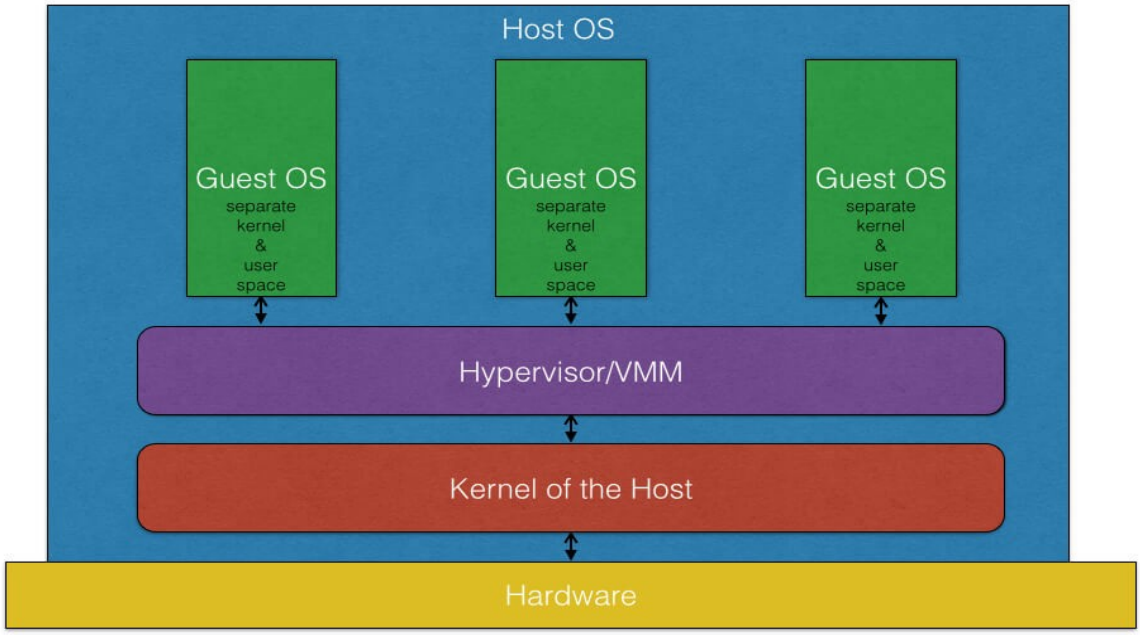
The need for containers

Hypervisor based virtualization technologies have existed for a long time now. Since a hypervisor or full virtualization mechanism emulates the hardware, you can run any operating system on top of any other, Windows on Linux, or the other way around. Both the guest operating system and the host operating system run with their own kernel and the communication of the guest system with the actual hardware is done through an abstracted layer of the hypervisor.

