An essay on virtualization, containerization and its significance to DevOps

Simon Jäger
DD2482
KTH Royal Institute of Technology

Nowadays, virtualization has come to be used as a broad term. Not unexpectedly, as the concept can be applied in many areas around computer science. This has in turn allowed a multitude of exciting technologies and tools to be invented. Virtualization has been instrumental in the evolution and adoption of DevOps practices. Many of which we will explore in this paper.

1. Introduction

Virtualization describes the means of separating resources from actual physical implementations. This is done to lift some of the restrictions and boundaries that physical implementations instill in operating systems, applications, services and software development. For example, virtualization can provide software with more memory, well-beyond the physical limitation (*virtual memory*) - using installed RAM (random-access memory), HDDs (hard disk drive) and memory paging [1].

Naturally, virtualization techniques can be implemented across many layers; computation, networking, storage, operating systems, applications and more. With virtualization techniques for the entire hardware stack, entire computers can be virtualization - often called *virtual machines* [1].

The technological innovations and improvements that virtualization brings may not seem evident at first glance. To grasp the impact of virtualization, its impact on software development practices ought to be explored [1].

Before virtualization, a computer would be limited to running one operating system at the time. With software and hardware tightly coupled. Such strong coupling is error-prone and multiple applications and services running on the same machine would increase the risk of conflicts. Often, running multiple software simultaneously on the same machine would be avoided - rendering hardware and machines underutilized, inflexible and ultimately increasing costs [1].

Virtualization provides independence of different layers, such as between hardware and the operating system or applications. With software designed for a set of hardware, the role of virtualization is to serve that specification before the software - through a virtual implementation, a transparent interface. Leaving virtual machines unaware of the fact that its resources are in fact virtualized [1].

Subsequently, virtual machines can be provisioned in any quantity on any machine. This allows physical resources to be utilized much better; through safely being able to encapsulate software in virtual machines and run them simultaneously. Should any of the applications or services impact the environment in such a way that it would be left unstable or unusable (e.g. kernel crashes, overconsumption of resources) - only that virtual machine would be impacted. Leaving other virtual machines unaffected and the host machine intact. As such, operating systems and applications have come to regularly be encapsulated into this single unit - partly something that initiated the innovation and adoption of container virtualization [1].

1.1 Hypervisors

For virtualization to be achieved, the interface between virtual resources and physical resources need to be handled - through translations. Additionally, virtual machines need to be managed from a lifecycle perspective (e.g. creation, resumption, deletion). These tasks are performed by what is called a *hypervisor*, sometimes referred to as a VMM (virtual machine monitor). The machine where a hypervisor is run is called the *host machine*, whereas its virtual machines are called *guest machines*. There are two types of hypervisors, *Type 1* and *Type 2* [2].

Type 1 hypervisors run on the physical hardware of the host machine. Sometimes referred to as a bare-metal hypervisor. These hypervisors are considered the most efficient, as they need not to load an operating system before executing - creating a minimal layer between virtual machines and hardware. Additionally, this creates a very highly secure environment, by not having to rely on an intermediate operating system [2].

Type 2 hypervisors generally execute in the context of an operating system. These hypervisors depend on the operating system to relay work onto the hardware and allow for a much broader set of hardware configurations for virtualization [1].

Type 2 hypervisors provide virtualization at a higher level (software) compared to Type 1. This leads to Type 2 hypervisors being avoided for critical workloads due to the added latency and security risks. Such as for data center computing where virtual machines might not be trusted, as different customers might end up sharing the same hardware. Type 2 hypervisors are heavily used for container virtualization [2].

Hardware acceleration has played a big role in enabling both Type 1 and Type 2 hypervisors. Without investments from large CPU manufacturer (such as Intel and AMD) the hypervisors would be left to perform many of the compute intensive tasks (e.g. address translation, virtualization nesting) needed to perform virtualization without low-level support [2].

1.2 Levels of virtualization

With virtualization as a broad concept, it can subsequently be applied throughout various layers of implementation. Beginning at the ISA (instruction set architecture) level, an interpreter will translate instructions during execution one by one between the source ISA and target ISA. Without a direct match of instructions, multiple target instructions may be needed to perform the function of the source instruction [3].

For virtualization at the hardware level, virtual machines - the hypervisor will assume the responsibility of hosting the interface between physical resources and virtualized resources. Each virtual machine will run its own operating system and software as if its running directly on the hardware [3].

Unfortunately, hardware virtualization has proven suboptimal with regards to resource utilization when running multiple virtual machines. As each virtual machine must maintain its own operating system and required dependencies (e.g. shared libraries), storage and memory is wasted.

With innovations to improve resource utilization, operating system level virtualization, *containerization*, has become a significant tool alongside the introduction of Docker and Kubernetes. This provides isolated containers (separated user-space instances) on the host machine that share the operating system kernel, binaries and libraries through read-only mechanisms. This has allowed containers to be much smaller and provide great startup/shutdown times. While the containers perceive themselves as regular servers [3].

Libraries have also come to be virtualized through a process that controls the linking between applications and the operating system [3]. In example, this type of virtualization is leveraged by the software WINE, which allows Windows applications and services to run in Linux environments [6].

