What Containers Are and Why You Should Care

Sheldon Zhao

December 13, 2019

1 Introduction

In 2014, a senior software engineer at Google told journalist: “Everything in Google runs in a container.”\(^1\) Today, over two billion containers are started each week to orchestrate some of Google’s most popular services, such as search, Gmail, and YouTube.\(^2\) In 2016, Microsoft announced that it will integrate containers and container management services into its cloud.\(^3\) Amazon Cloud Services, too, has introduced its own EC2 Container Service to facilitate container usages.\(^4\) Reports have shown that the container industry is set to grow from 762 million dollars in 2016 to over 2.7 billion dollars in 2020.\(^5\) All of this buzz begs the question: what are containers and why should we care?

Container technology is a lightweight virtualization tool that has greatly enabled commercial cloud computing and expedited software production.\(^6\) In this survey paper, the

\(^1\)Clark, Jack, “Google: ‘EVERYTHING at Google runs in a container”, The Register, (San Francisco), May 21, 2014
\(^2\)“Containers at Google”, Google Cloud, https://cloud.google.com/containers/
author attempts to provide a thorough and foundational understanding of containers as a virtualization tool. In addition to theory, which is common in existing literature, the author will employ ample examples and scenarios to illustrate what containers are, how containers work, and what containers can achieve. We will also look into the future of containers and understand its impact on the future of virtualization. This paper is for readers with a working understanding of operating systems. Understanding of virtualization will be helpful but is not required. For those approaching from a practical standpoint, this paper can expedite your learning process in approaching common tools; for those approaching from a theoretical standpoint, this paper can become a jumping off point for future research.

2 What is a container?

To answer this question in a short sentence, container is a virtualization tool. To expand on this statement, we will need to answer at a few questions: What is virtualization? How can we perform virtualization (hypervisors and containers)? What is the mechanism of containers? And what are the differences between hypervisors and containers?

2.1 Virtualization

Let us consider a few situations. First, imagine we host an email service with one million users. We need to run one program for each active user and we need to provide some sort of isolation to guarantee privacy for the users. It is unrealistic to get one machines for each user; how can we provide our service efficiently?

Second, imagine that we have a MacBook running macOS, but we need to run an
application for Ubuntu. How can we run that application on our own machine?

Lastly, imagine that we need to install two pieces of software on our Linux server: software 1 requires Python 3.5 and PostgresSQL 9.4, while software 2 requires Python 3.4 and PostgresSQL 9.6. Having duplicate libraries is possible, but it also presents cumbersome tasks for developers as well as users.\(^7\) How can we resolve this efficiently?

The aforementioned situations are but a few common scenarios that require virtualization. **Virtualization** in general describes the process of creating an abstract representation of something, usually with software.\(^8\) The thing being represented can be a server, a machine, a storage device, or more. To achieve such abstract representation, a virtualization tool would create a layer of removal between the virtualized and those attempting to access the virtualized, in order to provide and manage access to the virtualized hardware or software.\(^9\) In this paper, we focus on virtualization related to operating systems. We will show how OS virtualization can help resolve all the issues at presented the beginning of this section.

![Figure 1: Differences between a Type-1 Hypervisor and a Type-2 Hypervisor](image)


\(^9\)Ibid.,
2.2 Virtualization and Hypervisor

One popular way to achieve OS virtualization is through using a **hypervisor**. A hypervisor is a piece of software or hardware that isolates the machine from the OS. The hypervisor sits on hardware called the host machine and provides a virtual platform for guest operating systems, which are usually referred to as **Virtual Machines (VMs)**. Each VM running on the hypervisor has its own operating system that supports the applications running on the VM.\(^{10}\)

Major functions of a hypervisor include regulating hardware access, managing devices, and providing isolation for all its VMs.\(^{11}\) \(^{12}\) A VM running on top of a hypervisor thinks it is talking directly to the hardware, when in fact it is only communicating with a virtualized machine managed by the hypervisor.]

