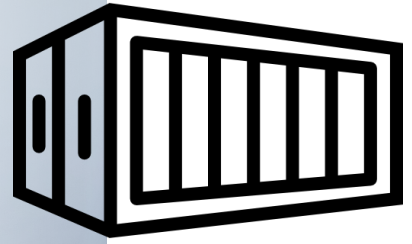
CLOUD NATIVE COMPUTING

CLOUD-NATIVE

Unit:

Container

(3) Container Laufzeitumgebungen (Docker)



creative commons

CLOUD NATIVE COMPUTING

Urheberrechtshinweise

Diese Folien werden zum Zwecke einer praktikablen und pragmatischen Nutzbarkeit im Rahmen der **CCo 1.0 Lizenz** bereitgestellt.

Sie dürfen die Inhalte also kopieren, verändern, verbreiten, mit eigenen Inhalten mixen, auch zu kommerziellen Zwecken, und ohne um weitere Erlaubnis bitten zu müssen.

Eine Nennung des Autors ist nicht erforderlich (aber natürlich gern gesehen, wenn problemlos möglich).

Diese Folien sind insb. für die Lehre an Hochschulen konzipiert und machen daher vom **§51 UrhG (Zitate)** Gebrauch.

Die CCo Lizenz überträgt sich nicht auf zitierte Quellen. Hier sind bei der Nutzung natürlich die Bedingungen der entsprechenden Quellen zu beachten.

Die Quellenangaben finden sich auf den entsprechenden Folien.

KAPITEL 8

Standardisierung von Deployment Units (Container)

8 Standardisierung von Deployment Units

8.1 Hintergrund (PaaS)

8.2 Betriebssystem-Virtualisierung

8.3 Container Runtime Environments
- Kernel-Namespaces
- Process Capabilities
- Control Groups
- Union Filesystems
- High- und Low-Level Container-Laufzeitumgebungen

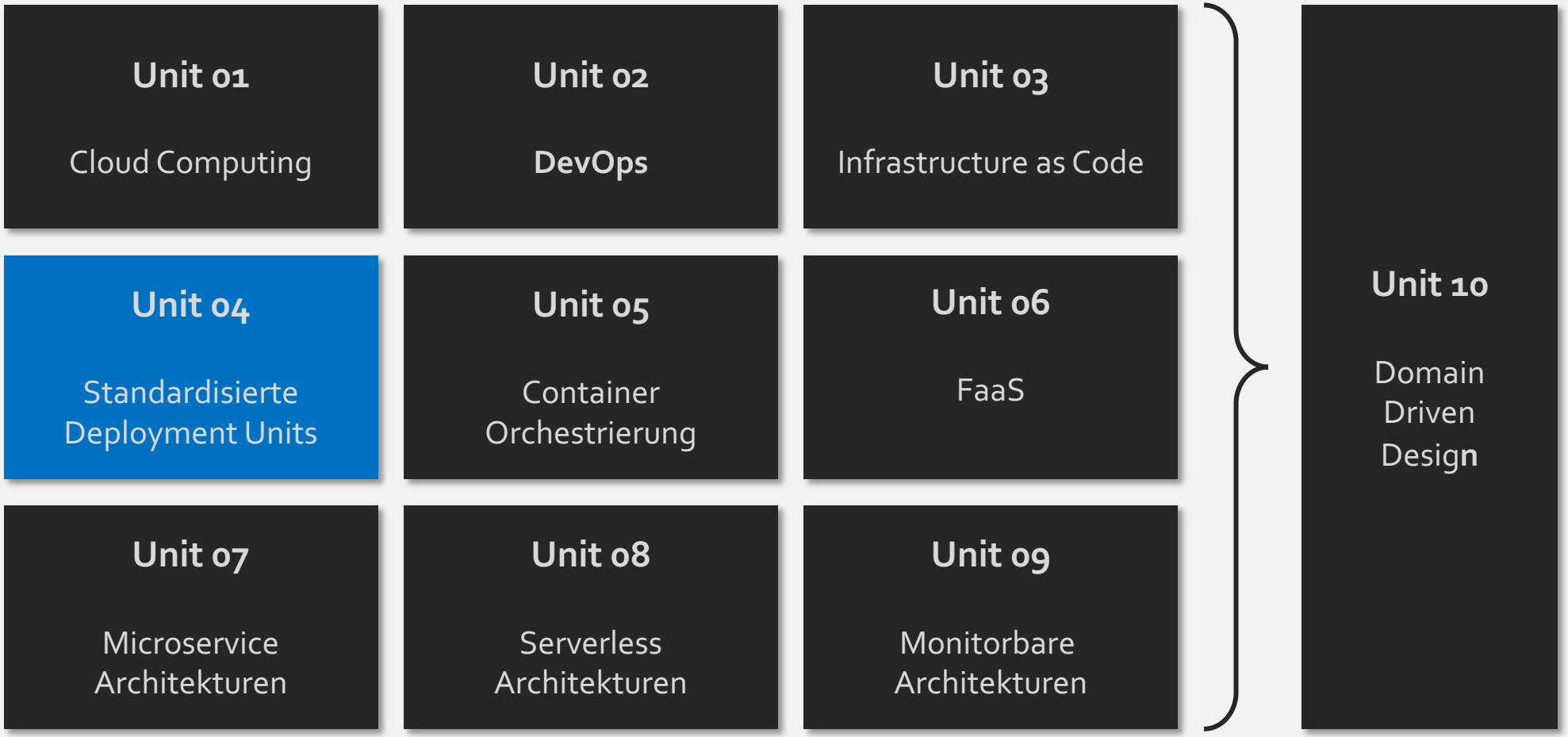
8.4 Bau und Bereitstellung von Container-Images

