A Crash Course in Docker

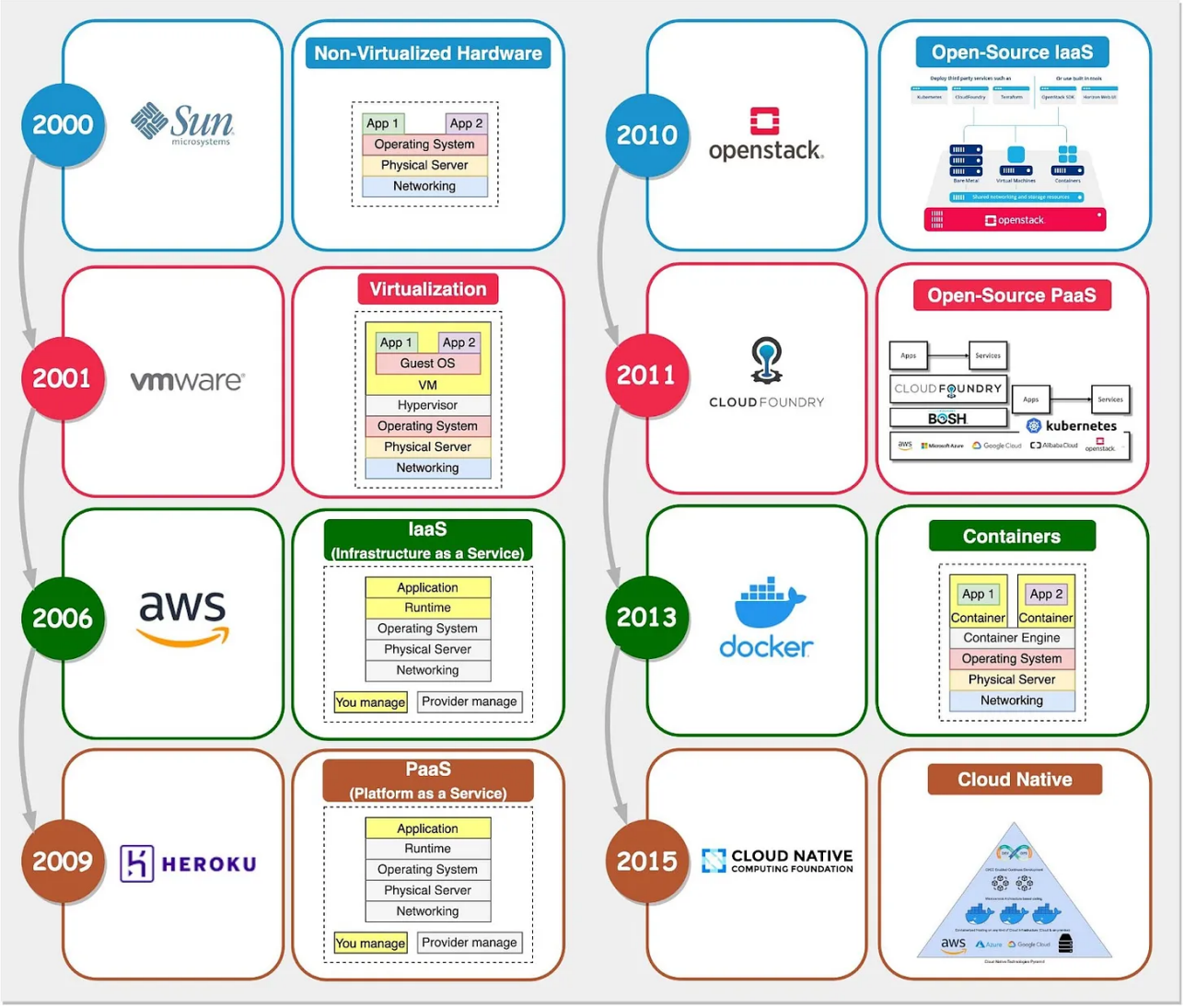
BYTEBYTEGO
NOV 9, 2023 · PAID

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.