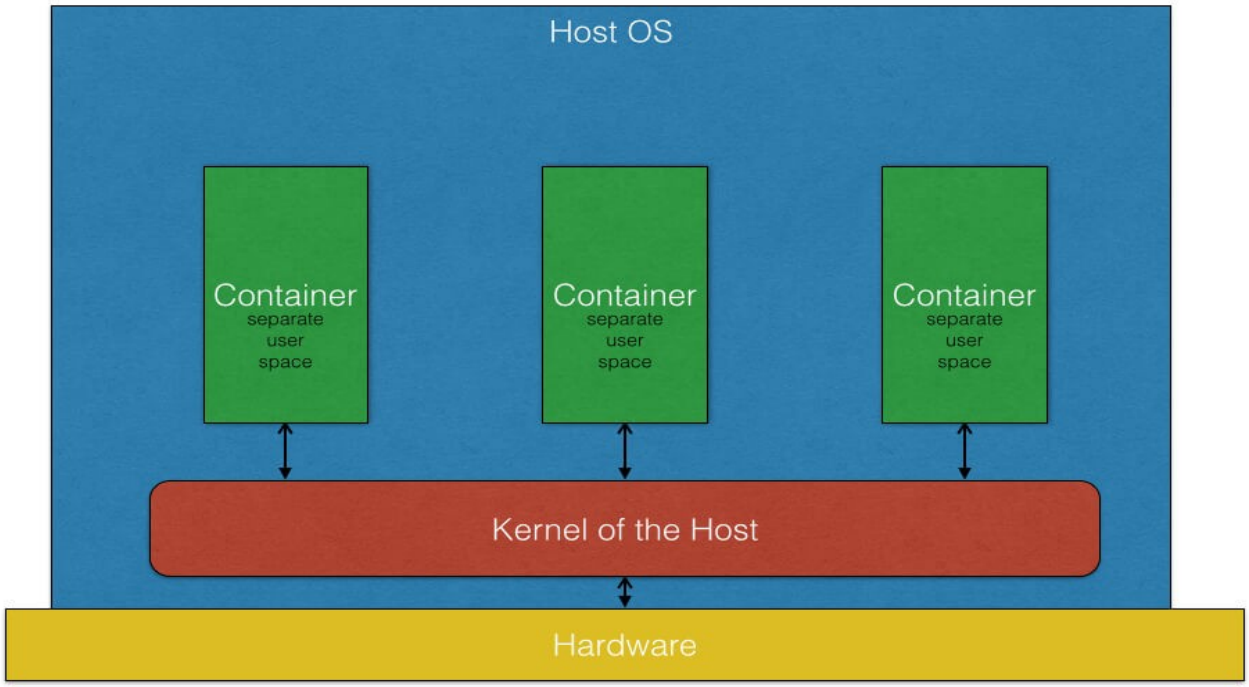


Hypervisor based Virtualization

This approach usually provides a high level of isolation and security as all communication between the guest and host is through the hypervisor. This approach is also usually slower and incurs significant performance overhead due to the hardware emulation. To reduce this overhead, another level of virtualization called "operating system virtualization" or "container virtualization" was introduced which allows running multiple isolated user space instances on the same kernel.

What are containers?

Containers are the products of operating system virtualization. They provide a lightweight virtual environment that groups and isolates a set of processes and resources such as memory, CPU, disk, etc., from the host and any other containers. The isolation guarantees that any processes inside the container cannot see any processes or resources outside the container.



Operating System/Container Virtualization

The difference between a container and a full-fledged VM is that all containers share the same kernel of the host system. This gives them the advantage of being very fast with almost 0 performance overhead compared with VMs. They also utilize the different computing resources better because of the shared kernel. However, like everything else, sharing the kernel also has its set of shortcomings.

- Type of containers that can be installed on the host should work with the kernel of the host. Hence, you cannot install a Windows container on a Linux host or vice-versa.
- Isolation and security -- the isolation between the host and the container is not as strong as hypervisor-based virtualization since all containers share the same kernel of the host and there have been cases in the past where a process in the container has managed to escape into the kernel space of the host.

Common cases where containers can be used

As of now, I have noticed that containers are being used for two major uses - as a usual operating system or as an application packaging mechanism. There are also other cases like using containers as routers but I don't want to get into those in this blog.

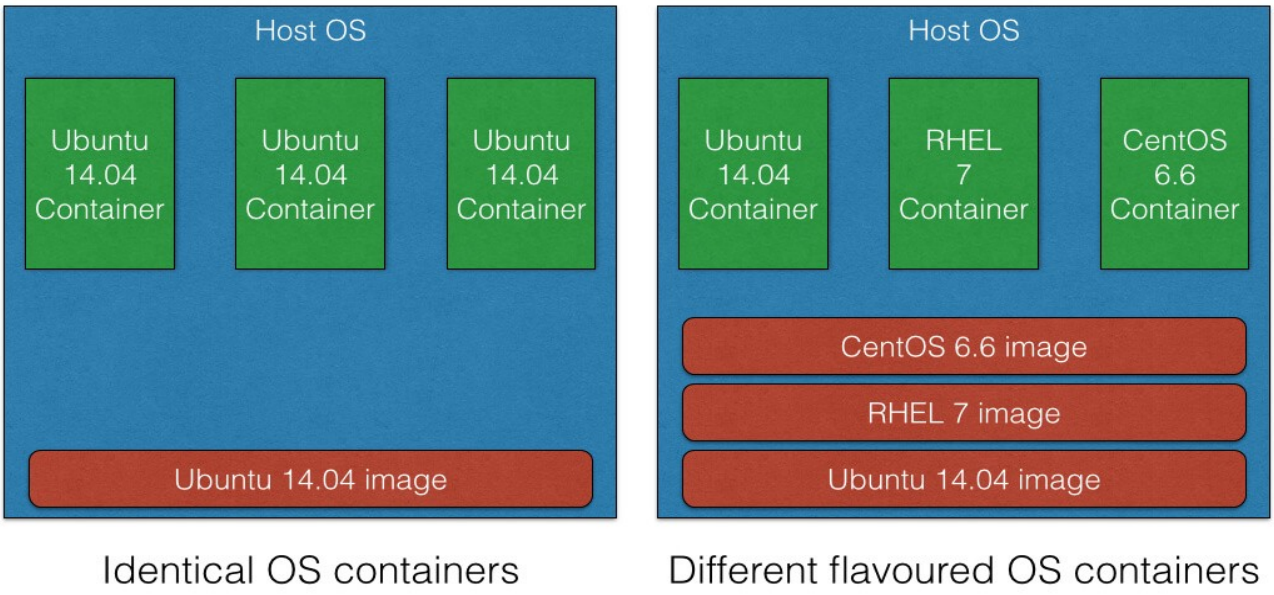
I like to classify the containers into special types based on how they can be used. Although I will also point out that it is not a must to use a container technology for just that case, and you may very well use it for other cases. I've classified them this way because I find certain technologies easier to use for certain cases. Based on the two uses I mentioned above I've classified containers as OS

