

Cloud Computing

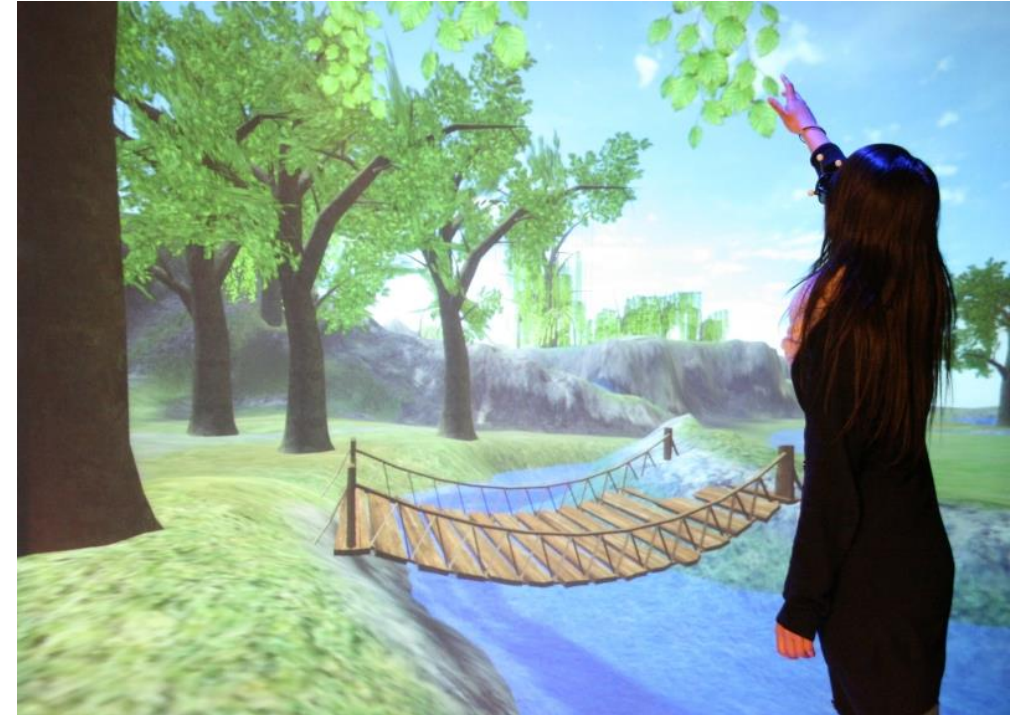
Kapitel 3: Virtualisierung

Florian Lautenschlager, Dr. Josef Adersberger

Grundlagen zur Virtualisierung

Virtualisierung

- **Virtualisierung:** die Erzeugung von virtuellen Realitäten und deren Abbildung auf die physikalische Realität.
- Zweck:
 - **Multiplizität** → Erzeugung mehrerer virtueller Realitäten innerhalb einer physikalischen Realität
 - **Entkopplung** → Bindung und Abhängigkeit zur Realität auflösen
 - **Isolation** → Physikalische Seiteneffekte zwischen den virtuellen Realitäten vermeiden



<http://www.techfak.uni-bielefeld.de>

Virtualisierungsarten

Virtualisierung ist stellvertretend für mehrere grundsätzlich verschiedene Konzepte und Technologien:

■ Virtualisierung von Hardware-Infrastruktur

1. Emulation
2. Voll-Virtualisierung (Typ-2 Virtualisierung)
3. Para-Virtualisierung (Typ-1 Virtualisierung)

■ Virtualisierung von Software-Infrastruktur

4. Betriebssystem-Virtualisierung (*Containerization*)
5. Anwendungs-Virtualisierung (*Runtime*)

Virtualisierung und Cloud Computing

- Entkopplung von der Hardware für mehr Flexibilität im Betrieb und Robustheit bei Ausfällen.
- Normierung von Ressourcen-Kapazitäten auf heterogener und wechselnder Hardware („S-Instanz“, „XL-Instanz“).
- Zentrale Steuerung und Bereitstellung von Rechen-Ressourcen über die mit Virtualisierung bereitgestellte Software-Defined-Resources.

Virtualisierungsarten

Was wird virtualisiert?

■ Prozessor

- Virtuelle Rechenkerne
- Dispatching von Prozessor-Befehlen auf echte Rechenkerne

■ Hauptspeicher

- Virtuelle Hauptspeicher-Partition
- Management der realen Repräsentation (im RAM, auf Festplatte, Ballooning)

■ Netzwerk

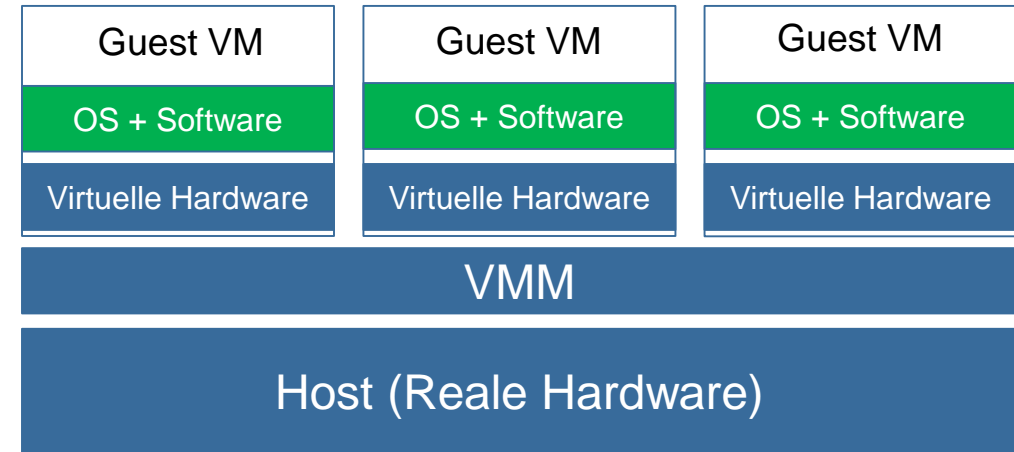
- Virtuelle Netzwerkschnittstellen und virtuelle Netzwerk-Infrastrukturen (VLAN)
- Brücken zwischen virtuellen und realen Netzwerken

■ Storage

- Virtuelle Festplatten-Laufwerke. Abbildung auf Dateien im realen Dateisystem. Volumen entweder vor-allokiert oder dynamisch wachsend.
- Virtuelle SANs (Storage Area Networks) über Aufteilung der Daten eines virtuellen Laufwerks auf viele Storage-Einheiten.

Hardware-Virtualisierung

- Durch Hardware-Virtualisierung werden die Ressourcen eines Rechnersystems aufgeteilt und von mehreren unabhängigen Betriebssystem-Instanzen genutzt.
- Anforderungen der Betriebssystem-Instanzen werden von der Virtualisierungssoftware (Virtual Machine Monitor, VMM) abgefangen und auf die real vorhandene Hardware umgesetzt.



Host

- Der Rechner der eine oder mehrere virtuelle Maschinen ausführt und die dafür notwendigen Hardware-Ressourcen zur Verfügung stellt.