Application level virtualization (also known as process level virtualization), is the means of executing a compiled high-level language. Examples of this technology is the JVM (Java Virtual Machine) [4] and Microsoft .NET CLR (Common Language Runtime) [5] as both provide an abstraction for which the application can run on. This has proven greatly beneficial for enabling cross-platform execution and application isolation [3].

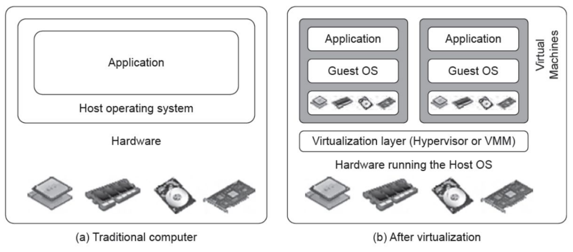


Figure 1: Layers of virtualization [3].

Virtualization, throughout the various layer serve the foundation of abstraction, replication and isolation in software development.

2. Containerization

Containerization has become an instrumental technology in software development. With a lightweight footprint through shared components with the host machine and swift boot times - it is easy to see how applications and services can benefit from being bundled into containers.

Containerization involves packaging an application with its configuration and various dependencies. Although the container will be running individual processes, most will share a common operating system - this is what allows improved utilization of hardware [7].

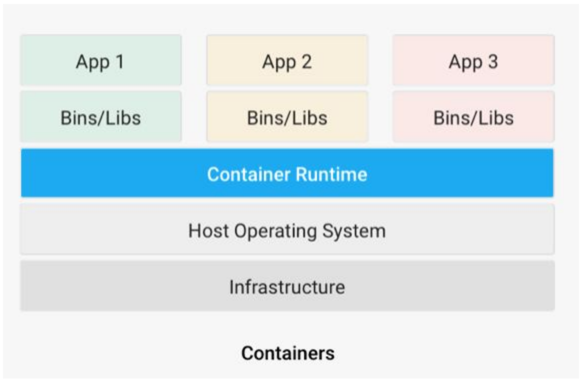


Figure 2: Containers running on a host machine [7].

As software engineers begun bundling their solutions, it was naturally discovered that applications could be broken down further and separated into their own units - their own containers (separation of concern). This has allowed application and services to be distributed and developed as part of a network of decoupled services. It has improved the fault tolerance of solutions, by removing a single point of failure (should one part of the solution crash, everything will be affected). Additionally, well defined bundles allow for better disaster recovery - where a container can be quickly restarted, should it crash. Each container would then communicate with other containers through APIs [7].

This strategy of building solutions has also lead way for *microservice* architectures - where scalability becomes a trivial task. Each microservice can be scaled independently of others, allowing for certain parts of solutions to meet new rising demands [7].

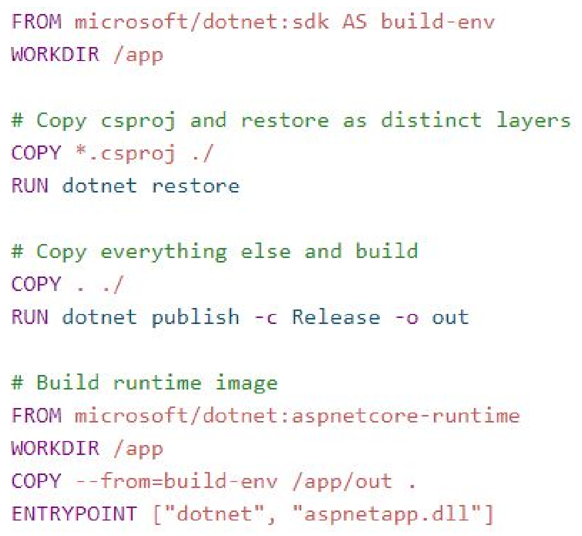
While sharing a common operating system and resources, containers provide an isolated environment where the container itself will not be able to reach outside the bounds of itself - unless explicitly allowed by the orchestration engine. However, containerization do introduce a set of security concerns. Sharing the same operating system means that a threat or compromise of the host machine, will lead to the entire system being compromised. Furthermore, any threat protection software will today only concern itself with the host machine - leaving containers uninspected [7].

Entire ecosystems of tools and resources have sprung up throughout the years for containerization, which many large companies are placing large investments upon (e.g. Microsoft, Google, IBM). Docker and Kubernetes have been instrumental technologies in the adoption and evolution of containerization [7].

2.1 Docker

Docker is a cross-platform and open-source project that is developed to ease the lifecycle management of containers and building solutions based on containers [8].