8.5 Faktoren gut betreibbarer Container
- Codebase
- Abhängigkeiten und Konfigurationen
- Unterstützende Services und Port Binding
- Build-, Release- und Run-Phase
- Horizontale Skalierung über Prozesse
- Umgebungen, Logs und Betrieb

8.6 Zusammenfassung

INHALTSVERZEICHNIS

Überblick über Units und Themen dieses Moduls



INHALTE

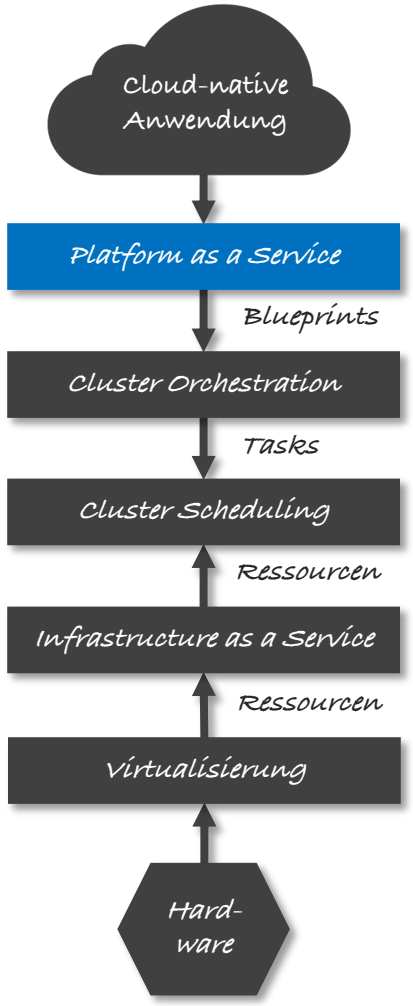
Hintergrund
- Platform as a Service
- Das PaaS-Problem
- Das CaaS-Versprechen

Betriebssystem-Virtualisierung
- OS-Virtualisierung
- Linux-basierte Techniken zur OS-Virtualisierung
- Standardisierung von Deployment-Einheiten => Container

Laufzeitumgebungen für Container
- Container Laufzeitumgebungen und Standards
- Docker
- Image Building und Registries

Container-Pattern
- Container (Anti-)Pattern
- 12-Factor Apps

CLOUD NATIVE COMPUTING



CONTAINER RUNTIME ENVIRONMENTS

Definition und Entstehungsgeschichte

CLOUD NATIVE COMPUTING

- Container sind keine First-Class Objekte im Linux-Kernel
- Container basieren auf u.a. auf Kernel-Features wie Namespaces, Control Groups und LSMs (Linux Security Modules)
- Container ermöglichen eine sichere, isolierte und überwachte Ausführungsumgebungen für Prozesse

- Es gibt zwei Haupttypen von Container Runtime Environments
  - Low-Level (Host-fokussiert) und
  - High-Level (Multi-Host-/Cluster-fokussiert)

Eine **Container-Runtime** ist eine Software, die die zum Ausführen von Containern erforderlichen Komponenten ausführt und verwaltet. Diese Tools erleichtern die sichere Ausführung und effiziente Bereitstellung von Containern unter Verwendung der genannten Kernel-Techniken.