Guest

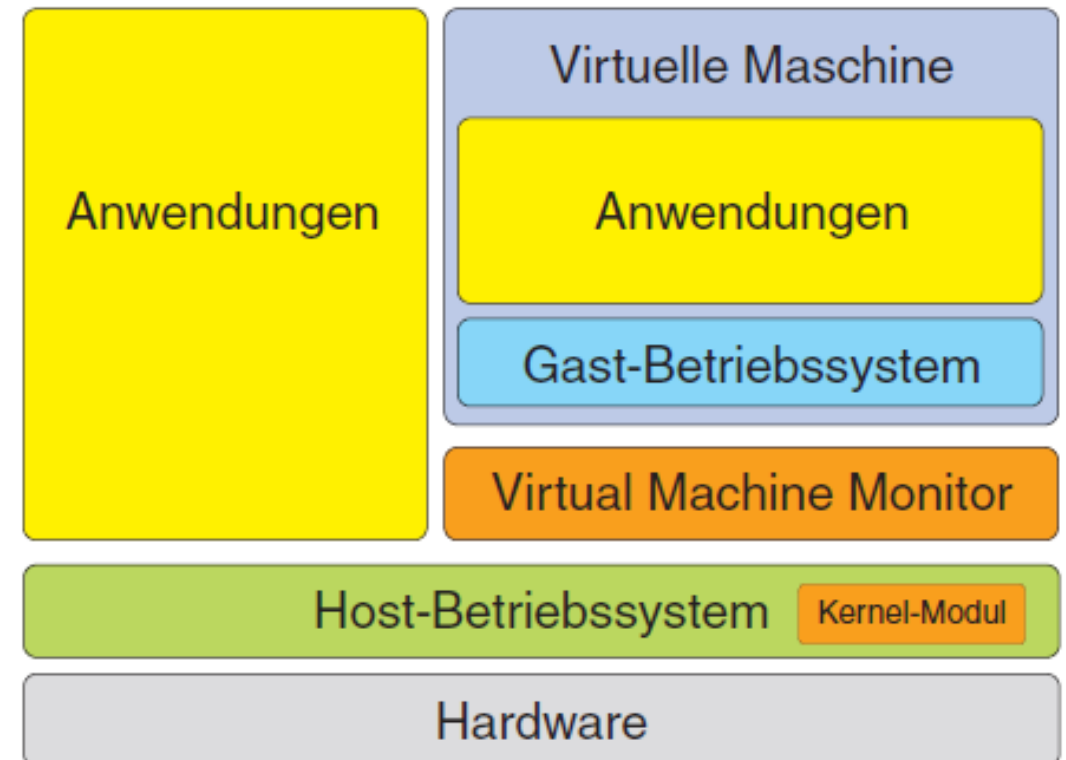
- Eine lauffähige / laufende virtuelle Maschine

VMM (Virtual Machine Monitor)

- Die Steuerungssoftware zur Verwaltung der Guests und der Host-Ressourcen

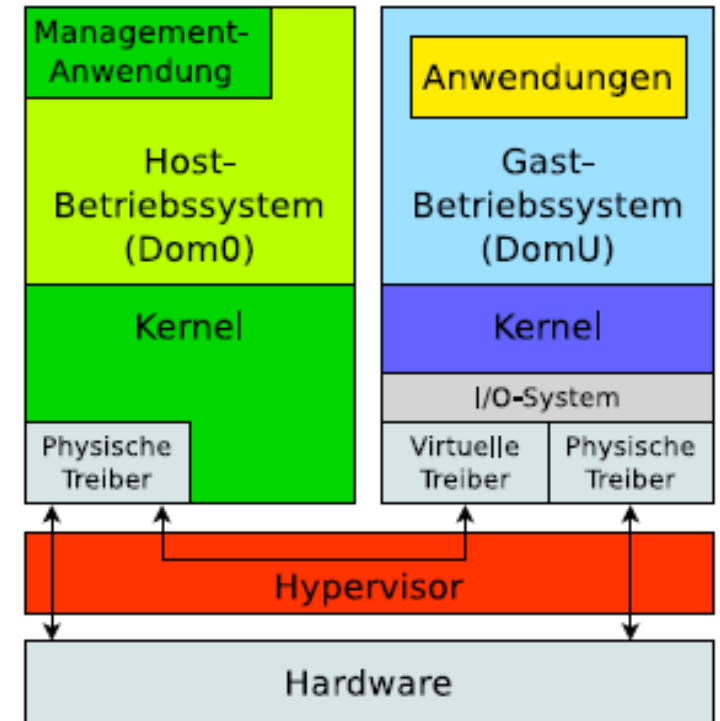
Hardware-Virtualisierung: Voll-Virtualisierung

- Jedem Gastbetriebssystem steht ein eigener virtueller Rechner mit virtuellen Ressourcen wie CPU, Hauptspeicher, Laufwerken, Netzwerkkarten, usw. zur Verfügung
- Der VMM läuft hosted als Anwendung unter dem Host-Betriebssystem (Typ 2 Hypervisor)
- Der VMM verteilt die Hardwareressourcen des Rechners an die VMs
- Teilweise emuliert der VMM Hardware, die nicht für den gleichzeitigen Zugriff mehrerer Betriebssysteme ausgelegt ist (z.B. Netzwerkkarten, Grafikkarten)
- Leistungsverlust: 5-10%.



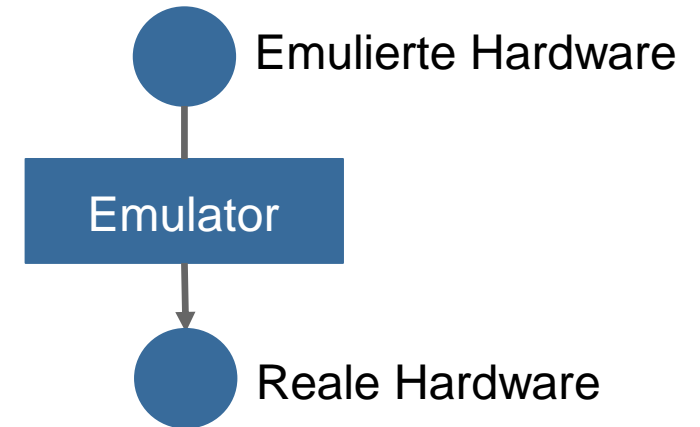
Hardware-Virtualisierung: Para-Virtualisierung

- Der Hypervisor läuft direkt auf der verfügbaren Hardware. Er entspricht somit einem Betriebssystem, das ausschließlich auf Virtualisierung ausgerichtet ist.
- Das Gast-Betriebssystem muss um virtuelle Treiber ergänzt werden, um mit dem Hypervisor interagieren zu können.
 - Dem Gast-Betriebssystem stehen keine direkt low-level virtualisierten Hardware-Ressourcen (CPU, RAM, ...) zur Verfügung sondern eine API zur Nutzung durch die virtuellen Treiber.
- Unterstützte Betriebssysteme und Hardware-Varianten aus Sicht des Gastes eingeschränkt pro Hypervisor-Implementierung.
- Der Hypervisor nutzt die Treiber eines Host-Betriebssystems, um auf die reale Hardware zuzugreifen. Damit brauchen im Hypervisor nicht aufwändig eigene Treiber implementiert werden.
- Leistungsstärkste Virtualisierung (Leistungseinbuße: 2-3%)

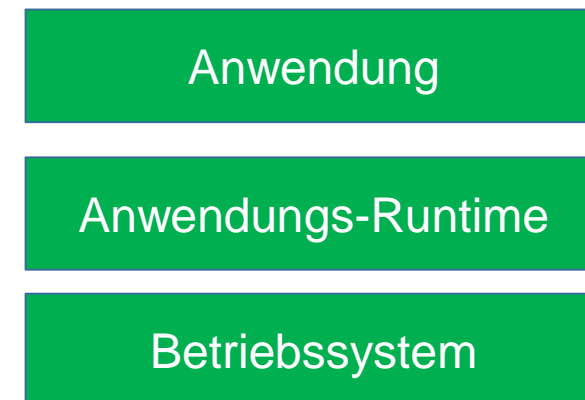


Zur Vollständigkeit: Was ist Emulation und Anwendungs-Virtualisierung?

