VMs, Containers and their relationship to DevOps

Jonas Johansson
jonasjo5@kth.se

April 2020

Contents

1 Introduction

Containers are sometimes referred to as lightweight virtual machines (VMs). While the explanation might be good for analogy, it is in fact not true and might contribute to the misconception that they bring the same functionality [1]. Another prevalent belief is that containers are "better" than VMs and that containers will replace VMs in near future.

In this essay, we will cover the basic architecture of both VMs and containers and look at some of the key differences and similarities between the tools. We will also look at some use cases and how the techniques can be used in combination.

2 Background

To fully understand the comparison and use cases of VMs and containers, it is required to have some basic knowledge of the tools. In this section we will cover a basic explanation of both VMs and containers and the architecture of them both.

2.1 Virtual Machines

A virtual machine is a computer system, emulated in software. Virtual machines can provide the same functionality as physical computers as they are capable of running complete operating systems [2]. To understand the structure and components of a virtual machine, let's look at the following figure:

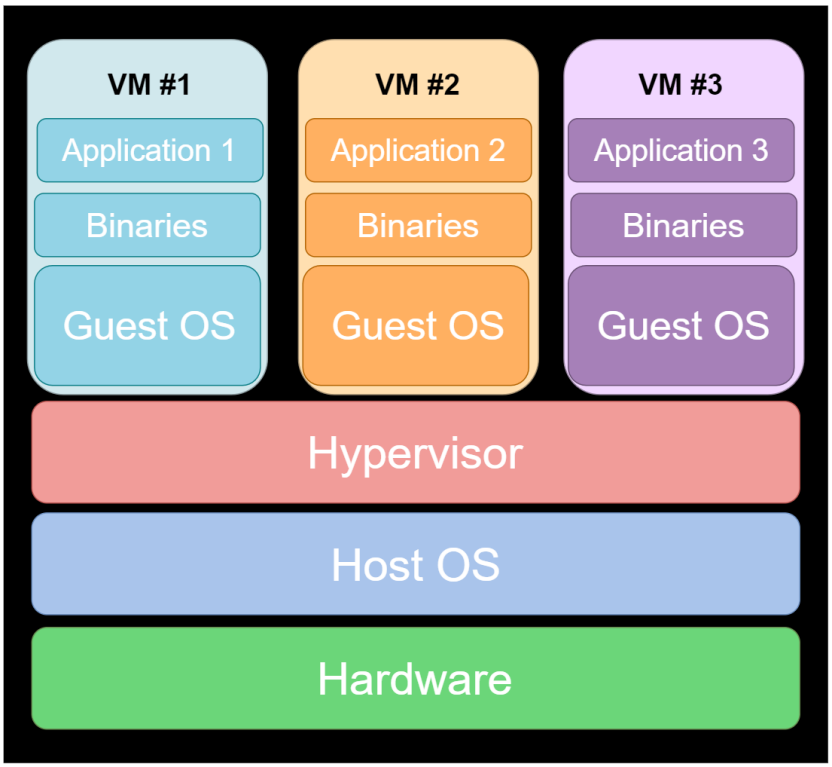

Figure 1: Illustrates a system bottom-up where virtual machines are used to run 3 applications on 3 operating systems. Inspired by CircleCI [3].

In short, the components of figure 1 can be described as:

Hardware: means the underlying hardware or infrastructure. This could be e.g. your physical computer or a server.

Host OS: is the operating system on which the virtual machine is run. E.g, if you run a virtual machine on your Windows computer, Windows is the host OS [4]. Depending on which type of hypervisor is used, a host OS might not be needed [5].

Hypervisor: is the process that creates, runs and monitors the virtual machines. A hypervisor allows several virtual machines to be run on one host computer, virtually sharing its resources. There are two types of hypervisors. Type 1, "bare metal" hypervisors, runs directly on the hardware, hosting its own lightweight operating system. A host operating system is therefore not needed when running a bare metal hypervisor.
Type 2 or "hosted" hypervisors are software that runs on the host OS. All interaction between the virtual machines and the host OS, such as system calls is

managed by the hypervisor [5].