In this newsletter, we'll explore the history of container technology, the specific innovations that powered Docker's meteoric rise, and the Linux fundamentals enabling its magic. We'll explain what Docker images are, how they differ from virtual machines, and whether you need Kubernetes to use Docker effectively. By the end, you'll understand why Docker has become the standard for packaging and distributing applications in the cloud.

Tracing the Path from Bare Metal to Docker

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



Reference: Openstack

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.

Lightweight Containerization

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.

Application Packaging

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 applications

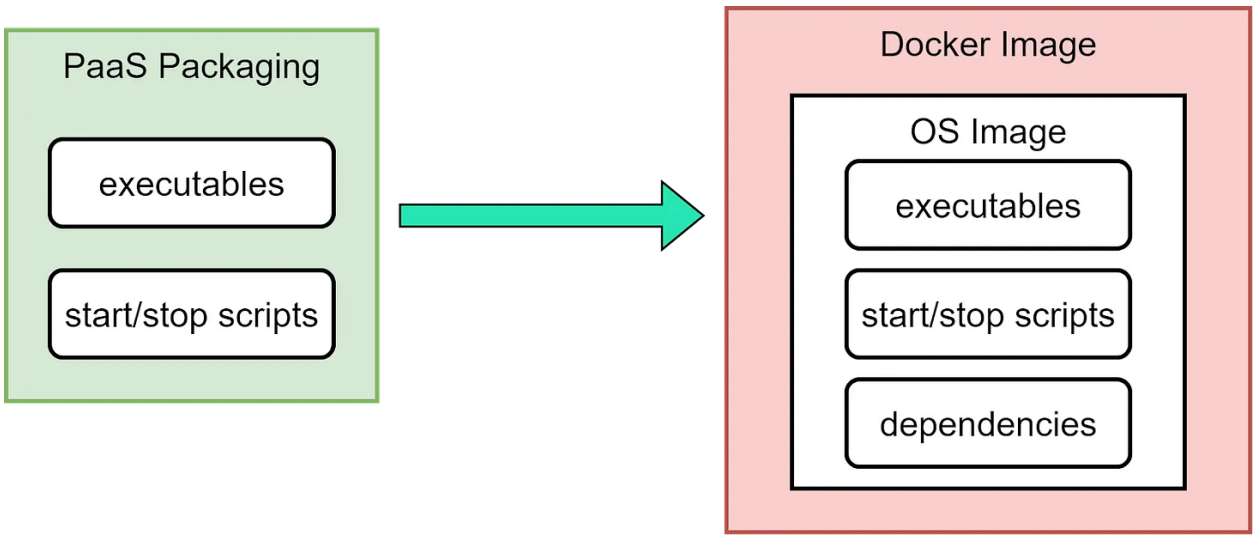
Cloud Foundry relied on Linux containers under the hood to provide isolated application sandbox environments. However, this core container technology powering Cloud Foundry was not exposed as a user-facing feature or highlighted as a key architectural component.

The companies offering Cloud Foundry PaaS solutions overlooked the potential of unlocking containers as a developer tool. They failed to recognize how containers could be transformed from an internal isolation mechanism to an externalized packaging format.

Docker became popular by solving two key PaaS packaging problems with container images:

1. Bundling the app, configs, dependencies, and OS into a single deployable image
2. Keeping the local development environment consistent with the cloud runtime environment

The diagram below shows a comparison.



This elegantly addressed dependency and compatibility issues that plagued PaaS. But Cloud Foundry did not adapt to support Docker images fast enough. This allowed

Docker images to proliferate in the cloud computing environment.