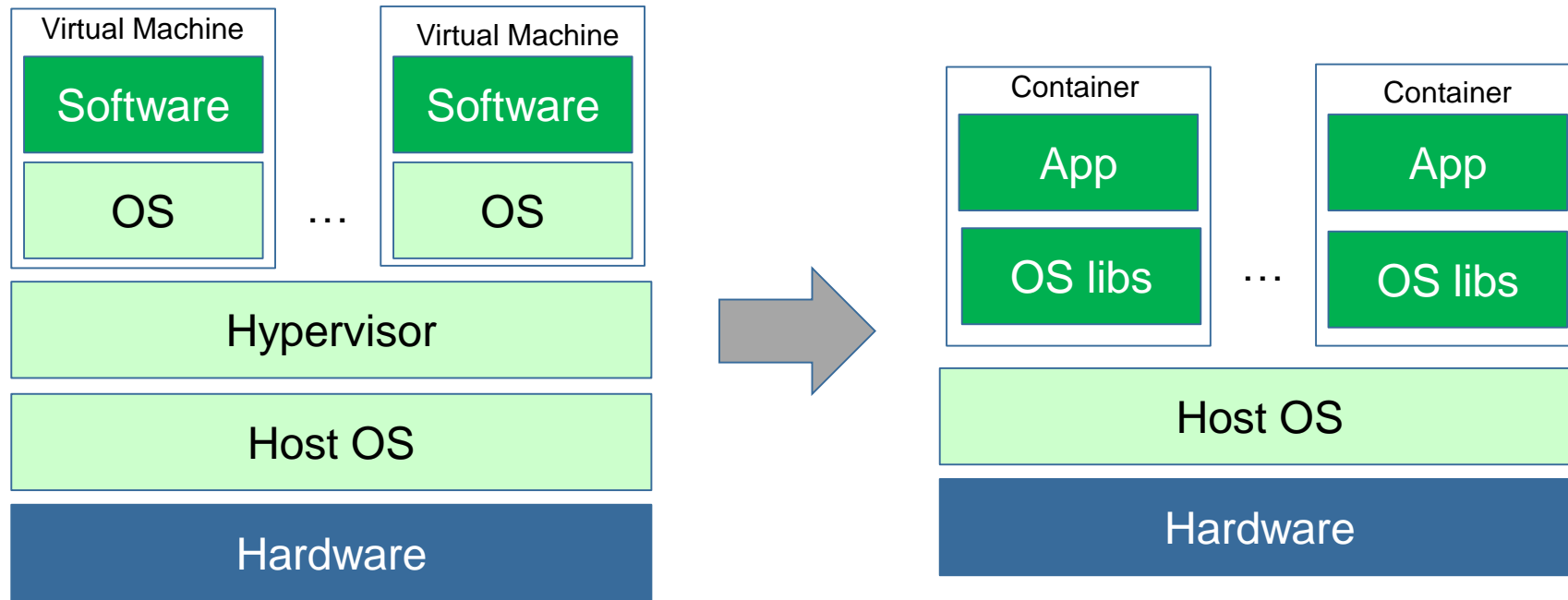
- **Emulation:** Bildet die Hardware eines nicht vorhandenen oder nicht kompatiblen Rechnersystems oder Teile eines entsprechenden Rechnersystems nach. Zweck u.A.: Alte Software konservieren.
(Beispiel: PearPC)



- **Anwendungs-Virtualisierung:** Stellt Anwendungen eine Programmierschnittstelle und eine Laufzeitumgebung (Runtime) zur Verfügung, die komplett vom darunterliegenden Betriebssystem entkoppelt. Zweck u.A.: Portable Anwendungen.
(Beispiele: JVM, CLR)



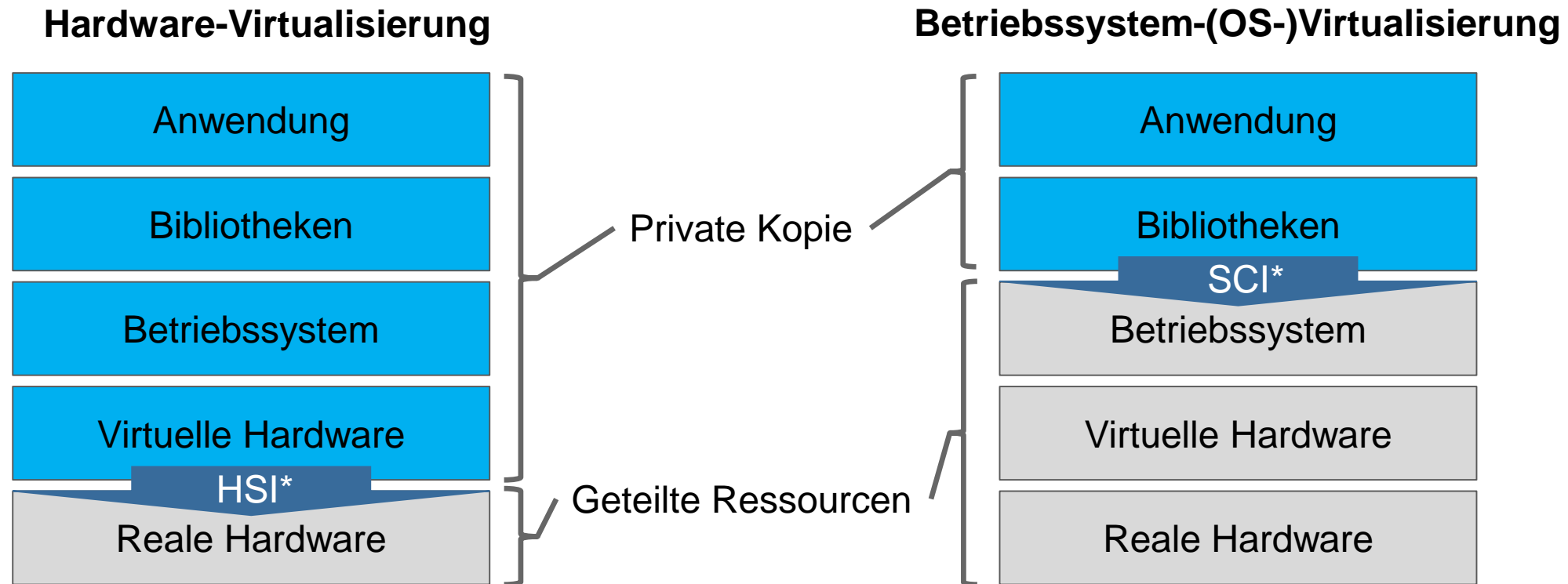
Betriebssystem-Virtualisierung



- Leichtgewichtiger Virtualisierungsansatz: Es gibt keinen Hypervisor. Jede App läuft direkt als Prozess im Host-Betriebssystem. Dieser ist jedoch maximal durch entsprechende OS-Mechanismen isoliert (z.B. Linux LXC).
 - Isolation des Prozesses durch Kernel Namespaces (bzgl. CPU, RAM und Disk I/O) und Containments
 - Isoliertes Dateisystem
 - Eigene Netzwerk-Schnittstelle
- CPU- / RAM-Overhead in der Regel nicht messbar (~ 0%)
- Startup-Zeit = Startdauer für den ersten Prozess

Hardware- vs. Betriebssystem-Virtualisierung

*) HSI = Hardware Software Interface
SCI = System Call Interface



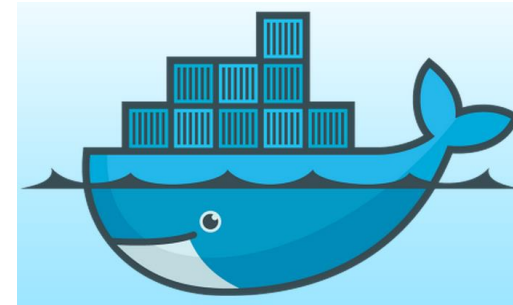
- Stärkere Isolation
- Höhere Sicherheit