Guest OS: is the operating system run by the virtual machine. Each virtual machine runs its own guest OS, which means that the different VMs run on the same host can have different operating systems [3]. E.g. VM#1 can run Windows 10, while VM#2 runs Ubuntu 18.04 and VM#3 runs CentOS.

Binaries: Since VMs are self-contained, each VM needs its own set of binaries and libraries required to run the desired software [3].

Application: is quite self-explanatory. This refers to the application that you want to run within the VM.

2.2 Containers

A container is an isolated runtime environment where applications can be run on a host operating system [6]. Instead of emulating the underlying hardware, a container emulates an operating system.

A container can be described as having two states, running and at rest. A container at rest is called a container image or container repository, which is a set of files saved on disk [7]. More precisely, Docker explains a container image as "*a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings* [8]".

When running a container image, the files and meta-data are unpacked and handed over to the OS kernel. Historically, there are several different container image formats, but the industry is currently moving towards a unified format under the Open Container Initiative [7].

The structure and components of a system running software via containers are visualized in figure 2:

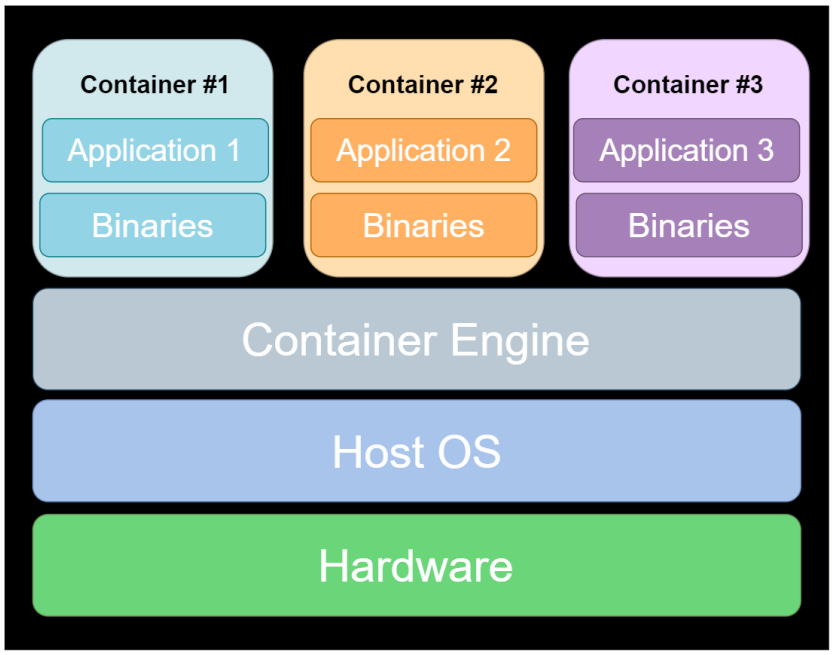

Figure 2: Illustrates a system bottom-up where containers are used to run 3 applications on 1 host operating system. Inspired by Docker [8]

Hardware: as described in section 2.1, the underlying hardware or infrastructure of your system.

Host OS: is the operating system where the container engine is installed. Since containers are an abstraction at the app layer, a host OS is needed to run containers. The OS must also be supported by the container engine [8].

Container Engine: is run on the host OS and is the environment for managing, and from the user's perspective, running the containers. Some of the tasks handled by the container engine are, handling user input, allocating resources, enforcing security, pulling container images, and running or calling the container runtime [7].

Binaries: and libraries are built into packages, called container images. Instead of running on a guest OS, the container engine runs those images [3].

Applications: are typically packaged together with its libraries and dependencies into the container image. Just as with VMs, each application is isolated [3].

2.2.1 Container Orchestration

Managing a handful of containers might be a doable task, but for large enterprises that need containerized applications by the thousands, container orchestration becomes a valuable tool. A container orchestration tool automates the process of many container managing tasks, such as deploying your images, scaling e.g by replicating containers, moving containers to avoid downtime, etc [9].

