Chapter 1. The What and Why of Containers

Containers are fundamentally changing the way we develop, distribute, and run software. Developers can build software locally, knowing that it will run identically regardless of host environment—be it a rack in the IT department, a user's laptop, or a cluster in the cloud. Operations engineers can concentrate on networking, resources, and uptime—and spend less time configuring environments and battling system dependencies. The use and uptake of containers is increasing at a phenomenal rate across the industry, from the smallest startups to large-scale enterprises. Developers and operations engineers should expect to regularly use containers in some fashion within the next few years.

Containers are an encapsulation of an application with its dependencies. At first glance, they appear to be just a lightweight form of virtual machines (VMs)—like a VM, a container holds an isolated instance of an operating system (OS), which we can use to run applications.

However, containers have several advantages that enable use cases that are difficult or impossible with traditional VMs:

- Containers share resources with the host OS, which makes them an order of magnitude more efficient. Containers can be started and stopped in a fraction of a second. Applications running in containers incur little to no overhead compared to applications running natively on the host OS.
- The portability of containers has the potential to eliminate a whole class of bugs caused by subtle changes in the running environment—it could even put an end to the age-old developer refrain of "but it works on my machine!"
- The lightweight nature of containers means developers can run dozens of containers at the same time, making it possible to emulate a production-ready distributed system. Operations engineers can run many more containers on a single host machine than using VMs alone.
- Containers also have advantages for end users and developers outside of deploying to the cloud. Users can download and run complex applications without needing to spend hours on configuration and installation issues or worrying about the changes required to their system. In turn, the developers of such applications can avoid worrying about differences in user environments and the availability of dependencies.

More importantly, the fundamental goals of VMs and containers are different—the purpose of a VM is to fully emulate a foreign environment, while the purpose of a container is to make applications portable and self-contained.

Containers Versus VMs

Though containers and VMs seem similar at first, there are some important differences, which are easiest to explain using diagrams.

Figure 1-1 shows three applications running in separate VMs on a host. The hypervisor[1] is required to create and run VMs, controlling access to the underlying OS and hardware as well as interpreting system calls when necessary. Each VM requires a full copy of the OS, the application being run, and any supporting libraries.

In contrast, Figure 1-2 shows how the same three applications could be run in a containerized system. Unlike VMs, the host's kernel[2] is shared with the running containers. This means that containers are always constrained to running the same kernel as the host. Applications Y and Z use the same libraries and can share this data rather than having redundant copies. The container engine is responsible for starting and stopping containers in a similar way to the hypervisor on a VM. However, processes running inside containers are equivalent to native processes on the host and do not incur the overheads associated with hypervisor execution.

Both VMs and containers can be used to isolate applications from other applications running on the same host. VMs have an added degree of isolation from the hypervisor and are a trusted and battle-hardened technology. Containers are comparatively new, and many organizations are hesitant to completely trust the isolation features of containers before they have a proven track record. For this reason, it is common to find hybrid systems with containers running inside VMs in order to take advantage of both technologies.