There are two types of hypervisors: Type-1 (or bare metal), and Type-2 (or hosted). A Type-1 hypervisor lays directly in between the hardware and the operating system, while a Type-2 hypervisor lays in between a guest OS and a host OS - like an application in the host OS.\(^{13}\)

Hypervisor provides solutions to the problems in Section 2.1. For the first scenario, instead of a million machines, we use one very powerful machine with a type-one hypervisor. Every time a user accesses their email, we fire up a virtual machine that runs our email application. Due to the mechanism of the hypervisor, each virtual machine

---

\(^{10}\) *Ibid.*


\(^{12}\) Further explanation of the mechanism of a hypervisor can be found at https://blackberry.qnx.com/content/dam/qnx/whitepapers/2017/what-is-a-hypervisor-and-how-does-it-work-pt1.pdf

is isolated from the others, so the users are unaware that they are sharing a machine. For the second scenario, we simply install a Type-2 hypervisor on our MacBook. Using that hypervisor, we can run Ubuntu on top of macOS, and run our desired application in this fashion. For the last scenario, we can install a Type-1 hypervisor on our server, then run one VM with Linux, Python 3.5, and PostgresSQL 9.4 for software 1, and another VM with Linux, Python 3.4, and PostgresSQL 9.6 for software 2. Problems solved.

2.3 Virtualization and Containers

Containers provide a different solution to virtualization. The container approach assumes the immutability of the kernel. Immutability means that certain low-level functions of the OS, especially the kernel, is required by all programs during virtualization, therefore we do not need multiple instances of the kernel as we would for VMs.\textsuperscript{14} As a result, containers sit on top of the kernel and abstract the operating system instead of the hardware.\textsuperscript{15}

From the perspective of the operating system, a container is a process between the OS, usually known as the host operating system, and the application.\textsuperscript{16} We can create an isolated environment inside the container using tools provided by the host OS and the hardware, so that the computing environment inside the container is different from the environment of the host OS. In this way, different programs can run inside different containers that share a kernel with the host OS while enjoying separation from both the host OS and other containers. Because of this dependency on the host OS, containers for different OSes are different, but all provide similar functionalities.

\textsuperscript{14}Goncalves da Silva et al.
\textsuperscript{15}Ibid.
\textsuperscript{16}Ibid.
Containers can also address all the situations mentioned in Section 2.1. For the email problem, we need one powerful machine on which we install the operating system needed to run our software. Then we can simply fire up a container with our email application each time the user uses our service. In order to run the Ubuntu software on MacBook, we can simply launch a container running Ubuntu on our MacOS, then run the software inside this container. For the last problem, we just have to produce two containers: one with Python 3.4, PostgresSQL 9.6, and software 1; the other with Python 3.5, PostgresSQL 9.4, and software 2. Then, we can run the two programs inside the two containers on a Linux machine.

![Figure 2: Differences between a Type-1 Hypervisor and a Containers](image)

2.4 Mechanism of Containers

The term “container technology” simplifies the complex processes behind operating system abstraction. Three things are key to the functionality of containers: file system isolation, process isolation, and hardware resources management. Since early development in the field of containers took place in Linux, we will look at Linux containers

---

17 Docker provides an Ubuntu image that simulates almost all functionalities of Ubuntu. It is commonly used by developers to test Ubuntu softwares. More details at https://hub.docker.com/_/ubuntu

18 This is not true for other guest OSes.

---
to see how these three functionalities are achieved. Containers on other OSes performs similar tasks.

File system isolation allows the abstracted OS to have its own, separate file system. Earliest containers achieve this through a simple root change.\textsuperscript{19} Each Linux process has a root directory and it can only see the content inside its current root.\textsuperscript{20} In this way, the current running process and its children see a set of files and directories in its root without knowledge of the file system of the host OS. To create a container with root change is simple: we first create a folder in the present working directory, then copy the desired files to this folder, and lastly change the root of the current running process to this new folder with a \texttt{chroot} command. To provide more complex functionalities, we can start to create a more complex file system inside this new root, such as mounting directories containing libraries for programs inside the container.