Red Hat provides a short, concise definition of container orchestration: "*Container orchestration automates the deployment, management, scaling, and networking of containers* [10]". There are several tools available for container orchestration, such as Docker Swarm [11], and Kubernetes [12].

3 Differences between containers and VMs

In this section, we will look at some of the key differences and similarities between VMs and containers. As different properties might be beneficial or unfavorable depending on the context, the traits of VMs and containers are not divided into simple pros and cons.

One of the most prominent differences between VMs and containers is their type of virtualization, which is explained in section 2. In short, containers are an abstraction at the app layer of a system and is hence dependent on the host OS kernel. VMs are an abstraction of physical hardware and are capable of running their own guest OS.

3.1 Storage

Virtual Machines are usually measured by the gigabyte. Each virtual machine generally contains its own operating system, libraries, binaries and application. The space needed is heavily dependent on the application as the system that is being emulated either can be lightweight or an entire server, OS or database [13].

A container image is usually measured by the megabyte as it only contains the application itself and the files needed for it to run. Also, according to Red Hat, containers "*are often used to package single functions that perform specific tasks (known as a microservice)* [13]", which further contributes to keeping the average size of container images small.

The most significant difference between VMs and containers in terms of storage is that containers share OS and thus have no need storing their own guest OS [3].

3.2 Performance

In a study from 2017, Zehng et al. [14] benchmarked the performance of Docker and VMWare Workstation 12 Pro to demonstrate the performance differences between VMs and containers. Their study showed that containers generally performed significantly better than VMs and in some cases even had performance comparable to the physical machine. They found however, that containers performed worse than VMs when testing storage data throughput using byte-sized data.

There seems to be a general consensus that containers are more lightweight than VMs and performs better at runtime since no emulation of hardware is needed and the hardware can be reached directly via the host OS [15, 8, 3].

3.3 Security

VMs provide complete isolation from other VMs and the host OS as they are only allowed to interact with hypervisor services. This means that even an application run with root privileges on the guest OS is unable to breach the isolation [16].

While VMs generally have a high level of security and isolation of processes, bugs and vulnerabilities in the hypervisor could potentially be exploited. Malware that has penetrated one VM and manages to attack the hypervisor is called hyper-jacking [17]. In 2015, Jason Geffner, a senior security researcher at CrowdStrike, revealed such a hypervisor vulnerability, called the VENOM vulnerability. Through the virtual floppy drive code included in many popular VM platforms, he managed to escape the guest OS and execute code on the host. The vulnerability had the potential to affect many datacenters [18].

Even though hyper-jacking is considered a real-world threat to virtualized environments, there are no known successful malicious attacks [17].

Containers also provide applications to run isolated from each other. Compared to a VM, the isolation of containers are more lightweight as they share resources and host operating system. Instead of running a complete isolated system, containers are isolated on process-level with shared kernel. One way to strengthen the security is to use a lightweight VM to isolate each container [19].

The concept of having containers are not by default secure as the enforcement of security and isolation relies on the container engine [15]. There have been several incidents concerning popular container services and their APIs. One example is the "Severe Privilege Escalation Vulnerability" in Kubernetes, where attackers could gain root access to a Kubernetes cluster [20].

Both containers' and VMs' image formats impose an indirect risk to confiden-

tiality and integrity. If images aren't treated with the same care as the data they contain, they are not protected against modification or unauthorized access [17].

3.4 Portability

Portability means the ability of porting or moving an application from one host environment to another. This includes moving the application between hosts with different OS, different versions of the same OS, or even different architectures [21].

Depending on the scenario, the portability of containerized applications varies. If an application is built and packaged into a container, it would run on any host running a supported OS from the same family without being recompiled. All you have to do is to move the container images over, which is usually managed by a container orchestration system [21].

Since containers are dependent on the host OS, their portability between operating systems is limited. A specific example is that it is not possible to natively run a Linux container on a computer running Windows operating systems. There are however workarounds to the problem, e.g. Docker for Windows runs Linux containers in a VM, making the support seem seamless for the end-user [22].

