Docker and containers: from their birth up to the present day

5 August 2021 - 11 min. read

Containers | Docker

If you work in the Information Technology field, you will surely have heard of *software containers* regardless of your job position. Maybe you work on them every day, maybe you've worked on them a few times (like me), or maybe you've just heard of them.

We have heard about this technology for no more than 10 years, but the container concept is much older.

This article aims to revisit the history of containers and understand where they come from up to the present day. We will discuss the main standard (Open Container Initiative) and clarify the different technological components (image, container, runtime). Finally, we will see possible alternatives and some applications in the AWS world.

What is a container?

It's a big deal giving a simple answer to this question, let's start with what is reported on the Docker homepage:

A software container is a software unit that contains code and all its dependencies so that the application can be run in the same way in different computational environments.

Let's try to analyze the answer deeper. When an application is packaged in a container, it can be run on different machines with the certainty that it will run in the same way. From this point of view, it's very similar to a Virtual Machine.

But the container is much more than that.

Let's start with the etymology of the word *container*. We are all familiar with the containers used on ships, trains, and trucks to transport goods. So, let's imagine *software containers* as a sort of envelope for our application, with their own size, standard, and locking mechanisms to the outside world. Thanks to its shape, any container can be moved from one side to the other without major problems, regardless of its content.

But what is the content of a software container? It's a set of dependencies, libraries, together with the application code, saved in the form of an image (*container image*) that can be run on any machine that supports the execution of containers. It follows that the same software, contained in a container, will behave the same way whether it is run on the developer's machine, on the on-premise server, or on the virtual machine in the cloud.

In general, any application also needs its dependencies, which can vary from machine to machine (due to different hardware architecture and operating systems). To keep consistency, the code should be prepared together with its dependencies.

Indeed, Virtual Machines (VMs) have been used in the software industry for years. The virtual machine is nothing more than an emulation of a computer, including the operating system, on which you can install all your necessary dependencies, making it independent from the host machine (code, dependencies, and operating system). However, it becomes challenging to use VMs as a software delivery tool, as you should use the entire image, including the operating system. I guess everyone has had to create a virtual machine on their computer and found its slowness (try it!)

Another problem with VMs is that they are tied to virtual hardware (via the hypervisor). A developer shouldn't worry about storage, networking, or processor type (or at least, not low-level). Do you remember the initial metaphor? A container should be independent of who transports it (ship, train, truck, etc.).

We have already said this, but another problem with VMs is performance; they require many hardware resources and generally suffer from high boot times. A virtual machine, although virtual, is still a complete machine, it is not suitable for the delivery of software.

In summary, to release reliable and repeatable software on different computers, we need an airtight box to put our code. This box should be agnostic to the system it runs on so that the developer can focus on developing the software and its close dependencies, not including machine details. Furthermore, it must be more performant than a virtual machine.

Container history

Let's briefly revisit the history of containers, including features of the Linux kernel, first open-source projects, and first companies that saw the potential of this technology, up to their widespread diffusion thanks to Docker.

1979: Unix V7

This year, a Linux kernel feature, *chroot*, is released, allowing you to change the root directory of a process. This result is only the beginning of the isolation of processes, a necessary mechanism in today's containers.

2001: Linux VServer

Linux VServer is among the first software supporting the so-called *jail mechanism,* a sort of virtualization at the operating system level that allows isolating and partitioning resources on a machine.

2002: Namespaces

The Linux kernel releases a new feature: *namespaces*. Namespaces allow you to partition hardware resources between a set of processes, limiting their visibility to the rest of the system.

2001-2007:

Many companies start investing in this technology: Solaris container (Oracle), Open VZ

2007: Cgroups

This is another feature of the Linux kernel. Indeed, the *cgroup* (short for control groups) is a Linux kernel feature that limits the processes' resources (CPU, memory, disk I / O, network, etc.).

2008: LXC

LXC (Linux Containers) was the first example of a modern container engine. Indeed, it exploits features of the Linux kernel used by the most recent container engines (cgroups, namespaces)

2013: Docker

Starting from LXC, Docker was born in 2013 as open-source software to run containers. Since then, the world of containers is about to change.