containers and application containers.

OS containers

OS containers are virtual environments that share the kernel of the host operating system but provide user space isolation. For all practical purposes, you can think of OS containers as VMs. You can install, configure and run different applications, libraries, etc., just as you would on any OS. Just as a VM, anything running inside a container can only see resources that have been assigned to that container.

OS containers are useful when you want to run a fleet of identical or different flavors of distros. Most of the times containers are created from templates or images that determine the structure and contents of the container. It thus allows you to create containers that have identical environments with the same package versions and configurations across all containers.



Container technologies like LXC, OpenVZ, Linux VServer, BSD Jails and Solaris zones are all suitable for creating OS containers.

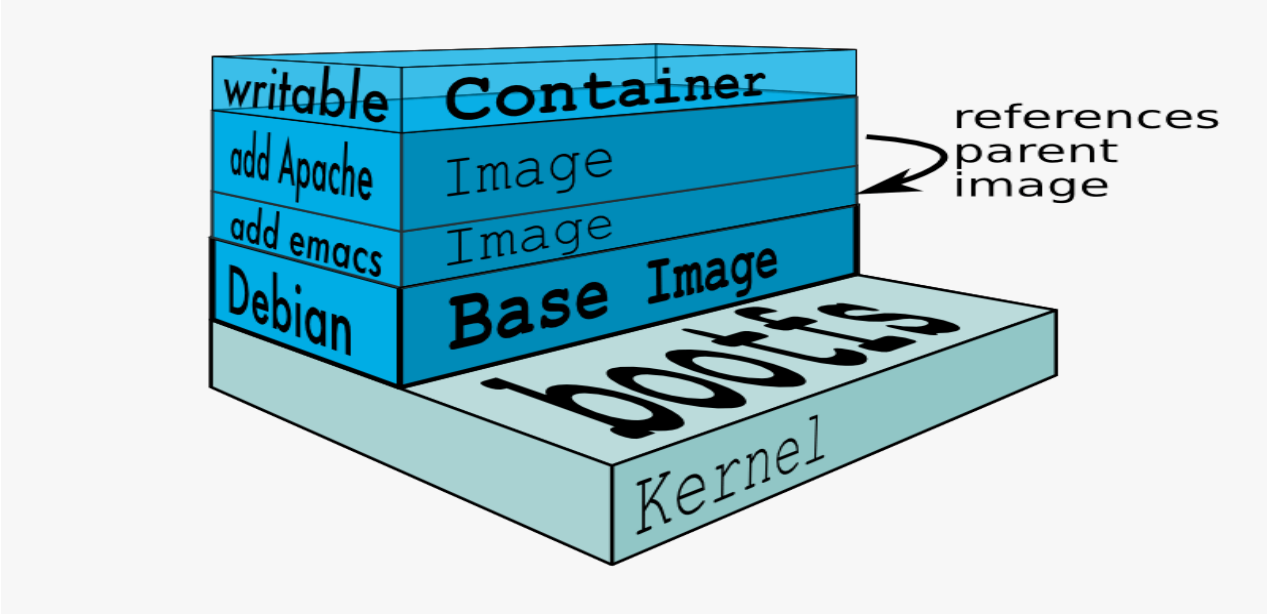
Application containers

While OS containers are designed to run multiple processes and services, application containers are designed to package and run a single service. Container technologies like Docker and Rocket are examples of application containers. So even though they share the same kernel of the host there are subtle differences make them different, which I would like to talk about using the example of a Docker container:

Run a single service as a container

When a Docker container is launched, it runs a single process. This process is usually the one that runs your application when you create containers per application. This very different from the traditional OS containers where you have multiple services running on the same OS.

Layers of containers



Any `RUN` commands you specify in the Dockerfile creates a new layer for the container. In the end when you run your container, Docker combines these layers and runs your containers. Layering helps Docker to reduce duplication and increases the re-use. This is very helpful when you want to create different containers for your components. You can start with a base image that is common for all the components and then just add layers that are specific to your component. Layering also helps when you want to rollback your changes as you can simply switch to the old layers, and there is almost no overhead involved in doing so.

Built on top of other container technologies

Until some time ago, Docker was built on top of LXC. If you look at the Docker FAQ, they mention a number of points which point out the differences between LXC and Docker.

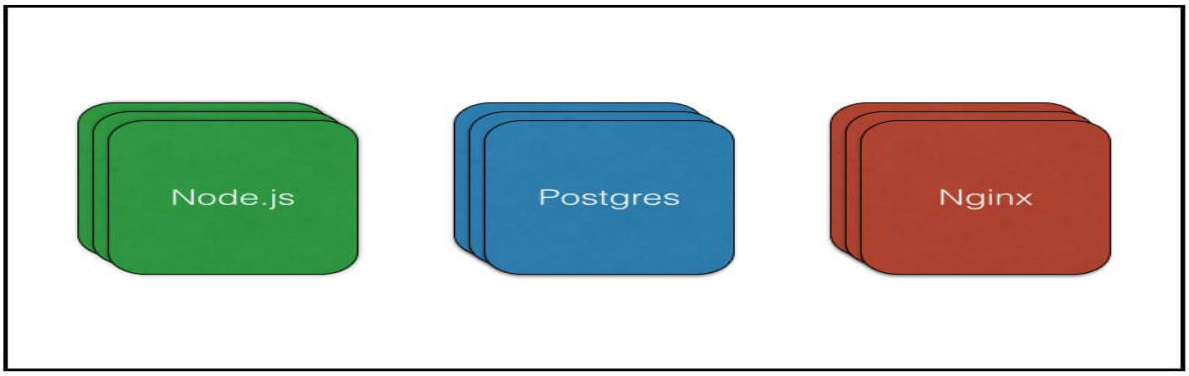
The idea behind application containers is that you create different containers for each of the components in your application. This approach works especially well when you want to deploy a distributed, multi-component system using the microservices architecture. The development team gets the freedom to package their own applications as a single deployable container. The operations teams get the freedom of deploying the container on the operating system of their choice as well as the ability to scale both horizontally and vertically the different applications. The end state is a system that has different applications and services each running as a container that then talk to each other using the APIs and protocols that each of them supports.

In order to explain what it means to run an app container using Docker, let's take a simple example of a three-tier architecture in web development which has a `PostgreSQL` data tier, a `Node.js` application tier and an `Nginx` as the load balancer tier.

In the simplest cases, using the traditional approach, one would put the database, the Node.js app and Nginx on the same machine.

Typical 3-tier architecture in the simplest sense

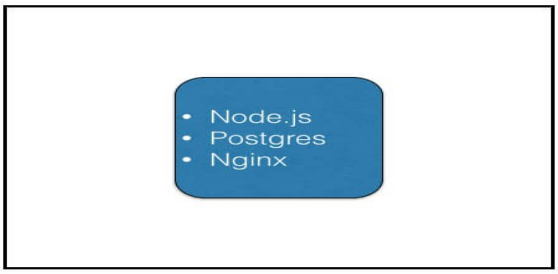
Deploying this architecture as Docker containers would involve building a container image for each of the tiers. You then deploy these images independently, creating containers of varying sizes and capacity according to your needs.