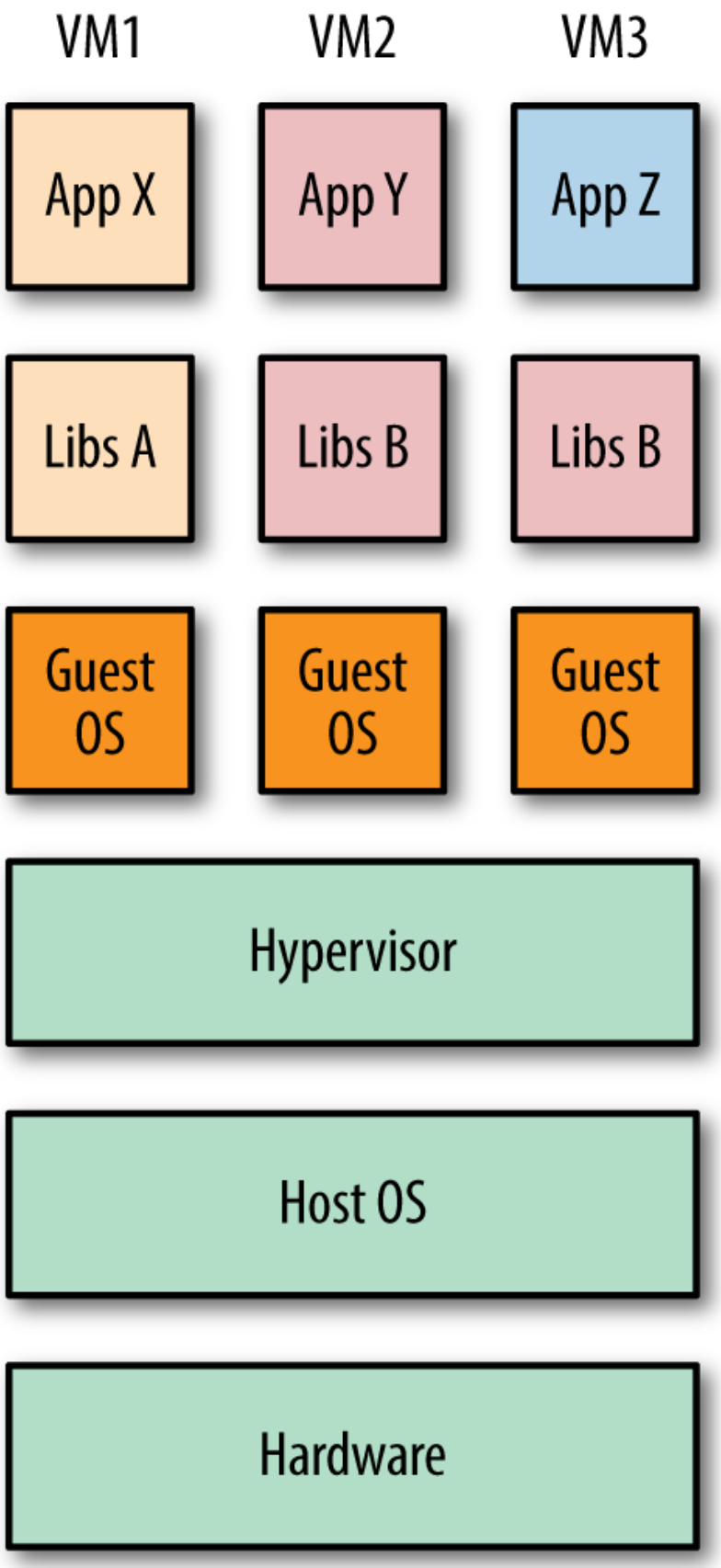
VM1
VM2
VM3
App X
App Y
App Z
Libs A
Libs B
Libs B
Guest OS
Guest OS
Guest OS
Hypervisor
Host OS
Hardware

Figure 1-1. Three VMs running on a single host

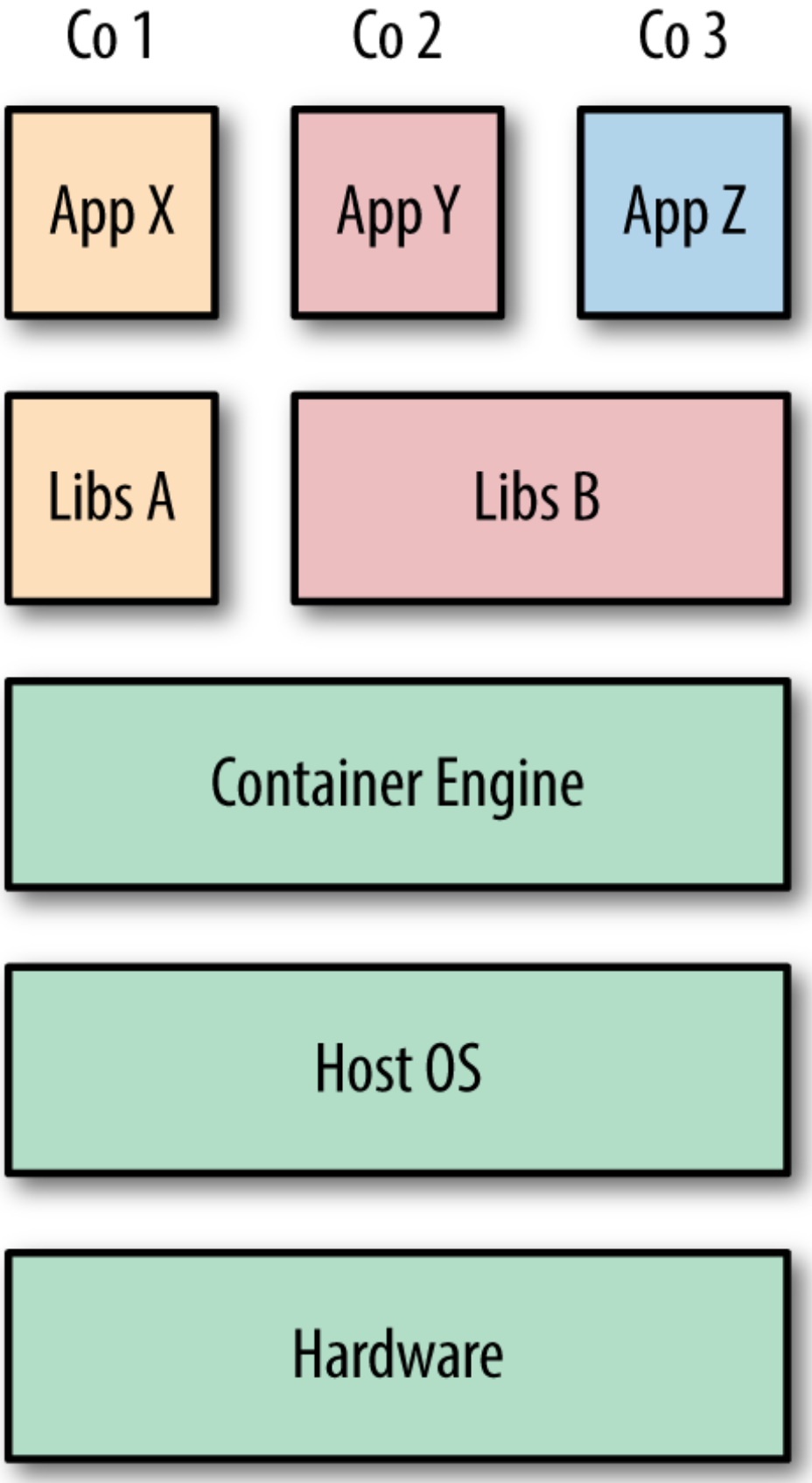
Co 1
Co 2
Co 3
App X
App Y
App Z
Libs A
Libs B
Container Engine
Host OS
Hardware