To organize the container’s file system in a more sophisticated manner, we introduce the \textbf{union file system}. The union file system allows directories and files mounted from different file systems to be overlaid in a simple and transparent manner.\textsuperscript{21} Each union file system contains three layers: a read-only base layer, an overlay layer, and a diff layer. The base layer refers to the directories and files mounted from outside file systems, which is kept read-only to prevent changes. The overlay layer is what the user sees and interacts with.\textsuperscript{22} The user is allowed to make changes to anything they desire without affecting the base layer thanks to the diff layer, which automatically stores any differences between the overlay and the base as a result of changes to the union file system beyond mounting. In Linux, users can construct a union file system with

\begin{footnotesize}
\begin{itemize}
\item[\textsuperscript{19}] Burns, Brendan et al., “Borg, Omega, and Kubernetes - Lessons learned from three container-management systems over a decade”, ACMqueue, Volume14, issue 1 (2016): 24 pages
\item[\textsuperscript{20}] “chroot(1)”, Linux Manual
\item[\textsuperscript{21}] “Union file system”, The Docker EcoSystem
\item[\textsuperscript{22}] Ibid., https://washraf.gitbooks.io/the-docker-ecosystem
\end{itemize}
\end{footnotesize}
the `mount -t overlay` command.\(^{23}\) Figure 3 illustrates the construction of a sample union file system. Here, we create a directory called `SimpleContainer`. Inside it, we create four folders: `base`, `diff`, `overlay`, and `workdir`. We set up the various layers according to the parameters of `mount -t overlay`. Lastly, we `cd` into `overlay`, where we can interact with this new union file system. Almost all modern containers use variations of this technique for file system isolation.\(^{24}\)

Isolation of processes allows for processes inside a container to run in an environments separated from the host OS and the other containers. For Linux containers, this is achieved through namespaces. In Linux, each process is associated with multiple namespaces that specifies its accessibility, priorities, ownership, and more. A process can only perform actions that its namespaces allow. Linux containers manipulates seven Linux namespaces: `mnt`, `pid`, `net`, `ipc`, `uts`, `user` and `cgroup`.\(^{25}\) One of the most important namespaces for process manipulation is `pid`,\(^{26}\) which we will now investigate in order to illustrate how containers utilize namespaces.

`pid` stands for process identifier.\(^{27}\) In Linux, each process has one `pid` for each `pid` namespace it belongs to so as to signify its identity.\(^{28}\) Since all Linux processes are organized in a tree-like fashion with parent-child hierarchy, a `pid` namespace is usually a collection of child processes under a certain parent process. The process at the root of each namespace has `pid 1`, signifying its special status.\(^{29}\) Inside a namespace, all processes are aware of each other’s existence and certain privileged processes

\(^{23}\) `mount(8)`, *Linux Manual*
\(^{24}\) Burn, Brendan et al.\(^{\text{Ibid.}}\)
\(^{26}\) *Ibid.*
\(^{27}\) `pid|_namespaces(7)`, *Linux Manual*
\(^{28}\) *Ibid.*
\(^{29}\) *Ibid.*
can inspect and kill other processes.\textsuperscript{30} We can create nested namespaces by creating a child namespace underneath a parent namespace by declaring a process as a new root, giving it \texttt{pid} 1 in the new child namespace, while maintaining its original \texttt{pid} in the parent namespace. All the children of this process will automatically belong to this new child namespace.\textsuperscript{31} The parent namespace is aware of all processes in the child namespace, but processes in the child namespace cannot see anything in the parent namespace. This effect effectively creates isolation between processes in and outside the namespace.\textsuperscript{32} Each Linux container creates unique children namespaces for all seven available namespaces.

One of the seven namespaces, \texttt{cgroup}, focuses on hardware resource management. In Linux, a \texttt{cgroup} is a collection of processes that share the same limits and priorities of resource usage.\textsuperscript{33} A Linux container essentially creates a new \texttt{cgroup} for each container and sets specific resource limitations for this new \texttt{cgroup}. \texttt{cgroup} helps containers avoid competition with the host OS and each other.\textsuperscript{34} We will see how

\texttt{mkdir SimpleContainer}
\texttt{cd SimpleContainer}
\texttt{mkdir base diff overlay workdir}
\texttt{sudo mount -t overlay \}
\texttt{-o lowerdir=base,upperdir=diff,workdir=workdir \}
\texttt{overlay \}
\texttt{overlay \}
\texttt{cd overlay}

Figure 3: Sample code for constructing a Linux union file system

\textsuperscript{31}Ibid.,
\textsuperscript{32}Grunert
\textsuperscript{33}“cgroup|\texttt{S(7)}”, Linux Manual
\textsuperscript{34}Burns
namespaces and cgroup are configurated in Section 2.5.