Over time it has implemented its container manager using its libraries (*libcontainer*).

2013 - Today

Since 2013, technologies have been consolidated, standards (OCI), container orchestrators (Kubernetes, Docker Swarm), and several alternatives (micro Virtual Machines) have been created.

Docker and the explosion of containers

Quite often, the words *Docker* and *container* are confused, but why? Simply because it is thanks to Docker that containers have become so popular.

Docker is a set of PaaS (Platform as a Service) products that enable the development and delivery of containerized software.

dotCloud, now Docker Inc, released Docker to facilitate software development by creating "standard boxes". Starting from the Linux kernel features previously seen, Docker wanted to facilitate the use of these low-level features by providing easy-to-use interfaces.

Docker open-sourced three key aspects that facilitated the use of containers, thus favoring their wide use:

- A standard for container format

- Tools for packaging (build) containers
- Tools for launching and running containers

As we have already seen, the image of a container is nothing more than a self-contained software package with source code, libraries, and related dependencies.

Once created, these images are standard and static. They no longer change (just like the contents of containers that are shipped from one part of the world to another). It is not the purpose of this article to go into the detail of the Docker image, but we can say that the container, once it's running, can only use a *writable layer* that is separated from the layer in which the image of our container is saved. Indeed, data stored inside a container are ephemeral.

Which are the steps to create a container image with Docker? Simple, by writing a "recipe", called *Dockerfile*: a text file that consists of precise instructions on how to package our standalone software. Looking at the Dockerfile, developers can conclude at a glance what the final content of the image will be and have full control over it.

Below is an example of Dockerfile using Ubuntu 18 as the starting image (FROM), copy the contents of the directory where the Dockerfile is into the *app* folder (COPY), execute a command while creating the image (RUN), and defines the command to be executed at the startup of the container (CMD)

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Simple, isn't it?

But Docker's features don't end there: once the image has been created, developers or other team members can exchange it among themselves and run it on their computer, on-premise, or in the cloud without having to install anything other than the Docker engine, using a convenient, and easy to use, Command Line Interface (CLI). The Docker runtime container does nothing but read the image content and execute it. Thanks to Linux namespaces, each container will run in a separate environment from the others.

Have you seen the benefits of Docker? Can you understand the reason for its popularity?

Docker did not upset the concept of container (portability, better performance than traditional VMs, the ability to modularize monolithic applications, offer greater scalability), but it was the first company to offer the right tools for software delivery, speeding up the process. Not surprisingly, these are also the years of the DevOps revolution (agility, flexibility, scalability in the delivery of software)

From Docker to OCI (Open Container Initiative)

From the first releases of Docker, many people started using containers as the standard unit for delivering their software. Companies also began to use Docker, and his team was not always able to meet all the technical and business needs that the market demanded. In response, several runtimes were born from the community, with different implementations and capabilities, together with new tools for creating and launching containers.

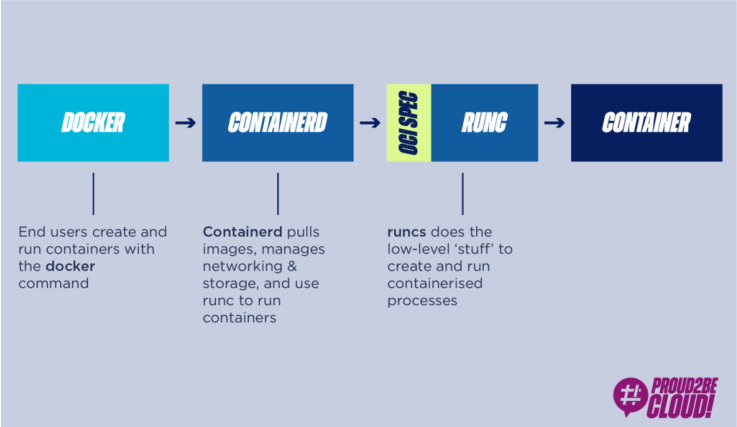
To ensure that different container runtimes could launch images produced by different tools, Docker and other community members founded the Open Container Initiative (OCI) in 2015 to define a standard around the world of containers.