Figure 1-2. Three containers running on a single host

Docker and Containers

Containers are an old concept. For decades, Unix systems have had the `chroot` command, which provides a simple form of filesystem isolation. Since 1998, FreeBSD has had the jail utility, which extended chroot sandboxing to processes. Solaris Zones offered a comparatively complete containerization technology around 2001 but was limited to the Solaris OS. Also in 2001, SWsoft (now Parallels, Inc.) released the commercial Virtuozzo container technology for Linux and later open sourced the core technology as OpenVZ in 2005.[3] Then Google started the development of CGroups for the Linux kernel and began moving its infrastructure to containers. The Linux Containers (LXC) project started in 2008 and brought together CGroups, kernel namespaces, and chroot technology (among others) to provide a complete containerization solution. Finally, in 2013, Docker brought the final pieces to the containerization puzzle, and the technology began to enter the mainstream.

Docker took the existing Linux container technology and wrapped and extended it in various ways—primarily through portable images and a user-friendly interface—to create a complete solution for the creation and distribution of containers. The Docker platform has two distinct components: the Docker Engine (which is responsible for creating and running containers); and the Docker Hub (a cloud service for distributing containers).

The Docker Engine provides a fast and convenient interface for running containers. Before this, running a container using a technology such as LXC required significant specialist knowledge and manual work. The Docker Hub provides an enormous number of public container images for download, allowing users to quickly get started and avoid duplicating work already done by others. Further tooling developed by Docker includes Swarm, a clustering manager; Kitematic, a GUI for working with containers; and Machine, a command-line utility for provisioning Docker hosts.

By open sourcing the Docker Engine, Docker was able to grow a large community around Docker and take advantage of public help with bug fixes and enhancements. The rapid rise of Docker meant that it effectively became a *de facto* standard, which led to industry pressure to move to develop independent formal standards for the container runtime and format. In 2015, this culminated in the establishment of the Open Container Initiative, a "governance structure" sponsored by Docker, Microsoft, CoreOS, and many other important organizations, whose mission is to develop such a standard. Docker's container format and runtime forms the basis of the effort.

The uptake of containers has largely been driven by developers, who for the first time were given the tools to use containers effectively. The fast startup time of Docker containers is essential to developers who crave quick and iterative development cycles where they can promptly see the results of code changes. The portability and isolation guarantees of containers ease collaboration with other developers and operations; developers can be

sure their code will work across environments, and operations can focus on hosting and orchestrating containers rather than worrying about the code running inside them.