2.5 Modern Containers

Nowadays, we do not need to write complicated shell scripts to construct our desired containers. Organizations such as Open Container Initiative (OCI) have standardized container format and container runtime specifications to facilitate faster setup, wider adaptation, and easier sharing.\(^35\) We will look at how these two standards have shaped the containers that we use now.

The standard container format encodes the structure of container image files. A container image file is an immutable file that specifies the structure of the union file system and its content.\(^36\) Similar to the union file system, an image file contains one base image and multiple overlay images. When compiled, the image file constructs the desired file system from the bottom layer up to the top layer, then it places a R/W layer on top of the directories to allow access to the new file system.\(^37\) Image files are written in a very low-level language and is thus very hard to construct and interpret.\(^38\) However, there are service providers out there that help us create and manage them with ease. One popular service provider is Docker.\(^39\) With Docker, users can create container images with a Dockerfile, which is written in a bash-like script.\(^40\)

Figure 4 is an example of a simple Dockerfile with four commands. The first command

\(^{35}\)“Open Container Initiative”, https://www.opencontainers.org

\(^{36}\)“About images, containers, and storage drivers”, docker docs, https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers/

\(^{37}\)Ibid.

\(^{38}\)“Open Container Initiative”, https://opencontainers.org


\(^{40}\)“Dockerfile reference”, docker docs, https://docs.docker.com/engine/reference/builder
specifies the base layer, which is the `ubuntu:18.04` image. The second command copies the present working directory (in this case, on the Docker registry) into `/app`. The third command executes the make function on `/app`, and the last command specifies which command to run when we launch the container, which in this case, is to run python on `app.py`. Figure 5 shows the structure of the image file created by this Dockerfile with a R/W layer on top to suggest that it is currently at runtime.

The runtime specification standardizes how containers are constructed and compiled. These specifications are usually in the form of a JSON file, as we can see in Figure 6. The file specifies the details surrounding root change, namespaces and resource limitations. Together, the runtime specification file and the image file form a filesystem bundle. Whenever a user builds this bundle, the resulting process is a container. A modern container, therefore, is essentially an instance of the container image file configured according to set of runtime specifications.

### 2.6 Differences between Hypervisors and Containers

By now, we can see that hypervisors and containers each provides a solution to virtualization. The two solutions, however, have distinct advantages and limitations.

Containers are superior to hypervisors in two important ways. First off, containers are lightweight, which means that they require less storage and memory. Since containers assume the immutability of the kernel of the host OS, container images do not contain much information about the host OS, and are thus significantly smaller than

41 “Runtime Spec”, GitHub, https://github.com/opencontainers/runtime-spec
43 Ibid.
44 Vitor et al.
VM images. Secondly, the smaller size usually means faster boot time, since it takes less time for the machine to load the container image into main memory and start execution.

However, the benefits of containers do not render hypervisors obsolete. First, since containers require a host OS, it is impossible to use containers for programs that are incompatible with the underlying kernel of the host OS. For example, it will be very hard, if not impossible, to run a Windows software on a Linux system with containers. One of the biggest advantage of a hypervisor, therefore, is its ability to host multiple operating systems all at once. In addition, security is a major concern for containers. Although containers are separated from one another through process isolation and file

---

45 Ibid.,
46 Ibid.,
47 “What is a Container?”, Docker, https://www.docker.com/resources/what-container
49 Ibid.,
> runc spec
> cat config.json
{