- Geringeres Volumen der privaten Kopie
- Geringerer Overhead

Hardware-Virtualisierung: Vagrant und VirtualBox



Betriebssystem- Virtualisierung: Docker



Hardware-Virtualisierung: Vagrant und VirtualBox



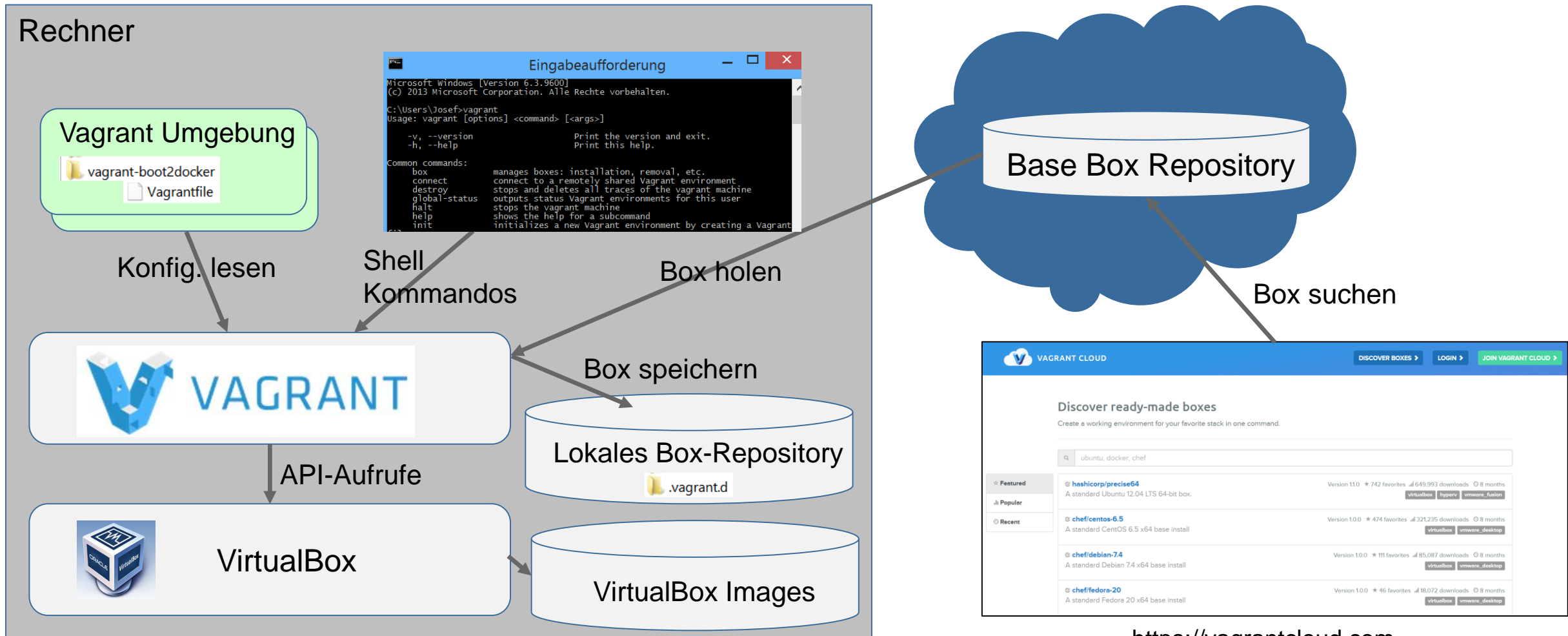
+



Open Source Typ 2 Virtualisierungssoftware (Voll-Virtualisierung) für Windows, Linux, OS X und Solaris.

Automationssoftware für virtuelle Umgebungen auf einem Rechner. Virtuelle Maschinen per Kommandozeile erstellen und steuern.

Vagrant: Eine schematische Übersicht.



Das Vagrantfile beschreibt die zu erstellende virtuelle Maschine.

```
# -*- mode: ruby -*-  
# vi: set ft=ruby :  
  
# Vagrantfile API/syntax version. Don't touch unless you know what you're doing!  
VAGRANTFILE_API_VERSION = "2"
```

Vagrantfiles werden in
Ruby geschrieben

```
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
```

```
  # My base box  
  config.vm.box = "chef/ubuntu-14.04"
```

Definition der Basis-Box

```
  # Define shell provisioning  
  config.vm.provision :shell, path: "bootstrap.sh"
```

Konfiguration der Provisionierung

```
  # Define docker provisioning  
  config.vm.provision "docker" do |d|  
    d.run "nginx1", image: "dockerfile/nginx", args: "-p 8080:80", daemonize: true  
    d.run "nginx2", image: "dockerfile/nginx", args: "-p 9080:80", daemonize: true  
    d.run "haproxy", image: "dockerfile/haproxy", args: "-p 80:80 --link nginx1:nginx1 --link nginx2:nginx2 -v /vagrant:/haproxy-override"  
  end
```

```
  # Configure VirtualBox  
  config.vm.provider "virtualbox" do |v|  
    v.memory = 1024  
    v.cpus = 4  
  end
```

Konfiguration des Virtualisierungs-Providers

```
  # Forward ports  
  config.vm.network :forwarded_port, host: 80, guest: 80  
  config.vm.network :forwarded_port, host: 8080, guest: 8080  
  config.vm.network :forwarded_port, host: 9080, guest: 9080
```

Konfiguration des Netzwerks

```
end
```

Ein typischer Arbeitsablauf mit Vagrant.

#	Befehle auf Kommandozeile	Bedeutung
1	<code>md <box-dir></code> <code>cd <box-dir></code>	Verzeichnis für Vagrant Umgebung erstellen und dorthin wechseln
2	<code>vagrant init [<box-name>] [<box-url>]</code>	Eine Vagrant Umgebung initialisieren. Dabei wird zunächst nur eine Datei <i>Vagrantfile</i> erstellt und initial mit dem Namen und der URL der Box (falls angegeben) initialisiert.
3		Vagrantfile anpassen nach Bedarf (z.B. IP vergeben, Port-Mapping zwischen Host und Guest, Verzeichnis-Share zwischen Host und Guest, ...)
4	<code>vagrant up</code>	Startet die virtuelle Maschine (Box → virtuelle Maschine) und konfiguriert sie entsprechend dem Vagrantfile
5	<code>vagrant ssh</code>	Per SSH auf die virtuelle Maschine verbinden
6	<code>exit</code>	Die SSH Kommandozeile in der virtuellen Maschine verlassen
7	<code>vagrant halt</code>	Die virtuelle Maschine stoppen

Weitere nützliche Kommandos:

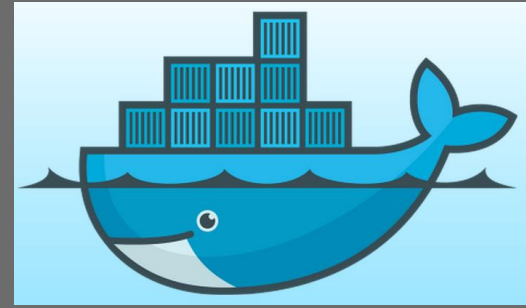
- `reload`: Startet eine VM neu und aktualisiert die Konfiguration entsprechend dem Vagrantfile
- `package`: Erstellt aus einer virtuellen Maschine wieder eine Box

Weitere Kommandos: <http://docs.vagrantup.com/v2/cli/index.html>

Vagrant Befehle auf Kommandozeile

- vagrant box add – allows you to install a box (or VM) to the local machine
- vagrant box remove – removes a box from the local machine
- vagrant box list – lists the locally installed Vagrant boxes
- vagrant init – initializes a project to use Vagrant
- vagrant up – starts up the vagrant VM
- vagrant suspend – saves the state of the current VM.
- vagrant resume – will load up the suspended VM.
- vagrant halt – will shut down the VM, saving configuration. (restart with 'up' command)
- vagrant destroy – will destroy the VM with all config changes.
- vagrant reload – apply Vagrant configuration changes (like port forwarding) without rebuilding the VM.
- vagrant status – tells you the current state of the Vagrant project's VM
- vagrant gem – install Vagrant plugins via RubyGems
- vagrant ssh – short cut to SSH into the running VM
- vagrant package – create a distribution of the VM you have running.
- vagrant <command> -help - Command that will provide man pages for a vagrant command.

Containerization mit Docker



Google Runs All Software In Containers

May 28, 2014 by Timothy Prickett Morgan



The overhead of full-on server virtualization is too much for a lot of hyperscale datacenter operators as well as their peers (some might say rivals) in the supercomputing arena. But the ease of management and resource allocation control that comes from virtualization are hard to resist and this has fomented a third option between bare metal and server virtualization. It is called containerization and Google recently gave a glimpse into how it is using containers at scale on its internal infrastructure as well as on its public cloud.