Typical 3-tier architecture using Docker containers

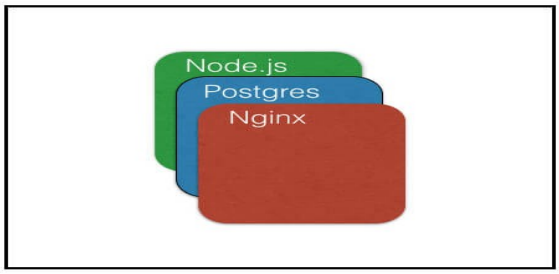
Summary

So in general when you want to package and distribute your application as components, application containers serve as a good resort. Whereas, if you just want an operating system in which you can install different libraries, languages, databases, etc., OS containers are better suited.



OS containers

- Meant to used as an OS - run multiple services
- No layered filesystems by default
- Built on cgroups, namespaces, native process resource isolation
- Examples - LXC, OpenVZ, Linux VServer, BSD Jails, Solaris Zones

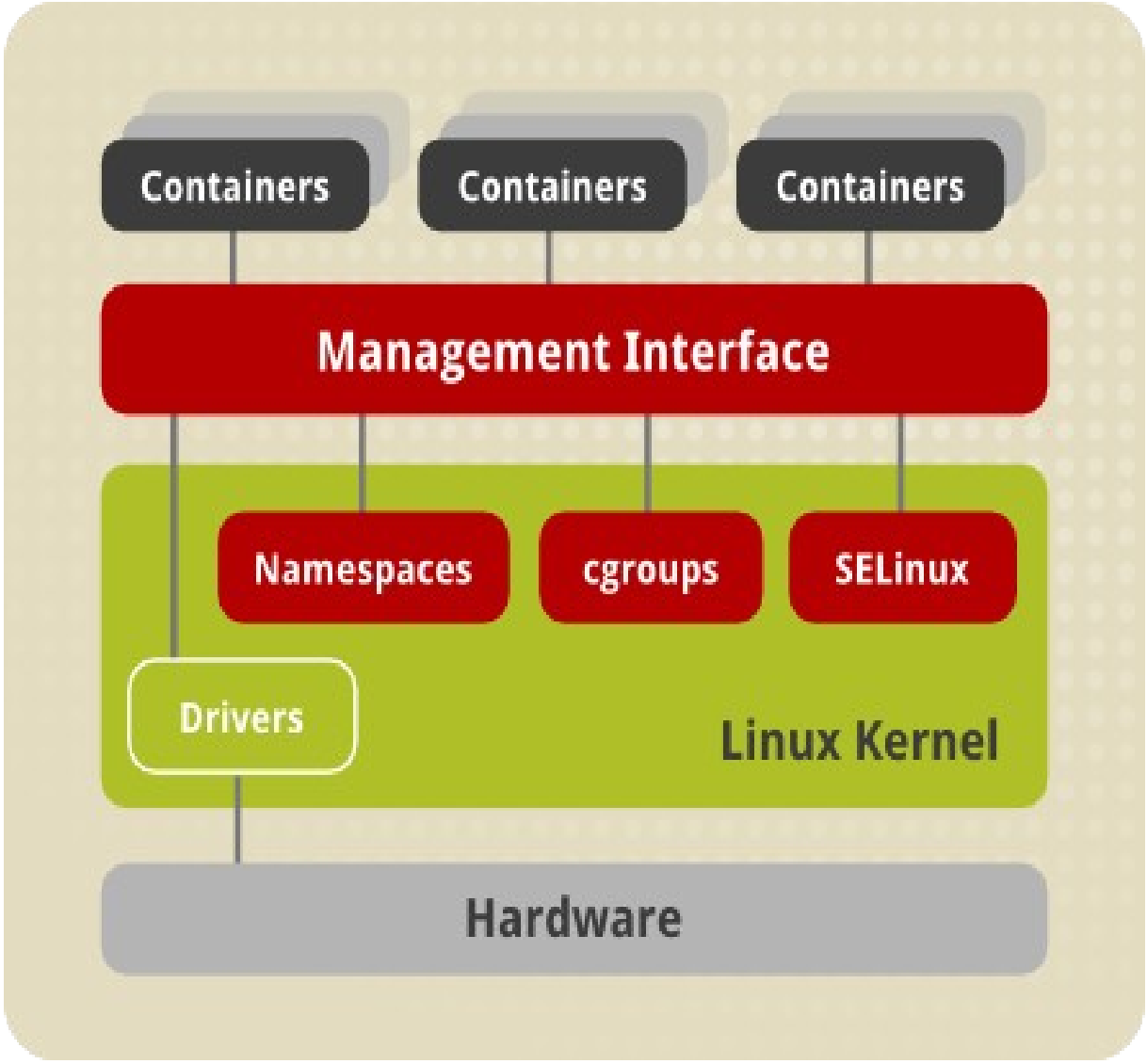


App containers

- Meant to run for a single service
- Layered filesystems
- Built on top of OS container technologies
- Examples - Docker, Rocket

The first thing you need to know is Docker's philosophy : run **one process** isolated in a container. You won't run an OS in a Docker container, you will run a process inside a container with a root filesystem content based on a linux distribution of your choosing. Ubuntu is a choice among others.

The following scheme illustrates the architecture of Linux Containers in Red Hat Enterprise Linux



Namespaces

The kernel provides process isolation by creating separate **namespaces** for containers. Namespaces enable creating an abstraction of a particular global system resource and make it appear as a separated instance to processes within a namespace.

Mount namespaces isolate the set of file system mount points seen by a group of processes so that processes in different mount namespaces can have different views of the file system hierarchy.

UTS namespaces This allows each container to have its own hostname and NIS domain name, which is useful for initialization and configuration scripts based on these names.

IPC namespaces isolate certain interprocess communication (IPC) resources, two containers can create shared memory segments and semaphores with the same name, but are not able to interact with other containers memory segments or shared memory.

PID namespaces allow processes in different containers to have the same PID, so each container can have its own init (PID1) process that manages various system initialization tasks as well as containers life cycle. Also, each container has its unique `/proc` directory. Note that from within the container you can monitor only processes running inside this container.

Network namespaces provide isolation of network controllers, system resources associated with networking, firewall and routing tables. You can add virtual or real devices to the container, assign them their own IP Addresses and even full iptables rules. You can view the different network settings by executing the `ip addr` command on the host and inside the container.