"ociVersion": "1.0.0",
"process": {
  "terminal": true,
  "user": { "uid": 0, "gid": 0 },
  "args": ["sh"],
  "env": [  
    "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
    "TERM=xterm"
  ],
  "cwd": "/",
  "capabilities": {
    "bounding": ["CAP_AUDIT_WRITE", "CAP_KILL", "CAP_NET_BIND_SERVICE"],
    ["
  ],
  "rlimits": [ { "type": "RLIMIT_NOFILE", "hard": 1024, "soft": 1024 } ],
  "noNewPrivileges": true
},
"root": { "path": "rootfs", "readonly": true },
"hostname": "runc",
"mounts": [
  {
    "destination": "/proc",
    "type": "proc",
    "source": "proc"
  },
  ["
],
"linux": {
  "resources": { "devices": [ { "allow": false, "access": "rwm" } ] },
  "namespaces": [  
    { "type": "pid" },
    { "type": "network" },
    { "type": "ipc" },
    { "type": "uts" },
    { "type": "mount" }
  ],
  "maskedPaths": [  
    "/proc/kcore",
    ["
  ],
  "readonlyPaths": [  
    "/proc/asound",
    ["
  ]
]
}
system isolation, they still share an underlying host OS. These OSes, be it Linux or Windows, has a much larger vulnerability surface than a hypervisor does. As a result, should a bug exist in the host OS, all containers running on that machine could be compromised.\textsuperscript{50} Lastly, containers are not always faster. After all, containers run on a host OS that provides overheads. For programs that require the speed of bare metal, hypervisor is still the safer option.\textsuperscript{51}

As of now, hypervisors and containers are symbiotic, each great for different purposes. Many servers and data centers run VMs and containers concurrently to take advantage of both solutions.\textsuperscript{52}

\section{Why should you care?}

Now that we understand the conceptual benefits of containers, we pivot to their applications. In fact, many of the benefits of containers become evident when we examine how various industries have adopted containers in their workflow. In this section, we will introduce a few scenarios where containers have brought forth significant benefits. We will also see a few existing solutions that further the effectiveness of containers. Hopefully, some of these scenarios can speak to why you should care about containers.

\textsuperscript{50}Manco, Felipe et. al., “My VM is Lighter than your Container”, \textit{Proceedings of SOSP '17: ACM SIGOPS 26th Symposium on Operating Systems Principles}, (Shanghai, China), October 28, 2017, 16 pages.

\textsuperscript{51}The slow down due to host OS overhead is debated. While researcher such as Christopher Tozzi believes that containers are slower than bare metal due to OS overhead, Roberto Morabito et al. have not detected such an issue in their empirical studies. The author included this to encourage further debate.

\textsuperscript{52}Manco et al.,
3.1 Virtualization with Limited Resources

In Section 2.6, we explored how containers could be very space efficient compared to traditional VMs. When storage and memory are scarce, one should consider switching from VMs to containers for their virtualization needs. Using a hypervisor, the Ubuntu 18.04 operating system is around 1.8GB.\footnote{\url{https://ubuntu.com/download/desktop}} The MacOS container image file for ubuntu acquired from Docker is only around 64.2MB.\footnote{\url{https://hub.docker.com/_/ubuntu/}} Not only is this helpful for storage, but for execution as well. Smaller image files means smaller memory requirement, which allows us to run our programs more densely.

3.2 Improving Workflow in Software Development

Around the time of writing, an increasing number of companies, including Citizens Bank, Liberty Mutual and more, are using Docker to improve their software production workflow.\footnote{\url{https://www.docker.com/customers}} In fact, containers’ effect in software development is often viewed as one of their most important contributions to the tech industry.\footnote{\url{https://apiumhub.com/tech-blog-barcelona/top-benefits-using-docker/}}

From the developer’s perspective, there are a few benefits. First, containers abstract the hardware and the environment so that developers no longer need to worry about developing for specific machines and systems.\footnote{\url{https://www.vmware.com/solutions/cloud-native-apps.html}} This liberates a decent amount of time for any development work. In addition, containers help maintain consistent runtime environment even during transportation, which is very helpful for collaboration.\footnote{Ibid.} For example, software developers can create a Dockerfile to package both the software and the environment into one singular Docker image and store this image file on a git-like
repository such as Docker Hub from Docker. Members from other teams, such as IT Ops, can pull the image from Docker Hub and directly execute the program in a container, saving time from configuring and debugging the runtime environment. As a result, new software and new updates reach users faster. Internal studies from Docker has seen the speed of delivery of new software triple after teams adopt containers. Lastly, as more and more software distributors ship their software to their customers as image files through container registries, companies like Docker are providing more sophisticated functionality for distribution. Now, developers can specify which nodes can run which versions of their software, providing great flexibility and control.