Docker consists of multiple tools and parts, one of which is the *Docker engine*. It is responsible for running the containers on the host machine. A container is defined through the definition of a *Dockerfile*. This is a text file with a set of properties that tell Docker how to build a *Docker image* (container image, binaries). It includes multiple details, such as which operating system it should run, environment variables, files, networking configurations and how to start the application. The Docker engine will use the Docker image to create and start the container (using the Docker run utility). Whereas the one Docker image can be used to provision multiple instances of the solution (or part of) with ease[8].



```
FROM microsoft/dotnet:sdk AS build-env
WORKDIR /app

# Copy csproj and restore as distinct layers
COPY *.csproj ./
RUN dotnet restore

# Copy everything else and build
COPY . ./
RUN dotnet publish -c Release -o out

# Build runtime image
FROM microsoft/dotnet:aspnetcore-runtime
WORKDIR /app
COPY --from=build-env /app/out .
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

Figure 1: Example of a Dockerfile [9].

Furthermore, Docker also hosts a large repository for Docker images named *Docker Hub* - where Docker images can be uploaded and downloaded. Many of which are from various software companies [8].

2.2 Kubernetes

An instrumental technology within the containerization space has been *Kubernetes*. It's an open source platform to manage containerized solutions - and works well together with Docker (but Docker is not required). Kubernetes eases lifecycle management, networking, load balancing and security of containers, across multiple different hosts - called *nodes*. Such nodes can be different types of machines; running Docker, virtual machines, bare-metal servers. These are consolidated into what is defined as a *cluster*. This is ideal for improving high-availability of solutions, as spreading out containers on more hardware reduces the risk of any single point of failure. Kubernetes brings mechanisms to ensure that containers are kept into a desired state, through replication and restarts to recover. This allows software engineers to improve the scalability of their solutions. As more hardware becomes available, it can transparently extend the cluster with additional nodes. In short, Kubernetes allows for orchestration and automation of orchestrion of containers [10].

3. Significance to DevOps

Containerization has contributed to both software engineers and IT administrators in terms of enabling DevOps practices. Through tools such as Docker, environments have been possible to define (e.g. Dockerfile) and used to deploy solutions identically on both local environments, test, staging and production environments. This have improved CI (continuous integration) practices, whereas clean environments can quickly be spun up for automated testing and other quality assurances. It's important to note that with the ability to completely define the environments and software, comes a multitude of benefits. It integrates the environment definition into source control, allowing it to be tested and validated throughout a CI system - a responsibility that used to be given to IT administrators (operations) to handle somewhat independently [11].

CD (continuous deployment) has evolved further with the combination of containerization and vast container repositories (e.g. Docker Hub). Where hosting and versioning of containers can be abstracted and retrieved with ease as part of build and release pipelines. This has also improved recoverability measures, such as being able to deploy an earlier working version of the application. As any previous version of container image, contains both the solution and its desired environment in a single unit [11].

Subsequently, software engineers have had to invest less into hardware constraints - allowing them to divert more time and energy onto software development [11].

In conclusion, containerization brings incredible abilities to define solutions in its entirety; its binaries and environment with tools such as Docker. Orchestration allows for container lifecycle management to happen across multiple machines, to be automated and to produce high-availability and scalability. These tools abstract hardware concerns away from the software engineers and promotes automation. Something that ties well into CI/CD practices, DevOps. Allowing software engineers to focus on building great software.

References

[1] VMware, Inc. 2006. *Virtualization Overview*. [Online]. Available at: https://www.vmware.com/pdf/virtualization.pdf. [Accessed April 20, 2019].

[2] Bigelow S. 2010. *What's the difference between Type 1 and Type 2 hypervisors?*. [Online]. Available at: https://searchservervirtualization.techtarget.com/feature/Whats-the-difference-between-Type-1-and-Type-2-hypervisors. [Accessed April 22, 2019].

[3] SNS Courseware. *Implementation Levels of Virtualization*. [Online]. Available at: http://www.snscourseware.org/snscenew/files/CW_5b38f57cc2ba1/Implementation%20Levels%20of%20Virtualization.pdf. [Accessed April 23, 2019].

[4] Net-informations.com, Inc. *What is Java virtual machine?*. [Online]. Available at: http://net-informations.com/java/intro/jvm.htm. [Accessed April 23, 2019].

[5] Microsoft. 2019. *Common Language Runtime (CLR) overview*. [Online]. Available at: https://docs.microsoft.com/en-us/dotnet/standard/clr. [Accessed April 23, 2019].

[6] WineHQ. *What is Wine?*. [Online]. Available at: https://www.winehq.org/. [Accessed April 23, 2019].

[7] Shaan R. 2019. *What Is Containerization?*. [Online]. Available at: https://hackernoon.com/what-is-containerization-83ae53a709a6. [Accessed April 24, 2019].

[8] Yegulalp S. 2019. *What is Docker? Docker containers explained*. [Online]. Available at: https://www.infoworld.com/article/3204171/what-is-docker-docker-containers-explained.html. [Accessed April 24, 2019].

[9] Docker, Inc. 2019. *Dockerize a .NET Core application*. [Online]. Available at: https://docs.docker.com/engine/examples/dotnetcore/. [Accessed April 24, 2019].

[10] Ratan V. 2017. *Docker: A Favourite in the DevOps World*. [Online]. Available at: http://opensourceforu.com/2017/02/docker-favourite-devops-world/. [Accessed April 25, 2019].

[11] Velez G. 2019. *Kubernetes vs. Docker: A Primer*. [Online]. Available at: https://containerjournal.com/2019/01/14/kubernetes-vs-docker-a-primer/. [Accessed April 25, 2019].