U.a. Docker begann mit Bau von Tools rund um LXC, um Container entwickler- und benutzerfreundlicher zu gestalten. Docker gründete u.a. die **"Open Container Initiative (OCI)"**, um Containerstandards festzulegen, und öffnete einige ihrer Containerkomponenten als Open Source **libcontainer**-Projekt.

Docker mag die bekannteste und meist genutzte Container-Runtime Enviroment sein, aber sie ist nicht die einzige!

CONTAINER RUNTIME ENVIRONMENTS

Überblick über Low-Level OCI-Runtimes

CLOUD NATIVE COMPUTING

Zusätzlich zu nativen Laufzeiten, die den containerisierten Prozess auf demselben Host-Kernel ausführen, gibt es einige Sandbox- und virtualisierte Implementierungen der OCI-Spezifikation

Native Container Runtimes:

runC (go) ist das Ergebnis aller Arbeiten von Docker an libcontainer und OCI. Dies ist die De-facto-Standardlaufzeit für Low-Level-Container.

Railcar (Rust) war eine von Oracle entwickelte OCI Runtime-Implementierung und wurde in Rust geschrieben. Railcar wurde mittlerweile aufgegeben.

crun (C) ist eine von Redhat geleitete OCI-Implementierung. Es war eine der ersten Runtimes, die cgroups v2 unterstützten.

rkt ist eine teilweise OCI-kompatible Runtime-Implementierung. Es unterstützt das Ausführen von Docker- und OCI-Images ist jedoch nicht mit allen OCI-Schnittstellen kompatibel.

Sandboxed/Virtualized Runtimes:

gVisor und **Nabla** sind Beispiele für Runtime Umgebung, die eine weitere Isolierung des Hosts vom containerisierten Prozess ermöglichen. Der containerisierte Prozess wird auf einer Unikernel- oder Kernel-Proxy-Schicht ausgeführt, die dann im Namen des Containers mit dem Host-Kernel interagiert. Daher haben diese Runtimes eine reduzierte Angriffsoberfläche und schützen so den Host.

runV, **clearcontainers** und **kata-containers** sind Beispiele für virtualisierte Laufzeitumgebung. Dies sind Implementierungen der OCI Runtime-Spezifikation, die von einer Schnittstelle der virtuellen Maschine unterstützt werden. Sie starten eine virtuelle Maschine mit einem Standard-Linux-Kernel-Image und führen den "containerisierten" Prozess in dieser VM aus.

Low-Level OCI Runtimes fokussieren die Erstellung und Ausführung von Containern auf einem Host.

CONTAINER RUNTIME INTERFACE (CRI)

Erweiterte Anforderungen durch Container Plattformen, High-Level Container Runtimes

Bei der Einführung des Kubernetes-Container-Orchestrators war die Docker-Runtime noch fest integriert. Als Kubernetes jedoch schnell populär wurde, brauchte die Community alternative Laufzeitunterstützung.

Im Gegensatz zu OCI-Runtime Umgebungen, die das Erstellen von Containern auf einem Host fokussieren, verfügt ein CRI über die erforderliche Funktionalität, um Container in dynamischen Cloud-Umgebungen auszubringen. Darüber hinaus delegieren CRIs normalerweise die eigentliche Containerausführung an eine OCI-Runtime. Durch die Einführung des CRI können Container-Plattformen von der zugrunde liegenden Container-Laufzeit-Umgebung entkoppelt werden.

containerd ist Dockers High-Level-Runtime. Standardmäßig wird **runC** unter der Haube verwendet. Wie die übrigen Container-Tools, die von Docker stammen, handelt es sich um einen aktuellen De-facto-Standard. Containerd verfügt über ein Plugin-Design, um Low-Level-Runtimes wie Kata mittels containerd steuern zu können.