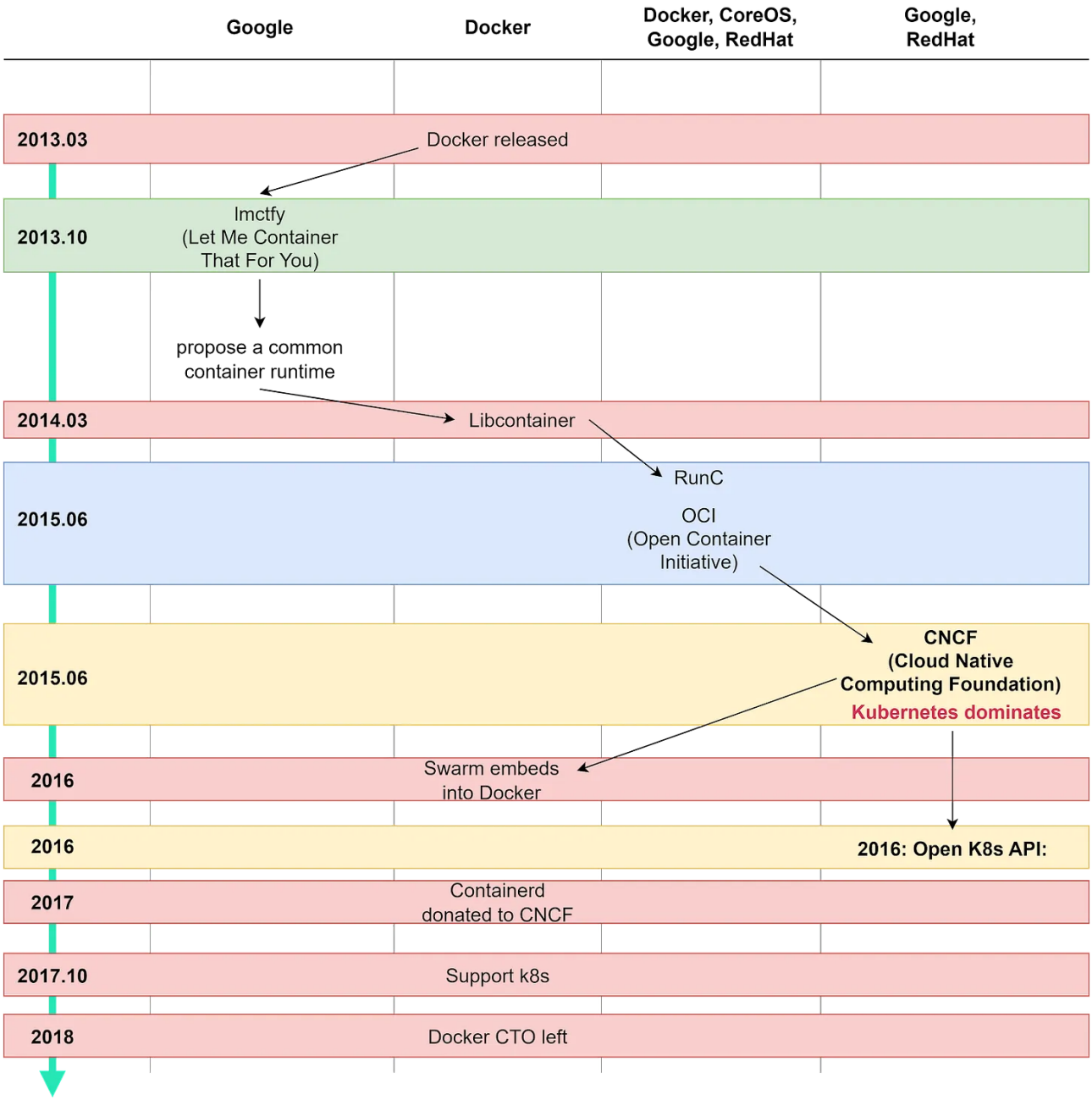
From Docker to Kubernetes

Docker won early popularity because it innovated in application packaging and deployment. Its initial success was largely due to this novel method of isolating applications in lightweight containers.

As Docker's popularity grew, the company sought to expand its offerings beyond containerization. It ventured to expand into a full PaaS platform. This led to the development of Docker Swarm for cluster management and the acquisition of Fig (later Docker Compose) to enhance orchestration capabilities.

Docker’s aspirations caught the attention of some tech giants. Companies like Google, RedHat, and other PaaS companies wanted in on this hot new technology.

Let’s see what happened between 2013 and 2018 with the diagram below:



In 2013, after Docker's initial release, Google open-sourced its internal container technology called lmctfy (Let Me Container That For You). However, lmctfy failed to gain much community traction. Google then proposed collaborating with Docker on a common container runtime, but Docker declined.

Google decided it needed to develop its own container solution leveraging its years of internal experience running containers at scale.

In 2014, Docker released its own container runtime called Libcontainer.