Control Groups (cgroups)

The kernel uses **cgroups** to group processes for the purpose of system resource management. Cgroups allocate CPU time, system memory, network bandwidth, or combinations of these among user-defined groups of tasks.

Secure Containers with SELinux

From the security point of view, there is a need to isolate the host system from a container and to isolate containers from each other. The kernel features used by containers, namely cgroups and namespaces, by themselves provide a certain level of security.

However, this can not prevent a hostile process from breaking out of the container since the entire system is not namespaced or containerized. Another level of separation, provided by SELinux, is therefore needed.
Security-Enhanced Linux (**SELinux**) is an implementation of a mandatory access control (MAC) mechanism,

This architecture provides a secure separation for containers as it prevents root processes within the container from interfering with other processes running outside this container. The containers created with Docker are automatically assigned with an SELinux context specified in the SELinux policy.

Linux Containers Compared to KVM Virtualization

KVM virtual machines require a kernel of their own. Linux containers share the kernel of the host operating system. It is usually possible to launch a much larger number of containers than virtual machines on the same hardware.

Both Linux Containers and KVM virtualization have certain advantages and drawbacks that influence the use cases in which these technologies are typically applied:

KVM virtualization:

- The resource-hungry nature of virtual machines (as compared to containers) means that the number of virtual machines that can be run on a host is lower than the number of containers that can be run on the same host.
- Running separate kernel instances generally provides separation and security. The unexpected termination of one of the kernels does not disable the whole system.

- Guest virtual machine is isolated from host changes.

Linux Containers:

- Linux Containers are designed to support isolation of one or more applications.
- System-wide changes are visible in each container. For example, if you upgrade an application on the host machine, this change will apply to all sandboxes that run instances of this application.
- Since containers are lightweight, a large number of them can run simultaneously on a host machine. The theoretical maximum is 6000 containers and 12,000 bind mounts of root file system directories.