We are talking about billions of containers being fired up a week here, just so you get a sense of the scale.

<http://www.enterprisetech.com/2014/05/28/google-runs-software-containers>



...

@cloud_opinion



Following

"Team, please adopt containers immediately."

"why?"

"Because they are hot"

"But, why are they hot"

"Because they are hot, damn it"

RETWEETS

4

FAVORITES

6



12:01 PM - 21 Jul 2015

https://twitter.com/cloud_opinion/status/623568543771045888

Containerization mit Docker



<http://www.srf.ch/kultur/im-fokus/brasilien/favelas-im-wandel-die-armen-muessen-weichen>



Standard-Betriebs- und Software-Logistik-Format

Docker

- Docker ist eine Automationsumgebung für Betriebssystem-Virtualisierung.
- Aktuell unterstützt Docker Linux als Host-Betriebssystem. Eine Windows-Variante ist in Arbeit und erscheint mit Windows Server 2016.
- Docker ist als Werkzeug eines Cloud-Anbieters entstanden und ist mittlerweile eines der sichtbarsten und aktivsten Open-Source-Ökosysteme.

In a Nutshell, docker...

... has had 19,640 commits made by 1,298 contributors representing 328,622 lines of code

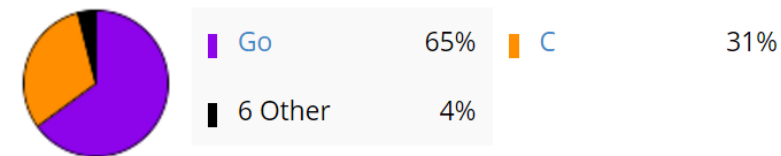
... is mostly written in Go with an average number of source code comments

... has a young, but established codebase maintained by a very large development team with stable Y-O-Y commits

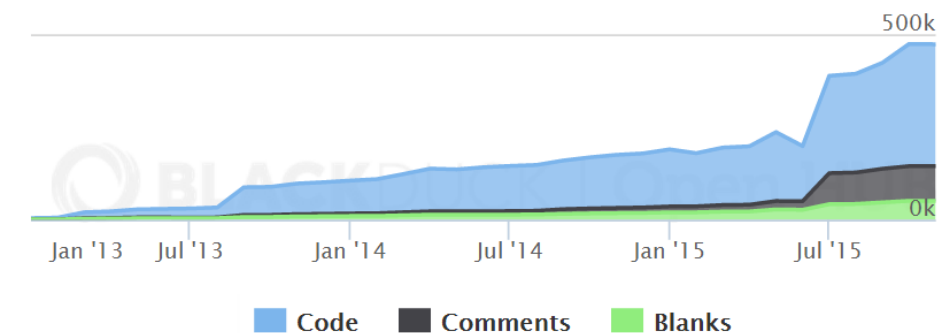
... took an estimated 88 years of effort (COCOMO model) starting with its first commit in January, 2013 ending with its most recent commit 3 days ago

<https://www.openhub.net/p/docker>

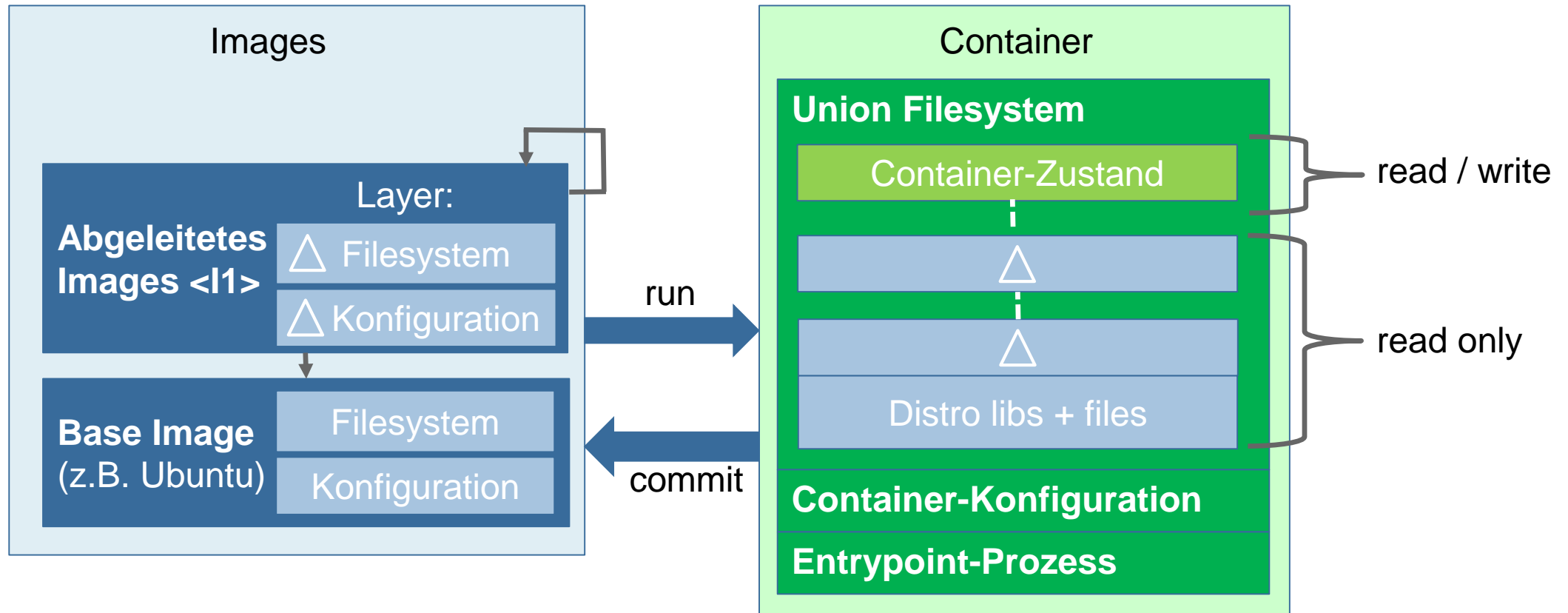
Languages



Lines of Code



Im Zentrum von Docker stehen Images und Container.