In 2015, Docker, CoreOS, Google, and other players including RedHat came together to announce that Libcontainer would be renamed to RunC and governed by the Open

Container Initiative (OCI) foundation. This was a move orchestrated by Google to establish neutral governance over key container technology.

Meanwhile, Google and RedHat started work on the Cloud Native Computing Foundation (CNCF) to challenge Docker's dominance more broadly.

Docker still embedded Swarm into its engine to compete in orchestration. But CNCF chose a different route - instead of a monolithic platform, they opened up Kubernetes' API to spur an ecosystem of modular cloud-native tools.

This strategy was brilliant. It birthed star projects like Prometheus, Envoy, OpenTracing, and more. Kubernetes also benefited from Google's Borg orchestrator, which powered its container workloads internally for over a decade.

Docker struggled to keep up. It donated the Containerd runtime to CNCF and eventually added Kubernetes support to Docker Enterprise. But the battle was lost for Docker as an orchestration platform - CNCF and Kubernetes won in container orchestration. In 2018, Docker's CTO resigned as Kubernetes cemented its leadership role.

However, Docker the technology remains essential and widely used. Docker containers continue to thrive as the standard for packaging applications in the cloud native world, underpinned by OCI standards.

The history between Docker and Kubernetes illustrates how open collaboration can outmaneuver attempts at centralized control. By opening up the orchestration layer through Kubernetes, CNCF reshaped the landscape of container technology. With the historical context, let's now dive deeper into the technical details that make containers possible.

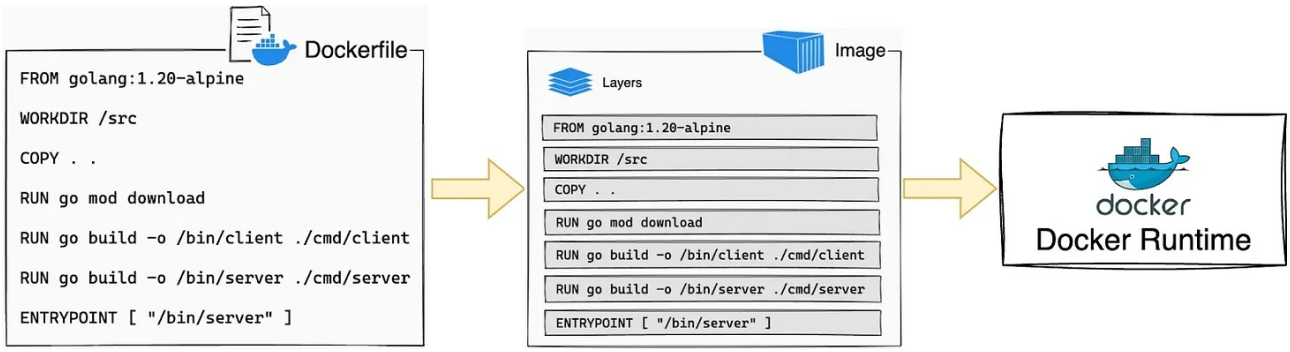
Image & Runtime

A container image bundles up code, dependencies, configs, and a root filesystem into a single static artifact. It is like a compiled executable, containing everything needed to run the application code in an isolated environment.

The Docker engine unpacks this image and starts a container process to run the application in an isolated namespace, like a sandboxed process on Linux.

The image provides a static view of the application to be run. The container runtime gives a dynamic instance of the image executing. The image is the build artifact, while the runtime provides an isolated environment to launch it.

This clean separation of immutable image and ephemeral runtime is a key concept, as illustrated in the diagram below:



Source: Docker docs

The bottom layers of the image provide the base OS and dependencies. The top layer adds in the application code and configs. When started by the container runtime, the image creates an isolated process sandbox to run dynamically.

This separation of static image and dynamic runtime solves dependency issues in PaaS. The image acts as a consistent packaging format from dev to test to prod. The OS-level isolation ensures portability across environments.

Let's look at the Linux capabilities powering this image/runtime separation.

Cgroups, Namespace, and rootfs

Namespace - A container is a special type of process

When starting a container, it is just a specially isolated process on the host Linux OS. For example:

```
$ docker run -ti debian /bin/bash
```

This starts a Bash process in a new container. The -i flag keeps STDIN open and -t allocates a pseudo-TTY.