The original Docker images have therefore become OCI image specifications.

Provided an OCI-compliant image, any runtime container that supports the OCI runtime can read the contents of an image and run it in an isolated environment.

Docker has not only donated the image creation standard, but also its container runtime, runc, and the containerd daemon, thus creating a completely modular ecosystem, as follows:

- **Docker**: The Docker suite includes the CLI to use commands with which to create and manage containers
- **Containerd**: a daemon that manages the lifecycle of containers. Save and download images, manage memory and networking,
- **Runc**: the low-level container runtime, the one who actually creates and runs the containers. It is written in Go



OCI is a group of tech companies that maintains a specification for the image of containers and how they should be performed. It arises from the desire to be able to choose between different runtimes that conform to the same specifications, having different low-level implementations

And Windows OS?

So far, we have talked about containers closely related to the Linux world. But in all of this, where does Windows rank? Let's try to answer.

On windows, you basically have two ways to run containers:

- Directly on the host machine, with what Microsoft calls Windows Server Container. Just like the containers described so far, indeed, they share a single kernel.
- Using a virtual machine managed by Hyper-V, a hypervisor system from Microsoft. Each Hyper-v container is a totally isolated virtual machine, each with its kernel

In the Linux world, Docker is almost completely open-source and supports all Linux distros. To use containers on this operating system as well, Docker and Windows work closely together to bring and maintain the containers on this operating system.

Non-OCI containers and alternatives

But a question arises: are there alternatives to the OCI standard? Of course! Let's see some examples:

- **Crun**: A runtime container written in C (unlike runc, which is written in Go).
- **Kata-runtime**: born from the katacontainers project. It partly reflects the OCI standard, with the difference that it is not containers that are executed, but micro Virtual Machines (we will deal with them shortly!).

- **gVisor**: Google opensource project to create containers with their own Kernel. They only implement the OCI runtime container (called runsc). We can consider them a cross between containers (in terms of Linux namespaces) and micro VMs. The applications that run inside the gVisor system aim to be more secure, starting from the fact that they communicate less with the underlying Linux kernel, thus reducing the attack surface from unwanted workloads.
- **Firecracker**: A runtime optimized for serverless workloads. It is the technology used by AWS as a runtime for Lambda and Fargate. Firecracker runs containerized applications inside a micro VM. Read micro VMs optimized for single applications. If you are curious, the project is available on Github!

Among the alternatives that do not comply with the OCI standard, there are, therefore, different solutions to containers, we are talking above all about micro VMs. Are we leaping into the past then? Absolutely not! Generally speaking, we can say that micro VMs are very light VMs, with their own reduced version of the kernel, which offers greater isolation and process security levels than containers. It explains why, for example, AWS has chosen this mode to offer greater security to its customers who choose serverless computational power (Lambda and Fargate).

Container and Cloud Computing

Containers are now widely used as a software delivery tool, but can they also be done in the cloud? Of course!

AWS offers services such as ECS (Elastic Container Service), already discussed in many articles on our blog, and EKS (Elastic Kubernetes Service) as container orchestrators. Furthermore, nothing prevents you from being able to autonomously orchestrate your own containers on EC2 machines, perhaps in autoscaling, but where would the advantage be?

Personally, in container-based architectures, I see the latter as the basic infrastructure unit to be managed, thus forgetting the maintenance of the underlying servers. If we think about it, it is precisely the philosophy behind containers: to have self-contained software elements, independent of the host on which they run (in terms of hardware and OS). For this reason, when I use ECS I choose it in Fargate mode: we build, manage and maintain containers, not servers!

To conclude

This article retraced the history of containers, from when they were born to their current applications. We covered the main advantages and differences compared to a classic virtual machine-based architecture.

Are you using containers? Or are you interested in learning more about the topic? Let us know in the comments!

PS: Did you know that it is possible to run your own containers on Lambda since the last AWS re: Invent?

Alessandro Bertini

DevOps Engineer @ beSharp. I deal with Cloud-Native software development, strongly oriented to the serverless paradigm!Passionate about board games and video games (as the best geeks do!)