As VMs run applications in their own guest OS, they support all the portability scenarios mentioned above [21]. Containers are however much more portable in the sense that they are significantly easier to move between supported systems. This is especially true when using orchestration tools [13].

4 VM, container, or both?

As both VMs and containers have their strengths and weaknesses depending on the context, they have different uses. It is however, not always clear which one is better and the decision might depend on security or performance requirements.

VMs' strong isolation properties make them very useful in applications with high security demands. An extreme example of this is that they can be used for analyzing malware. A VM can be a safe environment for running malware and analyzing their behavior. E.g. some malware uses encryption to hide their binaries but decrypts at runtime. By running and analyzing the malware in a VM, the encryption key can be retrieved and the malware signature recognized [23].

A VM is also the alternative to use if you want to run multiple applications together or where the full functionality of an OS is needed [24].

Due to their high performance and low overhead costs, containers are beneficial in most cases where the functionality of a complete OS and the strong isolation properties of VMs are not needed. Containers are especially suitable if you want to run a larger number of small isolated processes [3].

In many cases it can be beneficial to use containers together with VMs. The underlying host OS of your containers might as well be the guest OS of a VM. An example of this, provided by CircleCI is if you're running a web hosting company. You would probably use a VM to separate each customer but the web application services might be broken up into several container images [3].

4.1 Combining the tools, a more specific example

To better understand how VMs and containers coexist in a DevOps context, this section covers a more specific example of container orchestration. Kubernetes has recently become one of the most popular DevOps tools and is therefore used in the example, using terminology from the Kubernetes documentation [25]. As mentioned in section 2.2.1, Kubernetes is a container orchestration tool that facilitates managing large quantities of containerized software.

We keep things simple and begin by looking at a small Kubernetes cluster.

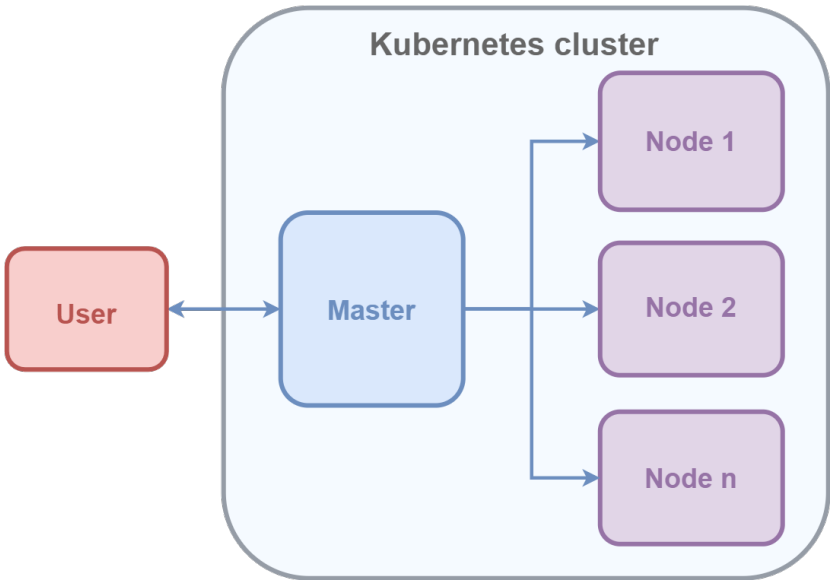

Figure 3: Illustrates a Kubernetes cluster with master and node components. Inspiration was taken from [26, 27]

Master
The Kubernetes master is the API server of the cluster. The master manages

the whole cluster, which nodes to assigning application services to, load balancing and scaling the cluster, interaction between nodes, etc. All user interaction with the Kubernetes cluster is done via the master component [28].

Nodes
Nodes are the worker machines of Kubernetes clusters. They can be run either on physical machines or more commonly, in VMs. Each node is managed by the master component and contains the services needed to run pods. The nodes are not inherently created by Kubernetes but are rather provided by cloud services or are a pool of your own private systems [29].