If we run the command ps in the container, we only see the /bin/bash process with PID 1. The container is isolated from viewing other host processes.

However, the host OS can still see the container's process, just with a different PID. This namespace isolation is achieved using the clone() system call:

```
int pid = clone(main_func, stack_size, CLONE_NEWPID | SIGCHLD, NULL);
```

The CLONE_NEWPID flag creates the process in a new PID namespace. So within the container, its Bash process PID is 1, but on the host, it has a different PID.

Namespace isolation provides separation of not just process IDs, but other resources as well. For example:

- *Mount namespace* provides isolation of the list of mounts seen by the processes in each namespace instance.
- *Network Namespace* provides isolation of the system resources associated with networking: network devices, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, etc.

Cgroups - Limit container resource usage

Namespace isolation provides separation between containers and the host. But containers still share the underlying host OS kernel and resources.

This differs from virtual machines, which run a full guest OS on virtualized hardware. VMs do not share resources with the host or other VMs.

Containers take a different approach - they run on the real host OS while using cgroups and namespaces for isolation. This makes them more lightweight than VMs.

We need a way to limit and restrict how much of the host resources a container can use. This is accomplished using Control Groups (cgroups).

Cgroups allow setting quotas and limits on resources like:

- CPU - limit CPU time allowed for the container

- Memory - restrict max memory the container can use
- Disk I/O - throttle disk access bandwidth
- Network - control incoming/outgoing traffic

For example, we can limit a container's CPU usage to 20 milliseconds out of every 100 milliseconds,

```
$ echo 20000 > /sys/fs/cgroup/cpu/container/cpu.cfs_quota_us

$ echo 100000 > /sys/fs/cgroup/cpu/container/cpu.cfs_period_us

$ echo $PID > /sys/fs/cgroup/cpu/container/tasks
```

The equivalent Docker command would be:

```
$ docker run -ti –cpu-period 100000 –cpu-quota 20000 debian /bin/bash
```

Cgroups ensure containers only use their fair share of resources on a host and don't starve other containers. This provides resource management within the OS for containerized processes.

rootfs - The container filesystem

Containers should have a clean filesystem instead of inheriting everything from the host OS. The chroot command can be used to change a process's root directory.

For example:

```
$ chroot /tmp/home /bin/bash
```

This will make /tmp/mycontainer the root filesystem for the Bash process.