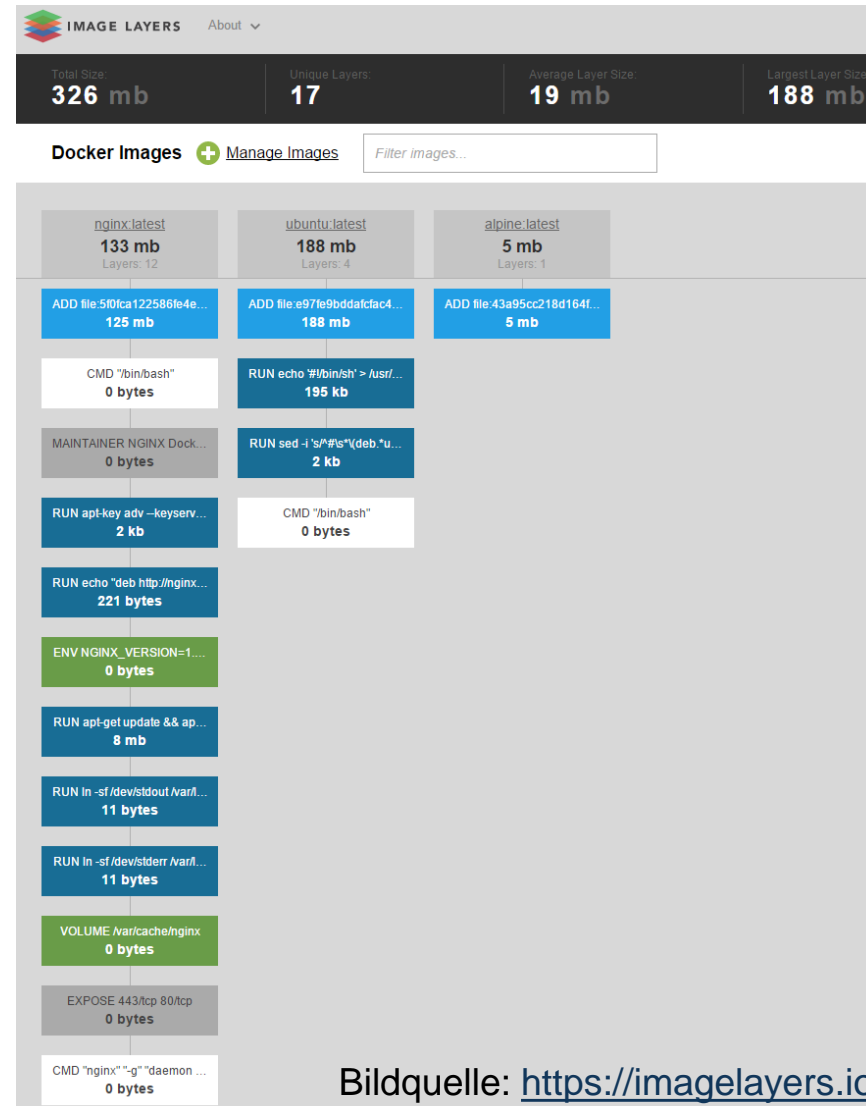


Ruhender und transportierbarer Zustand

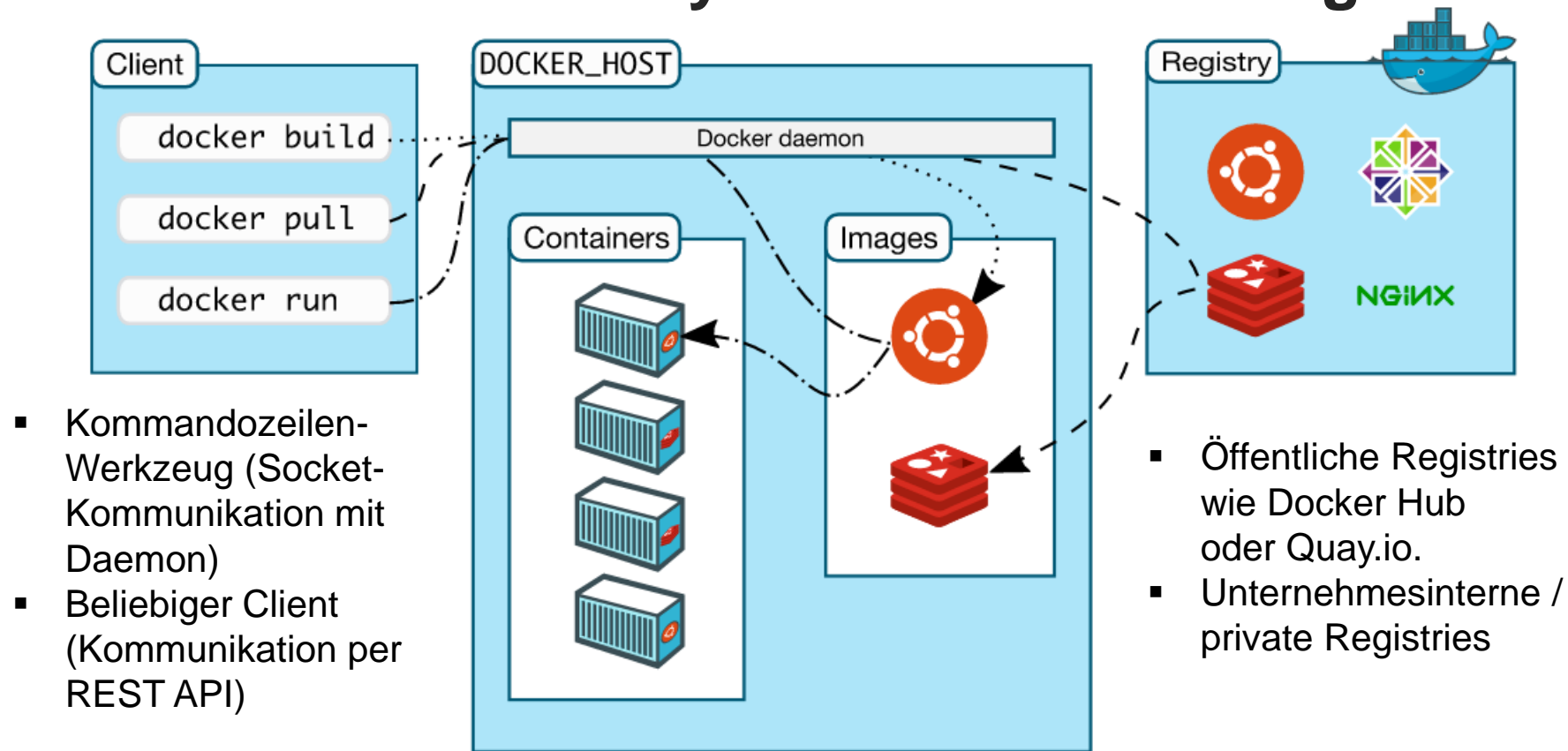
Laufender Zustand

Ein Container läuft so lange wie sein Entrypoint-Prozess im Vordergrund läuft. Docker merkt sich den Container-Zustand.

Visualisierung der Image Layer eines konkreten Images mit dem Werkzeug „Image Layers“.



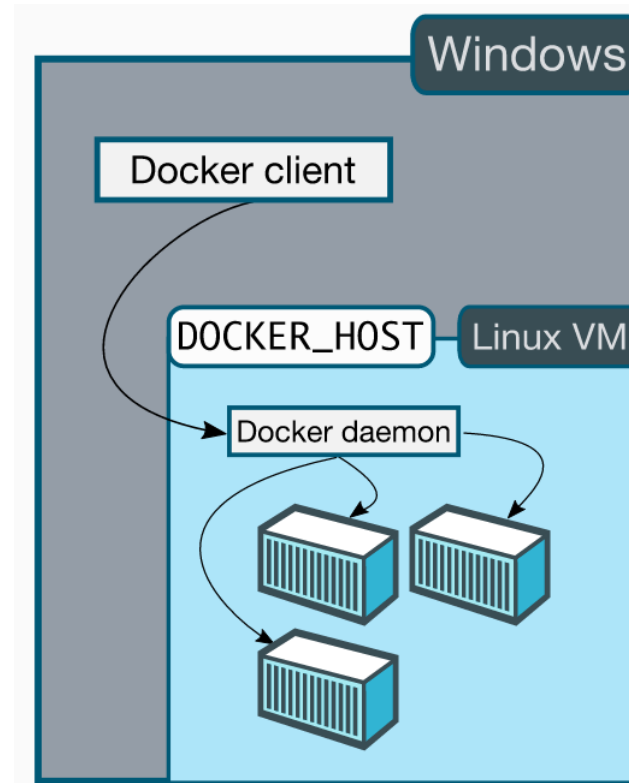
Docker ist eine Automationsumgebung für Anwendungs-Container auf Basis Betriebssystem-Virtualisierung.



Der Docker Daemon ist die zentrale Steuerungseinheit und läuft direkt als Prozess im Host-Betriebssystem. Er verwaltet alle lokalen Container und Images auf dem Host.