The changes brought about by Docker are significantly changing the way we develop software. Without Docker, containers would have remained in the shadows of IT for a long time to come.

THE SHIPPING METAPHOR

The Docker philosophy is often explained in terms of a shipping metaphor, which presumably explains the Docker name. The story normally goes something like this: when goods are transported, they have to pass through a variety of different *means*, possibly including trucks, forklifts, cranes, trains, and ships. These means have to be able to handle a wide variety of goods of different sizes and with different requirements (e.g., sacks of coffee, drums of hazardous chemicals, boxes of electronic goods, fleets of luxury cars, and refrigerated racks of lamb). Historically, this was a cumbersome and costly process, requiring the manual labor of dock workers to load and unload items by hand at each transit point (Figure 1-3).

The transport industry was revolutionized by the introduction of the intermodal container. These containers come in standard sizes and are designed to be moved between modes of transport with a minimum of manual labor. All transport machinery is designed to handle these containers, from the forklifts and cranes to the trucks, trains, and ships. Refrigerated and insulated containers are available for transporting temperature-sensitive goods, such as food and pharmaceuticals. The benefits of standardization also extend to other supporting systems, such as the labeling and sealing of containers. This means the transport industry can let the producers of goods worry about the contents of the containers so that it can focus on the movement and storage of the containers themselves.

The goal of Docker is to bring the benefits of container standardization to IT. In recent years, software systems have exploded in terms of diversity. Gone are the days of a LAMP[4] stack running on a single machine. A typical modern system may include JavaScript frameworks, NoSQL databases, message queues, REST APIs, and backends all written in a variety of programming languages. This stack has to run partly or completely on top of a variety of hardware—from the developer's laptop and the in-house testing cluster to the production cloud provider. Each of these environments is different, running different operating systems with different versions of libraries on different hardware. In short, we have a similar issue to the one seen by the transport industry—we have to continually invest substantial manual effort to move code between environments. Much as the intermodal containers simplified the transportation of goods, Docker containers simplify the transportation of software applications. Developers can concentrate on building the application and shipping it through testing and production without worrying about differences in environment and dependencies. Operations can focus on the core issues of running containers, such as allocating resources, starting and stopping containers, and migrating them between servers.

Figure 1-3. Dockers working in Bristol, England, in 1940 (by Ministry of Information Photo Division Photographer)

Docker: A History

In 2008, Solomon Hykes founded dotCloud to build a language-agnostic platform-as-a-service (PaaS) offering. The language-agnostic aspect was the unique selling point for dotCloud—existing PaaSs were tied to particular sets of languages (e.g., Heroku supported Ruby, and Google App Engine supported Java and Python). In 2010, dotCloud took part in Y Combinator's accelerator program, where it was exposed to new partners and began to attract serious investment. Some companies might have been reluctant to make such as decision (who wants to give away their magic beans), but the major turning point came in March 2013 when dotCloud recognized that Docker would benefit enormously from becoming a community-driven project.

Early versions of Docker were little more than a wrapper around LXC paired with a union filesystem, but the uptake and speed of development was shockingly fast. Within six months, it had more than 6,700 stars on GitHub and 175 nonemployee contributors. This

led dotCloud to change its name to Docker, Inc., and to refocus its business model. Docker 1.0 was announced in June 2014, just 15 months after the 0.1 release. Docker 1.0 represented a major jump in stability and reliability—it was now declared "production ready," although it had already seen production use in several companies, including Spotify and Baidu. At the same time, Docker started moving toward being a complete platform rather than just a container engine, with the launch of the Docker Hub, a public repository for containers.

Other companies were quick to see the potential of Docker. Red Hat became a major partner in September 2013 and started using Docker to power its OpenShift cloud offering. Google, Amazon, and DigitalOcean were quick to offer Docker support on their clouds, and several startups began specializing in Docker hosting, such as Stack Dock. In October 2014, Microsoft announced that future versions of Windows Server would support Docker, representing a huge shift in positioning for a company traditionally associated with bloated enterprise software.

DockerConEU in December 2014 saw the announcement of Docker Swarm, a clustering manager for Docker and Docker Machine, a command-line interface (CLI) tool for provisioning Docker hosts. This was a clear signal of Docker's intention to provide a complete and integrated solution for running containers and not allowing themselves to be restricted to only providing the Docker engine.

That same December, CoreOS announced the development of rkt, its own container runtime, and the development of the appc container specification. In June 2015, during DockerCon in San Francisco, Solomon Hykes from Docker and Alex Polvi from CoreOS announced the formation of the Open Container Initiative (then called the Open Container Project) to develop a common standard for container formats and runtimes.