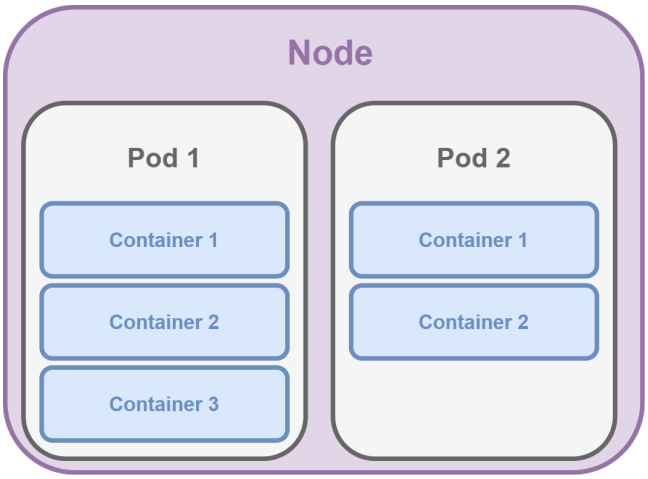

Figure 4: Shows a node hosting 2 pods, which runs 3 and 2 containers respectively.

As shown in figure 4, each node can run multiple pods. A pod is the smallest Kubernetes object that is deployed by the master component. Pods are the execution units of a Kubernetes application that encapsulates and runs one or several containers. There are several container runtimes supported by Kubernetes, where Docker is the most common [30].

VMs
While it is possible to run Kubernetes on physical machines VMs are in fact the most common way of deploying Kubernetes components [29]. In our example it would mean running the master component and every node on its own VM. This serves as one of many possible examples of the coexistence of VMs and containers.

5 Conclusions

The concept of running applications in isolation can be confusing when trying to differentiate containers from VMs. There are however fundamental differences between the two techniques. Containers are an abstraction at the app layer, where containerized applications share OS kernel and resources via the container engine. Containers offer low overhead and storage costs, which makes them suitable for running large numbers of isolated processes. Due to their dependency on their host OS, containers lack the full OS functionality provided by VMs.

VMs are an abstraction of physical hardware and emulates a complete system in software. VMs hosts their own guest OS and may provide applications with full OS functionality. The guest OS comes at the cost of extra disk space and memory needed. The isolation provided by VMs is stronger compared to the isolation provided by containers, which makes VMs more suitable in scenarios where a high level of security in terms of isolation is needed.

Both VMs and containers are common tools and both have their respective use cases. They are not necessarily competitors as they complement each other and can be used in combination.

References

[1] The Linux Foundation. *Containers Are Not Lightweight VMs*. URL: https://www.linuxfoundation.org/blog/2017/05/containers-are-not-lightweight-vms/. (accessed: 04.18.2020).

[2] Inc VMware. *Virtual Machine*. URL: https://www.vmware.com/topics/glossary/content/virtual-machine. (accessed: 04.14.2020).

[3] CircleCI. *Virtual Machines vs Docker Containers: a Comparison*. URL: https://circleci.com/blog/virtual-machines-vs-docker-containers-a-comparison. (accessed: 04.14.2020).

[4] Oracle. *1.2 Some Terminology*. URL: https://docs.oracle.com/en/virtualization/virtualbox/6.0/user/virtintro.html. (accessed: 04.14.2020).

[5] Inc VMware. *What is a Hypervisor?* URL: https://www.vmware.com/topics/glossary/content/hypervisor.html. (accessed: 04.14.2020).

[6] Docker Inc. *Docker overview*. URL: https://docs.docker.com/get-started/overview/. (accessed: 04.14.2020).

[7] Scott McCarty. *A Practical Introduction to Container Terminology*. URL: https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction. (accessed: 04.15.2020).

[8] Docker Inc. *What is a Container?* URL: https://www.docker.com/resources/what-container. (accessed: 04.15.2020).