3.3 Improving Convenience and Security in Software Installation

Containers also make using software much easier. To begin with, containers eliminate the need to configure the environment when we install a new piece of software. Software installation is troublesome when new software conflicts with legacy software when it comes to dependencies and libraries. Educational institutions like Yale, for example, now installs certain software in the form of a container when managing their servers.

Downloading software as a container is very easy. Programs like Docker have made it so easy that a mere pull command could get the trick done. Docker client automates the entire process of downloading images and specs, downloading dependencies, and compiling these files. However, these programs must be executed within containers, so one need to build them with Docker as well. Figure 7 shows the process of in-

---

59 Example extrapolated from VMWare official lecture series: https://www.youtube.com/watch?v=cCTLjAdIQho
61 Pearce et al.
62 Quote from Professor Stanley Eisenstat.
63 “Docker Overview”, https://docs.docker.com/engine/docker-overview/
64 Ibid.
stalling the latest version of ProstgresQL onto a MacBook running macOS.\textsuperscript{65}

Container registries also provide services in terms of vulnerability scanning and signing. On Docker Hub, for example, images from verified publishers receive official certifications to protect users from illicit container images.\textsuperscript{66}

![Figure 7: Installing Postgres with Docker – installation is completely automated](image)

\textbf{3.4 Running Massively Distributed Software}

When running multiple instances of the same program on the cloud, companies such as Google heavily favor containers, as seen through the billions of containers they start each week.\textsuperscript{67} While the initial benefit may have come from the small memory footprint of containers (as discussed in Section 3.1), the technology has evolved to provide more

\textsuperscript{65}“Scan images for vulnerabilities”,\docker docs, https://docs.docker.com/ee/dtr/user/manage-images/scan-images-for-vulnerabilities/

\textsuperscript{66}“Docker Hub Publisher Image Trust Chain”,\docker docs, https://docs.docker.com/docker-hub/publish/trustchain/

\textsuperscript{67}Clark
To orchestrate the large number of containers running on their servers, Google has been developing and utilizing container management systems.\textsuperscript{69} After a few generations, Google’s Kubernetes has become one of the widest adopted container management services.\textsuperscript{70} Like VMware for virtual machines, Kubernetes efficiently schedules jobs across processors, accounts for hardware malfunctions, manages resource limitations, facilitates state transitions, and more.\textsuperscript{71}

We now look at a simple example of Kubernetes performing container scheduling and migration. Suppose we have image $I$ and we would like to have four instances of $I$ named $I_1, I_2, I_3, I_4$ running on three machines $M_1, M_2, M_3$. For the first three instances, Kubernetes deploys one on each machine, and then randomly assigns $I_4$ to $M_1$. Now, unexpectedly, $M_2$ shuts down. Kubernetes detects this issue and intelligently reassigns $I_2$, which is currently not running, to $M_3$ in order to ensure even resource utilization.\textsuperscript{72}

As a result, container management systems allow for more efficient usage of servers and other machines, particularly in distributed systems and cloud services. Services like Kubernetes have propelled the commercialization of cloud computing.\textsuperscript{73} In 2019, Major firms such as Slack, Shopify and thousands of other companies utilize containers with Kubernetes to facilitate the execution of their massively distributed software on the cloud.\textsuperscript{74}

\textsuperscript{68}Burns et al. \hfill \textsuperscript{69}Ibid. \hfill \textsuperscript{70}Ibid. \hfill \textsuperscript{71}“Kubernetes”, https://kubernetes.io \hfill \textsuperscript{72}The aforementioned example is extracted and clarified from VMWare’s official YouTube tutorial on Kubernetes (https://www.youtube.com/watch?v=PH-2FfFD2PU) and Kubernetes’s documentation(https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/) \hfill \textsuperscript{73}Medel, Víctor et al., “Characterising resource management performance in Kubernetes”, Computers and Electrical Engineering, Volume 68 (May 2018): p. 286. \hfill \textsuperscript{74}“Kubernetes user case studies”, https://kubernetes.io/case-studies/
3.5 Better Hardware Deployment

Due to the highly abstract nature of the container environment, code is increasingly less hardware and OS dependent (as discussed in Section 3.2). For infrastructure teams and system managers, this change allows for hardware upgrades without disrupting the functionality of the code. With containers, hardware upgrades can happen not only through vertical scaling (more resources on each individual machine), which could be difficult, but also horizontal scaling (adding more machines), which is significantly easier. This allows for extremely flexible hardware deployment and faster upgrade in computational powers, and thus better runtime performances.