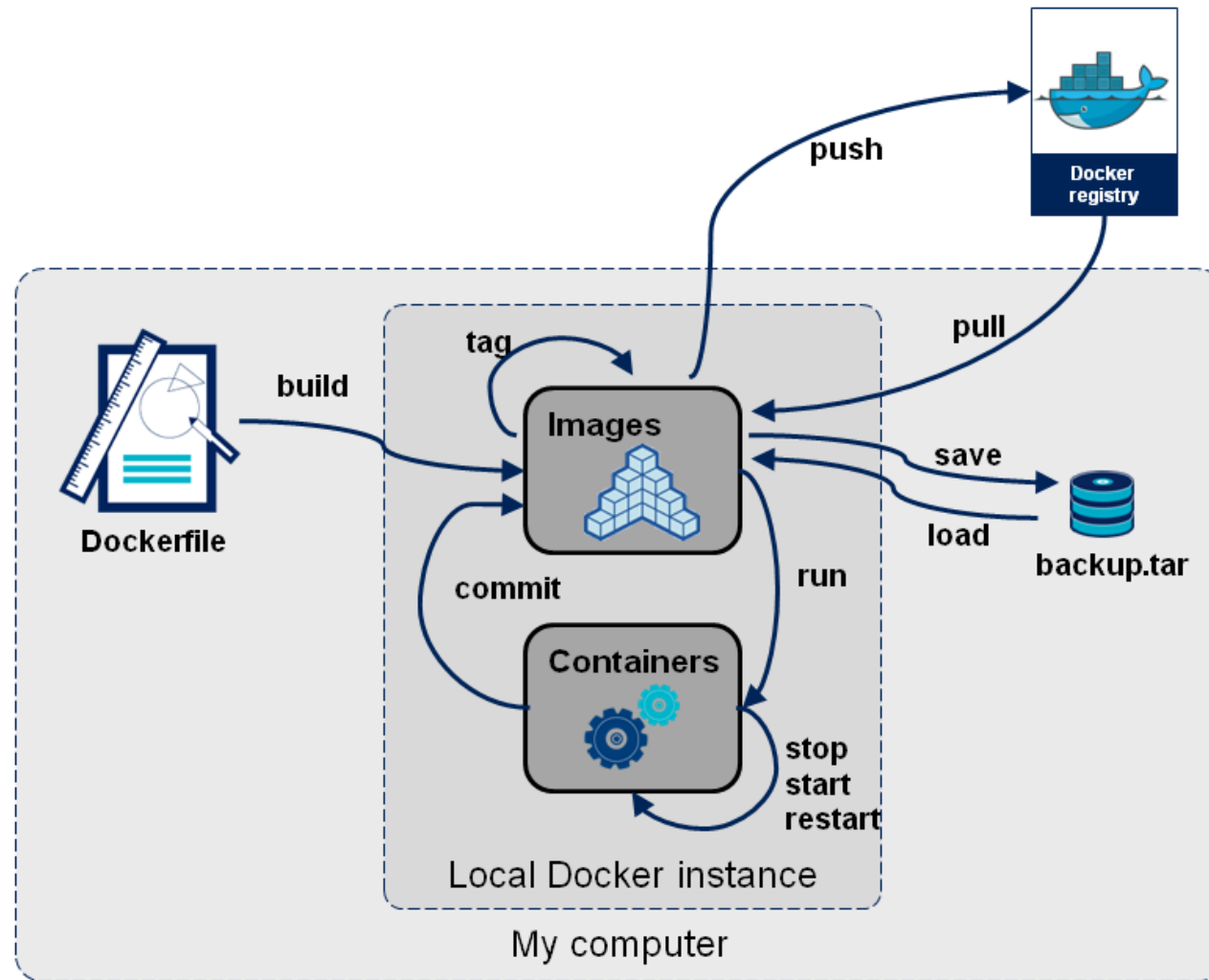
Bildquelle: <http://docs.docker.com/engine/introduction/understanding-docker>

Beispiel: boot2docker / Docker Machine



Bildquelle: <http://docs.docker.com/engine/installation/windows>

Das Big Picture von Docker.



Ein typischer Arbeitsablauf mit Docker.

Images:

- `busybox`: Mini-OS (2MB) für Testzwecke
- `alpine`: Mini-OS (5MB) mit Paketmanager
- `ubuntu`: Maxi-OS (188MB)

Befehle auf Kommandozeile	Bedeutung
<code>docker images</code>	Gibt alle lokalen Images aus
<code>docker run</code> <code>-d</code> <code>-v <volume mounts></code> <code>-p <host-port>:<container-port></code> <code><image> <entrypoint process></code>	Docker Image ausführen (typischer Container-Start) <ul style="list-style-type: none">▪ im Hintergrund▪ Mit Host-Verzeichnis im Guest gemountet▪ Mit Port-Forwarding von Host auf den Container▪ Image-Name und Einstiegsprozess
<code>docker run</code> <code>-ti</code> <code><image> /bin/sh</code>	Docker Image ausführen (mit Shell in den Container) <ul style="list-style-type: none">▪ Mit Konsolen-Forwarding (interaktive Konsole)▪ Image-Name und Shell (alternativ „/bin/bash“)
<code>docker ps -a</code>	
<code>docker commit <container> qaware/foo</code>	Docker Container als lokales Image speichern
<code>docker kill <container></code>	Docker Container beenden
<code>docker rmi -f <image></code>	Lokales Image löschen

Wichtige Befehle für das Container Debugging

Befehle auf Kommandozeile	Bedeutung
<code>docker inspect <container></code>	Metadaten zu einem Container ausgeben (z.B. IP)
<code>docker logs</code>	Syslog des Containers ausgeben
<code>docker top <container></code>	Die Prozesse in einem Container ausgeben (<code>ps -a</code> im Container)
<code>docker exec -ti <container> /bin/sh</code>	Per Kommandozeile in einen laufenden Container verbinden
<code>docker stats</code>	Laufzeit-Statistiken zu den laufenden Docker Containern (z.B. CPU-Verbrauch, IO-Intensität, ...)

Docker Befehle auf der Kommandozeile

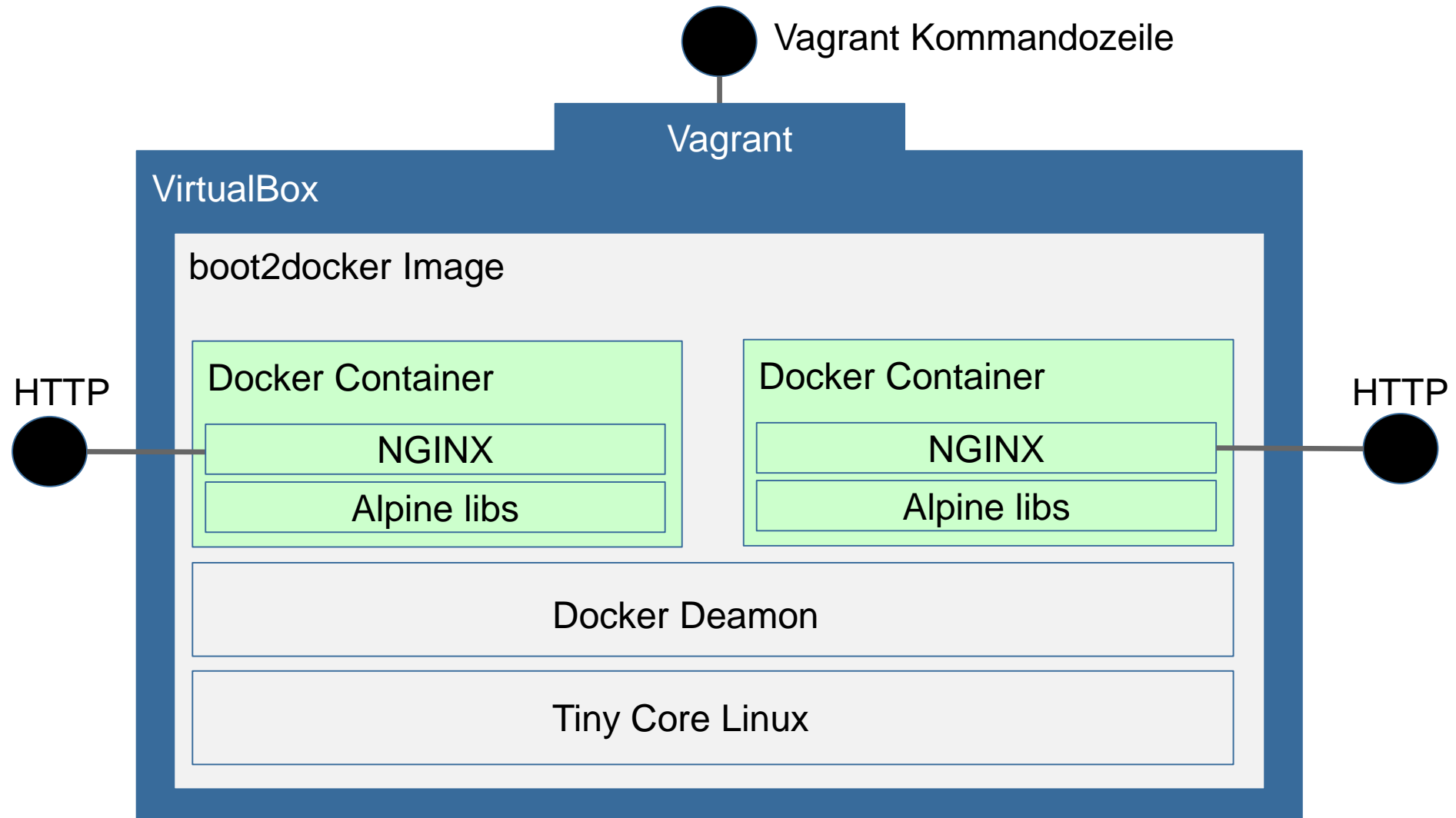
- docker create creates a container but does not start it.
- docker run creates and starts a container in one operation.
- docker stop stops it.
- docker start will start it again.
- docker restart restarts a container.
- docker rm deletes a container.
- docker kill sends a SIGKILL to a container.
- docker attach will connect to a running container.
- docker wait blocks until container stops.