Then, in June 2015, the FreeBSD project announced that Docker was now supported on FreeBSD, using ZFS and the Linux compatibility layer. In August 2015, Docker and Microsoft released a "tech preview" of the Docker Engine for Windows server.

With the release of Docker 1.8, Docker introduced the content trust feature, which verifies the integrity and publisher of Docker images. Content trust is a critical component for building trusted workflows based on images retrieved from Docker registries.

Plugins and Plumbing

As a company, Docker, Inc., has always been quick to recognize that it owes a lot of its success to the ecosystem. While Docker, Inc., was concentrating on producing a stable, production-ready version of the container engine, other companies such as CoreOS, WeaveWorks, and ClusterHQ were working on related areas, such as orchestrating and networking containers. However, it quickly became clear that Docker, Inc., was planning to provide a complete platform out of the box, including networking, storage, and

orchestration capabilities. In order to encourage continued ecosystem growth and ensure users had access to solutions for a wide range of use cases, Docker, Inc., announced it would create a modular, extensible framework for Docker where stock components could be swapped out for third-party equivalents or extended with third-party functionality. Docker, Inc., called this philosophy "Batteries Included, But Replaceable," meaning that a complete solution would be provided, but parts could be swapped out.[5]

At the time of writing, the plugin infrastructure is in its infancy, but is available. There are several plugins already available for networking containers and data management.

Docker also follows what it calls the "Infrastructure Plumbing Manifesto," which underlines its commitment to reusing and improving existing infrastructure components where possible and contributing reusable components back to the community when new tools are required. This led to the spinning out of the low-level code for running containers into the runC project, which is overseen by the OCI and can be reused as the basis for other container platforms.

64-Bit Linux

At the time of writing, the only stable, production-ready platform for Docker is 64-bit Linux. This means your computer will need to run a 64-bit Linux distribution, and all your containers will also be 64-bit Linux. If you are a Windows or Mac OS user, you can run Docker inside a VM.

Support for other native containers on other platforms, including BSD, Solaris, and Windows Server, is in various stages of development. Because Docker does not natively do any virtualization, containers must always match the host kernel—a Windows Server container can only run on a Windows Server host, and a 64-bit Linux container will only run on a 64-bit Linux host.

MICROSERVICES AND MONOLITHS

One of the biggest use cases and strongest drivers behind the uptake of containers are *microservices*.

Microservices are a way of developing and composing software systems such that they are built out of small, independent components that interact with one another over the network. This is in contrast to the traditional *monolithic* way of developing software, where there is a single large program, typically written in C++ or Java.

When it comes to scaling a monolith, commonly the only choice is to *scale up*, where extra demand is handled by using a larger machine with more RAM and CPU power. Conversely, microservices are designed to *scale out*, where extra demand is handled by provisioning multiple machines the load can be spread over. In a microservice architecture, it's possible to only scale the resources required for a particular service, focusing on the bottlenecks in the system. In a monolith, it's scale everything or nothing, resulting in wasted resources.

In terms of complexity, microservices are a double-edged sword. Each individual microservice should be easy to understand and modify. However, in a system composed of dozens or hundreds of such services, the overall complexity increases due to the interaction between individual components.

The lightweight nature and speed of containers mean they are particularly well suited for running a microservice architecture. Compared to VMs, containers are vastly smaller and quicker to deploy, allowing microservice architectures to use the minimum of resources and react quickly to changes in demand.

For more information on microservices, see *Building Microservices* by Sam Newman (O'Reilly) and Martin Fowler's Microservice Resource Guide.

1 The diagram depicts a *type 2* hypervisor, such as VirtualBox or VMware Workstation, which runs on top of a host OS. *Type 1* hypervisors, such as Xen, are also available where the hypervisor runs directly on top of the bare metal.

2 The kernel is the core component in an OS and is responsible for providing applications with essential system functions related to memory, CPU, and device access. A full OS consists of the kernel plus various system programs, such as init systems, compilers, and window managers.

3 OpenVZ never achieved mass adoption, possibly because of the requirement to run a patched kernel.

4 This originally stood for Linux, Apache, MySQL, and PHP—common components in a web application.

5 Personally, I've never liked the phrase; all batteries provide much the same functionality and can only be swapped with batteries of the same size and voltage. I assume the phrase has its origins in Python's "Batteries Included" philosophy, which it uses to describe the extensive standard library that ships with Python.