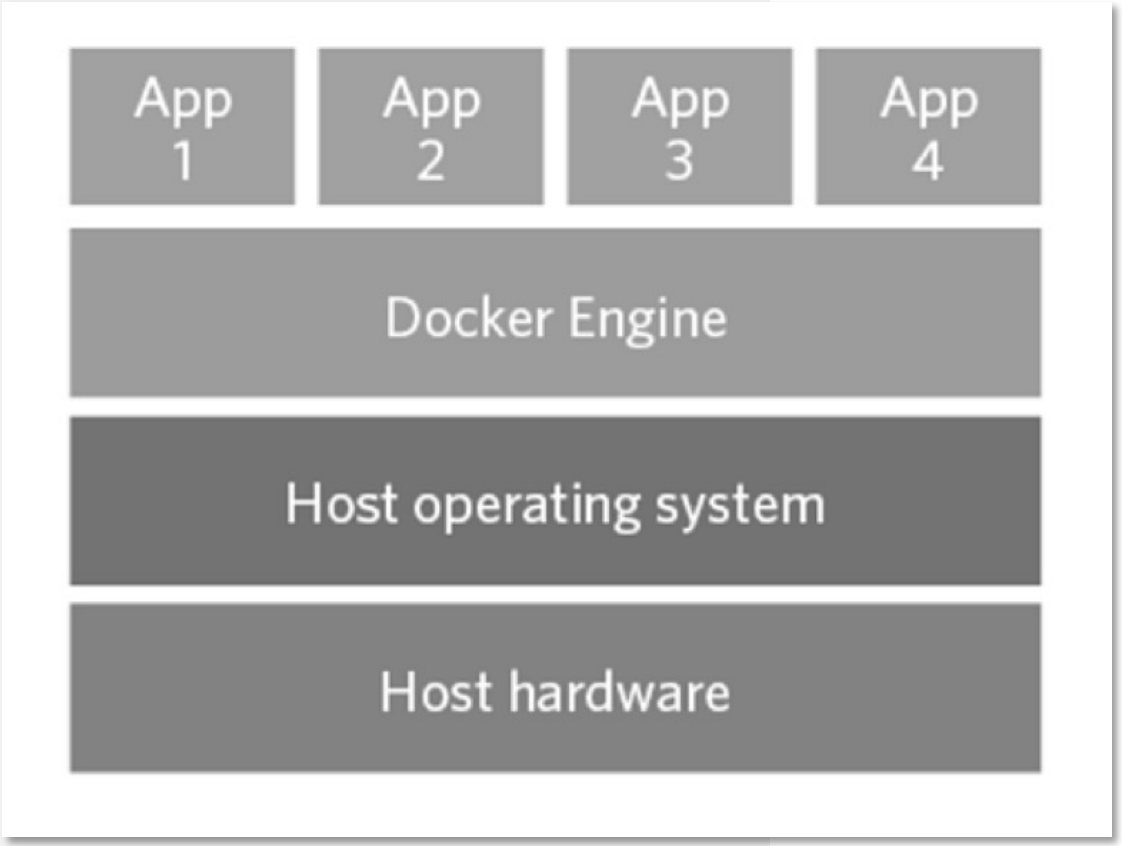
cri-o (Redhat) ist eine leichtgewichtige CRI-Implementierung, die speziell für Kubernetes entwickelt wurde. Es soll als leichte Brücke zwischen dem CRI und einer unterstützenden OCI-Laufzeit dienen. Standardmäßig verwendet cri-o **runC** als OCI-Runtime. Da es vollständig OCI-kompatibel ist, funktioniert cri-o aber auch mit OCI-konformen Low-Level Laufzeiten wie **Kata** oder **crun**.

High-Level CRI Runtimes fokussieren die Erstellung und Ausführung von Containern in geclusterten Container-Plattformen (z.B. Kubernetes).