Weitere Kommandos: <https://coderwall.com/p/2es5jw/docker-cheat-sheet-with-examples>, <https://docs.docker.com/reference>

Weiterführende Themen

- Das Docker Ökosystem
- Security-Mechanismen in Docker und Absicherung von Docker (Daemon benötigt aktuell noch root-Rechte!)
- Netzwerk-Themen jenseits des Wirings (z.B. DNS, NAT, ...)
- Die unterschiedlichen Docker Filesystem-Backends
- Produktionsreife Docker Container
- Auswahl der passenden Implementierungen (z.B. Filesystem)
- Monitoring von Docker Containern
- Orchestrierung und Verknüpfung von Docker-Containern
- Docker von Innen

Die Übung



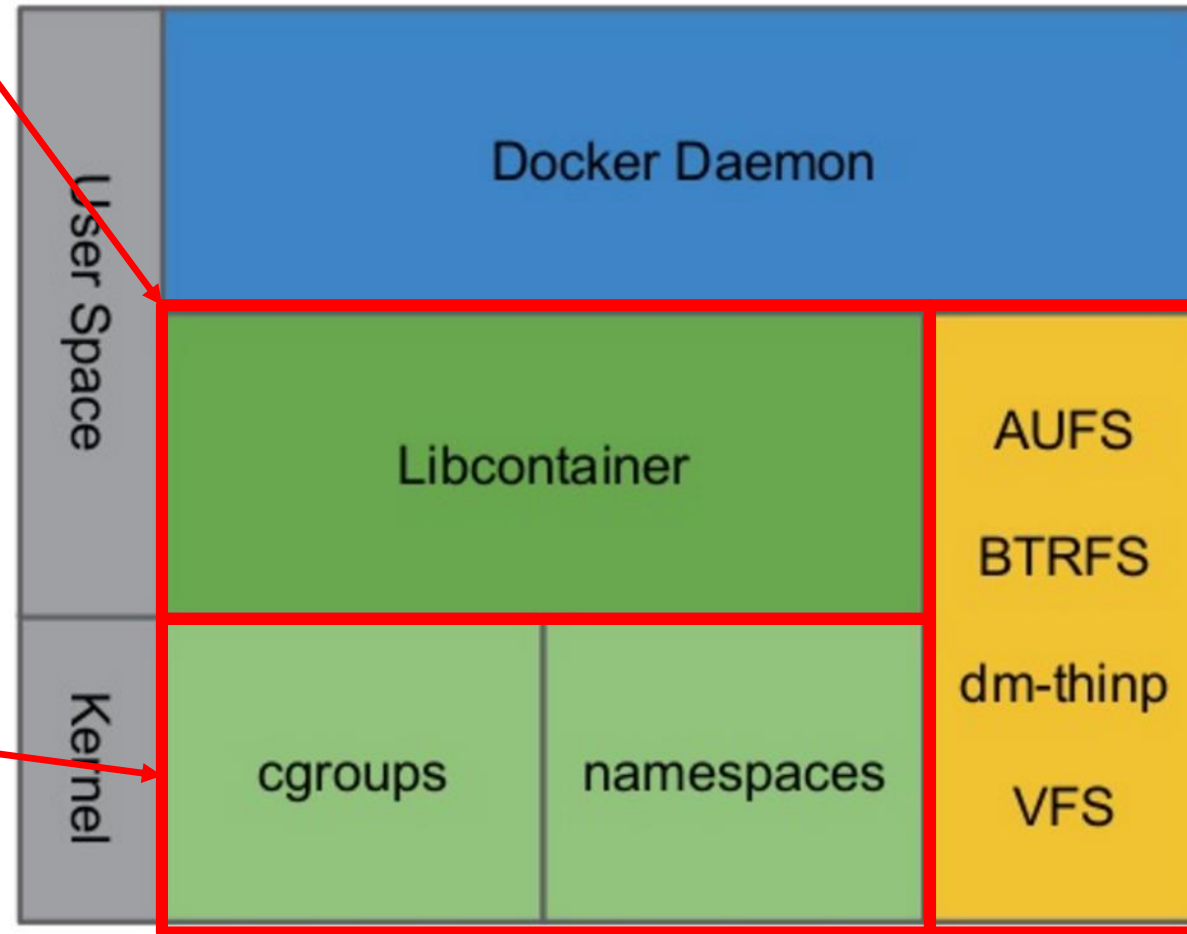
Anhang: Docker von Innen

Docker von Innen: Die Bausteine von Docker

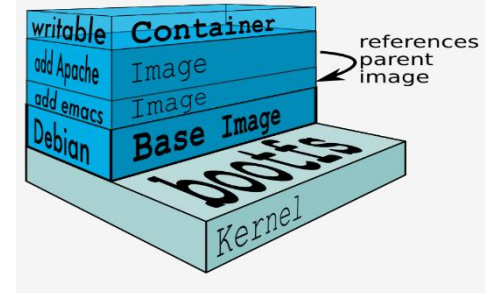
Entkopplung

Aktuell wird neben Libcontainer auch LXC und bald auch das standardisierte Format OCI (<https://www.opencontainers.org>) unterstützt.

Isolation



Multiplizität



<http://docs.docker.com/terms/layer/>

Union file systems (UnionFS)

- Ein Union File Systems funktioniert über Layers
- Diese werden zu einem gemeinsamen File System zusammen-gefasst

<http://de.slideshare.net/RohitJnagal/docker-internals>

Linux Cgroups (Isolation durch Grenzen)

- Ein Feature des Linux-Kernels, das maßgeblich durch Google entwickelt wurde
- Gruppiert Prozesse zu Gemeinschaften mit definiertem und beschränktem Ressourcen-Zugriff auf:
 - Prozessor
 - Hauptspeicher
 - I/O (insb. Netzwerk)
 - Disk
- Die Prozess-Gruppen können geschachtelt sein
- Cgroups stellt dabei für die Prozessgruppen sicher, dass
 - Die Ressourcen limitiert sind und die definierten Grenzen nicht überschritten werden
 - Die aktuell verbrauchten Ressourcen kontinuierlich gemessen und protokolliert werden
 - Dass bei Überschreitung der definierten Grenzen die Prozess-Gruppen eingefroren und neu gestartet werden

Linux Kernel Namespaces (Isolation durch Sichtbarkeit)

- Ein Feature des Linux-Kernels, das die Sicht auf das System einschränkt bzgl.
 - Prozessraum / Prozess-Ids
 - Netzwerk-Schnittstellen
 - Host-Name
 - Dateisystem-Mounts
 - IPC (Inter-Prozess-Kommunikation)
 - Benutzerkonten
- Namespaces können geschachtelt sein