[9] boxboat. *What Is Container Orchestration?* URL: https://boxboat.com/2019/01/25/what-is-container-orchestration/. (accessed: 04.21.2020).

[10] Red Hat. *What is container orchestration*. URL: https://www.redhat.com/en/topics/containers/what-is-container-orchestration. (accessed: 04.17.2020).

[11] Docker Inc. *Swarm mode overview*. URL: https://docs.docker.com/engine/swarm/. (accessed: 04.17.2020).

[12] The Linux Foundation. *Swarm mode overview*. URL: https://kubernetes.io/. (accessed: 04.17.2020).

[13] Red Hat. *Containers vs VMs*. URL: https://www.redhat.com/en/topics/containers/containers-vs-vms. (accessed: 04.15.2020).

[14] Zheng Li et al. "Performance Overhead Comparison between Hypervisor and Container Based Virtualization". eng. In: *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 2017, pp. 955–962. ISBN: 9781509060283.

[15] Donald Firesmith. *Docker overview*. URL: https://insights.sei.cmu.edu/sei_blog/2017/09/virtualization-via-containers.html. (accessed: 04.15.2020).

[16] Inc VMware. *Security and Virtual Machines*. URL: https://pubs.vmware.com/vsphere-51/index.jsp?topic=%2Fcom.vmware.vsphere.security.doc%2FGUID-3D54E1F7-AC12-48C1-A5AE-55D3A82CF606.html. (accessed: 04.16.2020).

[17] Cloud Security Alliance. *Best Practices for Mitigating Risks in Virtualized Environments*. 2015. URL: https://downloads.cloudsecurityalliance.org/whitepapers/Best_Practices_for_%20_Mitigating_Risks_Virtual_Environments_April2015_4-1-15_GLM5.pdf.

[18] CrowdStrike. *VENOM, VIRTUALIZED ENVIRONMENT NEGLECTED OPERATIONS MANIPULATION*. URL: https://venom.crowdstrike.com/. (accessed: 04.16.2020).

[19] Jason Gerend. *Containers vs. virtual machines*. URL: https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/containers-vs-vm. (accessed: 04.14.2020).

[20] National Vulnerability Database. *CVE-2018-1002105 Detail*. URL: https://nvd.nist.gov/vuln/detail/CVE-2018-1002105. (accessed: 04.16.2020).

[21] Christopher Tozzi. *For Portability, VMs Still Beat Docker Containers*. URL: https://containerjournal.com/features/portability-vms-still-beat-docker-containers/. (accessed: 04.16.2020).

[22] Microsoft. *Linux containers on Windows 10*. URL: https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/linux-containers. (accessed: 04.16.2020).

[23] Stichting Cuckoo Foundation. *Cuckoo automated malware analysis*. URL: https://cuckoosandbox.org/. (accessed: 04.17.2020).

[24] David Zomaya. *Container vs. Hypervisor: What's the Difference?* URL: https://www.cbtnuggets.com/blog/certifications/cloud/container-v-hypervisor-whats-the-difference. (accessed: 04.14.2020).

[25] The Kubernetes Authors. *Standardized Glossary*. URL: https://kubernetes.io/docs/reference/glossary/?fundamental=true. (accessed: 04.18.2020).

[26] Janakiram MSV. *Kubernetes: an overview*. URL: https://thenewstack.io/kubernetes-an-overview/. (accessed: 04.18.2020).

[27] Andrew Martin. *11 Ways (Not) to Get Hacked*. URL: https://kubernetes.io/blog/2018/07/18/11-ways-not-to-get-hacked/. (accessed: 04.18.2020).

[28] The Kubernetes Authors. *Master-Node Communication*. URL: https://kubernetes.io/docs/concepts/architecture/master-node-communication/. (accessed: 04.18.2020).

[29] The Kubernetes Authors. *Nodes*. URL: https://kubernetes.io/docs/concepts/architecture/nodes/. (accessed: 04.18.2020).

[30] The Kubernetes Authors. *Pod Overview*. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/. (accessed: 04.18.2020).