CONTAINERIZATION

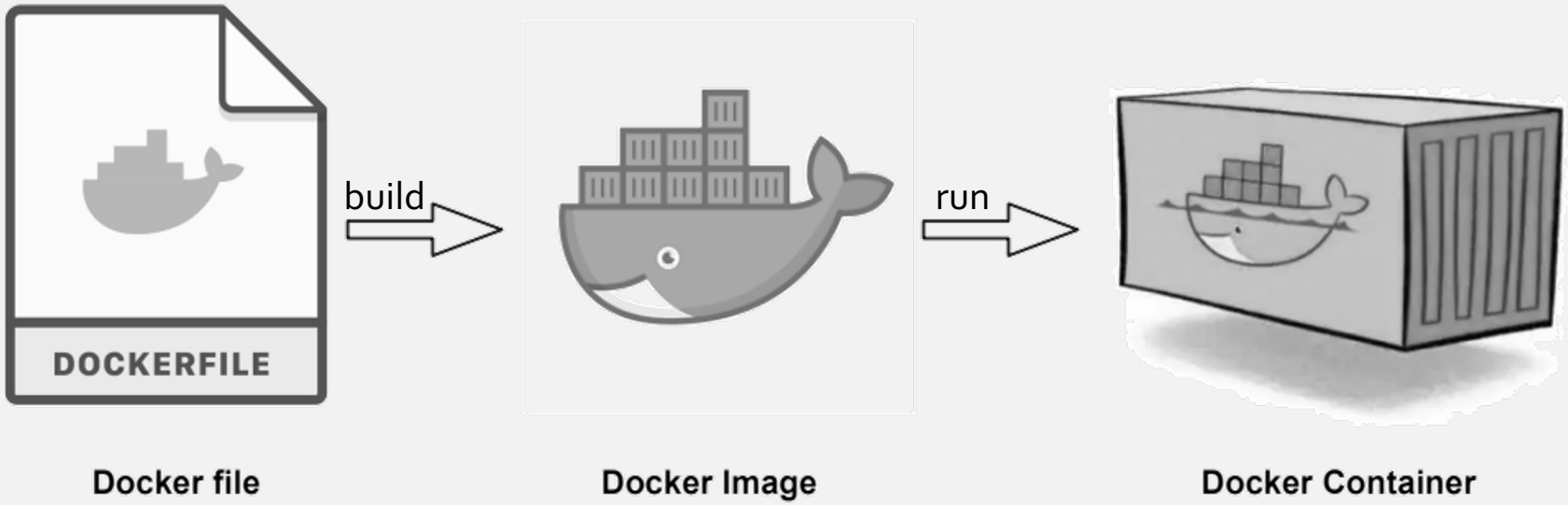
Mit Docker

- Docker ist eine komplette Laufzeitumgebung für (Linux-basierte) Betriebssystem-Virtualisierung.
- Docker ist als Werkzeug eines Cloud-Anbieters (dotCloud) entstanden und mittlerweile eines der sichtbarsten und aktivsten Open-Source-Ökosysteme.
- Docker unterstützt Linux, Windows und Mac OS (als virtualisierte OCI-Runtime).
- Docker hat das Container Ökosystem maßgeblich geprägt.
- Docker kann als De-Facto-Standard im Bereich der Containerisierung angesehen werden.



DOCKER

Images und Container



Docker file

Docker Image

Docker Container

Definition eines Images

„Image as Code"