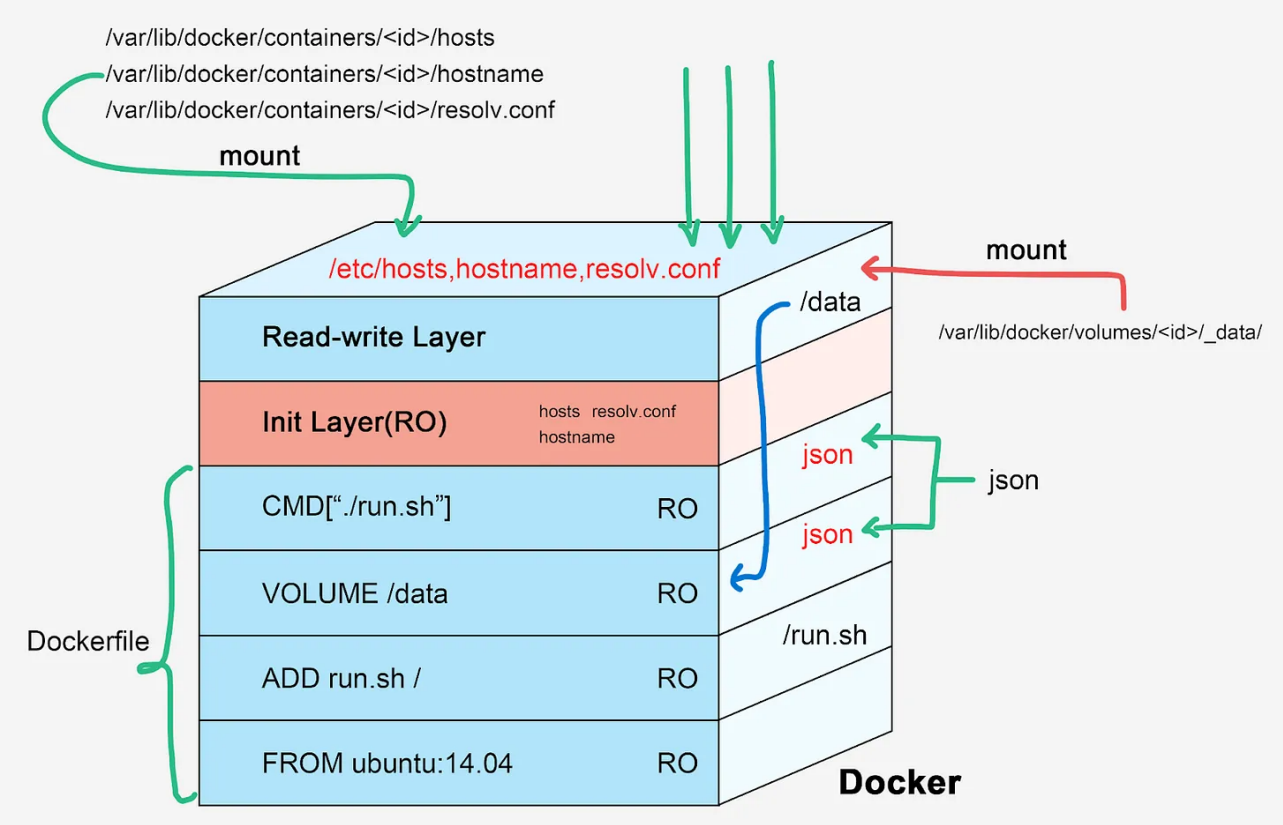
Docker builds on chroot and mount namespaces with layered images and copy-on-write filesystems. A Dockerfile defines the steps to assemble an image - each

command creates a new filesystem layer. When a container is started from the resulting image, a thin writable layer is added on top.

The Dockerfile gives complete control over the image contents. Each instruction adds a new layer, enabling immutable infrastructure patterns. The running container gets a writable layer to modify the state, without affecting the underlying image defined in the Dockerfile.

This separation of immutable image from writable container state is a key benefit of Docker's image/runtime architecture. The Dockerfile provides a recipe for building portable and immutable images that can be reliably reproduced.

The diagram illustrates this:



Source: Astrotech

The bottom layers provide the root filesystem from the image. The top writable layer captures changes at runtime.

When the container stops, the writable layer can be discarded, leaving a pristine image filesystem for the next container.

This approach enables immutable infrastructure patterns - containers can be freely started, topped, and replaced without worrying about persisting state.

The Docker Landscape Today

While the core Docker technology remains popular, especially in Kubernetes environments, the way Docker images are built and run has evolved:

- Dockerfile is still the standard for defining Docker images, but many teams now build images through CI/CD pipelines instead of manually running docker build.
- The containerd and CRI-O runtimes are gaining adoption as lower-level alternatives to the Docker engine, especially in Kubernetes clusters.
- Docker Swarm has declined in usage for orchestration. Kubernetes is now the dominant orchestrator managing containers at scale.
- Docker Compose remains popular for local development and testing containers.

While the container landscape has diversified, Docker's fundamental impact on developer workflow through standardized packaging remains. But the surrounding ecosystem has shifted as new runtimes, orchestrators and developer tools emerged.

Summary

In this newsletter, we explored the history of containers and Docker's origins in tackling PaaS packaging issues.

We discussed how Docker achieved lightweight isolation for processes using namespaces and cgroups built into the Linux kernel. This allowed portable and reproducible images to be defined using Dockerfiles and easily shared.

Docker revolutionized application delivery by finally solving dependency and environment inconsistencies. It quickly became the standard for packaging and deploying apps in the cloud.

While Docker usage has evolved with new runtimes and orchestrators, it remains a fundamental technology for development and DevOps teams operating in the cloud. Docker provided key innovations in image building, containerization, and reproducible environments that changed application development forever.

66 Likes · 1 Restack

Comments

Write a comment...