-bit version of one of these Ubuntu versions:

- Ubuntu Oracular 24.10
- Ubuntu Noble 24.04 (LTS)
- Ubuntu Jammy 22.04 (LTS)
- Ubuntu Focal 20.04 (LTS)

Docker Engine for Ubuntu is compatible with x86_64 (or amd64), armhf, arm64, s390x, and ppc64le (ppc64el) architectures.

https://docs.docker.com/engine/install/ubuntu/

References:

https://k21academy.com/docker-kubernetes/what-are-containers/#3

https://blog.bytebytego.com/p/a-crash-course-in-docker

https://www.dynatrace.com/news/blog/kubernetes-vs-docker/

https://codefresh.io/learn/kubernetes-management/kubernetes-tools/

https://blog.inedo.com/devops/top-50-docker-tools/

https://pureinfotech.com/install-windows-subsystem-linux-2-windows-10/

https://stackoverflow.com/questions/38775954/sudo-docker-compose-command-not-found

https://askubuntu.com/questions/1308897/ubuntu-installed-but-windows-subsystem-for-linux-has-no-installed-distributions

https://github.com/microsoft/WSL/issues/5393

https://docs.microsoft.com/en-us/windows/wsl/setup/environment

https://docs.microsoft.com/en-us/windows/wsl/basic-commands

https://docs.microsoft.com/en-us/windows/wsl/troubleshooting

https://stackoverflow.com/questions/63497928/ubuntu-wsl-with-docker-could-not-be-found

https://betterprogramming.pub/how-to-install-docker-without-docker-desktop-on-windows-a2bbb65638a1

https://askubuntu.com/questions/1030179/package-docker-ce-has-no-installation-candidate-in-18-04

https://stackoverflow.com/questions/62681041/ubuntu-18-04-on-wsl2-logon-failure-the-user-has-not-been-granted-the-requeste

https://docs.microsoft.com/en-us/windows/wsl/compare-versions