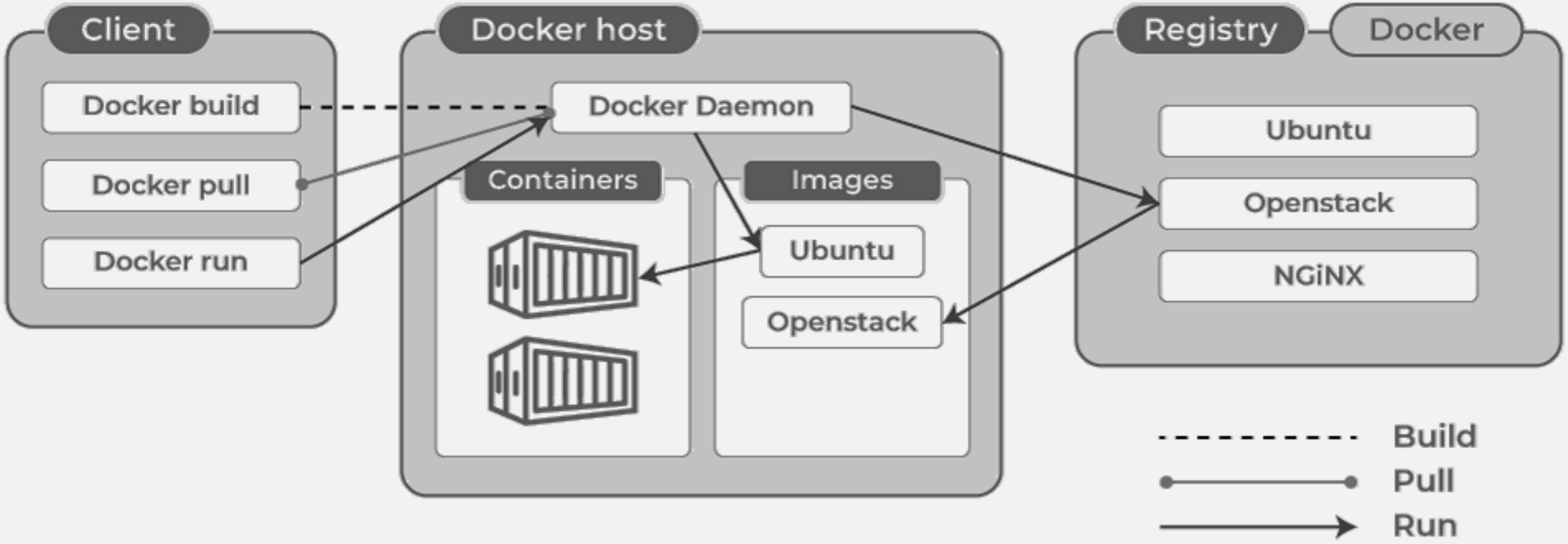
Ruhender und transportierbarer Zustand

Laufender Zustand

Ein Container läuft so lange wie sein Entrypoint-Prozess im Vordergrund läuft.

DOCKER

Architektur



Kommandozeilen-Werkzeug

`docker build, run, …`

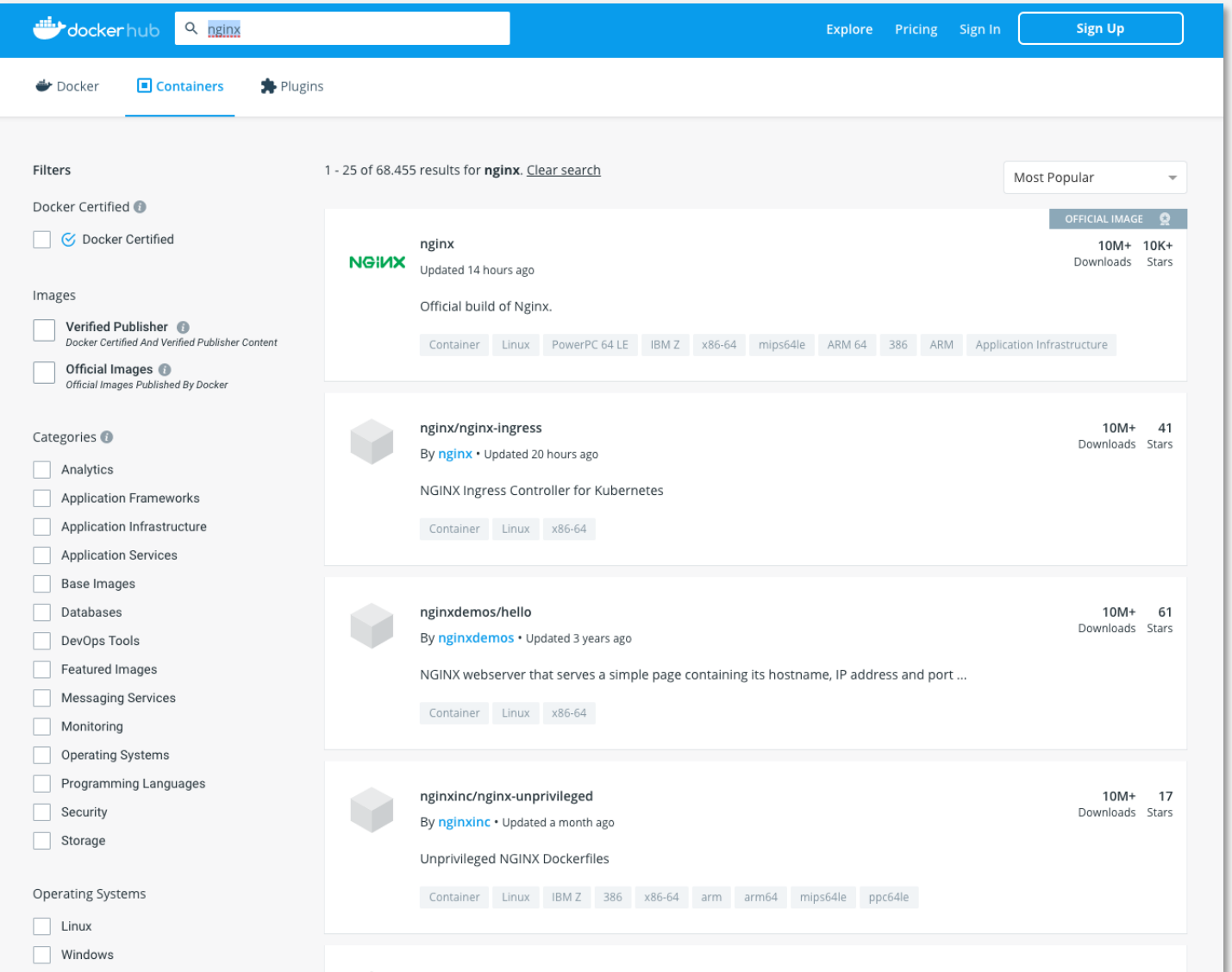
- **Zentrale Steuerungseinheit**
- Daemon-Prozess im Host-Betriebssystem
- Verwaltet alle lokalen Container und Images auf dem Host