4 What is the future?

In Section 2.6, we explored many drawbacks to containers. These criticisms, alongside many more, have been key to the improvement of containers. Many new functionalities nowadays come as a response to previous issues. One example is Kubernetes, which set out to address a common criticism that containers are hard to orchestrate at a large scale. Right now, researches into improving containers continue. In this section, we will look at the frontiers of containers and virtualization at large.

---

76 Quote from VMWare official lecture series: https://www.youtube.com/watch?v=cCTLjAdIQho
77 Burns
4.1 More Secure Containers

In Section 2.6, we discussed how containers’ dependency on the host OS poses a larger surface of attack. Docker, for example, has been exposed to major security issues that could compromise its users.\textsuperscript{78} In their essay, Sultan et al. explore four threat models surrounding containers: application attacking the container from the inside, inter-container attack, containers attacking the host, and the host attacking containers. Many of these issues, especially inter-container attacks, have been fixed with the usage of namespaces, \textit{cgroups}, and other kernel-provided features.\textsuperscript{79} Hardware solutions also exist. One example would be the Intel SGX, which is an extension to any Intel processor that can protect containers (or processes) from malicious kernel or hypervisor.\textsuperscript{80}

Sultan et al. suggest that research into container security is a large and divergent field because each of the four forms of attack merits a set of solutions. Solutions in progress include blockchain verification tools for container images, better vulnerability assessment tools, defense mechanisms against spectre attacks, digital forensic tools to analyze attacks, and many more.\textsuperscript{81}

4.2 LightVM - a VM Better Than Containers

Instead of fixating on containers, researchers are also seeking alternatives. One approach is to make VMs more lightweight. Recently, a group lead by Manco et al. produced LightVM, which is a lightweight virtual machine that is even faster than


\textsuperscript{79}Sultan et al.

\textsuperscript{80}Ibid.,

\textsuperscript{81}Ibid.,
Docker containers in some instances. LightVM uses a tool called Tinyx to create trimmed-down Linux virtual machines according to the need of the software. The reduced VM is only tens of megabytes. Running on Xen (a Type-1 hypervisor), the VM runs on par with the `fork\exec` on Linux, and two orders of magnitude faster than similar containers.

Right now, LightVM is very limited in functionality. It is more of a proof of concept than it is a threat to containers or existing virtual machines. LightVM, however, proves the need for better lightweight virtualization tools that don’t compromise between performance and security.

5 Conclusion/How Should You Approach Containers?

Containers are lightweight virtualization tools that are taking over in a flash. At this point, you should have a solid understanding of what containers are, how they work, and what are their applications. After reading this article, the author recommends that readers who are interested in virtualization should take a more hands-on approach with containers. If you are a system administrator, you could install Docker client and try to install a piece of software with Docker. If you are a programmer or IT ops person, you can consider switching over to containers for your next project to see if it streamlines your workflow. If you are a researcher or student interested in operating systems, you can construct a container from the ground up to understand its mechanism and to discover its vulnerabilities.

---

82 Manco et al.
83 Ibid.,
84 Ibid.,
85 Ibid.,
86 Ibid.,
Container technology pushes us to think about better ways we can achieve virtualization. With more lightweight and flexible virtualization tools, companies and individuals can better utilize existing hardware to produce what is currently unimaginable.

6 Future Work

This paper provides a very discursive overview on many topics. Interested readers can consider the following topics for further research:
1. Comparing the speed of softwares running on containers versus running on bare metal
2. Evaluating how Kubernetes improves hardware utilization
3. Survey on containers and cloud native apps
4. Survey on container alternatives (lightweight virtualization tools such as LightVM)
5. Research into a specific solutions to security issues mentioned in this paper, or
6. Hack a Docker container.