- **Öffentliche Registries**
- (z.B. DockerHub, Quay.io)

- **Private Registries**
- (bspw. Gitlab Docker Image-Registry)

HUB.DOCKER.COM

Die öffentliche Standardregistry für Docker Images

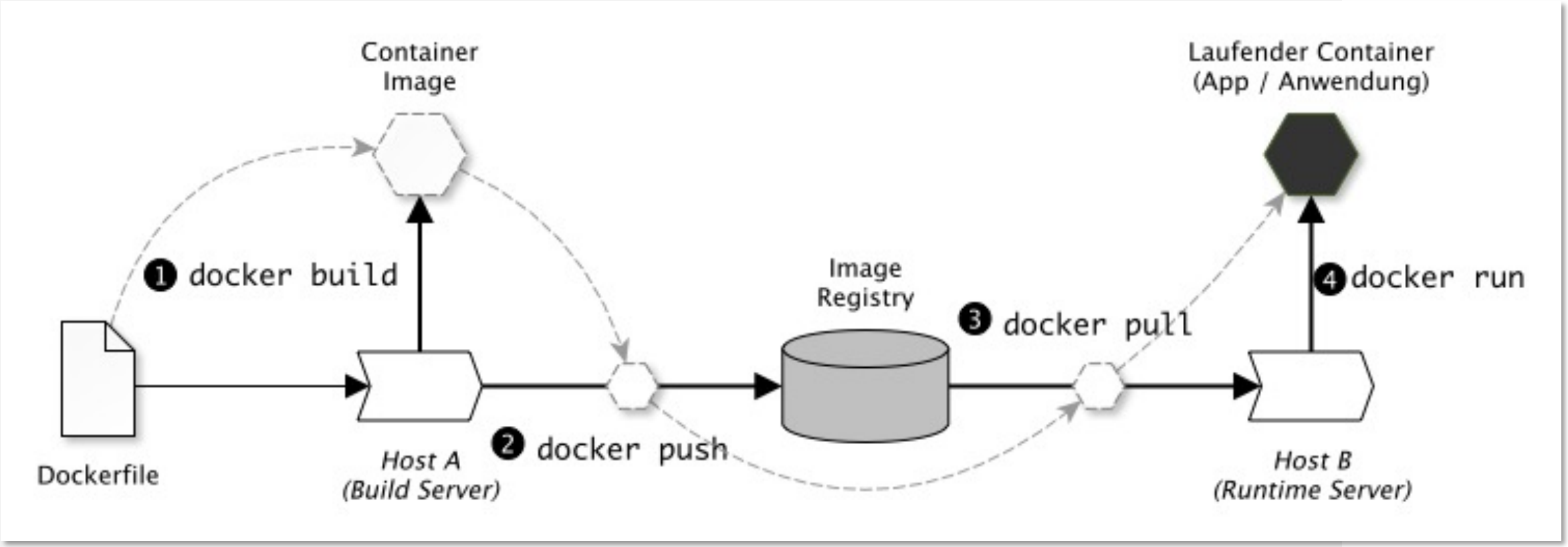


Aber es gibt weitere Registries, z.B.:

- **quay.io**
  (https://quay.io/search)
- **gcr.io**
  Google Container Registry
- **Treescale**
  (https://treescale.com)
- …

DOCKER

Workflow



Bei der lokalen Entwicklung sind der **Build Host A** und der **Runtime Host B** oft dieselbe Maschine

DOCKER

Dockerfile

```
FROM qaware/alpine-k8s-ibmjava8:8.0-3.10
LABEL maintainer="QAware GmbH <qaware-oss@qaware.de>"

RUN mkdir -p /app

COPY build/libs/zwitscher-service-1.0.1.jar /app/zwitscher-service.jar
COPY src/main/docker/zwitscher-service.conf /app/

ENV JAVA_OPTS -Xmx256m

EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/app/zwitscher-service.jar"]
```

Basis-Image von dem abgeleitet wird

Metainformationen

Führt ein Command im Container aus (wie in Shell)

Kopiert Dateien vom Host in den Contaier

Setzt Umgebungsvariablen im Container

Dokumentiert welche Ports zum Host exponiert werden können

Command, das ausgeführt wird, wenn Container gestartet wird

Quelle: QAware GmbH
https://github.com/qaware/cloudcomputing

DOCKER

Typische Kommandos eines Workflows

Command	Action
`docker build -t <image> .`	Build Docker image with given tag in current directory
`docker images`	Prints all local images

`docker run`
`  -d`
`  -v <volume mounts>`
`  -p <host-port>:<container-port>`
`  <image> <entrypoint process>`
	Run a Docker image: Creates and runs a container.
	▪ in background
	▪ with host directory mounted into the container
	▪ with port forwarding from host to container
	▪ image name (and optional entrypoint process)

`docker run`
`  -ti`
`  <image> /bin/sh`
	Run a Docker image and open a shell within the container
	▪ … with forwarding of local terminal
	▪ Image name and shell (or „`/bin/bash`“)

`docker ps -a`	Prints all containers (without –a = only running containers)
`docker commit <container> qaware/foo`	Store container as local image
`docker kill <container>` `docker rm <container>`	Terminate container (send SIGKILL to entrypoint process) Remove container
`docker rmi -f <image>`	Remove local image

DOCKER

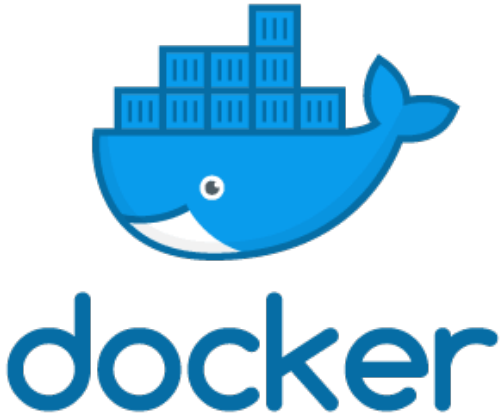
Nur Versuch macht kluch …

GitLab

CLOUD NATIVE COMPUTING

Klonen Sie dieses Repository:

```
git clone https://git.mylab.th-luebeck.de/cloud-native/lab-docker-demo.git
```



https://www.docker.com/get-started

```
# Bau eines Images
> docker build –t wikisearch .

# Ausführen eines Containers
> docker run –p 8888:80 wikisearch

# Zugriff auf einen Port
> http://localhost:8888?wiki=Lübeck
```

AUSBLICK

CLOUD NATIVE COMPUTING

Hintergrund

- Platform as a Service
- Das PaaS-Problem
- Das CaaS-Versprechen

Betriebssystem-Virtualisierung

- OS-Virtualisierung
- Linux-basierte Techniken zur OS-Virtualisierung
- Standardisierung von Deployment-Einheiten => Container

Laufzeitumgebungen für Container

- Container Laufzeitumgebungen und Standards
- Docker
- Image Building und Registries

Container-Pattern

- Container (Anti-)Pattern
- 12-Factor Apps

KONTAKT

Disclaimer

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de